MillenniumDB: An Interoperable & Efficient Graph Database

ABSTRACT

In this paper, we present MillenniumDB: a graph database engine designed around a novel graph data model, called domain graphs, that provides a simple abstraction upon which a variety of popular graph models can be supported. On top of this model, the engine adapts and combines tried and tested techniques from relational data management, state-of-the-art algorithms for worst-case-optimal joins, as well as graph-specific algorithms for evaluating path queries. We present the main design principles underlying MillenniumDB, the abstract graph model and query semantics supported, the concrete data model and query syntax implemented, as well as the custom storage, indexing, query planning and query evaluation techniques used. We evaluate MillenniumDB over real-world data and queries from the Wikidata knowledge graph, comparing various internal join and path algorithms, as well as various systems. We find that in its best configuration, MillenniumDB outperforms other popular persistent graph database engines (including both enterprise and open source alternatives) that support similar query features.

1 INTRODUCTION

Recent years have seen growing interest in graph databases [4], wherein nodes represent entities of interest, and edges represent relations between those entities. In comparison with alternative data models, graphs offer a flexible and often more intuitive representation of particular domains [3]. Graphs forgo the need to define a fixed (e.g., relational) schema for the domain upfront, and allow for modeling and querying cyclical relations between entities that are not well-supported in other data models (e.g., tree-based models, such as XML and JSON). Graphs have long been used as an intuitive way to model data in domains such as social networks, transport networks, genealogy, biological networks, etc. Graph databases further enable specific forms of querying, such as path queries that find entities related by arbitrary-length paths in the graph. Graph databases have become popular in the context of NoSQL [10], where alternatives to relational databases are sought for specialized scenarios; Linked Data [19], where graph-structured data are published and interlinked on the Web; and more recently Knowledge Graphs [22], where diverse data are integrated at large scale under a common graph abstraction.

Alongside this growing interest, recent years have seen a growing number of models, languages, techniques and systems for managing and querying graph databases [3, 4]. In the context of NoSQL systems, Neo4j [48], which uses the query language Cypher [14], is a leading graph database system in practice.[1] Other popular graph database systems include ArangoDB [34], JanusGraph [36], OrientDB [42], TigerGraph [44], etc., which support Gremlin [30] and other custom graph query languages. We also find graph database systems supporting the RDF data model and SPARQL query language [1], including Allegrograph [33], Amazon Neptune [35], Blazegraph [45], GraphDB [8], Jena TDB [37], Stardog [43], Virtuoso [12], and (many) more besides [1]. In summary, there now exist many graph database systems to choose from.

Many of these graph databases implement their own graph model, query language, etc. An open challenge is to design a graph database engine that is both *interoperable*, i.e., able to seamlessly support the diverse graph data models now popular in practice; and *efficient*, i.e., achieving query performance comparable (or ideally better than) systems built with a specific graph data model in mind. These goals are key to use-cases involving diverse, large-scale knowledge graphs. One concrete example is that of the Wikidata knowledge graph [47], which is composed of billions of statements, tens of thousands of node types, thousands of edge types, complex meta-data on edges, etc. Wikidata's query service – currently powered by Blazegraph [45] – receives in the order of millions of queries per day [26]. Interoperability in this setting would allow clients to seamlessly query Wikidata using their preferred syntax; in fact, as argued in Section 2, none of the graph models implemented by the aforementioned engines is well-suited for knowledge graphs like Wikidata, whereas our abstract graph model is designed with such knowledge graphs in mind. Efficiency in this setting would support more clients posing increasingly complex queries in less time.

Inspired by use-cases such as Wikidata, we propose MillenniumDB: an open-source graph database engine designed from the ground-up in order to achieve both interoperability and efficiency. To achieve *interoperability*, rather than implement the different data models from scratch, we rather adopt a common graph data model, called domain graphs, which can represent popular graph models in practice, and upon which MillenniumDB is founded. We then abstract the key query features common to different graph database engines, capturing them first as a formal query language over which concrete query syntax can be layered. Within this combination, we cover novel features, such as returning shortest paths (like Cypher) that match regular expressions (like SPARQL). In terms of *efficiency*, it is not trivial to know, *a priori*, which techniques are well-suited for evaluating real-world workloads over our model. To achieve efficient query processing, we thus incorporate a mix of both traditional and state-of-the-art techniques for evaluating graph patterns and path queries, adapting them to the domain graph model, and evaluating them empirically in a real-world setting to ascertain which offer the best performance in practice.

Contributions. The contributions of this paper are as follows:

- the domain graph and property domain graph data models, which allow for succinctly representing graph data models popular in practice, including RDF graphs, RDF-star graphs, property graphs, and the Wikidata knowledge graph [47];
- a formal query language based on domain graphs that captures key features of popular query languages for graph databases, along with a concrete query syntax;
- an indexing scheme and query engine designed for domain graphs that incorporates both traditional and state-of-the-art

[1] See, e.g., https://db-engines.com/en/ranking/graph+dbms

techniques, with optimizations dedicated to the evaluation of graph patterns and path queries;
- experiments over the Wikidata knowledge graph [47], involving real-world graph data and queries, comparing algorithms internal to MillenniumDB as well as other graph database engines.

Our experimental results highlight the benefits, for example, of incorporating worst-case-optimal join algorithms when evaluating complex graph patterns (with many joins) versus a more traditional approach based on applying binary joins with a Selinger-based query engine. We further compare the performance associated with different graph search algorithms in the context of path queries. On a more practical note, we show that MillenniumDB, under optimal configurations, clearly outperforms prominent graph database systems – namely Blazegraph, Neo4j, Jena and Virtuoso – and discuss why. We further make available a first release of MillenniumDB as an open source graph database engine [38], which we plan to extend in future in order to support more query syntax, query features, transactional updates, index structures, and more.

Paper structure. The rest of this paper is structured as follows: In Section 2, we discuss graph data models that are popular in current practice, and propose *domain graphs* as an abstraction of these models used in MillenniumDB. In Section 3, we describe the query language of MillenniumDB, and how it takes advantage of domain graphs. In Section 4, we explain how MillenniumDB stores data and evaluates queries. In Section 5, we provide an experimental evaluation of the proposed methods on a large body of queries over the Wikidata knowledge graph. In Section 6, we provide some concluding remarks and ideas for future research.

Supplementary material. The source code of MillenniumDB is provided in full at [38]. Experimental data is given at [40]. We also provide an extended version of the paper at [39], with additional details on the query syntax and semantics used in MillenniumDB.

2 DATA MODEL

In this section, we present the graph data model upon which MillenniumDB is based, and discuss how it generalizes existing graph data models such as RDF and property graphs. We also show its utility in concisely modeling real-world knowledge graphs that contain higher-arity relations, such as Wikidata [47].

2.1 Domain Graphs

The structure of knowledge graphs is captured in MillenniumDB via *domain graphs*, which follow the natural idea of assigning ids to edges in order to capture higher-arity relations within graphs [20, 24, 25]. Formally, assume a universe Obj of objects (ids, strings, numbers, IRIs, etc.). We define domain graphs as follows:

Definition 2.1. A *domain graph* $G = (O, \gamma)$ consists of a finite set of objects $O \subseteq \text{Obj}$ and a partial mapping $\gamma : O \to O \times O \times O$.

Intuitively, O is the set of database objects and γ models edges between objects. If $\gamma(e) = (n_1, t, n_2)$, this states that the edge (n_1, t, n_2) has id e, type t, and links the source node n_1 to the target node n_2.[2]

[2]Herein, we say "*edge type*" rather than "*edge label*" to highlight that the type forms part of the edge, rather than being an annotation on the edge, as in property graphs.

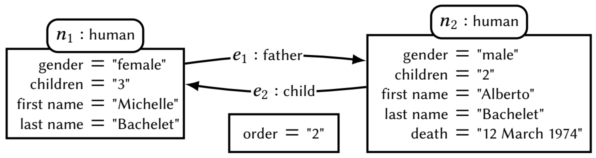


Figure 1: A property graph with two nodes and two edges. We use the order property on edge e_2 to indicate that Michelle Bachelet is the second child of Alberto Bachelet

We can analogously define our model as a relation:

$$\textsc{DomainGraph}(\text{source}, \text{type}, \text{target}, \underline{\text{eid}})$$

where eid (edge id) is a primary key of the relation.

The domain graph model of MillenniumDB already subsumes the RDF graph model [11]. Recall that an RDF graph is a set of triples of the form (a, b, c). An RDF graph can be visualized as a *directed labeled graph* [3, 27], which is a set of edges of the form $\textcircled{a} \xrightarrow{b} \textcircled{c}$, where a is the source node (aka *subject*), b the edge type (aka *predicate*), and c the target node (aka *object*). Alternatively, an RDF graph can be seen as a relation RDFGraph(source, type, target). To show how RDF is modeled in domain graphs, consider the following edge, claiming that Michelle Bachelet was the president of Chile.

$$\begin{tikzcd}\boxed{\text{Michelle Bachelet}} \arrow[r, "\textit{position held}"] & \boxed{\text{President of Chile}}\end{tikzcd}$$

We can encode this triple in a domain graph by storing the tuple (Michelle Bachelet, position held, President of Chile, e) in the DomainGraph relation, where e denotes a unique (potentially autogenerated) edge id, or equivalently stating that:

$$\gamma(\text{e}) = (\text{Michelle Bachelet, position held, President of Chile}).$$

The id of the edge itself is not needed in the RDF data model, but it can be used for modeling RDF-star (RDF*) graphs [17, 18] or named graphs based on RDF, as we will discuss in Section 2.2.

Property graphs [3] further allow nodes and edges to be annotated with labels and property–value pairs, as shown in Figure 1. Domain graphs can directly capture property graphs, where, for example, the property–value pair (gender, "female") on node n_1 can be represented by an edge $\gamma(e_3) = (n_1, \text{gender}, \text{female})$, the property–value pair (order, "2") on an edge e_2 becomes $\gamma(e_4) = (e_2, \text{order}, 2)$, the label on n_1 becomes $\gamma(e_5) = (n_1, \text{label}, \text{human})$, the label on e_1 becomes the type of the edge $\gamma(e_1) = (n_1, \text{father}, n_2)$, etc.[3] However, given a legacy property graph, there are some potential "incompatibilities" with the resulting domain graph; for example, strings like "male", labels like human, etc., now become nodes in the graph, generating new paths through them that may affect query results.

For stricter backwards compatibility with legacy property graphs (where desired), MillenniumDB implements a simple extension of the domain graph model, called *property domain graphs*, which allows for *external annotation*, i.e., adding labels and property–value pairs to nodes and edges without creating new nodes and edges. Formally, if $\mathcal{L}$ is a set of labels, $\mathcal{P}$ a set of properties, and $\mathcal{V}$ a set of values, we define a property domain graph as follows:

[3]To represent edges in property graphs that permit multiple labels, multiple edges with different types can be added (or the labels can be added on the edge ids).

Definition 2.2. A *property domain graph* is defined as a tuple $G = (O, \gamma, \mathtt{lab}, \mathtt{prop})$, where:

- (O, γ) is a domain graph;
- $\mathtt{lab} : O \to 2^{\mathcal{L}}$ is a function assigning a finite set of labels to an object; and
- $\mathtt{prop} : O \times \mathcal{P} \to \mathcal{V}$ is a partial function assigning a value to a certain property of an object.

Moreover, we assume that for each object $o \in O$, there exists a finite number of properties $p \in \mathcal{P}$ such that $\mathtt{prop}(o, p)$ is defined.

Domain graphs thus capture the graph structure of our model, while property domain graphs also permit annotating that graph structure with labels and property–value pairs. The property graph in Figure 1 can be represented with the following property domain graph $G = (O, \gamma, \mathtt{lab}, \mathtt{prop})$, where the graph structure is as follows:

$$O = \{n_1, n_2, e_1, e_2\}, \qquad \begin{aligned}\gamma(e_1) &= (n_1, \text{father}, n_2),\\ \gamma(e_2) &= (n_2, \text{child}, n_1),\end{aligned}$$

and the annotations of the graph structure are as follows:

lab(n_1) = human	prop(n_1, last name) = "Bachelet"
lab(n_2) = human	prop(n_2, gender) = "male"
prop(e_2, order) = "2"	prop(n_2, children) = "2"
prop(n_1, gender) = "female"	prop(n_2, first name) = "Alberto"
prop(n_1, children) = "3"	prop(n_2, last name) = "Bachelet"
prop(n_1, first name) = "Michelle"	prop(n_2, death) = "12 March 1974"

The relational representation of property domain graph then adds two new relations alongside DomainGraph:

$$\text{Labels}(\text{object}, \text{label}),$$
$$\text{Properties}(\underline{\text{object}, \text{property}}, \text{value}),$$

where $\underline{\text{object}, \text{property}}$ is a primary key of the second relation, with the first relation allowing multiple labels per object.

