Towards a Theory of Façade-X Data Access: Satisfiability of SPARQL Basic Graph Patterns

Luigi Asprino
Università Telematica San Raffaele, Rome, Italy

Enrico Daga[1]
Knowledge Media Institute (KMI), The Open University, Milton Keynes, UK

Abstract

Data integration is the primary use case for knowledge graphs. However, integrated data are not typically graphs but come in different formats, for example, CSV, XML, or a relational database. Façade-X is a recently proposed method for providing direct access to an open-ended set of data formats. The method includes a meta-model that specialises RDF to fit general data structures. This model allows to express SPARQL queries targeting data sources with those structures. Previous work formalised Façade-X and demonstrated how it can theoretically represent any format expressible with a context-free grammar, as well as the relational model. A reference implementation, SPARQL Anything, demonstrates the feasibility of the approach in practice. It is noteworthy that Façade-X utilises a fraction of RDF, and, consequently, not all SPARQL queries yield a solution (i.e. are satisfiable) when evaluated over a Façade-X graph. In this article, we consolidate Façade-X and we study the *satisfiability* of basic graph patterns. The theory is accompanied by an algorithm for deciding the satisfiability of basic graph patterns on Façade-X data sources. Furthermore, we provide extensive experiments with a proof-of-concept implementation, demonstrating practical feasibility, including with real-world queries. Our results pave the way for studying query execution strategies for Façade-X data access with SPARQL and supporting developers to build more efficient data integration systems for knowledge graphs.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Information systems → Graph-based database models; Information systems → Query languages for non-relational engines; Information systems → Mediators and data integration

Keywords and phrases Façade-X, RDF, SPARQL, Knowledge Graphs, Data Integration

Digital Object Identifier 10.4230/TGDK.4.2.5

Category Research

Supplementary Material *Software (Source Code)*: `https://zenodo.org/records/18196574` [15]
Dataset (Data): `https://zenodo.org/records/18133036` [14]

Funding *Enrico Daga*: EU funded projects (a) MultiPod: grant agreement 101178821, `https://cordis.europa.eu/project/id/101178821`; and (b) Climate Misinformation Surveillance and Analysis with Multidimensional GIS, `https://climatesense-project.eu/`, CHIST-ERA project.

Received 2025-07-15 **Accepted** 2026-04-03 **Published** 2026-09-03

Editors Stefania Dumbrava, George Fletcher, Matteo Lissandrini, and Riccardo Tommasini

Special Issue Data Management for (Knowledge) Graphs

Generative AI Declaration Generative AI was not used in the preparation of this article.

1 Introduction

Knowledge graphs use a graph-based model to integrate, manage, and extract value from diverse data sources at scale [26]. Thus, data integration is the primary application of Knowledge Graphs (KG). The W3C standard SPARQL 1.1 [23] serves as the authoritative language for engaging

[1] Corresponding author

with knowledge graphs serialised in RDF [32]. SPARQL provides functionalities for selecting, filtering, and aggregating data into a table format. Additionally, SPARQL allows the projection of results into an RDF structure using the CONSTRUCT query type. Consequently, research has investigated the use of SPARQL across a variety of applications beyond mere querying, particularly in the integration of diverse data sources [13, 33, 36]. In practice, methods often depend on enhancing SPARQL to retrieve data from non-RDF formats.

Recent studies [16] suggest utilizing an intermediate RDF model called Façade-X, which allows its components to be seamlessly converted into RDF Knowledge Graphs. This approach enables the development of software that offers *direct access* to source data as RDF, freeing knowledge engineers from the burden of managing diverse formats and associated languages they depend on. Façade-X uniquely addresses *re-engineering*, which means transforming resources by minimizing domain-specific concerns and concentrating on the syntactical meta-model, and allows users to focus on *remodelling*, that is, transforming the original domain model into a new representation [16]. In theory, Façade-X can represent any format defined by a context-free grammar, as well as the relational data model [8].

The Façade-X method informs the SPARQL Anything software and open source project, which also serves as its reference implementation[2].

```
email,name,surname
laura@example.com,Laura,Grey
craig@example.com,Craig,Johnson
mary@example.com,Mary,Jenkins
jamie@example.com,Jamie,Smith
```

Figure 1 Example CSV file.

Listing 1 A query for retrieving the surname of people whose first name is "Laura".

```
PREFIX xyz: <http://sparql.xyz/facade-x/data/>

SELECT ?surname WHERE {
  SERVICE <x-sparql-anything:https://sparql-anything.cc/example1.csv> {
    _:person xyz:surname ?surname .
    _:person xyz:name "Laura" .
  }
}
```

▶ Example 1. Suppose having the CSV file shown in Figure 1 as input[3]. Using SPARQL Anything, a SPARQL user can retrieve the surnames of people whose first name is "Laura" by running the query in Listing 1 obtaining the following result:

surname
"Grey"

[2] `http://sparql-anything.cc`
[3] Also available at `https://sparql-anything.cc/example1.csv`

This system (accessible through a web server, CLI, Java, or Python API [40]) has been utilized in numerous practical applications and is gaining recognition from the knowledge graph community in both research and industry [7][4]. The W3C Data Façades Community Group has been recently founded to develop Façade-X as a vendor-neutral, interoperable technology that can be implemented by various developers and organisations, including graph database vendors[5]

Currently, the execution of SPARQL queries on Façade-X data sources requires the materialisation of the source (in full or via a slicing approach [6]). An open question is how to support the implementation of more efficient query engines. It is noteworthy that Façade-X utilises a fraction of RDF, and, consequently, not all SPARQL queries yield a solution (i.e. are satisfiable) when evaluated over a Façade-X graph. For example, the query described in the Example 1 is satisfiable, whereas the query in the Listing 2 is not. This is because the Façade-X model does not allow for such a pattern: CSV data sources don't have any `rdf:type` triples; therefore, no results would be produced from its evaluation. These considerations can be made simply by looking at the query itself (rather than the input) and could save computational resources.

Listing 2 A query for retrieving the surname of people whose first name is "Laura".