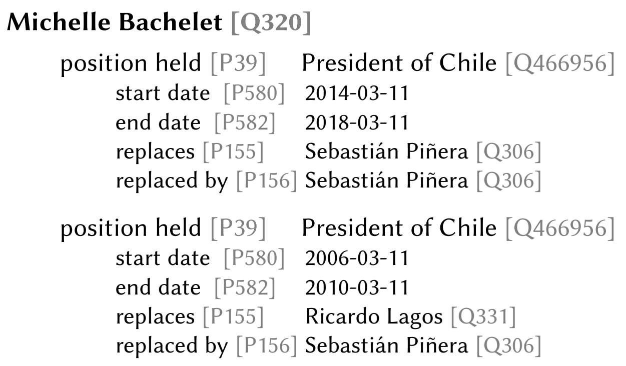
2.2 Why domain graphs?

Why did we choose domain graphs as the model of MillenniumDB? As discussed in the previous section, it can be used to model both directed labeled graphs (like RDF) as well as property graphs. It also has a natural relational expression, which facilitates its implementation in a query engine. But it is also heavily inspired by the needs of real-world knowledge graphs like Wikidata [22, 31, 47].

Consider the two Wikidata statements shown in Figure 2. Both statements claim that Michelle Bachelet was a president of Chile, and both are associated with nested *qualifiers* that provide additional information: in this case a start date, an end date, who replaced her, and whom she was replaced by. There are two statements for two distinct presidencies. Also the ids for objects (for example, Q320 and P39) are shown; any positional element can have an id and be viewed as a node in the knowledge graph.

Representing statements like this in a directed labeled graph requires some form of *reification* to decompose n-ary relations into binary relations [20]. For example, Figure 3 shows a graph where e_1 and e_2 are nodes representing n-ary relationships. The reification is given by the use of the edges typed as source, type and target. As before, we use human-readable nodes and labels, where in practice, the node $\boxed{\text{Sebastián Piñera}}$ will rather be given as the identifier $\boxed{\text{Q306}}$, and the edge type "replaces" will rather be given as "P155".

A number of graph models have been proposed to capture higher-arity relations more concisely, including property graphs [14] and



Michelle Bachelet [Q320]

position held [P39]		President of Chile [Q466956]
	start date [P580]	2014-03-11
	end date [P582]	2018-03-11
	replaces [P155]	Sebastián Piñera [Q306]
	replaced by [P156]	Sebastián Piñera [Q306]
position held [P39]		President of Chile [Q466956]
	start date [P580]	2006-03-11
	end date [P582]	2010-03-11
	replaces [P155]	Ricardo Lagos [Q331]
	replaced by [P156]	Sebastián Piñera [Q306]

Figure 2: Wikidata statement group for Michelle Bachelet

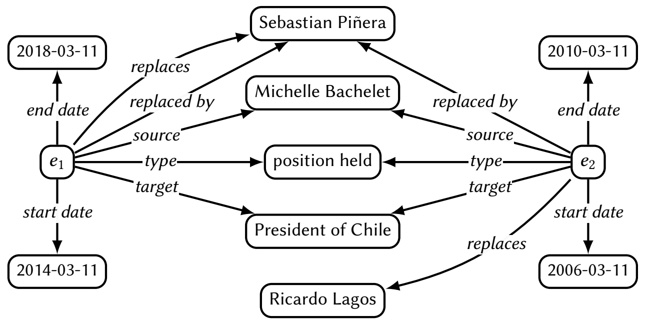
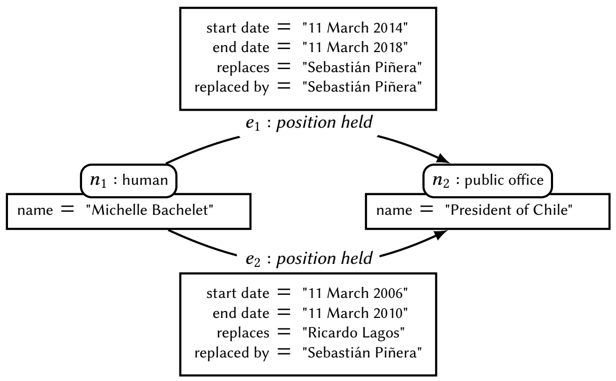


Figure 3: Directed labeled graph reifying the statements of Figure 2

RDF* [17, 18]. However, both have limitations that render them incapable of modeling the statements shown in Figure 2 without resorting to reification [23]. On the one hand, property graphs allow labels and property–value pairs to be associated with both nodes and edges. For example, the statements of Figure 2 can be represented as the property graph:


start date = "11 March 2014"
end date = "11 March 2018"
replaces = "Sebastián Piñera"
replaced by = "Sebastián Piñera"
e_1 : position held
n_1 : human
name = "Michelle Bachelet"
n_2 : public office
name = "President of Chile"
e_2 : position held
start date = "11 March 2006"
end date = "11 March 2010"
replaces = "Ricardo Lagos"
replaced by = "Sebastián Piñera"

Though more concise than reification, labels, properties and values are considered to be simple strings, which are disjoint with nodes; for example, "Ricardo Lagos" is neither a node nor a pointer to a node, but a string, which would complicate, for example, querying for the parties of presidents that Michelle Bachelet replaced.

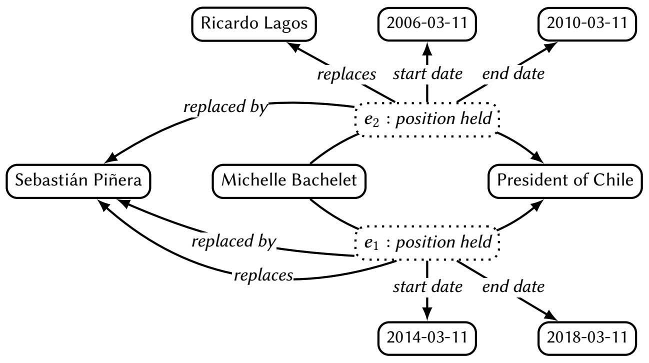
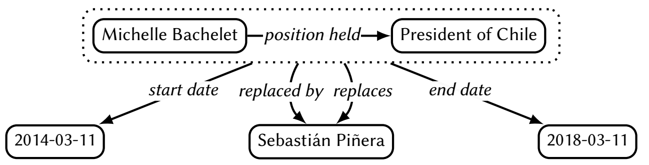


Figure 4: Domain graph for Figure 2

On the other hand, RDF* allows an edge to be a node. For example, the first statement of Figure 2 can be represented as follows in RDF*:



The node representing the edge is called a quoted triple [18]. However, we can only represent one of the statements (without reification), as we can only have one distinct node per edge; if we add the qualifiers for both statements, then we would not know which start date pairs with which end date, for example.[4]

The domain graph model allows us to capture higher-arity relations more directly. In Figure 4 we have one possible representation of the statements from Figure 2. We only show edge ids as needed (all edges have ids). We do not use the "property part" of our data model for external annotation, considering that the elements of Wikidata statements shown can form nodes in the graph itself.

Domain graphs are similar to *named graphs* in RDF/SPARQL. Both domain graphs and named graphs can be represented as quads. However, the edge ids of domain graphs identify each quad, which, as we will discuss in Section 4, necessitates fewer index permutations. Named graphs were proposed to represent multiple RDF graphs for publishing and querying. SPARQL thus does not support querying paths that span named graphs; to support path queries over singleton named graphs, all edges would need to be duplicated (virtually or physically) into a single graph [20]. Named graphs could be supported in domain graphs using a reserved term graph, and edges of the form $\gamma(e_3) = (e_1, \text{graph}, g_1)$, $\gamma(e_4) = (e_2, \text{graph}, g_1)$; optionally, *named domain graphs* could be considered in the future to support multiple domain graphs with quins.

The idea of assigning ids to edges/triples for similar purposes as described here is a natural one, and not new to this work. Hernández et al. [20] explored using singleton named graphs in order to represent Wikidata qualifiers, placing one triple in each named graph, such that the name acts as an id for the triple. In parallel with our work, recently a data model analogous to domain graphs has been independently proposed for use in Amazon Neptune, which the authors call 1G [25]. Their proposal does not discuss a formal definition for the model, nor a query language, storage and indexing, implementation, etc., but the reasoning and justification that they put forward for the model is similar to ours. To the best of our knowledge, our work is the first to describe a query language, storage and indexing schemes, query planner – and ultimately a fully-fledged graph database engine – built specifically for this model. Furthermore, with property domain graphs, we support annotation external to the graph, which we believe to be a useful extension that enables better compatibility with property graphs.

[4]A proposed workaround involves adding intermediate nodes to denote different *occurrences* of quoted triples, but this requires a reserved term [18].

Table 1: The features supported by graph models without reserved terms (NG = Named Graphs, PG = Property Graphs, DG = Domain Graphs, PDG = Property Domain Graphs)

	RDF	RDF*	NG	PG	DG	PDG
Edge type/label	✓	✓	✓	✓	✓	✓
Node label	✗	✗	✗	✓	✗	✓
Edge annotation	✗	✓	✓	✓	✓	✓
Node annotation	✓	✓	✓	✓	✓	✓
External annotation	✗	✗	✗	✓	✗	✓
Edge as node	✗	✓	✓	✗	✓	✓
Edge as nodes	✗	✗	✓	✗	✓	✓
Nested edge nodes	✗	✓	✓	✗	✓	✓
Graph as node	✗	✗	✓	✗	✗	✗

Table 1 summarizes the features that are directly supported by the respective graph models themselves without requiring *reserved terms*, which would include, for example, source, label and target in Figure 3 (all features except *External annotation* can be supported in all models with reserved vocabulary). Reserved terms can add indirection to modeling (e.g., reification [20]), and can clutter the data, necessitating more tuples or higher-arity tuples to store, leading to more joins and/or index permutations. The features are then defined as follows, considering directed (labeled) edges:

- *Edge type/label*: assign a type or label to an edge.
- *Node label*: assign labels to nodes.
- *Edge annotation*: assign property–value pairs to an edge.
- *Node annotation*: assign property–value pairs to a node.
- *External annotation*: nodes/edges can be annotated without adding new nodes or edges.
- *Edge as node*: an edge can be referenced as a node (this allows edges to be connected to nodes of the graph).
- *Edge as nodes*: a single unique edge can be referenced as multiple nodes.
- *Nested edge nodes*: an edge involving an edge node can itself be referenced as a node, and so on, recursively.
- *Graph as node*: a graph can be referenced as a node.

Some unsupported features in Table 1 are more benign than others; for example, *Node label* requires a reserved term (e.g., rdf:type), but no extra tuples; on the other hand, *Edge as node* requires reification, using at least one extra tuple, and also a reserved term.

Wikidata requires *Edge as nodes* for cases as shown in Figure 2 (since values such as Ricardo Lagos are themselves nodes). Only named graphs, domain graphs and property domain graphs can model such examples without reserved terms. Comparing named graphs and domain graphs, the latter sacrifices the "*Graph as node*" feature without reserved vocabulary to reduce indexing permutations (discussed in Section 4). Such a feature is not needed by Wikidata, but could be added in future versions. Property domain graphs further support external annotation, and thus better compatibility with legacy property graphs.

3 QUERY LANGUAGE

In this section, we present the syntax and semantics of the query language provided by our system, and then provide a comparison with other graph query languages.

3.1 Domain Graph Queries

Per our goal of supporting multiple graph models, MillenniumDB aims to support a number of graph query languages. However, no existing query language would take full advantage of the property domain graph model defined in the previous section. We have thus implemented a base query language, called DGQL, which closely resembles Cypher [14], but is adapted for the property domain graph model, and adds features of other query languages, such as SPARQL, that are commonly used for querying knowledge graphs [9, 22]. Herein we provide a guided tour of the syntax of DGQL. For space reasons, a formal definition of the semantics and syntax of the language is rather presented in an extended version [39].

A DGQL query takes the following high-level form:

```
MATCH Pattern
WHERE Filters
RETURN Variables
```

When evaluated over a property domain graph, such a query will return a multiset of mappings binding `Variables` to database objects (or values) that satisfy the `Pattern` specified in the **MATCH** clause and the `Filters` specified in the **WHERE** clause.

Querying objects. The most basic query will return all the objects (or more precisely, their ids) in our property domain graph. In MillenniumDB we can achieve this via the following query:

```
MATCH (?x)
RETURN ?x
```

Of course, one usually wants to select objects with a certain label, or a certain value in a specific property. For instance, if we want to select all people in the property (domain) graph from Figure 1 with two children, we could do this as follows:

```
MATCH (?x :human { children : 2 })
RETURN ?x, ?x.gender
```

This returns the ids of nodes with label human and value 2 for the property children, along with their value for the property gender, i.e., a single result $\{\{$?x $\mapsto n_2$, ?x.gender $\mapsto$ "male"$\}\}$. If n_2 did not have a gender, we would still want to return the node that makes the match, but signal that the gender is not specified. To do so, we would return $\{\{$?x $\mapsto n_2$, ?x.gender $\mapsto$ null$\}\}$.

If we wish to specify a range, we can rather use the **WHERE** clause:

```
MATCH (?x :human)
WHERE ?x.children >= 2
RETURN ?x, ?x.gender
```

which returns two solutions: $\{\{$?x $\mapsto n_2$, ?x.gender $\mapsto$ "male"$\}$, $\{$?x $\mapsto n_1$, ?x.gender $\mapsto$ "female"$\}\}$. If we replace >= with ==, the results would be the same as for the previous query.

Querying edges. In order to query over edges, we can write the following query, which returns γ, i.e., the relation DomainGraph:

```
MATCH (?x)-[?e :?t]->(?y)
RETURN *
```

The **RETURN** * operation projects all variables specified in the **MATCH** pattern, while the construct `(?x)-[?e :?t]->(?y)` specifies that we want to connect the object in ?x with an object in ?y, via an edge with type ?t and id ?e. This is akin to a query DomainGraph(?x, ?t, ?y, ?e) over the domain graph relation. Variable or constant edge types (e.g. ?t above) are prefixed by a colon.

We can also restrict which edges are matched. The following query in DGQL will return ids for edges of type child, where both nodes have the same last name, and the child is not the oldest; it also returns the order of the child:

```
MATCH (?x)-[?e :child]->(?y)
WHERE (?x.last_name == ?y.last_name) AND (?e.order > 1)
RETURN ?e, ?e.order
```

This returns $\{\{$?e $\mapsto e_2$, ?e.order $\mapsto$ 2$\}\}$ for the graph in Figure 1. If we need to query for the second oldest (an equality condition), we could use the syntax `(?x)-[?e :child {order : 2}]->(?y)` on the edge (or replace `(?e.order > 1)` with `(?e.order == 1)`). As seen here, the **WHERE** clause may use Boolean combinations.

Querying known objects. Knowing the id of an object, we might want to query more about it. For instance, in the domain graph of Figure 4 representing statements from Wikidata, we may query for the positions held by Michelle Bachelet as follows:

```
MATCH (Michelle Bachelet)-[:position_held]->(?x)
RETURN ?x
```

This would return duplicate results $\{\{$?x $\mapsto$ President of Chile$\}$, $\{$?x $\mapsto$ President of Chile$\}\}$ due to the two edges e_1 and e_2. Recall that for clarity we use human-readable ids, where for Wikidata, the node id `Michelle Bachelet` may rather be given as `Q320`, and `position held` by `P39`.

Path queries. A key feature of graph databases is their ability to explore paths of arbitrary length. DGQL supports two-way regular path queries (2RPQs), which specify regular expressions over edge types, including concatenation (/), disjunction (|), inverses (^), optional (?), Kleene star (*) and Kleene plus (+). For instance, if we need to find all descendants of people named Alberto, we could use the regular expression `:child+` in the following way:

```
MATCH (?x { first_name : "Alberto" })=[:child+]=>(?y)
RETURN ?y
```

This returns $\{\{$?y $\mapsto n_1\}\}$ over the graph in Figure 1, and would include children of n_1, and their children, recursively, if present. We use `=[]=>` (rather than `-[]->`) to signal a path query in DGQL. If we want to capture more results, we could replace `[:child+]` with an extended expression like `[(:child|^:father|^:mother)+]` which also traverses backwards along edges of type `father` and `mother`, and may thus return more results. We can also concatenate paths with `/`: the expression `[(:father|:mother)*/:sister]` can be used to find sisters, aunts, grand-aunts, and so forth.

DGQL can also return a single shortest path witnessing the query result via the following construct:

```
MATCH (?x { first_name : "Alberto" })=[?p :child+]=>(?y)
RETURN ?y, ?p
```

Unlike Cypher, we can return paths matching 2RPQs, not just Kleene star. Unlike SPARQL, we can return paths, not just pairs of nodes. The variable `?p` stores a shortest path connecting each `Alberto` with each descendant `?y`. No manipulation of path variables, apart from outputting the result, is currently supported in MillenniumDB, but a full path algebra will be supported in future versions.

Another novel feature of DGQL is the ability to return *all shortest paths* between a set of nodes. In the example above we would specify this by writing `[ALL ?p :child+]`. When the `ALL` keyword is placed before the path variable, all shortest paths are returned.

Basic and navigational graph patterns. Basic graph patterns [3] lie at the core of many graph query languages, including DGQL. These are graphs following the same structure as the data model, but where variables are allowed in any position. They can also be seen as expressing natural (multi)joins over sets of atomic edge patterns. In DGQL, they are given in the **MATCH** clause. Illustrating this, the following query finds pairs of nodes sharing a father:

```
MATCH (?x)-[:father]->(?z),
      (?y)-[:father]->(?z)
RETURN ?x, ?y
```

We evaluate basic graph patterns under *homomorphism-based semantics* [3], which allows multiple variables in a single result to map to the same element of the data. If we evaluate this query over Figure 1, we would thus get one solution: $\{\{?\text{x} \mapsto n_1, ?\text{y} \mapsto n_1\}\}$.

If we further allow path queries within basic graph patterns, we arrive at *navigational graph patterns* [3]. Suppose we need to find, for example, pairs of nodes that share a common ancestor along the paternal line and held the same position; we could write this as:

```
MATCH (?x)=[:father+]=>(?z),
      (?y)=[:father+]=>(?z),
      (?x)-[:position_held]->(?w)
      (?y)-[:position_held]->(?w)
WHERE ?x != ?y
RETURN ?x, ?y
```

We also explicitly filter results mapping `?x` and `?y` to the same node.

Taking advantage of the domain graph model. The DGQL query language allows us take full advantage of domain graphs, by allowing joins between edges, types, etc. To illustrate this, consider the following query, evaluated over the domain graph of Figure 4:

```
MATCH (Michelle Bachelet)-[?e :position_held]->
                                    (President of Chile),
      (?e)-[:replaces]->(?x),
      (?e)-[:start_date]->(?d)
RETURN ?x, ?d
```

The variable `?e` invokes a join between an edge and a node, returning two solutions: {{?x ↦ Ricardo Lagos, ?d ↦ 2006-03-11}, {?x ↦ Sebastián Piñera, ?d ↦ 2014-03-11}}. This query is thus used to return the list of presidents that were replaced by `Michelle Bachelet`, and the dates they were replaced by her. Illustrating a join on an edge type, consider the following query:

```
MATCH (?x)-[?e :?t]->(?y),
      (?t)=[:subproperty_of*]=>(parent)
RETURN ?x, ?y
```

This query returns all pairs of nodes with an edge whose type is `parent`, or a transitive sub-property of `parent`; for example, Wikidata defines `father` and `mother` to be sub-properties of `parent`.

Optional matches. DGQL also supports optional graph patterns, which behave akin to left outer joins. For instance, we may wish to query for presidents of Chile, and if available, the president they replaced, and if further available, the president that the latter president replaced. The following DGQL query accomplishes this:

```
MATCH (?x)-[?e1 :position_held]->(President of Chile),
      OPTIONAL {
        (?e1)-[:replaces]->(?y)
        OPTIONAL {
          (?y)-[?e2 :position_held]->(President of Chile),
          (?e2)-[:replaces]->(?z)
        }
      }
RETURN ?x, ?y, ?z
```

If evaluated on the domain graph of Figure 4, we would get two solutions: {{?x ↦ Michelle Bachelet, ?y ↦ Ricardo Lagos}, {?x ↦ Michelle Bachelet, ?y ↦ Sebastián Piñera}}, where in both cases, the variable `?z` is left *unbound* as the data is not available for the inner optional graph pattern. As shown, nested optionals are supported, but with the restriction that they form *well-designed patterns* [28].

Limits and ordering. Some additional operators that MillenniumDB supports are **LIMIT** and **ORDER BY**. These allow us to limit the number of output mappings, and sort the obtained results. For instance, if we need to obtain the ten most recent presidents of Chile, we could accomplish this with the following query:

```
MATCH (?x)-[?e :position_held]->(President of Chile),
      (?e)-[:start_date]->(?d)
ORDER BY DESC (?d)
RETURN ?x
LIMIT 10
```

Ordering is always applied before limiting results.

Formal definitions. In [39], we include an extended version of this paper, which contains a full specification of DGQL, as well as a formal definition of its semantics. A grammar for generating DGQL queries is presented in Appendix A of [39]. The abstract syntax and formal semantics of the language is given in Appendix B of [39].

3.2 Comparing Graph Query Languages

A variety of query languages for graphs have been proposed in recent years [3, 4]. This section compares DGQL with six representative query languages: Cypher [14] (Neo4j), SPARQL [16] (the standard query language for RDF Triple Stores), G-CORE [2] (LDBC),

Table 2: Query features supported by graph query languages (BGP = basic graph patterns, RGP = relational graph patterns, QE = querying edges, RPQ = regular path queries, NGP = navigational graph patterns, FPR = full path recovery). The symbol ∼ is used to indicate partial support of a feature.

Query language	*BGP*	*RGP*	*QE*	*RPQ*	*NGP*	*FPR*
DGQL	✓	∼	✓	✓	✓	∼
Cypher	✓	✓	∼	∼	∼	∼
SPARQL	✓	✓	∼	✓	✓	✗
G-CORE	✓	✓	∼	✓	✓	✓
GSQL	✓	∼	∼	✓	∼	✗
Gremlin	✓	∼	∼	∼	∼	∼
nGQL	✓	✓	∼	∼	∼	∼

GSQL [44] (TigerGraph), Gremlin [30] (supported by several systems like Amazon Neptune and JanusGraph) and nGQL [41] (NebulaGraph). For each query language, we evaluate its support (total or partial) for six query features, namely: basic graph patterns, relational graph patterns, querying edges, regular path queries, navigational graph patterns, and full path recovery. A summary of our comparison is shown in Table 2.

Every query language considered in Table 2 supports the notion of *basic graph pattern* (BGP), which, in its most general form, is a graph pattern structured like the data, but allowing variables to replace constants. In most cases, the result of a BGP is a relation (or table) consisting of results, and in some cases it is possible to construct/return a graph (like in G-CORE and SPARQL).

By considering that a graph pattern transforms a graph into a table, *relational graph patterns* allow the use of relational operators to combine the results of one or more graph patterns into a single relation. Full support of this feature in Table 2 indicates that a language provides join, optional, union, filter and negation of graph patterns. Partial support indicates that a language supports some of these operators, usually join and optional graph patterns, as is the case for DGQL (we plan to extend this in future).

Querying edges is a particular feature of DGQL, allowing for querying relationships involving edges. Other query languages provide partial support for querying edges, as they are restricted to query the labels and properties of the edges, require reserved vocabulary (reification), or have other restrictions (e.g., using named graphs in SPARQL over which paths cannot be resolved).

The *regular path queries* feature refers to matching paths based on 2-way regular expressions, with concatenation, disjunction, inverse, optional and Kleene star. Partial support indicates that a language offers a restricted group of such operators, such as in the case of Cypher, which supports only Kleene plus.

We use the term *navigational graph patterns* to represent the combination of basic graph patterns and regular path queries. These queries are akin to conjunctive (2-way) regular path queries.