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?s WHERE {
  SERVICE <x-sparql-anything:https://sparql-anything.cc/example1.csv> {
    ?s rdf:_1 ?o . ?x rdf:type ?s .
  }
}
```

In this article, we study the satisfiability of basic graph patterns under Façade-X. In addition, we contribute an algorithm to annotate any SPARQL BGP with possible solution patterns, to determine whether a given BGP can have a solution within Façade-X. We contribute:

- A consolidated formal definition of Façade-X and associated mappings to RDF;
- Characterisation of the satisfiability of SPARQL basic graph patterns queries within Façade-X;
- Algorithm for assessing whether a SPARQL basic graph patterns is satisfiable under Façade-X.

The rest of the article is structured as follows. The next section is dedicated to surveying related literature and contextualise our work. Section 3 introduces the formalisms and languages adopted in this article. Section 4 presents a consolidated version of Façade-X whose mapping to RDF is formalised in Section 5.1. Finally, Section 6 studies the conditions under which a BGP is satisfiable in Façade-X which are exploited in the algorithm presented in Section 7. Finally, Section 8 provides experiments with a proof-of-concept implementation, which demonstrates the practical feasibility and advantages of our theory, including an evaluation with real world queries. We conclude the article with Section 9.

2 Related work

We survey literature on Knowledge Graph Construction (KGC), i.e. systems and methods to onboard non-RDF data into RDF databases. We then look into approaches to extend SPARQL. Next, we focus on the evaluation of SPARQL queries, particularly basic graph patterns. Furthermore, we review the problem of satisfiability of graph patterns outside the RDF area. We conclude with a look at various approaches to query documents and provide a unifying view over heterogeneous resources.

[4] See also the activity on the open-source project page on GitHub: `http://github.com/sparql-anything/sparql.anything`

[5] W3C Data Façades Community Group: `https://www.w3.org/community/facade-x/`.

2.1 Knowledge graph construction

Mapping languages for transforming heterogeneous files into RDF are represented by RML [29, 47], also specialised to support data cleaning operations [44], and specific forms of data: relational [41], geospatial data [30], etc. RML has been adopted as reference mapping language in most of the KGC systems, such as SDM-RDFizer [27], Morph-CSV [11],and Morph-KGC [4]. Planning techniques have been studied to improve the efficiency of RML engines [28]. A recent work provides an algeabric foundation for RML based approaches [37], which establish a *query* operator as a primitive component in the definition of a mapping. Façade-X could be used as a common language to express queries over heterogeneous data sources *formally*.

Authors of an alternative to RML, based on the RDF constraing language ShEx [21], stress the importance of making mappings usable by end users. Indeed, literature acknowledges how these languages are built with machine-processability in mind [25], and how defining or even understanding the rules is not trivial. Façade-X provides a common abstraction allowing to express queries to non-RDF sources using basic graph patterns. It is worth noting that, although abstractions targeting individual data formats are already available in the literature, Façade-X keeps a single formalism and a single data model (i.e. RDF) for all data formats. Therefore, it paved the way for querying non-RDF data in SPARQL and building KGs, by making plain CONSTRUCT queries with a SPARQL engine supporting Façade-X data access. Moreover, since Façade-X is an abstract model that has been defined independently of RDF (as shown in Section 4, its specification is in predicate logic), it can potentially be applied to other languages (e.g. Datalog). Indeed, other query languages could benefit from the homogeneous overview of heterogeneous sources provided by Façade-X. Although this analysis could be extended to other frameworks for representing and querying data, our focus is on SPARQL. However, it is an open question *how to efficiently execute SPARQL queries on Façade-X data sources* that cannot be fully loaded in memory via a materialisation approach [6]. This problem motivates the present work.

2.2 Approaches to extend SPARQL

We examine methods for enhancing SPARQL. A common technique for augmenting SPARQL is to introduce custom functions that can be utilized in FILTER or BIND operators[6]. Query processing engines can enhance SPARQL through the use of so-called magic properties. This method involves defining custom predicates to guide specific actions during query execution[7]. A comparable method is employed to incorporate XPath and XQuery in Virtuoso, which also enables the embedding of SQL functions within SPARQL to directly access the underlying relational model[8]. SPARQL Generate [33] presents a technique for transforming data from varied sources into RDF by enhancing the SPARQL syntax with a new GENERATE operator [33]. This method introduces two additional operators, SOURCE and ITERATOR. Custom functions execute ad-hoc operations on supported formats, utilizing XPath or JSONPath. Nevertheless, there are methods to augment SPARQL without modifying the standard syntax. For instance, BASIL variables permit the formulation of parametric queries by imposing a convention on SPARQL variable names [17, 35]. SPARQL Anything leverages BASIL variables to facilitate parametric queries and file names. SPARQL Micro service [36] offers a framework that, based on an API mapping

[6] ARQ includes a library of custom functions that facilitate aggregates, such as calculating the standard deviation of a set of values. ARQ functions: `https://jena.apache.org/documentation/query/extension.html` (accessed 15/12/2020).

[7] For instance, this allows for the specification of intricate fulltext searches over literal values. Query processors can outsource execution to a fulltext engine (e.g., Lucene) and produce a set of query results as triple patterns

[8] Virtuoso Open Source. SPARQL inline in SQL: `http://docs.openlinksw.com/virtuoso/rdfsparqlinline/`.

specification, encapsulates web APIs as SPARQL endpoints and employs a JSON-LD profile to convert the API's JSON responses into RDF. Façade-X adopts a similar minimalist strategy and extends SPARQL by *overriding* the functionality of the SERVICE operator.

2.3 Evaluation of SPARQL graph patterns

Foundations of SPARQL query executions can be found in [26]. Seminal work in the area include studying the semantics of the query language [39, 3] and optimise the execution of graph pattern matching [46, 24, 19, 49]. Recent work tackles the efficient execution of SPARQL queries via translation in Datalog [2] or via combining logical and physical query plans [38].

The satisfiability problem is undecidable in SPARQL 1.0 (and, therefore, in 1.1 that is its extension) but becomes decidable for basic graph patterns [52]. Although decidability is NP-complete in theory, decidability of SPARQL BGPs is tractable in practices as the SPARQL BGPs are typically small [52]. Our work starts from the observation that Façade-X RDF graphs are less expressive than general RDF data. This motivates the study of evaluating basic graph patterns in the context of Façade-X graphs. However, it is certainly possible that optimization techniques for RDF pattern matching can be also applied to Façade-X. Hence, in the present work, we investigate the nature of Façade-X graphs and their corresponding basic graph patterns in SPARQL.

Another line of research analyses the satisfiability of BGPs in the context of conjunctive query (CQ) theory. Indeed, a SPARQL BGP can be viewed as a CQ over an RDF graph, thus inheriting the properties of CQs. A well-known result of Yannakakis [51] demonstrates that the evaluation of acyclic CQs is solvable in polynomial time. This result shows that the shape of a query fundamentally impacts the complexity of its evaluation. This perspective is relevant to Façade-X. Since the Façade-X data model only allows for rooted, acyclic structures, thus enforcing the tree-like nature of satisfiable BGPs.

2.4 Satisfiability of patterns in graphs

The problem of efficiently performing graph matching operations is significant beyond SPARQL [20, 34, 50, 9, 18]. While the general problem of subgraph pattern matching is known to be NP-complete [9], it seems that for a wide variety of statistics concerning trees, the expected time complexity for pattern matching is linear [45]. The satisfiability problem of tree patterns queries (within XPath and XQuery) is investigated in [31], where it is observed that some of the queries are NP-complete, while others can be resolved in polynomial time.

While Façade-X graphs can be seen as trees, when referring to static data sources, such as JSON documents or XML files (where two equal strings have different identity, because it derives from their position), their representation in RDF are not, because identical types and literals will be projected to unique RDF nodes. Instead, Façade-X/RDF graphs are *directed, acyclic, and rooted path graphs*. In the present work, we focus on satisfiability of SPARQL basic graph patterns on Façade-X RDF sources, and contribute a theory and related algorithm for deciding whether a graph pattern can have a solution in a possible Façade-X RDF graph.

2.5 Other approaches

Tools are available for automatically transforming data sources of several formats into RDF (Any23[9], JSON2RDF[10], CSV2RDF[11] to name a few). While these tools have a similar goal (i.e. enabling the user to access the content of a data source as if it was in RDF), the (meta)model used

[9] `http://any23.apache.org/` (now retired).
[10] `https://github.com/AtomGraph/JSON2RDF`
[11] `http://clarkparsia.github.io/csv2rdf/`

for generating the RDF data highly depends on the input format, thus limiting the homogeneity of data generated from heterogeneous data formats. In addition, none of those approaches are based on a common abstraction from heterogeneous formats.

A way to provide generalised data access is the list-of-lists approach. This is used by software libraries to onboard heterogeneus data structures, see, for example, Pandas[12] or the Visidata tool[13]. Façade-X follows a similar intuition, while targeting *graphs* as the general meta-model, rather than tabular representations.

Façade-X use cases typically involve documents. The approaches to querying documents have a tradition in computer science [12, 1, 48], and many format-specific languages have been developed for that (XPath, JsonPath, and CSS selectors are the most popular). Façade-X can be seen as a general approach to express queries over heterogeneus files. Here, we focus on the satisfiability of basic graph patterns against Façade-X RDF graphs. This analysis is the foundation for studying efficient execution strategies of SPARQL / Façade-X queries against serialized resources such as JSON or XML files.

3 Preliminaries

We introduce the reader to the formalisms on which our theory is based, namely RDF, SPARQL and Predicate Logic.

3.1 Predicate logic

As the façade-x model is defined in predicate logic, we provide the reader with a brief introduction to the main ingredients of this formalism. However, providing a full description of the syntax and semantic of PL is not the purpose of this section. For a complete definition of the syntax and semantic of PL we defer the reader to reference books, such as [42]. Predicate Logic (PL) is the logic to speak about objects, which are the domain of discourse (universe $\mathcal{U}$). PL is concerned about properties and relations about the objects of the universe which are expressed in form of predicates. Intuitively, a PL formula is combination of PL terms (variables and constants) with logical operators $\neg, \wedge, \vee, \Rightarrow$ and quantifiers $\exists, \forall$. An interpretation $\mathcal{I}$ of one or multiple PL formulas specifies what each predicate means, and the entities that can instantiate the variables. Given a set of predicates $P_1 ... P_n$ and function $f_1 ... f_m$ a PL interpretation is

$$\mathcal{I} = (\Delta^{\mathcal{I}}, P_1^{\mathcal{I}} ... P_n^{\mathcal{I}}, f_1^{\mathcal{I}} ... f_m^{\mathcal{I}})$$

where $\Delta^{\mathcal{I}} \subseteq \mathcal{U}$ is the domain, $P_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \cdots \times \Delta^{\mathcal{I}}$ (k-times where k is the arity of $\mathcal{P}_i$) is the interpretation of the predicate P_i, $f_i^{\mathcal{I}} : \Delta^{\mathcal{I}} \times \cdots \times \Delta^{\mathcal{I}} \to \Delta^{\mathcal{I}}$ (k-times where k is the arity of f_i) is the interpretation of the function f_i.

A PL open formula, also called *query*, is a PL formula with free variables (i.e. not quantified). Let `Vars` be the set of variables, given an interpretation $\mathcal{I}$, an assignment is a function $\alpha : \texttt{Vars} \to \texttt{Δ}^{\mathcal{I}}$.

Given a formula ϕ with free variables $(x_1 ... x_n)$ and an interpretation $\mathcal{I}$, the answer to the query is the tuple $(a_1 ... a_n)$, where a_i is an assignment of a free variable to an object of the domain of discourse $\Delta^{\mathcal{I}}$, such that $\mathcal{I}, (a_1 ... a_n) \models \phi$. We introduce $\mathcal{L}$ as the set of all possible queries, i.e. the query language, and the function `eval` which given a query ϕ and an interpretation $\mathcal{I}$ returns the list of assignments that satisfy ϕ over $\mathcal{I}$.

[12] `https://hevodata.com/learn/pandas-load-json/`
[13] `https://www.visidata.org/`

3.2 RDF and SPARQL

In this section, we use relational algebra notation to define the elements of the universe under consideration, indicated by $\mathcal{U}$, namely RDF graphs, SPARQL queries, and resource, data sources and values they contain that will be introduced in Section 6. We also use predicate logic to define the conditions that apply to the elements of the universe.

We start from the basic notions of RDF [32] and SPARQL [23]. We defer the reader to [26] and to the corresponding documentation for a complete description of these standards. Formally, let $\mathcal{I} \subseteq \mathcal{U}$, $\mathcal{B} \subseteq \mathcal{U}$, and $\mathcal{L} \subseteq \mathcal{U}$ be infinite sets of IRIs, blank nodes, and literals. The sets are assumed to be pairwise disjoint and we will collectively refer to them as *RDF terms*. A tuple $(s, p, o) \in \mathcal{T}$ (with $\mathcal{T}$ is the set $(\mathcal{I} \cup \mathcal{B}) \times (\mathcal{I}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$) is called *(RDF) triple* and we say s is the *subject* of the triple, p the *predicate*, and o the *object*. An *RDF graph* is a set of RDF triples, whereas an *RDF dataset* is a collection of named RDF graphs, each one identified by IRI, and a default RDF graph. *RDF graphs* can be expressed as a set of quads, i.e. elements of the set $\mathcal{I} \times \mathcal{T}$.

SPARQL is based on the idea of defining patterns to be matched against an input RDF dataset. Formally, considering the set of variables $\mathcal{V} \subseteq \mathcal{U}$, disjoint from the previously defined $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$, a *triple pattern* is a tuple of the form $(s, p, o) \in (\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$[14]. All the possible SPARQL triple patterns are enumerated in Table 1. A *basic graph pattern (BGP)* is a set of triple patterns.

A SPARQL query Q is composed of the following components: **(i)** the query type (i.e. SELECT, ASK, DESCRIBE, CONSTRUCT); **(ii)** the dataset clause; **(iii)** the graph pattern (recursively defined as being a BGP or the result of the composition of one or more graph patterns through one of several SPARQL operators that modify and combine the obtained results); **(iv)** the solution modifiers (i.e. LIMIT, GROUP BY, OFFSET).

This work focuses solely on the graph pattern itself, since the other components can be evaluated as defined in SPARQL 1.1. Consider a graph pattern consisting of two joined unordered triple patterns with a join condition defined on a single node. There are 6 possible joins matching nodes of the two triple patterns. The joins are listed in Table 2.

More complex join conditions involving multiple nodes can be derived by combining these joins. Most common graph structures can be classified according to a set of graph *motifs* [5, 43].These are reported in Table 3. For brevity, the motifs only contain nodes as variables, but these can be any valid combination of node types.

4 A consolidated version of Façade-X

Façade-X is the meta-model that results from the *abstraction* of all the basic data structures used to represent the source data formats and the *combination* of these data structures into a unified model. In this section, we present a consolidated version of Façade-X.

4.1 Resources and data sources

Before detailing the Façade-X, it is useful to formalise elements of its application context, i.e. the fact that data is serviced and shaped within a digital artifact of sort. We rely on the generic notion of *resource* to refer to a digital artifact that wraps some data[15]. In the Example 1, the Resource is the file identified by the URL `https://sparql-anything.cc/examples/simple.csv`, while the

[14] We are aware that the BGP specification in [23] includes literals in subject position. However, such patterns won't have a solution in RDF graphs, as defined in [32].

[15] To not be confused with the concept of Resource in RDFS (that is the most general type of RDF term) [22]

Table 1 The list of all possible SPARQL triple patterns.

Id	Pattern	Example
$\mathcal{VVV}$	$\mathcal{V}\ \mathcal{V}\ \mathcal{V}$	`?s ?p ?o`
$\mathcal{VVI}$	$\mathcal{V}\ \mathcal{V}\ \mathcal{I}$	`?s ?p <o>`
$\mathcal{VVB}$	$\mathcal{V}\ \mathcal{V}\ \mathcal{B}$	`?s ?p _:o`
$\mathcal{VVL}$	$\mathcal{V}\ \mathcal{V}\ \mathcal{L}$	`?s ?p "o"`
$\mathcal{VIV}$	$\mathcal{V}\ \mathcal{I}\ \mathcal{V}$	`?s <p> ?o`
$\mathcal{VII}$	$\mathcal{V}\ \mathcal{I}\ \mathcal{I}$	`?s <p> <o>`
$\mathcal{VIB}$	$\mathcal{V}\ \mathcal{I}\ \mathcal{B}$	`?s <p> _:o`
$\mathcal{VIL}$	$\mathcal{V}\ \mathcal{I}\ \mathcal{L}$	`?s <p> "o"`
$\mathcal{IVV}$	$\mathcal{I}\ \mathcal{V}\ \mathcal{V}$	`<s> ?p ?o`
$\mathcal{IVI}$	$\mathcal{I}\ \mathcal{V}\ \mathcal{I}$	`<s> ?p <o>`
$\mathcal{IVB}$	$\mathcal{I}\ \mathcal{V}\ \mathcal{B}$	`<s> ?p _:o`
$\mathcal{IVL}$	$\mathcal{I}\ \mathcal{V}\ \mathcal{L}$	`<s> ?p "o"`
$\mathcal{IIV}$	$\mathcal{I}\ \mathcal{I}\ \mathcal{V}$	`<s> <p> ?o`
$\mathcal{III}$	$\mathcal{I}\ \mathcal{I}\ \mathcal{I}$	`<s> <p> <o>`
$\mathcal{IIB}$	$\mathcal{I}\ \mathcal{I}\ \mathcal{B}$	`<s> <p> _:o`
$\mathcal{IIL}$	$\mathcal{I}\ \mathcal{I}\ \mathcal{L}$	`<s> <p> "o"`
$\mathcal{BVV}$	$\mathcal{B}\ \mathcal{V}\ \mathcal{V}$	`_:s ?p ?o`
$\mathcal{BVI}$	$\mathcal{B}\ \mathcal{V}\ \mathcal{I}$	`_:s ?p <o>`
$\mathcal{BVB}$	$\mathcal{B}\ \mathcal{V}\ \mathcal{B}$	`_:s ?p _:o`
$\mathcal{BVL}$	$\mathcal{B}\ \mathcal{V}\ \mathcal{L}$	`_:s ?p "o"`
$\mathcal{BIV}$	$\mathcal{B}\ \mathcal{I}\ \mathcal{V}$	`_:s <p> ?o`
$\mathcal{BII}$	$\mathcal{B}\ \mathcal{I}\ \mathcal{I}$	`_:s <p> <o>`
$\mathcal{BIB}$	$\mathcal{B}\ \mathcal{I}\ \mathcal{B}$	`_:s <p> _:o`
$\mathcal{BIL}$	$\mathcal{B}\ \mathcal{I}\ \mathcal{L}$	`_:s <p> "o"`

Table 2 The list of all possible single-node joins involving two SPARQL triple patterns.

Id	Join	Example
$S \bowtie P$	$\texttt{subj}_1 = \texttt{pred}_2$	`?j ?p1 ?o1 . ?s2 ?j ?o2`
$P \bowtie O$	$\texttt{pred}_1 = \texttt{obj}_2$	`?s1 ?j ?o1 . ?s2 ?p1 ?j`
$S \bowtie S$	$\texttt{subj}_1 = \texttt{subj}_2$	`?j ?p1 ?o1 . ?j ?p1 ?o2`
$P \bowtie P$	$\texttt{pred}_1 = \texttt{pred}_2$	`?s1 ?j ?o1 . ?s2 ?j ?o2`
$S \bowtie O$	$\texttt{subj}_1 = \texttt{obj}_2$	`?j ?p1 ?o1 . ?s2 ?p1 ?j`
$O \bowtie O$	$\texttt{obj}_1 = \texttt{obj}_2$	`?s1 ?p1 ?j . ?s2 ?p1 ?j`

content of the CSV file is the DataSource. We say that a resource contains one or more *data sources*, to refer to actual data contained in them. Intuitively, a CSV is a file (then, a resource) that includes some tabular data (hence, a single data source). A Spreadsheet is a file (then, a resource) containing many tabs, each one of them including tabular data (hence, possibly multiple data sources).

We introduce the unary predicates `Resource` and `DataSource` and the binary predicate `includes`. These predicates must comply with the following conditions: **(i)** Resources includes at least one data source; **(ii)** A data source must be included in one and only one resource. See Listing 3.

Listing 3 Formal definition of resources and data sources.

$$\forall \texttt{r}, \texttt{ds}.\texttt{includes}(\texttt{r}, \texttt{ds}) \Rightarrow \texttt{Resource}(\texttt{r}) \wedge \texttt{DataSource}(\texttt{ds})$$
$$\forall \texttt{r}.\exists \texttt{ds}.\texttt{Resource}(\texttt{r}) \Rightarrow \texttt{includes}(\texttt{r}, \texttt{ds})$$
$$\forall \texttt{ds}.\exists \texttt{r}.\texttt{DataSource}(\texttt{ds}) \Rightarrow \texttt{includes}(\texttt{r}, \texttt{ds})$$
$$\forall \texttt{ds}, \texttt{r}_1, \texttt{r}_2.\texttt{includes}(\texttt{r}_1, \texttt{ds}) \wedge \texttt{includes}(\texttt{r}_2, \texttt{ds}) \Rightarrow \texttt{r}_1 = \texttt{r}_2$$

Table 3 A list of possible SPARQL BGP motifs.

BGP Motif Name	Path	Cycle/Clique	Star	Snowflake
Motif	`?s ?p ?o . ?o ?p1 ?o1`	`?s ?p ?o . ?o ?p1 ?s`	`?s ?p ?o . ?s ?p1 ?o1 . ?s ?p2 ?o2`	`?s ?p ?o . ?s ?p1 ?o1 . ?s ?p2 ?o2 . ?o ?p3 ?o3 . ?o ?p4 ?o4`

4.2 Façade-X

We overview the basic data structures needed for representing the content of common resources including CSV, JSON, YAML, XML, HTML, Text, Markdown, Relational Databases, and Spreadsheets (e.g. XLS and XLSx)[16]. The framework proposed in [8] represents any data source as a combination of only three knowledge representation primitives, namely: sets of key-data pairs (*maps*), ordered sequence of elements (*lists*), and unary predicates (*types*). In fact, this framework interprets any data source as a sequence of values, regardless of its format, and then, arranges the values of the sequence in multiple structures according to their serialisation format. Crucially, all these structures can be captured using the three knowledge representation primitives mentioned above. In addition, the framework introduces the notion of *Container* as an abstraction of both lists and maps through the notion of *Slots*. A *Slot* is an allotted place for an object, namely a primitive value or another container, *"contained"* in a *Container*. *Container* may have a type. Façade-X is a representation of a data source in terms of Containers, Types, Slots, and Values. In this Section we provide a formal definition of the Façade-X. Figure 2 shows a diagrammatic representation of the Façade-X that provides an informal overview of the model.

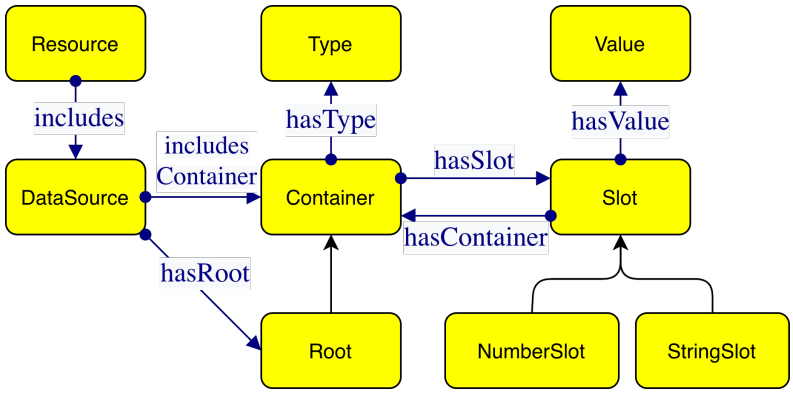

Figure 2 A diagrammatic representation of the Façade-X model. Unlabelled arcs represent subsumtion (e.g. Root is a type of Container).

We introduce the unary predicates: `Container`, `Slot`, `StringSlot`, `NumberSlot` and `Value`. We also introduce the binary predicates `hasSlot`, `hasValue`, `includesContainer` and `hasContainer`. Containers are collections of unique key/value pairs (slots) where: the key of the slot is unique in the collection and can be either a number (i.e. `NumberSlot`) or a sequence of alphanumeric characters (i.e. `StringSlot`); the value can be either a primitive value or another container. See Listing 4.

[16] See also `https://sparql-anything.readthedocs.io/stable/#supported-formats`.

Listing 4 Formal definition of containers, slots, and values.

$$
\begin{aligned}
&\forall \mathtt{c}, \mathtt{s}.\mathtt{hasSlot}(\mathtt{c},\mathtt{s}) \Rightarrow \mathtt{Container}(\mathtt{c}) \wedge \mathtt{Slot}(\mathtt{s})\\
&\forall \mathtt{s}, \mathtt{v}.\mathtt{hasValue}(\mathtt{s},\mathtt{v}) \Rightarrow \mathtt{Slot}(\mathtt{s}) \wedge \mathtt{Value}(\mathtt{v})\\
&\forall \mathtt{s}, \mathtt{c}.\mathtt{hasContainer}(\mathtt{s},\mathtt{c}) \Rightarrow \mathtt{Slot}(\mathtt{s}) \wedge \mathtt{Container}(\mathtt{v})\\
&\forall \mathtt{s}.\mathtt{NumberSlot}(\mathtt{s}) \Rightarrow \mathtt{Slot}(\mathtt{s})\\
&\forall \mathtt{s}.\mathtt{StringSlot}(\mathtt{s}) \Rightarrow \mathtt{Slot}(\mathtt{s})\\
&\nexists \mathtt{s}.\mathtt{StringSlot}(\mathtt{s}) \wedge \mathtt{NumberSlot}(\mathtt{s})\\
&\forall \mathtt{s}.\mathtt{Slot}(\mathtt{s}) \Rightarrow \mathtt{NumberSlot}(\mathtt{s}) \vee \mathtt{StringSlot}(\mathtt{s})\\
&\forall \mathtt{ds}, \mathtt{c}.\mathtt{includesContainer}(\mathtt{ds},\mathtt{c}) \Rightarrow \mathtt{DataSource}(\mathtt{ds}) \wedge \mathtt{Container}(\mathtt{c})\\
&\forall \mathtt{c}.\exists \mathtt{ds}.\mathtt{Container}(\mathtt{c}) \Rightarrow \mathtt{includesContainer}(\mathtt{ds},\mathtt{c})\\
&\forall \mathtt{ds}_1, \mathtt{ds}_2, \mathtt{c}.\mathtt{includesContainer}(\mathtt{ds}_1,\mathtt{c}) \wedge \mathtt{includesContainer}(\mathtt{ds}_2,\mathtt{c}) \Rightarrow \mathtt{ds}_1 = \mathtt{ds}_2
\end{aligned}
$$

Listing 5 Formal definition of the properties of slots.

$$
\begin{aligned}
&\forall \mathtt{v} \exists \mathtt{s}.\mathtt{Value}(\mathtt{v}) \Rightarrow \mathtt{hasValue}(\mathtt{s},\mathtt{v})\\
&\forall \mathtt{s} \exists \mathtt{c}.\mathtt{Slot}(\mathtt{s}) \Rightarrow \mathtt{hasSlot}(\mathtt{c},\mathtt{s})\\
&\forall \mathtt{c}_1, \mathtt{c}_2, \mathtt{s}.\mathtt{hasSlot}(\mathtt{c}_1,\mathtt{s}) \wedge \mathtt{hasSlot}(\mathtt{c}_2,\mathtt{s}) \Rightarrow \mathtt{c}_1 = \mathtt{c}_2\\
&\forall \mathtt{s} \exists \mathtt{x}.\mathtt{Slot}(\mathtt{s}) \Rightarrow \mathtt{hasContainer}(\mathtt{s},\mathtt{x}) \vee \mathtt{hasValue}(\mathtt{s},\mathtt{x})\\
&\nexists \mathtt{s}, \mathtt{x}.\mathtt{hasContainer}(\mathtt{s},\mathtt{x}) \wedge \mathtt{hasValue}(\mathtt{s},\mathtt{x})\\
&\forall \mathtt{c}_1, \mathtt{c}_2, \mathtt{s}.\mathtt{hasContainer}(\mathtt{s},\mathtt{c}_1) \wedge \mathtt{hasContainer}(\mathtt{s},\mathtt{c}_2) \Rightarrow \mathtt{c}_1 = \mathtt{c}_2\\
&\forall \mathtt{v}_1, \mathtt{v}_2, \mathtt{s}.\mathtt{hasValue}(\mathtt{s},\mathtt{v}_1) \wedge \mathtt{hasValue}(\mathtt{s},\mathtt{v}_2) \Rightarrow \mathtt{v}_1 = \mathtt{v}_2\\
&\nexists \mathtt{c}, \mathtt{s}.\mathtt{hasSlot}(\mathtt{c},\mathtt{s}) \wedge \mathtt{hasContainer}(\mathtt{s},\mathtt{c})\\
&\forall \mathtt{c} \nexists \mathtt{s}_1 ... \mathtt{s}_n, \mathtt{c}_1 ... \mathtt{c}_n.\mathtt{hasSlot}(\mathtt{c},\mathtt{s}_1) \wedge \mathtt{hasContainer}(\mathtt{s}_1,\mathtt{c}_1) \wedge \cdots\\
&\qquad \cdots \wedge \mathtt{hasSlot}(\mathtt{c}_{n-1},\mathtt{s}_n) \wedge \mathtt{hasContainer}(\mathtt{s}_n,\mathtt{c})\\
&\forall \mathtt{c}, \mathtt{s}_1, \mathtt{s}_2.\mathtt{hasSlot}(\mathtt{s}_1,\mathtt{c}) \wedge \mathtt{hasSlot}(\mathtt{s}_2,\mathtt{c}) \Rightarrow \mathtt{s}_1 = \mathtt{s}_2
\end{aligned}
$$

Terminology. We say that a container $\mathtt{c}$ *contains* a slot $\mathtt{s}$ if $(\mathtt{c},\mathtt{s})$ is in the interpretation of the predicate `hasSlot`. Moreover, we say that a slot $\mathtt{s}$ *holds* a value $\mathtt{v}$ (or a container $\mathtt{c}$) if $(\mathtt{s},\mathtt{v})$ (resp. $(\mathtt{s},\mathtt{c})$) is in the interpretation of `hasValue` (resp. `hasContainer`). In this case we can also say that $\mathtt{v}$ (resp. $\mathtt{c}$) is assigned to a slot. Finally, we say that a container $\mathtt{c}$ *recursively contains* a container $\mathtt{c}'$ if there exists a sequence $\mathtt{s}_0, ..., \mathtt{s}_n$ and containers $\mathtt{c}_1, ..., \mathtt{c}_n$ such that $(\mathtt{c},\mathtt{s}_0), (\mathtt{c}_1,\mathtt{s}_1) ... (\mathtt{c}_n,\mathtt{s}_n)$ is in the interpretation of `hasSlot` and $(\mathtt{s}_0,\mathtt{c}_1), (\mathtt{s}_1,\mathtt{c}_2), ..., (\mathtt{s}_n,\mathtt{c}')$ is in the interpretation of `hasContainer`.

A formal definition of the properties of slots is provided in Listing 5. Values must be assigned to a slot (cf. 5.1). Slots must be contained by a single container (cf. 5.2, 5.3). Slots must hold either a single container or a single value (cf. 5.4, 5.5, 5.7, 5.6). There cannot exists a container c and a slot s such that c contains s and s holds c. (cf. 5.8). Each container cannot recursively contain itself through a sequence of slots and containers (cf. 5.9). Each container can only be held by a single slot (it is not possible for two slots to hold the same container)(cf. 5.10).

Containers can have types (cf. 6.1). Every type must be assigned to a container (cf. 6.2). To this end we introduce the unary predicate `Type` and the binary predicate `hasType`. A formal definition of types is provided in Listing 6.

Listing 6 Formal definition of types.