Finally, a query language with *full path recovery* allows not only to search for some paths, but also to return such paths as objects that can be manipulated (with the nodes and edges in a path). Some query languages, like DGQL, partially support this feature by returning a path as a string. Languages like SPARQL do not support this feature: only the start and end nodes of a path are returned.

Table 2 focuses on core features for querying graphs [3], and thus omits features (e.g., borrowed from SQL) that are supported by some of the languages, and that are potentially very useful in practice, such as aggregations, solution modifiers, federation, etc. Such features can be layered atop the features mentioned.

4 SYSTEM ARCHITECTURE

In this section, we describe the internals underlying MillenniumDB designed to efficiently support the domain graph model.

MillenniumDB is founded on tried and tested relational techniques: it stores the domain graph model as several relations indexed in B+ trees. However it also uses algorithmic techniques recently suggested in the theoretical literature for evaluating queries [7, 46] – techniques not typically implemented in graph database systems – for supporting the domain graphs model in practice. Specifically, we combine three different techniques that are new to the architecture of graph database systems when used in conjunction. First, the data model is encoded as basic relations, indexed following different attributes orders, and where data objects (e.g., nodes, strings) are represented by ids. Second, we translate the evaluation of any query to several joins between basic relations, which we manage using worst-case optimal join algorithms, a novel evaluation technique recently proposed for relational database systems. Last, we combine join algorithms with the evaluation of path queries by compiling the path pattern into an automaton and running the query on the fly. These techniques, together, are at the heart of the performance of MillenniumDB, optimizing queries over the domain graphs model in practice. To the best of our knowledge, MillenniumDB is the first graph database system that efficiently supports an expressive graph model by exploiting these main techniques all at once.

In the following, we explain how one can store the domain graphs and index them. We then outline the query evaluation process and provide details on the algorithmic techniques used, like the worst-case optimal query plan and the evaluation of path queries.

Storage and indexing. Objects are represented internally as 8-byte identifiers. To optimize query execution, identifiers are divided into classes and the first byte of the identifier specifies a class it belongs to. The main classes in a domain graph $G = (O, \gamma, \texttt{lab}, \texttt{prop})$ are:

- *Nodes*, which are objects in the range of γ. They are divided in two subclasses: *named nodes*, which are objects in the domain graph for which an explicit name is available (e.g. Q320 in Wikidata), and *Millenniumymous nodes*, which are internally generated objects without an explicit name available to the user (similar to blank nodes in RDF [21]).
- *Edges*, which are objects in the domain and range of γ, and are always Millenniumymous, internally generated objects.
- *Values*, which are data objects like strings, integers, etc. These values are classified in two subclasses: *inlined values*, which are values that fit into 7 bytes of the identifier after the mask (e.g. 7 byte strings, integers, etc.), and *external values*, which are values longer than 7 bytes (e.g. long strings).

All records stored in MillenniumDB are composed of these identifiers. We will explain later how long strings for external values are handled.

To store property domain graphs, MillenniumDB deploys B+ trees [29]. For this purpose, we built a B+ tree template for fixed sized records, which store all classes of identifiers. To store a property domain graph $G = (O, \gamma, \texttt{lab}, \texttt{prop})$, we simply store and index in B+ trees the four components defining it:

- Objects($\underline{\text{id}}$) stores the identifiers of all the objects in the database (i.e., O).
- DomainGraph(source,type,target,$\underline{\text{eid}}$) contains all information on edges in the graph (i.e., γ), where eid is an edge identifier, and source, type, and target can be ids of any class (i.e., node, edge, or value). By default, four permutations of the attributes are indexed in order to aid query evaluation. These are: source-target-type-eid, target-type-source-eid, type-source-target-eid and type-target-source-eid.
- Labels(object,label) stores object labels (i.e., lab). The value of object can be any identifier, and the values of label are stored as ids. Both permutations are indexed.
- Properties($\underline{\text{object,property}}$,value) stores the property–value pairs associated with each object (i.e., prop). The object column can contain any id, and property and value are value ids. Aside from indexing the primary key, an additional permutation is added to search objects by property–value pairs.

All the B+ trees are created through a bulk-import phase, which loads multiple tuples of sorted data, instead of creating the trees by inserting records one by one. In order to enable fast lookups by edge identifier, we use the fact that this attribute is the key for the relation. Therefore, we also store a table called EdgeTable, which contains triples of the form (source, type, target), such that the position in the table equals to the identifier of the object e such that $\gamma(e) = (\text{source}, \text{type}, \text{target})$. This implies that edge identifiers must be assigned consecutive ids starting from zero, and they are generated in this way by MillenniumDB (they are not specified by the user). In total, we use ten B+ trees for storing the data.

To transform external strings and values (longer than 7 bytes) to database object ids and values, we have a single binary file called ObjectFile, which contains all such strings concatenated together. The internal id of an external value is then equal to the position where it is written in the ObjectFile, thus allowing efficient lookups of a value via its id. The identifiers are generated upon loading, and an additional hash table is kept to map a string to its identifier; we use this to ensure that no value is inserted twice, and to transform explicit values given in a query to their internal ids. Only long strings are currently supported, but the implementation interface allows for adding different value types in a simple manner.

Evaluating a query. In MillenniumDB, the execution pipeline follows the standard database template where the string of the query is parsed and translated into a logical plan, which is analyzed and converted then into a physical plan, and finally evaluated.

The interesting part is in how the *patterns* and *filters* of a DGQL query (see Section 3.1) are evaluated. Specifically, the patterns and filters are grouped together into a list of relations that can be edges, labels, properties, or path queries, forming a big *multi-way join* query. In essence, evaluating these joins is analogous to selecting an appropriate join plan for the relations representing the different elements. This also goes in hand with selecting the appropriate join algorithm for each of the joins. Given that edges, labels, and properties are all indexed, this will most commonly be index nested loops join. Paths on the other hand are not directly indexed. For this reason, they are pushed to the end of the join plan and joined via a nested-loop with the rest of the multi-way join [5].

MillenniumDB supports different mechanisms for evaluating the multi-way join formed by the pattern and filter of DGQL query.

- A worst-case optimal query plan as described in [23] is used whenever possible. This approach implements a modified leapfrog algorithm [46] in order to minimize the number of intermediate results that are generated.
- The classical relational optimizer which is based on cost estimation, and tries to order base relations in such a way as to minimize the amount of (intermediate) results. We currently support two modes of execution here:
  (i) Selinger-style join plans [32] which use dynamic programming to determine the optimal order of relations.
  (ii) In the presence of a large number of relations, a greedy planner [15] is used which simply determines the cheapest relation to use in each step.

Two particular points of interest are the worst-case optimal query planner, and the way that paths are evaluated. Both of these deploy state-of-the-art research ideas that are usually not, to the best of our knowledge, implemented in graph database systems. We provide some additional details on these next.

Worst-case optimal query plan. Evaluating multiple joins in a worst-case optimal way is done using a modified leapfrog algorithm [23]. While a classical join plan does a nested for-loop over relations, leapfrog performs a nested for-loop over variables [46]. Specifically, the algorithm first selects a variable order for the query, say (?x, ?y, ?z). It then intersects all relations where the first variable ?x appears, and over each solution for ?x returned, it intersects all relations where ?y appears (replacing ?x in its current solution), and so on to ?z, until all variables are processed and the final solutions are generated. We refer the reader to [46] and [23] for a detailed explanation. Two critical aspects for supporting this approach are indices and variable ordering, explained next.

To support the leapfrog algorithm, we should index all relations in all possible orders of their attributes, which greatly increases disk storage [23]. In MillenniumDB, we include four orders for DomainGraph, and all orders for Labels and Properties. By considering these orders, we can cover the most common join-types that appear in practice [9] by a worst-case optimal query plan. We use the classical relational optimizer if the plan needs an unsupported order or one of the relations uses a path query.

The leapfrog algorithm requires choosing a variable ordering, which is crucial for its performance. The heuristic we deploy for selecting the variable ordering mixes a greedy approach, and the ideas of the GYO reduction [49]. More precisely, we first order the variables based on the minimal cost of the relations they appear in and resolve ties by selecting the variable that appears in more distinct relations. The variables “connected" to the first one chosen

[5]This is not always the best option. However, based on extensive empirical evidence (see Section 5), this solution seems to be adequate in practice.

are then processed in the same manner (where connected meant appearing in the same relation) until the process can not continue. The isolated variables are then treated last.

Evaluating path queries. For evaluating a path query, the path pattern (2RPQ) is compiled into an automaton, and a "virtual" cross-product of this automaton and the graph is constructed on-the-fly, and navigated in a breadth-first manner[6], as commonly suggested in the theoretical literature [6, 7, 27]. Our assumption is that each path pattern will have at least one of the endpoints assigned before evaluation. This can be done either explicitly in the pattern, or via the remainder of the query. For instance, a path pattern `(Q1)=[P31*]=>(?x)` has the starting point of our search assigned to `Q1`. On the other hand, `(?x)=[P31*]=>(?y :Person)` does not have any of the endpoints assigned, however, the `(?y :Person)` allows us to instantiate `?y` with any node with the label `:Person`.

Intuitively, from a starting node (tagged with the initial state of the automaton), all edges with the type specified by the outgoing transitions from this state are followed. The process is repeated until reaching an end state of the automaton, upon which a result can be returned. This allows a fully pipelined evaluation of path queries, while only requiring at most a fixed amount of memory (the neighbors of the node on the top of the BFS queue). Additionally, the BFS algorithm also allows us to return a single shortest path between each pair of endpoints (see Section 3 for an example). Returning paths comes almost for free, given that they can be reconstructed using the set of visited nodes which is used for bookkeeping in the BFS algorithm. The algorithm can also be extended to return all shortest paths by keeping a list of predecessors that reach the node by a path of shortest length.

The implemented algorithm only requires two permutations of the DomainGraph relation: one for retrieving all the nodes successors via an edge of a specified type; and another for retrieving all such predecessors of a given node.

5 BENCHMARKING

In this section, we provide an experimental evaluation of the core graph querying features of MillDB addressing two key questions: (Q1) *Which join and path algorithms provide the best performance over domain graphs?* (Q2) *How does MillDB's performance compare with existing graph database engines?*

We base our experiments on the Wikidata knowledge graph [47], which is one of the largest and most diverse real-world knowledge graphs that is publicly available, and also provides a public log of real-world queries posted by Wikidata users that we can use for experiments [9, 26]. The experiments focus on two fundamental query features: (i) basic graph patterns (BGPs); and (ii) path queries. Regarding (Q1), we compare the performance of different join and path algorithms within MillDB. Regarding (Q2), we also provide a side by side comparison with several popular persistent graph database engines that support BGPs and at least the Kleene star feature for paths. We publish the data, queries, scripts, and configuration files for each engine online, together with the scripts used to load the data and run the experiments [40].

[6]BFS is the default evaluation method for property paths. The source code also includes four other algorithms, including DFS. We contrast these two basic implementations in Section 5. A detailed exploration of all options is outside of the scope of this paper.

Table 3: Wikidata Truthy sizes when loaded into each engine. The base dataset consists of roughly 1.25 billion triples.

MillDB	BlazeG	Jena	Jena LF	Virtuoso	Neo4J
203GB	70GB	110GB	195GB	70GB	112GB

Internal baselines. The base of our comparison is the MillDB (MillDB) implementation available at [**?**]. For comparing the performance of different join and path algorithms in MillDB (per Q1), we include internal baselines, where we test: (i) MillDB LF, which is the default version implementing the leapfrog triejoin algorithm; (ii) MillDB GR, which implements the greedy algorithm for selecting the join order; and (iii) MillDB SL, implementing the Sellinger join planner. Similarly, for path queries, we test (a) MillDB BFS, the default version of the engine; and (b) MillDB DFS, which evaluates path queries using the depth-first traversal.

Other engines. We also compare the performance of MillDB with five persistent graph query engines (per Q2). First, we include three popular RDF engines: Jena TDB version 4.1.0 [37], Blazegraph (BlazeG for short) version 2.1.6 [45], and Virtuoso version 7.2.6 [12]. We further include a property graph engine: Neo4J community edition 4.3.5 [48].[7] Finally, we also compare with Jena Leapfrog (Jena LF, for short) – a version of Jena TDB implementing a leapfrog-style algorithm [23] – in order to compare with an external graph database using a worst case optimal algorithm.

The machine. All experiments described were run on a single commodity server with an Intel®Xeon®Silver 4110 CPU, and 128GB of DDR4/2666MHz RAM, running the Linux Debian 10 operating system with the kernel version 5.10. The hard disk used to store the data was a SEAGATE ST14000NM001G with 14TB of storage.

The data. The base for our experiments is the Wikidata dataset. In particular, we used the truthy dump version 20210623-truthy-BETA [13], keeping only triples in which (i) the subject position is a Wikidata entity, and (ii) the predicate is a direct property. We call this dataset *Wikidata Truthy*. The size of the dataset after this process was 1,257,169,959 triples. The simplification of the dataset is done to facilitate comparison across multiple engines, specifically to keep data loading times across all engines manageable while keeping the nodes and edges necessary for testing the performance of BGPs and property paths. The size of the *Wikidata Truthy* dataset, when loaded into the respective systems, is summarized in Table 3. Default indices were used on Jena TDB, Blazegraph and Virtuoso. Jena LF stores three additional permutations of the stored triples to efficiently support the leapfrog algorithm for any join query, thus using more space. Neo4j by default creates an index for edge types (as of version 4.3.5). To speed up searches for particular entities and properties, we also created an index linking a Wikidata identifier (such as, e.g., Q510) to its internal id in Neo4j. We also tried to index literal values in Neo4j, but the process failed (the literals are still stored). MillDB uses extra disk space because of the additional indices needed to support worst-case optimal join over domain graphs (similar to the case of Jena LF).

[7]Though TigerGraph meets the technical requirements, its license currently restricts benchmarking and thus it is excluded.

How we ran the queries. We detail the query sets used for the experiments in their respective subsections. To simulate a realistic database load, we do not split queries into cold/hot run segments. Rather, we run them in succession, one after another, after a cold start of each system (and after cleaning the OS cache). This simulates the fact that query performance can vary significantly based on the state of the system buffer, or even on the state of the hard drive, or the state of OS's virtual memory. For each system, queries were run in the same order. We record the execution time of each individual query, which includes iterating over all results. We set a limit of 100,000 distinct results for each query, again in order to enable comparability as some engines showed instability when returning larger results. Jena and Blazegraph were assigned 64GB of RAM, and Virtuoso was set up with 64GB or more of RAM as is recommended. Neo4J was run with default settings, while MillDB had access to 32GB for main-memory buffer, and it uses an additional 10GB for in-memory dictionaries.

Handling timeouts. We defined a timeout of 10 minutes per query for each system. Apart from that, we note that most systems had to be restarted upon a timeout as they often showed instability, particularly while evaluating path queries. This was done without cleaning the OS cache in order to preserve some of the virtual memory mapping that the OS built up to that point. In comparison, even when provided with a very small timeout window, MillDB managed to return a non-trivial amount of query results on each query, and did not need to be restarted, thus allowing us to handle timeouts gracefully.

5.1 Basic Graph Patterns

We focus first on basic graph pattern queries. To test different query execution strategies of MillDB, we use two benchmarks: *Real-world BGPs* and *Complex BGPs*, which are described next.

Real-world BGPs. The Wikidata SPARQL query log contains millions of queries [26], but many of them are trivial to evaluate. We thus decided to generate our benchmark from more challenging cases, i.e., a smaller log of queries that timed-out on the Wikidata public endpoint [26]. From these queries we extracted their BGPs, removing duplicates (modulo isomorphism on query variables). We distinguish queries consisting of a single triple pattern (*Single*) from those containing more than one triple pattern (*Multiple*). The former set tests the triple matching capabilities of the systems, whereas queries in the latter set test join performance. *Single* contains 399 queries, whereas *Multiple* has 436 queries.

Real-world Single. Table 4 (top) summarizes the query times on this set, whereas Figure 5 (left) shows boxplots with more detailed statistics on the distributions of runtimes. Since these queries do not require joins, we show one variant for MillDB. MillDB is the fastest overall (median of 0.05 s), followed by Blazegraph (median of 0.09 s). In terms of average times and higher percentiles, MillDB more clearly outperforms other engines, being able to enumerate up to 100,000 results (the limit) more quickly due to decoding internal ids more quickly. In MillDB, values such as `P12` or `Q10` that fit within 7 bytes are inlined, and do not need to be dictionary decoded. The remaining dictionary fits entirely in available memory (~8 GB of RAM). In the other systems, dictionary decoding generates random accesses to the disk. The four SPARQL engines tested must store IDs as IRIs within the RDF model, which include relatively long prefixes. However, since RDF datasets typically have few prefixes repeated often, we could support full IRIs within MillDB with minimal overhead by encoding a prefix id for the top k prefixes in $\lceil \log_2(k) \rceil$ bits within the object identifier, keeping the small mapping from prefix id to string in memory. The Wikidata query service lists 32 prefixes, which would require 5 bits that would fit "for free" in the class byte (essentially considering each prefix to be a class).

Table 4: Summary of runtimes (in seconds) for BGPs

Engine	Supported	Error	Timeouts	Average	Median
Real-world Single (399 queries)					
MillDB	399	0	0	0.07	0.05
Blazegraph	399	0	0	2.21	0.09
Jena	399	0	0	14.10	0.34
Jena LF	395	4	0	10.08	0.44
Virtuoso	399	0	0	2.22	0.32
Neo4j	394	5	0	28.00	1.33
Real-world Multiple (436 queries)					
MillDB LF	436	0	0	4.84	0.24
MillDB GR	436	0	1	10.19	0.30
MillDB SL	436	0	1	10.04	0.27
Blazegraph	436	0	3	31.79	2.42
Jena	426	10	0	35.43	4.90
Jena LF	418	18	0	16.78	3.39
Virtuoso	436	0	0	7.87	5.11
Neo4j	405	31	0	75.55	6.84
Complex (850 queries)					
MillDB LF	850	0	0	0.38	0.10
MillDB GR	850	0	1	3.30	0.17
MillDB SL	850	0	1	3.51	0.17
Blazegraph	850	0	2	4.63	0.34
Jena	850	2	0	3.37	0.16
Jena LF	850	0	0	0.88	0.14
Virtuoso	850	0	0	1.00	0.19
Neo4j	850	10	0	17.92	0.66

Real-world Multiple. Table 4 (middle) and Figure 5 (middle) show the results for this set. Comparing different join execution strategies within MillDB, we can see the superiority of the leapfrog triejoin variant (particularly on average, i.e., for more complex queries). The Selinger variant of MillDB outperforms the greedy algorithm for join selection, but only marginally. Compared with existing graph engines, MillDB clearly outperforms the other systems on this query set. Its medians are an order of magnitude faster than those of Blazegraph, the next best contender. Indeed, the median of Blazegraph is an order of magnitude higher than all the queries in MillDB (excluding outliers, as per Figure 5). The difference is less sharp for averages, but MillDB LF still takes 60% of the time of Virtuoso, the next best contender.

Complex BGPs. This is a benchmark used to test the performance of worst-case optimal joins [23]. Here, 17 different complex join patterns were selected, and 50 different queries generated for each pattern, resulting in a total of 850 queries. Figure 5 (right), and Table 4 (bottom), show the resulting query times. In this case, the difference between the join algorithms of MillDB is more clear. The worst-case-optimal version (MillDB LF) is not only considerably more stable than the other two versions, but also twice as fast in the

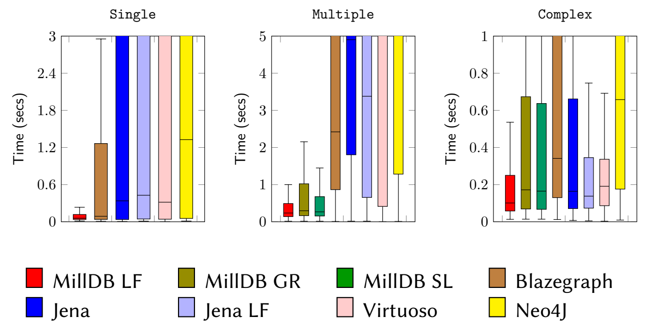


Figure 5: Boxplots showing distributions for BPG runtimes

median. We can also observe that MillDB GR wins out over MillDB SL on average (but not the median case). When comparing with other engines, the next-best competitor after MillDB LF is Jena LF, showing the benefits of worst-case optimal joins. Virtuoso follows not far behind, while MillDB GR, Jena, Blazegraph and Neo4j are considerably slower. Overall, MillDB LF offers the best performance for every statistic shown in the plot.

5.2 Path Queries

To test the performance of path queries, we extracted 2RPQ expressions from a log of queries that timed out on the Wikidata endpoint [26]. The original log has 2110 queries. After removing queries that do not use direct properties (which are absent in the *Wikidata Truthy* dataset), we ended up with 1683 queries. These were run in succession, each restricted to return at most 100,000 results. In the case of SPARQL engines, we added the `DISTINCT` keyword to remove duplicates caused by the rewriting of fixed-length path queries to unions of BGPs that are then evaluated under bag semantics. To make the comparison fair, the `DISTINCT` keyword was also added in MillDB queries. Each system was started after cleaning the system cache, and with a timeout of 10 minutes. Since these are originally SPARQL queries, not all of them were supported by Neo4J given the restricted regular-expression syntax it supports. We remark that MillDB and Neo4J were the only systems able to handle timeouts without being restarted.[8] In this comparison we do not include Jena LF since it uses the same execution strategy as Jena for property paths. Likewise, for MillDB, we introduce two internal baselines for breadth-first search (BFS) and depth-first search (DFS). The experimental results for these path queries are summarized in Table 5 and Figure 6.

In terms of our internal comparison, we can see that the DFS algorithm slightly outperforms BFS. The reason for keeping BFS as the default algorithm is twofold: (i) it significantly outperforms DFS when paths are also returned; and (ii) it supports returning all shortest paths between any pair of nodes. To illustrate point (i), we ran our experiments again, but now also returning a single path witnessing each query answer. In this case, the average for BFS is 5.9 sec, and median is 0.086 sec. On the other hand, when paths are returned, DFS takes 7.9 sec on average, and 0.1 sec median time.

[8]In fact, MillDB did not give any timeouts. However, we re-ran the experiments with a lower timeout, and observed that the system could recover from interrupting the query gracefully and was able to return the results found before being interrupted.

Table 5: Summary of runtimes (in seconds) for path queries

Engine	Supported	Error	Timeouts	Average	Median
MillDB BFS	1683	0	0	1.1	0.095
MillDB DFS	1683	0	0	1.1	0.072
Blazegraph	1683	2	44	27.6	0.396
Jena	1683	14	46	22.8	0.207
Virtuoso	1683	55	4	5.8	0.325
Neo4J	1622	0	42	23.3	0.328

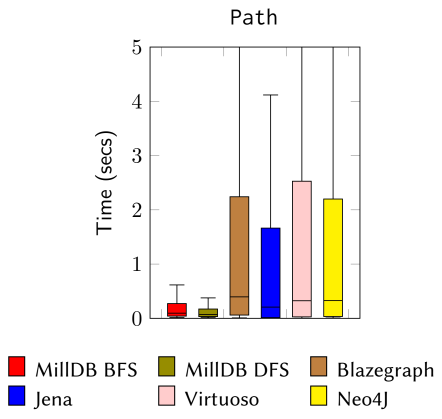


Figure 6: Boxplots of query times on property paths

Compared with other engines, MillDB is generally the fastest, and has the most stable performance, being able to run all the queries. Its average is near a second, i.e., five times faster than the next best contender (Virtuoso). Its median, below 0.1 seconds, is half the next one (Jena's). Even after removing the queries that timed-out on the other systems, they are considerably slower than MillDB. In particular, if we only consider the queries that run successfully on Virtuoso (i.e., excluding the 59 queries that timed-out or gave an error), we get an average time of 0.85 seconds and a median time of 0.086 seconds on MillDB, less than half the times of Virtuoso with these queries excluded. The boxplots further show the stability of MillDB: the medians of other baselines are over the third quartile of MillDB. Their third quartile is 5–10 times higher than that of MillDB, and higher than its topmost whisker.

To further test robustness, we also ran all of the queries *without limiting the output size* on MillDB. In this test, the engine timed out in only 15 queries, each returning between 800 thousand and 44 million results before timing out. When running queries to completion, MillDB BFS averaged 13.4 seconds per query (8 seconds if timeouts were excluded), with a median of 0.1 seconds (both with and without timeouts).