```
1 ∀c, t.hasType(c, t) ⇒ Container(c) ∧ Type(t)
2 ∀t.∃c.Type(t) ⇒ hasType(c, t)
```

The unary predicates Container, Slot, Value, and Type are pairwise disjoint. Disjointness axioms are provided in Listing 7.

Listing 7 Definition of the disjointness relation.

```
1 ∀x.Container(x) ⇒ ¬Slot(x)
2 ∀x.Container(x) ⇒ ¬Value(x)
3 ∀x.Container(x) ⇒ ¬Type(x)
4 ∀x.Slot(x) ⇒ ¬Container(x)
5 ∀x.Slot(x) ⇒ ¬Value(x)
6 ∀x.Slot(x) ⇒ ¬Type(x)
```

```
7 ∀x.Value(x) ⇒ ¬Container(x)
8 ∀x.Value(x) ⇒ ¬Slot(x)
9 ∀x.Value(x) ⇒ ¬Type(x)
10 ∀x.Type(x) ⇒ ¬Container(x)
11 ∀x.Type(x) ⇒ ¬Slot(x)
12 ∀x.Type(x) ⇒ ¬Value(x)
```

For each data source, there exists one and only one Root container (cf. 8.1, 8.7, 8.3). The Root container cannot be assigned to a slot (cf. 8.6). Containers that are not root must be assigned to a slot (cf. 8.8). To identify the root container we introduce the unary predicate `Root` as specialisation of Container (8.5) and the binary predicate `hasRoot` that connects a data source to its root container (cf. 8.2, 8.4). A formal definition of the Root container is provided in Listing 8.

Listing 8 Formal defition of Root container.

```
1 ∀ds.∃r.DataSource(ds) ⇒ hasRoot(ds, r)
2 ∀ds, r.hasRoot(ds, r) ⇒ DataSource(ds) ∧ Root(r)
3 ∀r.∃ds.Root(r) ⇒ hasRoot(ds, r)
4 ∀ds, r.hasRoot(ds, r) ⇒ includesContainer(ds, r)
5 ∀x.Root(x) ⇒ Container(x)
6 ∄r, s.Root(r) ∧ hasContainer(s, r)
7 ∀ds, r1, r2.hasRoot(ds, r1) ∧ hasRoot(ds, r2) ⇒ r1 = r2
8 ∀c∃s.Container(c) ∧ ¬Root(c) ⇒ hasContainer(s, c)
```

In Example 1, a container is generated for each row of the file. These containers are held by slots in a root container, which represents the entire table. A value is generated for each cell and associated with the corresponding row via a slot. An excerpt of this is shown in the Example 4.

4.3 Properties of the Façade-X

This section outlines some properties of the Façade-X model that will be used later to characterise the RDF knowledge graph constructed from the instances of the model.

▶ **Proposition 1** (Connectedness of containers). *Every non-root container is recursively contained by the root container.*

Proof. For each non-root container c, either **(i)** c is not assigned to any slot; **(ii)** or c is assigned to a slot. We can exclude (i) as by definition (8.8[17]) a non-root container must be assigned to a slot. As for (ii), we show that for each non-root container c there always exist a sequence of slots $s_r, s_1, ..., s_n$ and containers $r, c_1, ..., c_n$ such that $(r, s_r), (c_1, s_1)...(c_n, s_n)$ is in the interpretation of hasSlot and $(s_r, c_1), (s_1, c_2), ..., (s_n, c)$ is in the interpretation of hasContainer, i.e. r recursively contains c. The following conditions imply that each non-root container c must be in a sequence as above: **(a)** each non-root container must be assigned to a slot (8.8); **(b)** each slot is always contained by a container (5.2); **(c)** containers cannot be recursively contain themself (5.9). It is worth noting that (a) and (b) imply that for each non-root there is always a sequence of containers like that above which it belongs to. (c) implies that the sequence does not form a cycle. Therefore, the first container in the sequence must be the root container, since it cannot be assigned to any slot. ◀

▶ **Proposition 2.** *For each data source ds, there is one and only one sequence of containers and slots that connects the root of ds to its containers.*

Proof. This follows from the following facts: **(i)** There is always a single root for each data source (8.7); **(ii)** Every non-root container is recursively contained by the root container (1); **(iii)** Each slot is assigned to one and only one container (5.3); **(iv)** Each non-root container is assigned to one and only one slot (5.6). Therefore, (ii) implies that there is always a sequence of slots and containers that connects a container to a root, (ii) that there is always a single root, (ii) and (iv) guarantee that the sequence is one and only one (it is convenient to pass through the sequence backward, the last container of the sequence can be assigned to one and only one slot, which in turn can be assigned to one and only container and so on). ◀

Now that we have described the Façade-X model and demonstrated its properties, we introduce a function fx, called *façadify*. This function is required to convert a resource into an instance of the Façade-X model. Let $\mathfrak{I}$ be the set all possible interpretations and $\mathcal{R}$ the set of all possible resources, the function fx, called *façadify*, given a resource r returns an interpretation $\mathfrak{I}$ that complies with the Façade-X model, i.e. $fx : \mathcal{R} \to \mathfrak{I}$.

Summary of changes from [8]:

- We remove the predicates Key(k), hasKey(s,k), StringKey(k), and NumberKey(k), i.e. collapse them with the definition of slots. We add StringSlot and NumberSlot as a replacement. Without loss of generality, we imagine that the key of the slot is included in the identifier of the individual.
- We rename Class into Type.
- We solve an ambiguity regarding the distinction between data sources and root containers, that are now distinct. We add the predicate hasRoot(ds, r).
- We remove Name and hasName that were used to assign a name to a DataSource (we assume that the identifier of the data sources are unique).
- We clarify disjointness and connectedness.
- We provide formal definitions for the predicates DataSource, Resource and for integrity constraints that were previously only defined informally.

▶ Example 2. Table 4 show an example of usage of the introduced predicates for interpreting a CSV file.

[17] With this notation we refer to the line 8 of the definition 8.

Table 4 An example of usage of the introduced predicates for interpreting a CSV file[18].

Source: `ex.csv`	Façade-X Interpretation

name,surname
Laura,Grey
Craig,Johnson
	`includes(file:///ex.csv,ex.csv),`
	`hasRoot(ex.csv,root),hasSlot(root,rs.1),`
	`hasSlot(root,rs.2),hasSlot(root,rs.3),`
	`NumberSlot(rs.1),NumberSlot(rs.2),NumberSlot(rs.3),`
	`hasContainer(rs.1,row:1),hasContainer(rs.2,row.2)`
	`hasContainer(rs.3,row.3),`
	`hasSlot(row.1,rsr:11),hasSlot(row.1,rsr.12),`
	`hasSlot(row.2,rsr.21),hasSlot(row:2,rsr.22),`
	`hasSlot(row.3,rsr.31),hasSlot(row.3,rsr.32),`
	`hasValue(rsr.11,"name"),hasValue(rsr.12,"surname"),`
	`hasValue(rsr.21,"Laura"),hasValue(rsr.22,"Grey"),`
	`hasValue(rsr.31,"Craig"),hasValue(rsr.32,"Jonhnson")`
	`NumberSlot(rsr.11),NumberSlot(rsr.12),NumberSlot(rsr.21),`
	`NumberSlot(rsr.22),NumberSlot(rsr.31),NumberSlot(rsr.32)`

▶ Example 3. Table 5 show an example of usage of the introduced predicates for interpreting a JSON document.

Table 5 An example of usage of the introduced predicates for interpreting a JSON file.

Source: `ex.json`	Façade-X Interpretation

`{`
`"a":1,`
`"b":[1,2,3]`
`}`
	`includes(file:///ex.json,ex.json),`
	`hasRoot(ex.json,root),`
	`hasSlot(root,rs.a),StringSlot(rs.a),hasValue(rs.a,1),`
	`hasSlot(root,rs.b),StringSlot(rs.b),hasContainer(rs.b,rsb.c),`
	`hasSlot(rsb.c,rsbc.1),hasSlot(rsb.c,rsbc.2),hasSlot(rsb.c,rsbc.3),`
	`NumberSlot(rsbc.1),NumberSlot(rsbc.2),NumberSlot(rsbc.3),`
	`hasValue(rsbc.1,1),hasValue(rsbc.2,2),hasValue(rsbc.3,3)`

▶ Example 4. Table 6 show an example of usage of the introduced predicates for interpreting a XML file.

5 Façade-X RDF graphs

5.1 Mapping Façade-X to RDF

We define a mapping to RDF as a function $\mathcal{F} : \mathcal{L} \times \mathfrak{I} \to 2^{\mathcal{I} \times \mathcal{T}}$ which takes as input a query q over Façade-X and an interpretation $\mathcal{I}$ of the model, it evaluates q over $\mathcal{I}$ and it returns a set of quads[19]. The function $\mathcal{F}$ uses the assignments of the free variables of q to instantiate the quads.

[18] The example shows the CSV as a single container including a list of lists. SPARQL Anything allows to interpret the first line as a schema and generate an alternative representation, using column names as `StringSlot`s. However, in this work, we do not consider format-specific configurations. More details at: `https://sparql-anything.readthedocs.io/stable/formats/CSV/`.

[19] 2^A denotes the powerset of a set A, i.e. the set of all possible subsets of A.

Table 6 An example of usage of the introduced predicates for interpreting a XML file.

Source: `ex.xml`	Façade-X Interpretation

`<TEAM name="Chicago Bulls">`
`  <PLAYER name="Micheal Jordan"/>`
`</TEAM>`
	`includes(file:///ex.xml,ex.xml),`
	`hasRoot(ex.xml,root),hasType(root,TEAM),`
	`hasSlot(root,r.name),StringSlot(rname),`
	`hasValue(rname,"Chicago Bulls"),`
	`hasSlot(root,r.1),NumberSlot(r.1),`
	`hasContainer(r.1,r1c),hasType(r1c,PLAYER),`
	`hasSlot(r1c,r1c.name),StringSlot(r1c.name),`
	`hasValue(r1c.name,"Micheal Jordan")`

Listing 9 Definition of Façade-X namespaces.

```
1 prefix fx: <http://sparql.xyz/facade-x/ns/>
2 prefix xyz: <http://sparql.xyz/facade-x/data/>
```

The input of $\mathcal{F}$ is specified as a PL formula and the output is specified as a RDF graph template that include the free variable to instantiate the RDF terms. We use the symbol $\rightleftharpoons$ to indicate an input-output pair of $\mathcal{F}$. We note that such mappings are bidirectional and conversion is loss-less, meaning that one can go back and forth without loosing information.

It is worth noting that even though the purpose of mapping is different, formalising the mapping as a function that associates queries and data sources with RDF graphs is compatible with the framework proposed in [37] which associates queries with the positions of graphs to be generated. However, demonstrating the compatibility of the two theories is beyond the scope of this article.

Before providing the details of the function $\mathcal{F}$ we introduce elements that will be used in the transformation. We define two namespaces and preferred prefix names: `fx` and `xyz` (see Listing 9).

We define the mappings (see Listing 10) between Façade-X terms and RDF. These mappings are the input-output (i.e. query, RDF template) pairs of $\mathcal{F}$. We use the functions `IRI`, `Entity` and `Literal` to construct RDF terms on the basis of the arguments passed to the function. Depending from the configuration given by the KG engineer, the `Entity` constructs either an IRI or a Blank Node. Moreover, we introduce the function `ContainerMembershipProperty` that given $n \in \mathbb{N}^+$ constructs the IRI of the corresponding container membership property, i.e. `rdf:_n`. We introduce the primitive IRI `fx:root` to represent the unary predicate `Root`. Without loss of generality, we assume that **(i)** `ContainerMembershipProperty` can extract the cardinal order of the slot from its identifier; **(ii)** `IRI` extract the name of the slot from its identifier.

▶ **Corollary 3.** *The mapping rules of Definition 10 are bidirectional.*

Proof. This is can be demonstrated by showing that the mapping of Definition 10 is a bijective, i.e. a one-to-one correspondence of distinct elements. By definition the mapping rules define a correspondence of queries and generated quads. Therefore, we have only to show that the corresponding elements are distinct. Firstly, we note that queries and generated quads are distinct entities (i.e. it is not possible to convert one into the other using implications). As queries, we note that in each query (implicitly) involves distinct elements of the model: 1: Root; 2: Container and StringSlot; 3: Container and NumberSlot; 4: Value and StringSlot; 5: Value and NumberSlot;

Listing 10 Defintion of mappings of the Façade-X to RDF.

$$
\begin{aligned}
&1\ \texttt{hasRoot}(\texttt{ds},\texttt{c}) \rightleftharpoons \{(\texttt{IRI}(\texttt{ds}),\texttt{Entity}(\texttt{ds},\texttt{c}),\texttt{rdf:type},\texttt{fx:root})\}\\
&2\ \texttt{includesContainer}(\texttt{ds},\texttt{c}_1) \wedge \texttt{includesContainer}(\texttt{ds},\texttt{c}_2)\\
&\qquad \wedge\texttt{StringSlot}(\texttt{s}) \wedge \texttt{hasSlot}(\texttt{c}_1,\texttt{s}) \wedge \texttt{hasContainer}(\texttt{s},\texttt{c}_2) \rightleftharpoons\\
&\qquad \{(\texttt{IRI}(\texttt{ds}),\texttt{Entity}(\texttt{ds},\texttt{c}_1),\texttt{IRI}(\texttt{xyz:},\texttt{s}),\texttt{Entity}(\texttt{ds},\texttt{c}_2))\}\\
&3\ \texttt{includesContainer}(\texttt{ds},\texttt{c}_1) \wedge \texttt{includesContainer}(\texttt{ds},\texttt{c}_2)\\
&\qquad \wedge\texttt{NumberSlot}(\texttt{s}) \wedge \texttt{hasSlot}(\texttt{c}_1,\texttt{s}) \wedge \texttt{hasContainer}(\texttt{s},\texttt{c}_2) \rightleftharpoons\\
&\qquad \{(\texttt{IRI}(\texttt{ds}),\texttt{Entity}(\texttt{ds},\texttt{c}_1),\texttt{ContainerMembershipProperty}(\texttt{s}),\texttt{Entity}(\texttt{ds},\texttt{c}_2))\}\\
&4\ \texttt{includesContainer}(\texttt{ds},\texttt{c})\wedge\texttt{NumberSlot}(\texttt{s}) \wedge \texttt{hasSlot}(\texttt{c},\texttt{s}) \wedge \texttt{hasValue}(\texttt{s},\texttt{v}) \rightleftharpoons\\
&\qquad \{(\texttt{IRI}(\texttt{ds}),\texttt{Entity}(\texttt{ds},\texttt{c}),\texttt{ContainerMembershipProperty}(\texttt{s}),\texttt{Literal}(\texttt{v}))\}\\
&5\ \texttt{includesContainer}(\texttt{ds},\texttt{c})\wedge\texttt{StringSlot}(\texttt{s}) \wedge \texttt{hasSlot}(\texttt{c},\texttt{s}) \wedge \texttt{hasValue}(\texttt{s},\texttt{v}) \rightleftharpoons\\
&\qquad \{(\texttt{IRI}(\texttt{ds}),\texttt{Entity}(\texttt{ds},\texttt{c}),\texttt{IRI}(\texttt{xyz:},\texttt{s}),\texttt{Literal}(\texttt{v}))\}\\
&6\ \wedge\texttt{includesContainer}(\texttt{ds},\texttt{c})\wedge\texttt{hasType}(\texttt{c},\texttt{t}) \rightleftharpoons\\
&\qquad \{(\texttt{IRI}(\texttt{ds}),\texttt{Entity}(\texttt{ds},\texttt{c}),\texttt{rdf:type},\texttt{IRI}(\texttt{xyz:},\texttt{t}))\}
\end{aligned}
$$

Listing 11 The result of the application of the mapping to the input of Example 1.