5.3 Wikidata Complete

To show the scalability of MillDB, and to leverage its domain graph, we ran experiments with a full version of Wikidata to measure query performance. We call this dataset *Wikidata Complete*, and base it off the Wikidata JSON dump[9] version 20201102-all.json, which is

[9]It is important to note that JSON and RDF dumps of Wikidata do not result in precisely the same knowledge graph due to some restrictions of the particular reification used in RDF; however, they do result in very similar knowledge graphs.

Table 6: Average and median runtimes, in seconds, for MillDB on the complete version of Wikidata

	Single	Multiple	Complex	Paths
Average	0.08	4.04	0.35	1.04
Median	0.07	0.28	0.095	0.10

preprocessed and mapped to our data model. In *Wikidata Complete*, we model qualifiers (i.e. edges on edges), put labels on objects, and assign them properties with values. We use properties to store the language value of each string in Wikidata, and also to model elements of complex data values (e.g., for coordinates we would have objects with properties latitude and longitude, and similarly for amounts, date/time, limits, etc.). Each object representing a complex data value also has a label specifying its data type (e.g. `coord` for geographical coordinates). Qualifiers were loaded in their totality (i.e., not only the preferred qualifiers, but all specified ones). The only elements excluded from the data dump (for now) were sitelinks and references. This full version of Wikidata resulted in a knowledge graph with roughly 300 million objects, participating in 4.3 billion edges. The total size on disk of this data was 827GB in MillDB, i.e., more than four times larger than *Wikidata Truthy*. More details about this dataset can be found online [40].

We ran the same queries from the benchmarks (*Single*, *Multiple* and *Complex* BGPs, as well as *Paths*). The number of outputs on the two versions of the data, while not the same, was within the same order of magnitude averaged over all the queries. The results are presented in Table 6. As we can observe, MillDB shows no deterioration in performance when a larger database is considered for similar queries. This is mostly due to the fact that the buffer only loads the necessary pages into the main memory, and will probably require a rather similar effort in both cases. We also note that, again, no queries resulted in a timeout over the larger dataset.

5.4 Discussion

Regarding (Q1) – i.e., which join and path algorithms provide the best performance in this setting – we can conclude that the worst-case optimal join algorithm consistently outperforms the greedy and Selinger variants, being particularly notable in the case of more complex graph patterns with many joins (wherein Jena LF – also worst-case optimal – was the next best competitor). Worst-case optimal joins use more space for index permutations, but provide superior query runtimes. We see less difference between BFS and DFS: DFS is slightly faster for returning pairs of nodes connected by matching paths, while BFS is faster for returning paths.

Regarding (Q2) – i.e., how existing graph database systems compare with MillDB – we have again found that MillDB, when equipped with the best join and path algorithms, consistently outperforms other competitors in all query sets tested.

6 CONCLUSIONS AND LOOKING AHEAD

This paper presents MillenniumDB, a graph database system with persistent storage that implements the domain graph model, and is available under an open-source license.

Domain graphs adopt the natural idea of adding edge ids to directed labeled edges in order to concisely model higher-arity relations in graphs, as needed in Wikidata, without the need for reserved vocabulary or reification. They can naturally represent popular graph models, such as RDF and property graphs, and allow for combining the features of both models in a novel way. While the idea of using edge ids as a hook for modeling higher-arity relations in graphs is far from new (see, e.g., [20, 24, 25]), it is an idea that is garnering increased attention as a more flexible and concise alternative to reification. To the best of our knowledge, our work is the first to propose a formal data model that incorporates edge ids, a query language that can take advantage of them, and a fully-fledged graph database engine that supports them by design. We also propose to optionally allow (external) annotations on top of the graph structure, thus facilitating better compatibility with property graphs, whereby labels and property–values can be added to graph objects without adding new nodes and edges to the graph itself.

With respect to the query language, we have proposed a new query syntax inspired by Cypher, but that additionally enables users to take full advantage of the domain graph model by (optionally) referencing edge ids in their queries, and performing joins on any element of the domain graph. We further combine useful features present in both Cypher and SPARQL, in order to provide additional expressivity, such as returning the shortest path witnessing a result for a path query (as captured by a 2RPQ expression).

In the implementation of MillenniumDB, we combine both tried-and-trusted techniques that have been successfully used in relational database pipelines for decades [29] (e.g., B+ trees, buffer managers, etc.), with promising state-of-the-art algorithms for computing worst case optimal joins (leapfrog [46]) and evaluating path queries (guided by an an automaton [6, 27]). Our experiments over Wikidata, considering real-world queries and data at large-scale, show that this combination outperforms other persistent graph database engines that are commonly found in practice.

Looking to the future, we foresee extensions such as: returning entire graphs, supporting more complex path constraints, returning sets of paths, path algebra, just to name a few. Regarding more practical features, we aim to add support for full transactions, keyword search, a graph update language, existing graph query languages, and more besides. More importantly, given that MillenniumDB is published as an open source engine, we hope that the research community can view the MillenniumDB code base as a sandbox for incorporating their novel algorithms and ideas into a modern graph database, without the need to remake storage, indexing, access methods, or query parsers. Along these lines, we are currently working on adding an in-memory storage option to MillenniumDB using the ring [5]: a data structure based on the Burrows–Wheeler transform that supports worse-case optimal joins (over triples) in space similar to representing the graph itself. Initial tests show that the ring can store *Wikidata Truthy* in 50GB of space and improve median query times by a factor of 3, with average query times remaining similar. We are working on extending the ring to support edge ids and thus work with domain graphs. We also wish to explore the deployment of MillenniumDB for key use-cases; for example, we plan to provide and host an alternative query service for Wikidata, which may help to prioritize the addition of novel features and optimizations as needed in practice.

REFERENCES

[1] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2021. A Survey of RDF Stores & SPARQL Engines for Querying Knowledge Graphs. *CoRR* abs/2102.13027 (2021). arXiv:2102.13027 https://arxiv.org/abs/2102.13027

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutiérrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 1421–1432.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[4] Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008), 1:1–1:39. https://doi.org/10.1145/1322432.1322433

[5] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2021. Worst-Case Optimal Graph Joins in Almost No Space. In *SIGMOD International Conference on Management of Data*. ACM, 102–114. https://doi.org/10.1145/3448016.3457256

[6] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*. 175–188. https://doi.org/10.1145/2463664.2465216

[7] Jorge A. Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoc. 2017. Evaluating Navigational RDF Queries over the Web. In *Proceedings of the 28th ACM Conference on Hypertext and Social Media, HT 2017, Prague, Czech Republic, July 4-7, 2017*. 165–174. https://doi.org/10.1145/3078714.3078731

[8] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. *Semantic Web* 2, 1 (2011), 33–42. https://doi.org/10.3233/SW-2011-0026

[9] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679. https://doi.org/10.1007/s00778-019-00558-9

[10] Rick Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Rec.* 39, 4 (2010), 12–27. https://doi.org/10.1145/1978915.1978919

[11] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. https://www.w3.org/TR/rdf11-concepts/

[12] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8. http://sites.computer.org/debull/A12mar/vicol.pdf

[13] The Wikimedia Foundation. 2021. *Wikidata:Database download*. https://www.wikidata.org/wiki/Wikidata:Database_download

[14] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proc. of the International Conference on Management of Data (SIGMOD)*. ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[15] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

[16] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/

[17] Olaf Hartig. 2017. Foundations of RDF★ and SPARQL★ (An Alternative Approach to Statement-Level Metadata in RDF). In *Proc. of the Alberto Mendelzon International Workshop (AMW) (CEUR)*, Vol. 1912. CEUR-WS.org. http://ceur-ws.org/Vol-1912/paper12.pdf

[18] Olaf Hartig, Pierre-Antoine Champin, Gregg Kellogg, Andy Seaborne, Dörthe Arndt, Jeen Broekstra, Bob DuCharme, Ora Lassila, Peter F. Patel-Schneider, Eric Prud'hommeaux, Ted Thibodeau Jr., and Bryan Thompson. 2021. RDF-star and SPARQL-star. W3C Draft Community Group Report. https://w3c.github.io/rdf-star/cg-spec/2021-07-01.html

[19] Tom Heath and Christian Bizer. 2011. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00334ED1V01Y201102WBE001

[20] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. 2015. Reifying RDF: What Works Well With Wikidata?. In *Proc. of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS) (CEUR Workshop Proceedings)*, Vol. 1457. CEUR-WS.org, 32–47. http://ceur-ws.org/Vol-1457/SSWS2015_paper3.pdf

[21] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. 2014. Everything you always wanted to know about blank nodes. *J. Web Semant.* 27-28 (2014), 42–69. https://doi.org/10.1016/j.websem.2014.06.004

[22] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4 (2021), 71:1–71:37. https://doi.org/10.1145/3447772

[23] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. A Worst-Case Optimal Join Algorithm for SPARQL. In *Proc. of the International Semantic Web Conference (ISWC) (LNCS)*, Vol. 11778. Springer, 258–275. https://doi.org/10.1007/978-3-030-30793-6_15

[24] Filip Ilievski, Daniel Garijo, Hans Chalupsky, Naren Teja Divvala, Yixiang Yao, Craig Milo Rogers, Ronpeng Li, Jun Liu, Amandeep Singh, Daniel Schwabe, and Pedro A. Szekely. 2020. KGTK: A Toolkit for Large Knowledge Graph Manipulation and Analysis. In *International Semantic Web Conference (ISWC)*. Springer, 278–293.

[25] Ora Lassila, Michael Schmidt, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Ronak Sharda, and Bryan B. Thompson. 2021. Graph? Yes! Which one? Help! *CoRR* abs/2110.13348 (2021). arXiv:2110.13348 https://arxiv.org/abs/2110.13348

[26] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*. Springer, 376–394.

[27] Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*. 185–193.

[28] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[29] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database management systems*. McGraw-Hill.

[30] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, 1–10. https://doi.org/10.1145/2815072.2815073

[31] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618.

[32] Edward Sciore. 2020. *Database Design and Implementation - Second Edition*. Springer. https://doi.org/10.1007/978-3-030-33836-7

[33] AllegroGraph Team. 2021. AllegroGraph 7.1.0 Documentation. https://franz.com/agraph/support/documentation/current/

[34] ArangoDB Team. 2021. ArangoDB v3.7.11 Documentation. https://www.arangodb.com/docs/stable/

[35] Amazon Neptune Team. 2021. What Is Amazon Neptune? https://docs.aws.amazon.com/neptune/latest/userguide/intro.html

[36] JanusGraph Team. 2021. JanusGraph Documentation, v.0.5. https://docs.janusgraph.org/

[37] Jena Team. 2021. TDB Documentation. https://jena.apache.org/documentation/tdb/

[38] MillenniumDB Team. 2022. Millennium Source Code. https://github.com/MillenniumDB/MillenniumDB

[39] MillenniumDB Team. 2022. Millennium Whitepaper. https://arxiv.org/abs/2111.01540

[40] MillenniumDB Team. 2022. Wikidata Benchmark. https://github.com/MillenniumDB/benchmark

[41] Nebula Graph Team. 2022. Nebula Graph Query Language (nGQL) – version 3.0.2. https://docs.nebula-graph.io/3.0.2/

[42] OrientDB Team. 2021. OrientDB Manual – version 3.0.34. https://orientdb.org/docs/3.0.x/

[43] Stardog Team. 2021. Stardog 7.6.3 Documentation. https://docs.stardog.com/

[44] TigerGraph Team. 2021. TigerGraph Documentation – version 3.1. https://docs.tigergraph.com/

[45] Bryan B. Thompson, Mike Personick, and Martyn Cutcher. 2014. The Bigdata® RDF Graph Database. In *Linked Data Management*. Chapman and Hall/CRC, 193–237. http://www.crcnetbase.com/doi/abs/10.1201/b16859-12

[46] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. 96–106. https://doi.org/10.5441/002/icdt.2014.13

[47] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489

[48] Jim Webber. 2012. A programmatic introduction to Neo4j. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*. ACM, 217–218. https://doi.org/10.1145/2384716.2384777

[49] C. T. Yu and M. Z. Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*. 306–312.

APPENDIX

A SYNTAX OF MILLENNIUMDB QUERIES

The language supported by AnonDB borrows syntax from popular graph query languages SPARQL [16] and Cypher [14]. General structure of queries in AnonDB can be visualized as follows:

```
MATCH  MatchPattern
WHERE  Condition
RETURN Selectors
ORDER BY OrderSelectors
LIMIT Number
```

Figure 7: General structure of queries in MillenniumDB

Intuitively, the MATCH clause specifies the basic or navigational graph pattern which we will look for in our graph. The WHERE clause is used to filter result based on a selection, usually by restricting the values of some of the attributes of a matched object. The RETURN clause specifies which of the matched variables will be returned. The ORDER BY allows us to reorder the results based on the values of some output variables, while LIMIT cuts off the evaluation after a specific number of results had been found.