```
PREFIX fx:      <http://sparql.xyz/facade-x/ns/>
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

[ rdf:type  fx:root;
  rdf:_1    [ rdf:_1  "email"; rdf:_2  "name"; rdf:_3  "surname" ];
  rdf:_2    [ rdf:_1  "laura@example.com"; rdf:_2  "Laura";
              rdf:_3  "Grey" ];
  rdf:_3    [ rdf:_1  "craig@example.com"; rdf:_2  "Craig";
              rdf:_3  "Johnson" ];
  rdf:_4    [ rdf:_1  "mary@example.com"; rdf:_2  "Mary";
              rdf:_3  "Jenkins" ];
  rdf:_5    [ rdf:_1  "jamie@example.com"; rdf:_2  "Jamie";
              rdf:_3  "Smith" ]
] .
```

and 6: Type. In a similar fashion, this is also the case for the generated quads. While the first two elements are always an IRI derived from the DataSource and a Container, the property and the object are distinct: 1: `rdf:type`, `fx:root`; 2: `xyz:` property and Entity; 3: container membership property and Entity; 4: container membership property and Literal; 3: `xyz:` property and Literal; and 6: `rdf:type` and `xyz:` IRI. ◀

Listing 11 shows the result of applying the mapping rules in 10 to the input of Example 1. Considering the mapping to RDF defined above, we can show that the constructed quads meet a series of conditions.

▶ **Corollary 4.** *The following statements apply to the constructed RDF quads:*

(i) *The subject is*[20] *a* ***Container****;*

(ii) *If the property is not* `rdf:type`*, then it is a* ***Slot****;*

[20] "is" in this context means "is constructed on the basis of".

(iii) *If the property is* `rdf:type` *and the object is not* `fx:root`*, then the object is a* ***Type****;*
(iv) *If the property is a container membership property, then the property is a* ***NumberSlot****;*
(v) *If the object is a Literal, then the property is a* ***Slot*** *and the object from a* ***Value****.*

Proof. The statement (i) is demonstrated by observing that the subjects in the mapping rules of the Definition 10 is always an Entity constructed on the basis of a variable bound to a Container.

The statement (ii) is demonstrated by observing that all the rules that do not produce quads having `rdf:type` as property, namely 2, 3, 4 and 5, have as property an Entity or a container membership property derived from a slot.

The statement (iii) is demonstrated by observing that the only mapping rule that produces a quad having `rdf:type` as property and an object different from `fx:root` is the 6. The quad generated by this rule has as object an Entity derived from a type t.

The statement (iv) is demonstrated by observing that the only mappings comply with this condition are the 3 and 4 and in both rules the predicate is derived from a **NumberSlot**.

The statement (v) is demonstrated by observing that the only mappings comply with this condition are the 4 and 5 and in both rules the predicate is derived from a **Slot** and the object from a **Value**. ◀

Considering the implications demonstrated in the Corollary 4, the mapping to RDF provided in the Definition 10 and the Façade-X described in Section 4.2, we can show that further implications hold. Note that these implications apply to the constructed quads as well as to the facts implied by the statements in Corollary 4 (this is why they are provided separately).

▶ **Corollary 5.** *The following statements hold:*

(i) *If the object is an IRI and the property is a* ***Slot****, then the object is a* ***Container****;*
(ii) *If the object is a* ***Container****, then the property is a* ***Slot****;*
(iii) *If the object is a* ***Value****, then the property is a* ***Slot****.*

Proof. The statement (i) is demonstrated by observing that the only mapping rules that generate a IRI as object (via the function Entity) are the 2 and 4. In both, the predicate of the generated quad is a Slot.

The statement (ii) is demonstrated by observing that the only mapping rules that generate a Container as object are the 2 and 3. In both, the predicate of the generated quad is a Slot.

The statement (iii) is demonstrated by observing that the only mapping rules that generate a Value as object are the 3 and 4. In both, the predicate of the generated quad is a Slot. ◀

Finally, on the basis of the corollaries and the definitions above, we can show that further implications can demonstrated on the triple patterns satisfiable on the Façade-X graphs.

▶ **Corollary 6.** *The following conditions hold:*

(i) *If the object is* `fx:root`*, then the subject is derived from the* ***Root*** *and the property is either* `rdf:type` *or a variable that can match only with* `rdf:type`*, we call such a term* ***TypeProperty****;*
(ii) *If the object is a* ***Type****, then the property is a* ***TypeProperty****;*
(iii) *If the property of a triple pattern is neither a container membership property, nor a variable, nor a* ***TypeProperty****, then it must be a* ***StringSlot****.*

Proof. The statement (i) is demonstrated by observing that the only mapping rule in which `fx:root` appears is at line 1 and the subject of this constructed quad is a Container and the property is `rdf:type`. Therefore, any triple pattern that has `fx:root` as object, graph must have a **TypeProperty** as property to be satisfiable on a Façade-X graph.

Similarly, regarding the statement (ii) we can observe that a Type is generated only by the mapping 6 which generates a quad having `rdf:type` as property.

The statement (iii) is demonstrated by observing that if we exclude the mapping rules that generate a container membership property are the 3 and 4 and those generating a quad having `rdf:type` as property are the 1 and 6. Therefore, if we exclude also that the property of the triple is a variable, then the property must be a **StringSlot**. ◀

5.2 Properties of Façade-X RDF graphs

RDF graphs are usually interpreted as *directed*, *labelled* (mathematical) graphs, which we refer to as "m-graphs" to avoid confusion. An m-graph can be constructed from an RDF graph by treating the subjects and objects of the triples as nodes and representing the properties as labelled edges. Considering the properties of the Façade-X model demonstrated in the previous Sections we can derive some topological properties of the corresponding m-graph.

▶ **Theorem 7.** *Façade-X m-graphs built from a Façade-X RDF graphs are connected.*

Proof. We can observe that nodes of a Façade-X m-graph can be either Containers, Values, Types or `fx:root`. Proposition 1 demonstrates that for each container there is always a sequence of slots and containers that connect it the root. Moreover, Containers can be object only of quads having a Slot as property (see 5.(ii)), hence a sequence of Containers and Slots in the Façade-X model corresponds a path of Container nodes and Slot edges. Therefore, there is always a path that connects the Root node to any Container node. Finally, we can observe that Types, Values and `fx:root` are always directly connected to the Container they belong to (see 10.1, 10.4, 10.5 and 10.6). Hence, the Façade-X m-graphs are connected. ◀

▶ **Theorem 8.** *Façade-X m-graphs built from a Façade-X RDF graphs are acyclic.*

Proof. A cycle involving a container node is not possible, as this would imply that a container contains itself recursively, which would invalidate condition 5.9. Types, `fx:root` and Values are directly connected to the container they belong to only and can not have outgoing edges. Therefore, Façade-X m-graphs are acyclic. ◀

▶ **Theorem 9.** *Façade-X m-graphs have a single root.*

Proof. The only mapping rule matching the root container is the 10.1. For each DataSource there is a single solution for the query of this rule since there is one and only one root for each DataSource, see 8.3 and 8.7. ◀

▶ **Theorem 10.** *In Façade-X m-graphs, there always exist a single directed path connecting the root node to any container node.*

Proof. Proposition 2 demonstrates that for each Container c there is always a single sequence of containers and slots that connects the root of the data source to c. Proof of Theorem 7 demonstrates that a sequence of Containers and Slots in the Façade-X model corresponds a directed path of Container nodes and Slot edges. Therefore, each container is connected to the root of the its data source by one and only one path. ◀

▶ **Corollary 11.** *In Façade-X m-graphs, there is either a single directed path connecting any pair of container nodes, or there is none.*

Proof. This is a consequence of the Theorem 10. We observe that for a given pair of nodes x and y, either y lies on the path from the root to x, or it does not. ◀

6 Satisfiability of Basic Graph Patterns under Façade-X

We characterise the Basic Graph Patterns that admits a solution on a Façade-X RDF graph (BGP^{FX}). Although the number of satisfiable BGP^{FX} is infinite, we can characterise a BGP^{FX} from the Triple Patterns (TPs) that admit a solution on a Façade-X RDF Graph and the possible joins among them. Moreover, although the Façade-X model include their definition, Resource and DataSource are not involved in BGP expressions. In fact, a DataSource is mapped to a named graph (see Definition 10). A Façade-X Resource is mapped to an RDF Dataset, as intended by [53]. In what follows, we study basic graph patterns.

6.1 Satisfiable Triple Patterns

We remind some properties of a Façade-X RDF graph that can be derived by the theory outlined in the previous Sections.

- The subject must be a Container;
- The set of all possible predicates is the union of TypeProperty, SlotString and SlotNumber;
- The set of all possible objects is the union of Value, Container, FXRoot and Type;
- Container, TypeProperty, SlotString, SlotNumber, Value, Root and Type, of these sets are mutually disjoint by definition.

Given these conditions, we can have only 6 TPs compatible with a Façade-X graph (out of the 12 possible combinations). Table 7 shows the TPs resulting all the 12 possible combinations of Façade-X components, together with an indication of their satisfiability. Consider the Façade-X triple pattern "`Container SlotNumber Value`" (first row of the table). This is satisfiable, since the Façade-X graph can contain such a triple. The last column of the table indicates the SPARQL triple patterns that can match the corresponding Façade-X triple pattern. For example, the SPARQL triple pattern $\mathcal{VIL}$ (e.g. `?s rdf:_1 "some value".`) can match "`Container SlotNumber Value`" as the variable `?s` can be resolved to a container, `rdf:_1` is a `SlotNumber` and `"some value"` is a Literal. The Façade-X triple pattern "`Container SlotNumber Type`" (second row of the table) is not satisfiable as types in Façade-X occur only in triples with a `TypeProperty` as predicate.

6.2 Satisfiable Join conditions

Without loss of generality, we can study the satisfiability of BGP^{FX} consisting of two TPs. In fact, more complex BGPs can be derived by joining BPGs of one or two TPs. Since properties never occur as the subject or object in Façade-X, joins that equate the subject or object of one TP with the predicate of another TP are not permitted. Therefore, the joins $S \bowtie P$ and $P \bowtie O$ will always result in a empty solution. As a result, Façade-X TPs can be involved in four types of joins, namely: $S \bowtie S$, $P \bowtie P$, $S \bowtie O$, and $O \bowtie O$.

However, not all Façade-X TPs can be involved in all of these joins. Consider the TPs `C.SN.V` and `C.SN.C`. The only joins allowed are $S \bowtie S$, $P \bowtie P$, and $S \bowtie O$, but not $O \bowtie O$ since Values and Containers are disjoint.

For example, the following SPARQL BGPs are satisfiable:

- `?s rdf:_1 ?v . ?s rdf:_2 ?c .` which is example of $S \bowtie S$ join;
- `<i1> ?p "some value" . <i2> ?p <i3> .` which is example of $P \bowtie P$ join;
- `<i> rdf:_1 ?c . ?c rdf:_2 "some value" .` which is example of $S \bowtie O$ join;

Moreover, consider the following BGP.

```
?s1 rdf:_1 ?o .
?s2 rdf:_2 ?o .
```

Table 7 Triple Patterns result in all possible combinations of Façade-X components, together with an indication of their satisfiability on a Façade-X RDF graph.

Id	Façade-X Triple Pattern	Satisfiability	Corresponding SPARQL Triple Patterns
C.SN.V	Container SlotNumber Value	✓	$\mathcal{VVV}$, $\mathcal{VVB}$, $\mathcal{VVL}$, $\mathcal{VIV}$, $\mathcal{VIB}$, $\mathcal{VIL}$, $\mathcal{IVV}$, $\mathcal{IVB}$, $\mathcal{IVL}$, $\mathcal{IIV}$, $\mathcal{IIB}$, $\mathcal{IIL}$, $\mathcal{BVV}$, $\mathcal{BVB}$, $\mathcal{BVL}$, $\mathcal{BIV}$, $\mathcal{BIB}$, $\mathcal{BIL}$
C.SN.T	Container SlotNumber Type	–	–
C.SN.R	Container SlotNumber FXRoot	–	–
C.SN.C	Container SlotNumber Container	✓	$\mathcal{VVV}$, $\mathcal{VVI}$, $\mathcal{VVB}$, $\mathcal{VIV}$, $\mathcal{VII}$, $\mathcal{VIB}$, $\mathcal{IVV}$, $\mathcal{IVI}$, $\mathcal{IVB}$, $\mathcal{IIV}$, $\mathcal{III}$, $\mathcal{IIB}$, $\mathcal{BVV}$, $\mathcal{BVI}$, $\mathcal{BVB}$, $\mathcal{BIV}$, $\mathcal{BII}$, $\mathcal{BIB}$
C.SS.V	Container SlotString Value	✓	$\mathcal{VVV}$, $\mathcal{VVB}$, $\mathcal{VVL}$, $\mathcal{VIV}$, $\mathcal{VIB}$, $\mathcal{VIL}$, $\mathcal{IVV}$, $\mathcal{IVB}$, $\mathcal{IVL}$, $\mathcal{IIV}$, $\mathcal{IIB}$, $\mathcal{IIL}$, $\mathcal{BVV}$, $\mathcal{BVB}$, $\mathcal{BVL}$, $\mathcal{BIV}$, $\mathcal{BIB}$, $\mathcal{BIL}$
C.SS.T	Container SlotString Type	–	–
C.SS.R	Container SlotString FXRoot	–	–
C.SS.C	Container SlotString Container	✓	$\mathcal{VVV}$, $\mathcal{VVI}$, $\mathcal{VVB}$, $\mathcal{VIV}$, $\mathcal{VII}$, $\mathcal{VIB}$, $\mathcal{IVV}$, $\mathcal{IVI}$, $\mathcal{IVB}$, $\mathcal{IIV}$, $\mathcal{III}$, $\mathcal{IIB}$, $\mathcal{BVV}$, $\mathcal{BVI}$, $\mathcal{BVB}$, $\mathcal{BIV}$, $\mathcal{BII}$, $\mathcal{BIB}$
C.TP.V	Container TypeProperty Value	–	–
C.TP.T	Container TypeProperty Type	✓	$\mathcal{VVV}$, $\mathcal{VVI}$, $\mathcal{VVB}$, $\mathcal{VIV}$, $\mathcal{VII}$, $\mathcal{VIB}$, $\mathcal{IVV}$, $\mathcal{IVI}$, $\mathcal{IVB}$, $\mathcal{IIV}$, $\mathcal{III}$, $\mathcal{IIB}$, $\mathcal{BVV}$, $\mathcal{BVI}$, $\mathcal{BVB}$, $\mathcal{BIV}$, $\mathcal{BII}$, $\mathcal{BIB}$
C.TP.R	Container TypeProperty FXRoot	✓	$\mathcal{VVV}$, $\mathcal{VVI}$, $\mathcal{VVB}$, $\mathcal{VIV}$, $\mathcal{VII}$, $\mathcal{VIB}$, $\mathcal{IVV}$, $\mathcal{IVI}$, $\mathcal{IVB}$, $\mathcal{IIV}$, $\mathcal{III}$, $\mathcal{IIB}$, $\mathcal{BVV}$, $\mathcal{BVI}$, $\mathcal{BVB}$, $\mathcal{BIV}$, $\mathcal{BII}$, $\mathcal{BIB}$
C.TP.C	Container TypeProperty Container	–	–

We have that either `?o` is a value or it is a container; it cannot be both, as these are mutually exclusive. Therefore, if other constraints from different TPs require that `?o` be both a container and a value, the pattern is not satisfiable. Moreover, we will see that `?o` can only be a `Value`.

In general, assuming the order of the triple pattern is meaningful, we have 6 valid Façade-X TPs, hence 36 Façade-X basic graph patterns of two joined Façade-X TPs. If we do not consider the order of the TPs, there are 21 possible Façade-X graph patterns[21]. The join condition among these patterns can be one of the 4 above, resulting in 84 combinations.

To narrow down the number of satisfiable combinations, we can make the following considerations.

- $S \bowtie S$ is always satisfiable as it equates containers (see 4.(i));
- $S \bowtie O$ is satisfiable when the joined node of the first and the joined node of the second Façade-X triple pattern are of the same Façade-X type (otherwise the graph would not have a solution as the Façade-X types are pairwise disjoint, see Definition 7).

As a result, we have 42 possible Façade-X graph patterns, which are showed in Table 8.

[21] This is the number of all possible combinations without repetition of 2 elements from a set of 6, i.e. $\binom{6}{2}$, plus the number of Façade-X graph patterns where the same pattern is repeated twice, i.e. 6.

Table 8 The list of all possible joins of two Façade-X triple patterns with their corresponding admissible joining condition.

Façade-X Graph Pattern	Satisfiable Joins
`C.SN.V` $\bowtie$ `C.SN.V`	$S \bowtie S$ $P \bowtie P$ $O \bowtie O$
`C.SN.V` $\bowtie$ `C.SS.V`	$S \bowtie S$ $O \bowtie O$
`C.SN.V` $\bowtie$ `C.SS.C`	$S \bowtie S$ $S \bowtie O$
`C.SN.V` $\bowtie$ `C.TP.T`	$S \bowtie S$
`C.SN.V` $\bowtie$ `C.TP.R`	$S \bowtie S$
`C.SN.C` $\bowtie$ `C.SN.V`	$S \bowtie S$ $P \bowtie P$
`C.SN.C` $\bowtie$ `C.SN.C`	$S \bowtie S$ $P \bowtie P$ $S \bowtie O$ $O \bowtie O$
`C.SN.C` $\bowtie$ `C.SS.V`	$S \bowtie S$
`C.SN.C` $\bowtie$ `C.SS.C`	$S \bowtie S$ $S \bowtie O$ $O \bowtie O$
`C.SN.C` $\bowtie$ `C.TP.T`	$S \bowtie S$
`C.SN.C` $\bowtie$ `C.TP.R`	$S \bowtie S$
`C.SS.V` $\bowtie$ `C.SS.V`	$S \bowtie S$ $P \bowtie P$ $O \bowtie O$
`C.SS.V` $\bowtie$ `C.TP.T`	$S \bowtie S$
`C.SS.V` $\bowtie$ `C.TP.R`	$S \bowtie S$
`C.SS.C` $\bowtie$ `C.SS.V`	$S \bowtie S$ $P \bowtie P$
`C.SS.C` $\bowtie$ `C.SS.C`	$S \bowtie S$ $P \bowtie P$ $S \bowtie O$ $O \bowtie O$
`C.SS.C` $\bowtie$ `C.TP.T`	$S \bowtie S$
`C.SS.C` $\bowtie$ `C.TP.R`	$S \bowtie S$
`C.TP.T` $\bowtie$ `C.TP.T`	$S \bowtie S$ $P \bowtie P$ $O \bowtie O$
`C.TP.R` $\bowtie$ `C.TP.T`	$S \bowtie S$ $P \bowtie P$
`C.TP.R` $\bowtie$ `C.TP.R`	$S \bowtie S$ $P \bowtie P$ $O \bowtie O$

6.3 Satisfiable Basic Graph Patterns

Combining these patterns yields to more complex graph structures as those reported in Table 3. It is worth noting that we can consider a BGP as a conjunction of its sub-BGPs, if a sub-BGP violates the Façade-X model, then the BGP is unsatisfiable. Therefore, given a BGP, we could extract every possible sub-BGP from it and check whether any of these violate the Façade-X model. If we find a sub-BGP that violates Façade-X, then the BGP is unsatisfiable.

We observe that there are formulas of the model that can be violated without compromising the satisfiability of the BGP. In fact, a BGP is satisfiable if there is a *partial* mapping for its variables under which the BGP is isomorphic with a *portion* of the Façade-X RDF graph. Therefore, formulas requiring the existence of triples can be violated without compromising the satisfiability of the BGP, because the mandatory triples could be in a different portion of the graph. For example, the formula 5.1 implies that every value must be contained in a slot. However, BGPs that do not involve slots or values (e.g. `?s a ?t`) can violate formula 5.1 while still being satisfiable.

This approach, i.e. analysing the BGP by its sub components to assess the satisfiability, raises three questions:

(i) Which formulas, if violated, would make a BGP unsatisfiable?

(ii) How can we check whether a (sub-)BGP violates a model formula?

(iii) Which (sub-)BGPs do we need to extract?

Answer (i). We identify the categories for the model formulas: *type/disjoint implications* which are the formulas that allow us to reason about the types of the individuals; *min-cardinality constraints* (resp. *max-cardinality constraints*) which are the formulas that require that there

Table 9 Classification of the Façade-X model formulas by the categories identified: Min-Cardinality Constraint (Min), Max-Cardinality Constraint (Max), Type Implications (Type), Disjointness (Dis), Integrity Constraints (Int), BGP Inviolable (Inv) and BGP Inviolable By Construction of the Mapping Rules (Inv BC).

	Definition 3				Definition 4										Definition 5									
Form	1	2	3	4	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
Min		✓	✓										✓		✓	✓		✓						
Max				✓										✓			✓			✓	✓			✓
Type	✓				✓	✓	✓	✓	✓		✓	✓												
Disj																								
Int										✓									✓			✓	✓	
BGP Inv					✓														✓	✓	✓	✓	✓	✓
BGP Inv BC						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓						

	Definition 6		Definition 7											Definition 8							
Form	1	2	1	2	3	4	5	6	7	8	9	10	11	1	2	3	4	5	6	7	8
Min														✓		✓					✓
Max																				✓	
Type	✓	✓													✓		✓	✓			
Disj			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							
Int																			✓		
BGP Inv					✓							✓							✓	✓	
BGP Inv BC	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓			

must be a minimum (resp. maximum) of individuals/pairs in the interpretation of a model; and *integrity constraints* which are the formulas that impose that one or multiple individuals/pairs do not exist.

It is worth noticing that min-cardinality constraints can be violated without compromising the satisfiability of the BGP. These formulas imply the (implicit) existence of triples that can exist in a different portion of the Façade-X RDF graph. Instead, violating a maximum cardinality constraint, an integrity constraint, a type/disjoint implications renders a BGP unsatisfiable, since these constraints prevent triples from existing, whereas BGP requires them to exist. These call these kind of formulas *inviolable*.

Moreover, we note that there are formulas that, by definition of the mapping rules *alone* (see 10), can not be violated. For example, formula 4.6 can not be violated as NumberSlot and StringSlot are always generated by distinct rules. Instead, for formula 4.1 to be violated, we need to know that StringSlots or NumberSlots are both Slots, which is knowledge that cannot be derived from the mappings alone. Finally, it is worth noting that type implications themselves can only lead to the unsatisfiability of the BGP when considered alongside a disjoint implication.

Table 9 classifies the model formulas according to the above-defined categories and indicates wether the formula is inviolable or inviolable by construction.

Answer (ii). A BGP violates a model formula if there is a mapping between its nodes and the formula's variables that renders the formula invalid (i.e. if it makes the negation of the formula true). For example, consider the formula 5.8, the triple pattern `?x ?p ?x` violate it since we can assign to both `c` and `s` the variable `?x`. Then we can reformulate question (ii) as "how can we assign variables of the inviolable formulas to the nodes of the BGP?"

For each *inviolable* formula, we can construct, via the mapping rules in Definition 10, a *proxy graph* that violates the formula. This is the minimal graph that violates a formula and exemplifies the pattern that must be avoided in BGP to ensure satisfiability. Since mapping rules are bidirectional (see Theorem 3) assigning BGP nodes to formula variables is equivalent to assigning BGP nodes to proxy-graph nodes. Therefore, BGPs that are isomorphic to a proxy pattern violate an inviolable formula.

Table 10 Inviolable formulas and corresponding proxy graph.

Formula	Proxy Graph
4.1, 7.3, 7.10	`<i1> xyz:s <i2> . <i3> rdf:type <i1>`
5.5	`<i1> rdf:_1 "a" . <i1> rdf:_1 <i2>`
5.6	`<i1> rdf:_1 <i2> . <i1> rdf:_1 <i3>`
5.7	`<i1> rdf:_1 "1" . <i1> rdf:_1 "2"`
5.8	`<i1> rdf:_1 <i1>`
5.9	`<i1> rdf:_1 <i2> . <i2> rdf:_1 <i3> . <i3> rdf:_1 <i1>`
5.10	`<i1> rdf:_1 <i2> . <i1> rdf:_2 <i2>`
8.6	`<i1> rdf:type fx:root . <i2> rdf:_1 <i1>`
8.7	`<i1> rdf:type fx:root . <i2> rdf:type fx:root`

Table 10 shows the inviolable formulas and the corresponding proxy graphs. For brevity, Table 10 omits disjoint formulas. Such formulas can be violated by forcing a node to be of two disjoint types. Most of the formulas cannot be violated due to the construction of the mapping rules, either because the two types of a disjoint occur in different roles (e.g. slots and containers are placed as properties and subjects) or because they are materialised using different RDF node types (e.g. values and containers are literals and IRIs). These conditions can be validated by checking if the triple patterns of the BGP match the valid triple pattern indicated in Table 7. Therefore, the only disjoint that can be violated is between Type and Container.

▶ Remark 12. In addition to the above conditions, there is an extra constraint that must be met to ensure the BGP is satisfiable. This constraint is treated separately from the others as it implicitly emerges from the structural properties of the Façade-X m-graph. Since there is a single path connecting the root to its containers (see 10) and there is either a single path connecting any pair of container nodes, or there is none (see 11), backward paths of containers to the root must be compatible. For example, the following BGP is not satisfiable since `<iri>` which must be a container, and it is contained by two different containers: `<j1>` and `<j2>` (in case that `?j3=?j5`, `?p3=?p4`, `?p2=?p4`), or, `<j3>` and `<j5>` (if `j3` $\neq$ `j5`):

```
<j1> ?p2 ?j3 . ?j3 ?p3 <iri> .
<j4> ?p4 ?j5 . ?j5 ?p4 <iri> . <iri> ?p ?o .
```

Answer (iii). It is worth noting that, in order to determine whether a BGP violates an inviolable formula, we must first establish whether the BGP contains an invalid FX triple pattern, and secondly, whether it is isomorphic to a proxy graph in Table 10. If we exclude proxy graph associated to formula 5.9 that imposes that Façade-X RDF graphs are acyclic, conditions that would make the BGP unsatisfiable would span across one or two triple patterns. Therefore, assuming that the BGP is acyclic, we can determine its validity by extracting sub-BGPs of one or two triple patterns and checking the above-defined conditions.

7 Detecting satisfiable Basic Graph Patterns under Façade-X

We apply the Façade-X theory defined in the previous sections to determine the satisfiability of basic graph patterns in Façade-X. We first summarize the components of the theory and their role in addressing the problem and then present two algorithms to annotate basic graph patterns and determine their satisfiability.

7.1 Implementing the theory

We will introduce some preliminary definitions and supporting functions that will be used in later algorithms.

Satisfiable BGPs under Façade-X. We define G as the set of all possible RDF graphs, and G^{FX} as the set of possible RDF graphs that derive from the mappings of definition 10. Furthermore, we define BGP[22] as the set of SPARQL Basic Graph patterns and $BGP^{FX} \subset BGP$ as the set of BPGs having at least one solution against at least one possible $g \in G^{FX}$. Note that there is a strict inclusion between BGP^{FX} and BGP as we already demonstrated that not all triple patterns/joins are satisfiable under Façade-X (see Section 6.3).

Any $bgp \in BGP^{FX}$ must comply with the conditions determined by our theory, namely:

C1 No cycles are allowed in BGPs (see Theorem 8);

C2 Join conditions of the BGP must comply with those reported in Table 8;

C3 BGPs cannot have two alternative paths that lead from any node to a container (see remark 12);

C4 Mapping Rules (i.e. 10) and related corollaries 4, 5 and 6;

C5 There must be *at most* one root (see 9);

C6 Triple patterns of the BGP must comply with those reported in Table 7;

C7 None of the sub-BGPs included in bgp must be isomorphic to the proxy graphs defined for the inviolable formulas as reported in the Table 10.

We ignore the following properties of Façade-X RDF graphs:

- Connectedness (cf. 7 and 2): because BGP don't need to, as non-joined triple patterns will be executed as their product;
- Façade-X graphs are rooted (cf. 9): because the BGP can safely match a portion of the Façade-X graph;
- Min-Cardinality constraints and conditions defined in formulas that are inviolable by construction of the mapping rules (cf. Section 6.3).

Next, we introduce functions to avoid unsupported joins, cycles and multiple paths to container nodes. These functions guarantee that the conditions C2, C1 and 12 are satisfied.

We proceed by describing the procedure to remove unsupported joins.

▶ **Algorithm 1** (Detection of unsupported joins). *BGP^{FX} cannot have Subject/Property or Object/Property joins within or across triples. Listing 12 defines the function $hasUnsupportedJoin(bgp) \rightarrow Bool$ to detect unsupported joins.*

Listing 12 Pseudocode of the hasUnsupportedJoin function.