We define the formal syntax MillenniumDB query language in Figure 8. As an example of a query occupying many of these elements, consider the following:

```
MATCH (?x :Person)-[?e :knows]->(?y :Person {country:"Chile"})
      OPTIONAL {(?x)-[:studiedAt]->(?z)}
WHERE ?x.age >= 35 AND ?e.since == "1/1/2020"
RETURN ?x.age, ?y.age, ?z
ORDER BY ?x.age ASC, ?y.age DESC
LIMIT 1000
```

When evaluated on a social network graph, the query is trying to find the ages of all pairs of people such the first knows the second. Additionally, it is stating that the second person lives in Chile, that the first one has age greater than or equal 35, and that the edge connecting them was created on 1/1/2020. If the university where the first person studied is known, this is also returned (and is otherwise `null`). The results are ordered such that the age of the first person is ascending, and the age of the second person is descending. Finally, only 1000 results are solicited. The syntax here can be build rater easily using the specification of Figure 8. The most interesting part of the query is in the `MatchPattern`, which is composed of an `EdgePattern`, `(?x :Person)-[?e :knows]->(?y :Person {country:"Chile"})`, and an optional pattern `(?x)-[:studiedAt]->(?z)`, which is itself an `EdgePattern`.

B FORMAL DEFINITION OF DOMAIN GRAPH QUERIES

Queries in MillenniumDB are based on an abstract notion of *domain graph query*, which generalise the types of graph patterns used by modern graph query languages [3]. This query abstraction provides modularity in terms of how the database is constructed, flexibility in terms of what concrete query syntax is supported, and allows for defining its semantics and studying its theoretical properties in a clean way.

This section provides the formal definition of the MillenniumDB query language. From now on, assume an infinite set Var of variables disjoint with the set of objects Obj.

B.1 Basic graph patterns

At the core of domain queries are basic graph patterns.[10] A *basic graph pattern* is defined as a pair (V, ϕ) such that $\phi : (\mathrm{Obj} \cup \mathrm{Var}) \rightarrow (\mathrm{Obj} \cup \mathrm{Var}) \times (\mathrm{Obj} \cup \mathrm{Var}) \times (\mathrm{Obj} \cup \mathrm{Var})$ is a partial mapping with a finite domain and $V \subseteq \mathrm{var}(\phi)$, where $\mathrm{var}(\phi)$ is the set of variables occurring in the domain or in the range of ϕ. Thus, ϕ can be thought as a domain graph that allows a variable in any position, together with a set V of output variables (hence the restriction that each variable in V occurs in ϕ).

The evaluation of a basic graph pattern returns a set of solution mappings. A *solution mapping* (or simply *mapping*) is a partial function $\mu : \mathrm{Var} \rightarrow \mathrm{Obj}$. The *domain* of a mapping μ, denoted by $\mathrm{dom}(\mu)$, is the set of variables on which μ is defined. Given $v \in \mathrm{Var}$ and $o \in \mathrm{Obj}$, we use $\mu(v) = o$ to denote that μ maps variable v to object o. Besides, given a set V' of variables, the term $\mu_{|V'}$ is used to denote the mapping obtained by restricting μ to V', that is, $\mu_{|V'} : (\mathrm{dom}(\mu) \cap V') \rightarrow \mathrm{Obj}$ such that $\mu_{|V'}(v) = \mu(v)$ for every $v \in (\mathrm{dom}(\mu) \cap V')$ (notice that V' is not necessarily a subset of $\mathrm{dom}(\mu)$). Finally, for the sake of presentation, we assume that $\mu(o) = o$, for all $o \in \mathrm{Obj}$.

[10]Basic graph patterns correspond to conjunctive queries (CQs) over graphs.

$$\boxed{\text{Selectors}}$$

Selectors :− * | Variables

Variables :− v | $v.k$ | Variables, Variables   , $v \in \text{Var}, k \in \mathcal{K}$

$$\boxed{\text{MatchPattern}}$$

MatchPattern :− GraphPattern | (MatchPattern) OPTIONAL { MatchPattern }

GraphPattern :− GraphElement | GraphElement, (GraphPattern)

GraphElement :− NodePattern | EdgePattern | PropertyPath

NodePattern :− (Node? Labels? Properties?)

Node :− $v \in \text{Var}$ | $o \in \text{Obj}$

Labels :− :l | :l Labels   , $l \in \mathcal{L}$

Properties :− { PropertyList }

PropertyList :− k:val | k:val, PropertyList   , $k \in \mathcal{K}, val \in \mathcal{V}$

EdgePattern :− NodePattern -Edge?> NodePattern | NodePattern <-Edge? NodePattern

Edge :− [v? Type? Properties?]-   , $v \in \text{Var}$

Type :− :o | :?v , $o \in \text{Obj},\ v \in \text{Var}$

PropertyPath :− NodePattern =[pathExp]=> NodePattern | NodePattern <=[pathExp]= NodePattern

pathExp :− :o | ^pathExp | (pathExp/pathExp) | pathExp* |
pathExp+ | pathExp{n,m} | (pathExp | pathExp) | pathExp?   , $o \in \text{Obj}, \text{n}, \text{m} \in \mathbb{N}, \text{n} \leq \text{m}$

$$\boxed{\text{Conditions}}$$

Condition :− Comparison | (Condition AND Condition) | (Condition OR Condition)

Comparison :− $v.k \sim val$ | $v.k \sim v'.k'$, $v, v' \in \text{Var}, k, k' \in \mathcal{K}, val \in \mathcal{V}, \sim \in$ {==,<=,>=,<,>}

$$\boxed{\text{OrderSelectors}}$$

OrderSelectors :− v | v ASC | v DESC | $v.k$ | $v.k$ ASC | $v.k$ DESC | OrderSelectors, OrderSelectors   , $v \in \text{Var}, k \in \mathcal{K}$

Figure 8: Specification of query patterns in MillenniumDB. Blue symbols are taken literally. Red brackets in `MatchPattern` and `GraphPattern` are grouping specification for unambiguous parsing, and are not specified when writing the query. The `Number` going in the `LIMIT` command is any integer.

The evaluation of a basic graph pattern $B = (V, \phi)$ over a domain graph $G = (O, \gamma)$, denoted by $[\![B]\!]_G$, is defined as $[\![B]\!]_G = \{\mu_{|V} \mid \mu \in [\![\phi]\!]_G\}$, where:

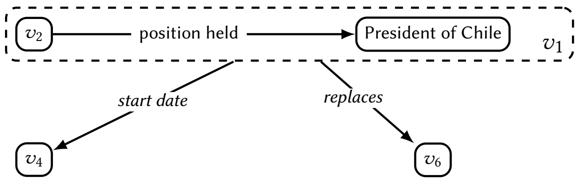
$$[\![\phi]\!]_G \;=\; \{\mu \mid \text{dom}(\mu) = \text{var}(\phi) \text{ and if } \phi(a) = (a_1, a_2, a_3), \text{then } \gamma(\mu(a)) = (\mu(a_1), \mu(a_2), \mu(a_3))\}.$$

For example, consider the basic graph pattern (V, ϕ) where $V = \{v_2, v_4, v_6\}$ and ϕ is given by the assignments:

$\phi(v_1) = (v_2,$ position held, President of Chile$)$,
$\phi(v_3) = (v_1,$ start date$, v_4)$, and
$\phi(v_5) = (v_1,$ replaces$, v_6)$.

In Figure 9, we provide a graphical representation of the above graph pattern, and the solution mappings obtained by evaluating the graph pattern over the property domain graph shown in Figure 4. The solution mappings are presented as a table with columns v_2, v_4, v_6 (i.e. the variables in V), and each row represents an individual mapping. In our definitions, different variables may map to the same object in a single solution. Thus, our notion of evaluation follows a homomorphism-based semantics, similar to query languages such as SPARQL [3].[11]

[11]Isomorphism-based semantics [3] – such as Cypher's no-repeated-edge semantics, which disallows the same solution to use the same edge twice – can be emulated by filtering solutions after they are generated.



v_2	v_4	v_6
`Michelle bachelet`	`2006-03-11`	`Ricardo Lagos`
`Michelle bachelet`	`2018-03-11`	`Sebastián Piñera`

Figure 9: Graphical representation of a basic graph pattern (left), and the tabular representation of the solution mappings (right) obtained by evaluating the basic graph pattern over the property domain graph shown in Figure 4 .

Supporting labels and properties. Observe that the formalization thus far only allowed to access elements of the function γ, and could not reason about labels, nor properties. In essence, up to now we only defined the semantics of queries over domain graphs. We now extend this definition to property domain graphs. More precisely, we wish to define the semantics of queries such as:

```
MATCH (?x :human)-[:knows {since:"1/1/2020"}]->(?y {name: "John"})
RETURN *
```

Following our approach of modelling a query similarly as a domain graph, we define a *basic graph pattern with properties* as a tuple $(V, \phi, \text{Qlab}, \text{Qprop})$, where:

- $\phi : (\text{Obj} \cup \text{Var}) \rightarrow (\text{Obj} \cup \text{Var}) \times (\text{Obj} \cup \text{Var}) \times (\text{Obj} \cup \text{Var})$ is a partial mapping;
- $V \subseteq \text{var}(\phi)$;
- $\text{Qlab} : (\text{Obj} \cup \text{Var}) \rightarrow \mathcal{L}$ is a partial mapping;
- $a \in \text{dom}(\text{Qlab})$ implies that there are x, y, z, w such that $\phi(x) = (y, z, w)$; and $a \in \{x, y, z, w\}$
- $\text{Qprop} : (\text{Obj} \cup \text{Var}) \times \mathcal{P} \rightarrow \mathcal{V}$ is a partial mapping;
- $(a, k) \in \text{dom}(\text{Qprop})$, for some $k \in \mathcal{P}$, implies that there are x, y, z, w such that $\phi(x) = (y, z, w)$; and $a \in \{x, y, z, w\}$

Basically, a basic graph pattern with properties extends basic graph patterns with a labelling function and a property checking function. Notice that the domain constraints on Qlab and Qprop serve to make sure that these are associated to some variable or object used in the core pattern. Given a property domain graph $G = (O, \gamma, \texttt{lab}, \texttt{prop})$, the semantics of a basic graph pattern with properties $BP = (V, \phi, \text{Qlab}, \text{Qprop})$, denoted $[\![BP]\!]_G$ is defined as $[\![BP]\!]_G = \{\mu_{|_V} \mid \mu \in [\![\phi, \text{Qlab}, \text{Qprop}]\!]_G\}$, where:

$$\begin{aligned}
[\![\phi, \text{Qlab}, \text{Qprop}]\!]_G \quad = \quad \{\mu \mid \quad & \text{dom}(\mu) = \text{var}(\phi); \\
& \phi(a) = (a_1, a_2, a_3) \text{ implies } \gamma(\mu(a)) = (\mu(a_1), \mu(a_2), \mu(a_3)); \\
& \text{Qlab(a)} = l \text{ implies } \texttt{lab}(\mu(a)) = l; \\
& \text{Qprop(a,k)} = v \text{ implies } \texttt{prop}(\mu(a), k) = v \ \}.
\end{aligned}$$

In essence, extending the definition to also support labels and properties with their values is akin to making the query pattern have the same structure as the property domain graph. This extension now allow us to query labels and properties as described in Section 3.

B.2 Navigational graph patterns

A characteristic feature of graph query languages is the ability to match paths of arbitrary length that satisfy certain criteria. We call basic graph patterns enhanced with this feature *navigational graph patterns*, and we define them next.