```
1 function hasUnsupportedJoin(bgp): Bool
2     subjects = {}, properties = {}, objects = {}
3     for (s, p, o) ∈ bgp:
4         subjects = subjects ∪ {s}
5         properties = properties ∪ {p}
6         objects = objects ∪ {o}
7         if s ∈ properties ∨ o ∈ properties ∨ p ∈ subjects ∨ p ∈ objects :
8             return True
9     return False
```

[22] We remind that a BGP is a set of triple patterns, then $BGP := 2^{(\mathcal{I}\cup\mathcal{B}\cup\mathcal{V})\times(\mathcal{I}\cup\mathcal{V})\times(\mathcal{I}\cup\mathcal{B}\cup\mathcal{L}\cup\mathcal{V})}$

Time Complexity of `hasUnsupportedJoin`. The time complexity depends on the number of triples in the bgp, hence $O(|bgp|)$.

We proceed to detect cycles in basic graph patterns.

▶ **Algorithm 2** (Detection of cycles). *Since Façade-X graphs are acyclic, BGP^{FX} also cannot have cycles. We define the function $hasCycle(bgp) \rightarrow Bool$ to detect cycles in BGPs. This function explores the bgp by performing a depth-first search (DFS, see `detectCycle`) for each triple pattern of the bgp. If a node is visited two times by the search, the function returns True. hasCycle and detectCycle are defined in Listing 13.*

Listing 13 Pseudocode of functions for detecting cycles.

```
function hasCycle(bgp): Bool
    for (s, p, o) ∈ bgp:
        if(detectCycle (bgp, {s}, s)):
            return True
    return False

function detectCycle(bgp, visited, last): Bool
    for (last, p, next) ∈ bgp:
        if next ∈ visited:
            return True
        else:
            return detectCycle(bgp, visited ∪ {last}, next)
    return False
```

Time Complexity of cycle detection. The time complexity of a depth-first search on a graph of n nodes and e edges is $O(n+e)$. We observe that the function `hasCycle` performs a DFS for each triple of the BGP (number of edges of the graph). Therefore in the worst case the time complexity is $O(|bgp|^2 + |bgp| * n)$.

Once the cycling detection is verified, we move to verify the constraints of our theory. To do that, we proceed to annotate the nodes in the pattern with the predicates resulting from the Façade-X mappings to RDF. We list the possible predicates in Table 11, which also summarises the following characteristics.

Ground predicates. Nodes in BGPs have roles related to the RDF triples (Subject, Predicate, Object) or roles resulting from mappings to the Façade-X model: Container, FXRoot (to represent the role of the term `fx:root`), Slot, SlotString, SlotNumber, Value, Type and TypeProperty. We organise these roles in a predicate hierarchy, where the top role is the RDF Node and the first layer Subject, Predicate, Object, which are in turn specialised by Façade-X predicates. Ultimately, the nodes in BGP^{FX} will necessarily be one of Container (Subject or Object), Type (Object), FXRoot (Object), TypeProperty (Property), SlotString (Slot which specialises Property), SlotNumber (Slot which specialises Property), or Value (Object). We name those *ground predicates* (GP), i.e. the set of Façade-X predicates that cannot be specialised. These are marked with a * in Table 11. Table 11 also summarises the disjointness between predicates (derived from Definition 7 and the analysis of satisfiable joins, see Table 8).

▶ **Definition 13** (Annotated Basic Graph Patterns). *We indicate the set of nodes of a BGP as $\mathcal{N}$, i.e. $\mathcal{N} := \mathcal{V} \cup \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ We define the function* `nodes` : `BGP` $\rightarrow 2^{\mathcal{N}}$ *which given a BGP returns the set of nodes it contains. We define a function* `annotate` : `BGP` $\rightarrow$ `Map`$(\mathcal{N}, \texttt{GP})$[23] *which associates every bgp with a set of pairs (node, ground predicate).*

[23] `Map(X,Y)` indicates the set of all possible dictionaries having elements in X as keys and Y as values.

Table 11 Façade-X RDF predicates hierarchy and disjointness. *Ground predicates* are marked with a *.

Predicate	Specialises	Disjoint with
Subject		Property
Property		Subject, Object
Object		Property
TypeProperty*	Property	Subject, Object, Slot, Type, Container
Type*	Object	Slot, Container, Value, FXRoot
Container*	Subject, Object	Property, Slot, Value, Type, FXRoot
Slot	Property	Subject, Object,TypeProperty
Value*	Object	Property, Subject,Type,Container,FXRoot,Slot
FXRoot*	Object	Subject, Property,Container,Value,Type
SlotNumber*	Slot	SlotString, Subject, Object
SlotString*	Slot	SlotNumber, Subject, Object

Relations between BGP and annotated BPGs. Note that a BGP can be annotated in many ways, i.e. $|annotate(bgp)| \geq 1. \forall bgp \in BGP$ Note that the existence of an annotation for a BGP does not guarantee that it is satisfiable.

Later, we present two algorithms that, given a generic $bgp \in BGP$, can generate all possible annotations that satisfy the theory. In the remaining of this section, we are going to summarise the elements of the theory and introduce pseudo-code that will be later applied in the algorithms.

In what follows, we distinguish the BGP node types (Variable, IRIs, Literal) from the predicates, i.e. the *annotated roles* that each node may have in a Façade-X graph matching scenario. We omit blank nodes, as they operate as existential qualifiers and thus behave like variables do [23].

▶ **Definition 14.** *According to the above theory, the algorithm implements the following rule for annotating a BGP. Note that if a rule raises an inconsistency when verified, the NS term is raised, which stands for "no solution".*

R1 *If a node is a subject, then it is a Container;*

R2 *If a node is the IRI fx:root, then it is FXRoot;*

R3 *If a node is the IRI rdf:type, then it is TypeProperty;*

R4 *If a Property is not a variable or the IRI rdf:type, then it is a Slot;*

R5 *If the Object of a triple is not a Variable and not the IRI fx:root but Property is rdf:type, then the Object is a Type;*

R6 *If the Object is FXRoot, then the Property is TypeProperty;*

R7 *If the Object is Type, then Property is TypeProperty;*

R8 *If the Property is a ContainerMembershipProperty, then the Property is SlotNumber;*

R9 *If the Property is neither a Variable, rdf:type, or ContainerMembershipProperty, then is a SlotString;*

R10 *If a node is a Literal, then it is a Value;*

R11 *If the Object is a IRI and Property is Slot, then Object is a Container;*

R12 *If the Object is a Value, then the Property is Slot;*

R13 *If the Object is an IRI not equal to* `fx:root` *and it is annotated as FXRoot, then raise NS;*

R14 *If the Object is a Type and is* `fx:root`*, then raise NS;*

R15 *If the Object is an IRI and is a Value, then raise NS;*

R16 *If the Object is a Container, then Predicate is Slot;*

R17 *If two triples have the same subject and predicate, and the predicate is a Slot, they also need to have matching objects - where either or are variables or are the same node;*

R18 *If there is a subject/object join when the object is FXRoot, then raise NS;*

R19 *If a node is Object and Container or FXRoot and there are disjoint paths (S⋈O joins) leading to it, then raise NS.*

Data structures used in the algorithms. We introduce the data structures which will be referred to by both algorithms. These are: **(i)** the input Basic Graph Pattern, i.e. a sequence of triples denoted by the variable `bgp`; **(ii)** the set of predicates of Façade-X indicated as `terms`; **(iii)** the Façade-X RDF terms hierarchy, as presented in Table 11, i.e. a map, named `specialisations`, which associates each Façade-X predicate to its direct super type; **(iv)** the Façade-X RDF terms disjointness Table 11, a map, named `disjointness` which associates each Façade-X predicate to the set of predicates it is disjoint with; **(v)** ground terms, i.e. Façade-X set of terms that cannot be specialised, indicated as `groundTerms`; **(vi)** top terms, i.e. terms that do not specialise other terms, indicated as `topTerms`.

Implementation of inference rules. We assume that the consistency rules presented above (see R1-R19) are implemented in form of implications (`head` ← `body`), meaning that the head is applied if the body is true. These rules are accessible through the variable `rules`. We also introduce the functions: **(i)** `body(rule, node, map) : Bool` which takes a rule a node of the BGP and a *map* ∈ `Map`($\mathcal{N}$, `terms`), it evaluates the body of the rule and returns True if the condition holds, False otherwise; **(ii)** `head(rule, node, map) : Bool` which applies the head of a rule (returns the implied FX term or NS).

▶ **Algorithm 3** (Checking constraints from inference rules). *We introduce the function* `check` *(Listing 14), which verifies the consistency of a bgp against the set of rules presented before. The function iterates through the nodes of the bgp, applying each rule in* `rules` *to each node. If the body of the rule is true, then it obtains the inferred term from the head of the rule. If the the inferred term is inconsistent with the other terms already associated with the node, it returns False; otherwise, it continues with the next rule. The function returns True if no inferred term raises an inconsistency.*

Listing 14 Pseudocode of check function.

```
function check(map, bgp): Bool
    for node → term in map:
        for rule in rules:
            if body (rule, node, map) then:
                result ← head(rule, node, map)
                if result = NS or result in disjointness[term]:
                    return False
    return True
```

Time complexity of `check`. The time complexity of the function `check` is $O(n * r)$ where n is the number of nodes in the bgp and r is the number of rules, i.e. R19. Since we can consider r a constant parameter, then we can conclude that `check` is linear in the number of nodes of the bgp.

match(n1,n2). We define a function `match(n1, n2) : Bool` that implements the SPARQL specification (SPARQL Basic Graph Pattern Matching) [23]. The function compares the nodes' types, if any of the two is a variable, returns True. If both aren't, it checks node equality (see Table 12).

Table 12 The *match(left,right)* function compares two nodes.

left	right	matches(left,right)
$\mathcal{V}$	$\mathcal{V}$	True
$\mathcal{V}$	$\mathcal{I}$	True
$\mathcal{V}$	$\mathcal{L}$	True
$\mathcal{I}$	$\mathcal{I}$	=
$\mathcal{I}$	$\mathcal{L}$	False
$\mathcal{L}$	$\mathcal{L}$	=

▶ **Algorithm 4** (Function satisfiesUniquePathToRoot).
We present the function `satisfiesUniquePathToRoot(bgp, map, node):Bool` *(see Listing 15 for the pseudocode) which guarantees that the constraint R19 is satisfied. The function takes as argument (i) bgp - a basic graph pattern; (ii) map* $\in$ `Dict`$(\mathcal{N},$ `terms`$)$ *- nodes with annotations (e.g. '?x -> Slot'); and (iii) a focus node n so that* $(n \rightarrow Container) \in map$ *or* $(n \rightarrow FXRoot) \in map$. *The algorithm operates as follows:*

1. *From bgp, construct the set* `triplePairs` *containing pairs of distinct triples that have* `node` *as object (cf. line 2). This can be an empty set.*
2. *We collect backward triple paths. For each pair of triples so that* $Triple(\_,\_,node)$, *and each triple in the pair, get the lists of triples having recursively* $S \bowtie O$ *joins from the BGP (cf. function* `walkBackwards`*). backwardsTriplePathPairs is a list of tuples, holding two backwards paths for each triple pair.*
3. *Next, for each triple path pair, we transform each triple path in a sequence of backward node paths starting from node (cf. function* `nodesPath`*). backwardsNodePathPairs is a list of tuples, holding two backwards node paths for each pair.*
4. *We walk each paths' pair.*
5. *Paths can have the same length or not. When there is a node in both paths, we check whether they are compatible, ie. we match the nodes in the same position, relying on the match function defined before. If two nodes don't match, the bgp does not satisfy the constraint at R19.*
6. *If length does not match (lines 10-16), we verify that the shortest path of the two ends with a node that is not being annotated as FXRoot. Since both paths must be compatible for the bgp to be satisfiable, if the shortest path ends with a root container, the bgp is not satisfiable (cf. line 26).*
7. *If the algorithm proceeds, all conditions are satisfied: the solution graph does not need to have alternative unmatching paths to a container or root node.*

Time complexity of `satisfiesUniquePathToRoot`. In the worst case, the function **satisfiesUniquePathToRoot** constructs a path pair for each node. Since the dominant part of the function involves going through pairs of paths and checking that each node does not invalidate any conditions, the time complexity of the function is $O(n^2)$.

7.2 Correctness and Completeness with respect to Façade-X

To guarantee the satisfiability of a BGP, it is necessary and sufficient to verify the conditions C1-C7. Here, we summarise how all the conditions are captured by our framework.

C1, C2 and C3. As discussed earlier, C1, C2 and C3 are verified by executing the functions `hasCycle`, `hasUnsupportedJoin`, and `satisfiesUniquePathToRoot`.

Listing 15 Pseudocode of the satisfiesUniquePathToRoot function.