A popular way to express criteria that paths should match is through regular expressions on their labels, aka. *regular path queries* (*rpqs*). More precisely, an *rpq expression* r is defined by the following grammar:

$$r \quad ::= \quad \varepsilon \ \mid \ o \in \text{Obj} \ \mid \ (r/r) \ \mid \ (r + r) \ \mid \ r^- \ \mid \ r^*.$$

The semantics of an rpq expression r is defined in terms of its evaluation on a property-domain graph G, denoted by $[\![r]\!]_G$, which returns a set of pair of nodes in the graph that are connected by paths satisfying r. More precisely, assuming that $G = (O, \gamma)$, $o \in \text{Obj}$ and r, r_1, r_2 are rpq expressions, we have that:

$$\begin{aligned}
[\![\varepsilon]\!]_G &= \{(o, o) \mid o \in O\}, \\
[\![o]\!]_G &= \{(o_1, o_2) \mid \exists o' \in \text{Obj} : \gamma(o') = (o_1, o, o_2)\}, \\
[\![(r_1/r_2)]\!]_G &= \{(o_1, o_2) \mid \exists o' \in \text{Obj} : (o_1, o') \in [\![r_1]\!]_G \text{ and } (o', o_2) \in [\![r_2]\!]_G\}, \\
[\![(r_1 + r_2)]\!]_G &= [\![r_1]\!]_G \cup [\![r_2]\!]_G, \\
[\![r^-]\!]_G &= \{(o_1, o_2) \mid (o_2, o_1) \in [\![r]\!]_G\}.
\end{aligned}$$

Moreover, assuming that $r^1 = r$ and $r^{n+1} = r/r^n$ for every $n \geq 1$, we have that:

$$[\![r^*]\!]_G \quad = \quad [\![\varepsilon]\!]_G \cup \bigcup_{k \geq 1} [\![r^k]\!]_G.$$

Other rpq expressions widely used in practice can be defined by combining the previous operators. In particular, $r? = \varepsilon + r$ and $r^+ = r/r^*$.

A *path pattern* is a tuple (a_1, r, a_2) such that $a_1, a_2 \in \mathrm{Obj} \cup \mathrm{Var}$ and r is an rpq expression. As for the case of basic graph patterns, given a path pattern p, we use the term $\mathrm{var}(p)$ to denote the set of variables occurring in p. Moreover, the evaluation of $p = (a_1, r, a_2)$ over a property-domain graph G, denoted by $[\![p]\!]_G$, is defined as:

$$[\![p]\!]_G \quad = \quad \{\mu \mid \mathsf{dom}(\mu) = \mathrm{var}(p) \text{ and } (\mu(a_1), \mu(a_2)) \in [\![r]\!]_G\}.$$

For example, the expression (`Michelle Bachelet`, `(replaced by)`$^+$, v) is a path pattern that returns all the Presidents of Chile after Michelle Bachelet. Given a set ψ of path patterns, $\mathrm{var}(\psi)$ also denotes the set of variables occurring in ψ, and the evaluation of ψ over a property-domain graph G is defined as:

$$[\![\psi]\!]_G \quad = \quad \{\mu \mid \mathsf{dom}(\mu) = \mathrm{var}(\psi) \text{ and } \mu_{|\mathrm{var}(p)} \in [\![p]\!]_G \text{ for each } p \in \psi\}.$$

A *navigational graph pattern* is a triple (V, ϕ, ψ) where (V', ϕ) is a basic graph pattern for some $V' \subseteq V$, ψ is a set of path patterns, and $V \subseteq \mathrm{var}(\phi) \cup \mathrm{var}(\psi)$. The semantics of a navigational graph pattern $N = (V, \phi, \psi)$ is defined as:

$$[\![N]\!]_G \quad = \quad \{\mu_{|V} \mid \mathrm{var}(\mu) = \mathrm{var}(\phi) \cup \mathrm{var}(\psi), \mu_{|\mathrm{var}(\phi)} \in [\![\phi]\!]_G \text{ and } \mu_{|\mathrm{var}(\psi)} \in [\![\psi]\!]_G\}.$$

Hence, the result of a navigational graph pattern $N = (V, \phi, \psi)$ is a set of mappings μ projected onto the set V of output variables, where μ satisfies the structural restrictions imposed by ϕ and the path constrainst imposed by ψ. Notice that multiple rpq expressions can link the same pair of nodes. This is similar to the existential semantics of path queries, as specified in the SPARQL standard [16].

Given a domain graph $G = (O, \gamma)$, we define paths over the directed labeled graph that forms the range of γ; in other words, we do not allow for matching paths that emanate from an edge object. Such a feature could be considered in the future. We may also consider adding semantics for shortest paths, additional criteria on node or edges in the path, etc.

B.3 Relational graph patterns

As previously discussed (and seen in the example of Figure 9), graph patterns return relations (tables) as solutions. Thus we can – and many practical graph query languages do – use a relational-style algebra to transform and/or combine one or more sets of solution mappings into a final result.

Towards defining this algebra, we need the following terminology. Two mappings μ_1 and μ_2 are *compatible*, denoted by $\mu_1 \sim \mu_2$, if $\mu_1(v) = \mu_2(v)$ for all variables v which are in both $\mathsf{dom}(\mu_1)$ and $\mathsf{dom}(\mu_2)$. If $\mu_1 \sim \mu_2$, then we write $\mu_1 \cup \mu_2$ for the mapping obtained by extending μ_1 according to μ_2 on all the variables in $\mathsf{dom}(\mu_2) \setminus \mathsf{dom}(\mu_1)$. Given two sets of mappings Ω_1 and Ω_2, the *join* and *left outer join* between Ω_1 and Ω_2 are defined respectively as follows:

$$\begin{aligned} \Omega_1 \bowtie \Omega_2 \quad &= \quad \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}, \\ \Omega_1 ⟕ \Omega_2 \quad &= \quad (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2). \end{aligned}$$

With this terminology, a *relational graph pattern* is recursively defined as follows:

- If N is a navigational graph pattern, then N is also relational graph pattern;
- If R_1 and R_2 are relational graph patterns, then $(R_1 \ \text{AND} \ R_2)$ and $(R_1 \ \text{OPT} \ R_2)$ are relational graph patterns.

The evaluation of a relational graph pattern R over a property-domain graph G, denoted by $[\![R]\!]_G$, is recursively defined as follows:

- if R is a navigational graph pattern N, then $[\![R]\!]_G = [\![N]\!]_G$;
- if R is $(R_1 \ \text{AND} \ R_2)$ then $[\![R]\!]_G = [\![R_1]\!]_G \bowtie [\![R_2]\!]_G$;
- if R is $(R_1 \ \text{OPT} \ R_2)$ then $[\![R]\!]_G = [\![R_1]\!]_G ⟕ [\![R_2]\!]_G$.

B.4 Selection conditions

In addition to match a graph pattern against a property-domain graph, we would like to filter the solutions by imposing selection conditions over the resulting objects (i.e. nodes and edges). More precisely, a *selection condition* is defined recursively as follows: (a) if $v_1, v_2 \in \mathrm{Var}$ and $o \in \mathrm{Obj}$, $k_1, k_2 \in \mathcal{K}$, $v \in \mathcal{V}$ then $(v_1 = v_2)$, $(v_1 = o)$, $v_1.k_1 = v_2.k_2$ and $v_1.k_1 = v$ are selection conditions; and (b) if C_1, C_2 are selection conditions, then $(\neg\, C_1)$, $(C_1 \wedge C_2)$, $(C_1 \vee C_2)$ are selection conditions.

Given a property-domain graph $G = (O, \gamma, \mathtt{lab}, \mathtt{prop})$, a mapping μ, and a selection condition C, we say that μ satisfies C under G, denoted by $\mu \models_G C$, if one of the following statements holds:

- C is $(v_1 = v_2)$, $v_1, v_2 \in \mathsf{dom}(\mu)$ and $\mu(v_1) = \mu(v_2)$;
- C is $(v_1 = o_1)$, $v_1 \in \mathsf{dom}(\mu)$ and $\mu(v_1) = o_1$;
- C is $(v_1.k_1 = v_2.k_2)$, $v_1, v_2 \in \mathsf{dom}(\mu)$, $\mathtt{prop}(\mu(v_1), k_1) = \mathtt{prop}(\mu(v_2), k_2)$;
- C is $(v_1.k_1 = v)$, $v_1 \in \mathsf{dom}(\mu)$ and $\mathtt{prop}(\mu(v_1), k_1) = v$;
- C is $(\neg C_1)$, and it is not the case that $\mu \models_G C_1$;
- C is $(C_1 \wedge C_2)$, $\mu \models_G C_1$, and $\mu \models_G C_2$;
- C is $(C_1 \vee C_2)$ and either $\mu \models_G C_1$, $\mu \models_G C_2$, or both.

B.5 Solution modifiers

We consider an initial set of solution modifiers that allow for applying a final transformation on the solutions generated by a graph pattern. These include: return, which defines a set of elements (variables and properties) to be returned; order by, which orders the solutions according to a sort criteria; and limit, which returns the first n mappings in a sequence of solutions. Notice here that the solution mappings are not defined by the RETURN solution modifier, but rather by the relational graph pattern, and by selection conditions.

Let S be the set of strings, $v \in var$ and $k \in \mathcal{K}$. A *return mapping* is a function $\tau : \mathsf{S} \rightarrow \mathsf{Obj} \cup \mathcal{V}$. A *return element* is either a variable v or an expression $v.k$. Assume that there is a simple way to transform a selection element into a string in S. Given a sequence of selection mappings S and an integer n, the function $\mathtt{limit}(S, n)$ returns the first n elements of S when $n > 0$, and returns S otherwise.

Given a property-domain graph $G = (O, \gamma, \mathtt{lab}, \mathtt{prop})$, a mapping μ, and a sequence of return elements E, the function $\mathtt{ret}(\mu, E)_G$ produces a return mapping τ defined as follows: if $v \in E$ then "v" $\in \mathrm{dom}(\tau)$ and $\tau(\text{"}v\text{"}) = \mu(v)$; if $v.k \in E$ then "$v.k$" $\in \mathrm{dom}(\tau)$ and $\tau(\text{"}v.k\text{"}) = \mathtt{prop}(\mu(v), k)$. Moreover, given a set of return mappings Ω, the function $\mathtt{return}$ outputs a set of return mappings defined as $\mathtt{return}(\Omega, E)_G = \{\mathtt{ret}(\mu, E)_G \mid \mu \in \Omega\}$.

An *order modifier* is a tuple (e, β) where e is a return element and β is either $\mathtt{asc}$ or $\mathtt{desc}$. Given a sequence of return mappings S and an order modifier $o = (e, \beta)$, we say that S satisfies o, denoted $S \models o$, if it applies that: (i) β is $\mathtt{asc}$ and S satisfies an ascending order with respect to e; or (ii) β is $\mathtt{desc}$ and S satisfies a descending order with respect to e. Moreover, given a sequence of order modifiers $O = (o_1, \ldots, o_n)$, we say that S satisfies O, denoted $S \models O$, if it applies that: (i) $S \models o_1$ when $n = 1$; or (ii) $S \models o_1$ and, for every sub-sequence of selection mappings $S' \subseteq S$ it applies that $S' \models (o_2, \ldots, o_n)$ such that $\tau_i(e_1) = \tau_j(e_1)$ for any pair of selection mappings $\tau_i, \tau_j \in S'$, with $o_1 = (e_1, \beta_1)$.

B.6 Graph Queries

A *graph query* Q is defined as a tuple (R, C, E, O, n), where R is a relational graph pattern, C is a selection condition, E is a sequence of return elements, $O = \{o_1, \ldots, o_n\}$ is a sequence of order modifiers, and n is a positive integer. We assume that R is the unique mandatory component. Given a variable $v \in \mathrm{dom}(R)$, the rest of components have the following expressions by default: C is $v = v$, E is v, O is $(v, \mathtt{asc})$ and $n = 0$.

The evaluation of Q over G is defined as $\mathtt{limit}(S, n)$ where $S = \mathtt{return}(\Omega, E)_G$, $S \models O$, and $\Omega = \{\mu_{|V} \mid \mu \in [\![R]\!]_G \wedge \mu \models^G C\}$. We will assume that every graph query $Q = (R, C, E, O, n)$ satisfies the following two conditions: (i) For every sub-pattern $R' = (R_1 \ \mathrm{OPT} \ R_2)$ of R and for every variable v occurring in R, it applies that, if v occurs both inside R_2 and outside R', then it also occurs in R_1; (ii) It applies that $\mathbf{Var}(C) \subseteq \mathbf{Var}(R)$. Then, we say that Q is a *well-designed graph query*.

We finish this section noting that the semantics of a declarative query expression:

```
MATCH  R
WHERE  C
ORDER BY O
RETURN E
LIMIT n
```

is defined as the outputs of the graph query (R, C, E, O, n).