```
function satisfiesUniquePathToRoot(bgp, map, node): Bool
    triplePairs ← {((s1, p1, node), (s2, p2, node)) | (s1, p1, node) ∈ bgp ∧ (s2, p2,
        node) ∈ bgp ∧ (( s1 ≠ s2) ∨ ( p1 ≠ p2))}
    backwardsTriplePathPairs ← {(walkBackwards([left], bpg), walkBackwards([right], bgp))
        | (left, right) ∈ triplePairs}
    backwardsNodePathPairs ← {(nodesPath(left), nodesPath{right) | (left, right) ∈
        backwardsTriplePathPairs}
    for (leftList, rightList) in backwardsNodePathPairs:
        for i ∈ [0···max(|leftList|,|rightList|)]
            if |leftList|=|rightList|
                if match(leftList[i], rightList[i]) = False:
                    return False
            else:
                if i < |leftList|:
                    shortestEndNode = rightList[|rightList|]
                else:
                    shortestEndNode = leftList[|leftList|]
                if (shortestEndNode, _, r) \in bgp ∧ map[r] = FXRoot:
                    return False
    return True

function walkBackwards(list, bgp): List
    (s, p, o) ← list[|list|]
    if (sp, pp, s) ∈ bgp:
        list ← list + [(sp, pp, s)]
        return walkBackwards(list, bgp)
    else:
        return list

function nodesPath(triplePath): List
    path = []
    for (s, p, o) ∈ triplePath:
        if path = []:
            path = [o]
        else: path =
            path + [p, s]
    return path
```

The other conditions (C4, C6, C5, C7) are verified by the `check` function via the inference rules R1-R19, as follows.

C4 and C6. R1-R12 guarantee that the BPG's triple patterns comply with the Façade-X triple patterns and mapping rule:

- R1 implies that all the subject must be a Container;
- R3, R5, R6, R7, R8 and R9 imply that the predicate of a BGP's TP is either a Slot or TypeProperty.
- R5, R10, R11 imply that the object of a TP is a Container or a Type or a Value or FXRoot.

Note that if multiple incompatible predicates are associated with a node, the function `check` raises NS.

So far, satisfying the conditions guarantees that satisfiable Façade-X TPs are identified.

Next, we must demonstrate that unsatisfiable TPs are excluded. To this end, we go through the unsatisfiable patterns of Table 7, to demonstrate that they are excluded by the rules. `C.SN.T`, `C.SN.R`, `C.SS.T`, `C.SS.R` are excluded by the R11, which requires the Object of a Slot to be a Container that is disjoint with Type and FXRoot. `C.TP.V` is excluded by the R12, which

requires the property of a triple having Value as Object to be a Slot (and Slot is disjoint with TypeProperty). `C.TP.C` is excluded by the rule R16, which requires the property of a triple having Container as Object to be a Slot. Thus, we have shown how all unsatisfiable triple patterns are identified by the rules.

C5. Condition C5 is guaranteed by the R2 and R19. In fact, R2 guarantees that the IRI `fx:root` is annotated as FXRoot and R19 guarantees that there can not be multiple path leading to it.

C7. Last, we go through the Proxy Graph reported in Table 10 and we demonstrate that are all excluded by the rules. Proxy Graph for Formulas 4.1, 7.3, 7.10 raises a NS since `<i1>` is annotate as both Container (cf. R1) and Type (cf. R3 and R5) and they are disjoint predicates. 5.5, 5.6, and 5.7 are excluded by R17 since we have two triples have the same subject and predicate, the predicate is a Slot, but the objects do not match. 5.8 and 5.9 are cycles, then excluded by the function `hasCycle`. 5.10 is excluded by the function `satisfiesUniquePathToRoot`. 8.6 is excluded by R18 since it is a subject/object join when the object is FXRoot. Finally, 8.7 guarantees that the BGP has a single root and, as discussed earlier, this is enforced by R2 and R19.

7.3 Annotating graph patterns

We investigate two approaches to annotate basic graph patterns, ie. implementing the function of definition 13: (1) a top-down algorithm, addressing the annotation process as a search, and (2) a bottom-up algorithm, addressing the annotation process as constraint satisfaction one.

7.3.1 Top-down algorithm

A top-down algorithm could address the annotation of the BGP problem as searching the space of annotated patterns for any (or all) annotations that comply with the theory. If none is found, the input *bgp* is not satisfiable. Clearly, it would be sufficient to find one viable solution for the BGP to be satisfiable. However, the implementation allows for finding all possible valid annotated BGPs (by setting the argument complete as True).

▶ **Algorithm 5** (Top-down algorithm: annotation as search)**.** *The algorithm (whose pseudocode is provided in Listing 16) operates as follows:*

1. *It initialise an initial annotation start* $\in Map(\mathcal{N}, terms)$*, where each node is associated with its position in the triple it occurs in. For instance, given the bgp* {?a ?b ?c . ?c ?d ?e}*, it would generate the map* {(?a → *Subject*) (?b → *Property*) (?c → *Object*) (?d → *Property*) (?e → *Object*)}*;*
2. *Next, the recursive function* `search` *is called (cf. line 3);*
3. *For each node in the map, we first lookup for possible specialisations, and generate a copy of the map with a modified annotation (cf. line 9);*
4. *Next, we verify that the set of annotations is consistent with the constraints defined in 14 (cf. line 10);*
5. *If the bgp is consistent, and only ground predicates are mentioned in the annotation, the bgp is satisfiable. The current valid map is added to the solutions (cf. line 12). If complete is False, the solution is returned and the process interrupted, otherwise, we keep generating alternative solutions (cf. lines 6-7);*
6. *If the bgp is consistent but some annotations can still be specialised, we keep searching within the current space (cf. line 16).*

Listing 16 Pseudocode of annotate_topdown function.

```
1 function annotate_topdown(bgp, complete) : List
2    map ← { node → Subject | (node,_,_)∈bgp } ∪ { node → Predicate | (_,node,_)∈bgp }
         ∪ { node → Object | (_,_,node)∈bgp }
3    return search ( map , bgp, {}, complete)
4
5 function search (map , bgp, solutions, complete): List
6    for node → term in map:
7       for (subterm → term) in specialisations:
8          newMap ← copy(map)
9          newMap[node] ← subterm
10         if check(newMap, bgp):
11            if {term | node ← term ∈ newMap} ⊆ GP:
12               solutions ← solutions + [newMap]
13               if complete:
14                  return solutions
15            else:
16               solutions ← solutions + search(newMap, bgp, solutions, complete)
17   return solutions
```

Time complexity of `annotate_topdown`. The dominant part of the algorithm is the search function. This function specialises multiple times the nodes of the bgp and checks the validity of the annotation every time a term is specialised. Therefore, the complexity of the `annotate_topdown` depends on the number of nodes of the bgp (n), the number of specialisation relation in the predicate hierarchy |specialisations| and the complexity of `check`, i.e. $O(n * r)$. Furthermore, if the solution contains predicates other than ground ones, the function specialises the solution further. The process is repeated at most $|terms| - |GP|$ times. Therefore, the complexity of `annotate_topdown` is quadratic in the number of nodes in the bgp and linear in the number of specialisations and inference rules, i.e. $O(n^2 * r * |specialisations| * (|terms| - |GP|))$.

7.3.2 Bottom-up algorithm

The second algorithm uses the same components (inference rules) of the previous one, but operates by first annotating bgp triples with all *grounded predicates* specialising Subject, Property, or Object, and then filters out inconsistent combinations.

▶ **Algorithm 6** (Bottom-up algorithm). *A generate function produces valid annotations of a bgp. When the parameter complete equals to True, it stops when one valid solution is found.*

1. *We extract the list of unique nodes from the bgp (cf. line 2);*
2. *We populate a list with, for each node, the set of ground predicates that could match the nodes' position in any of the triples of the bgp (cf. lines 5-13). Note that $|nodesTerms| = |nodes|$ and for any index i in the list, $nodes[i] \rightarrow nodeTerms[i]$. We assume that $nodes[n]$ where n is a node of the BGP returns the set of ground predicates associated with n.*
3. *We refine the ground predicates by some of the inference rules (cf. Definition 14) whose application conditions are at the triple level (lines 15-22).*
4. *We generate possible annotations in line 16. hypotheses is a set of lists (tuples) of size |nodes|, each one with a possible combinations of ground predicates (line 24).*
5. *In the next steps, we verify each combination against our conditions with the check function (cf. lines 26-32).*
6. *If the argument complete is set to False, the process ends when the first valid solution is found (cf. lines 29-32).*
7. *Finally, all valid solutions are returned, or an empty set, if no solution was found (cf. line 33).*

Listing 17 Pseudocode of the bottom-up alogorithm.

```
1 function generate_bottomup(bgp, complete):
2     nodes ← [{s|(s,_,_)∈bpg} ∪ {p|(_,p,_)∈bpg} ∪ {o|(_,_,o)∈bpg}]
3     nodesTerms ← []
4
5     for node in nodes:
6         t ← {}
7         if (node, p, o) ∈ bgp:
8             t ← { term | term ∈ groundTerms ∧ topTerm(term) = 'Subject' }
9         if (s, node, o) ∈ bgp:
10            t ← t + { term | term ∈ groundTerms ∧ topTerm(term) = 'Predicate' }
11        if (s, p, node) ∈ bgp:
12            t ← t + { term | term ∈ groundTerms ∧ topTerm(term) = 'Object' }
13        nodesTerms ← nodesTerms + t
14
15    for (s, p, o) ∈ bgp:
16        nodes[s] = {'Container'}
17        if p is an IRI:
18            if p ∈ TypeProperty:
19                nodes[p] = {'TypeProperty'}
20                nodes[o] = nodes[o] \ {'Container', 'Value'}
21            else:
22                nodes[o] = nodes[o] \ {'Type', 'Root'}
23
24    hypotheses ← nodeTerms[1] × nodeTerms[2] × ··· × nodeTerms[|nodes|]
25
26    solutions = {}
27    for h ∈ hypotheses:
28        map = {nodes[i] → h[i] |1≤i≤|nodes|}
29        if check(map, bgp):
30            solutions ← solutions ∪ {map}
31            if complete = False:
32                return solutions
33    return solutions
```

Time complexity of `generate_bottomup`. The dominant part of the algorithm is the iteration at lines 27-32. This goes through the hypotheses, which are $n * |GP|$. Then, for each hypothesis, it generates an annotation (map) and checks its validity. Therefore, the time complexity of `generate_bottomup` is $O(n^2 * |GP| * r)$.

8 Experiments

We report on three experiments. The first compares the two algorithms described in Section 7 with a curated set of BGPs of increasing complexity, and provide empirical evidence of their performance. The second experiment focuses on real world queries, and has the objective of testing the relevance and practical feasibility of the approach. The third experiment shows the advantages of checking whether a query can be satisfied before evaluating it, considering the case of a knowledge graph construction benchmark.

8.1 Setting

We prepare a set of BGPs with different characteristics and consider two regimes: (a) find the first satisfiable annotation (ensure satisfiability); or (b) generate all satisfiable annotations (describe solution patterns).

Table 13 List of BGPs used in experiments.

BGP label	Satisfiable?	Triple patterns	Features
N_1T	No	1	only variables, no joins
N_2J	No	2	only variables, with joins
N_2P_R	No	2	multiple paths to fx:root
N_2T	No	2	only variables, no joins
N_3J	No	3	only variables, no joins
N_3P_C	No	3	multiple paths to a container
N_3P_R	No	3	multiple paths to fx:root
N_3T	No	3	only variables, no joins
N_4J	No	3	only variables, with joins
N_4P_C	No	4	multiple paths to a container
N_4T	No	4	only variables, no joins
N_5J	No	5	only variables, with joins
N_5P_C	No	5	multiple paths to a container
N_5T	No	5	only variables, no joins
S_1T	Yes	1	only variables, no joins
S_2J	Yes	2	only variables, with joins
S_2P_R	Yes	2	multiple paths to fx:root
S_2T	Yes	2	only variables, no joins
S_3J	Yes	3	only variables, with joins
S_3P_C	Yes	3	multiple paths to a container
S_3T	Yes	3	only variables, no joins
S_4J	Yes	4	only variables, with joins
S_4P_C	Yes	4	multiple paths to a container
S_4T	Yes	4	only variables, no joins
S_5P_C	Yes	5	multiple paths to a container
S_5T	Yes	5	only variables, no joins

The algorithms are implemented in Java. Reported results were performed on a MacBook Pro with 8-Core Intel Core i9 (2.3 GHz) processor and Java 17 and Maven 3.9. We report on results of ten executions, aggregated with average duration (in milliseconds) and standard deviation. Since the satisfiability assessment will be part of a process performed at query time, we set the upper limit of 5 seconds, after which we stop the execution and consider the algorithm not viable. We leave possible optimisations of the algorithms and experimenting with a broader class of BGPs to future work.

The proof-of-concept implementation and the experimental code is published and experiments can be reproduced [15].

8.2 Comparing two algorithms

We report on experiments with the two algorithms described in Section 7. Both implementations perform all checks described in Section 7 (unsupported joins and cycles), before proceeding with the annotation of the BGPs. With these experiments, we want to demonstrate the practical feasibility of the approach to solving Façade-X BGP satisfiability problem.

Table 14 Results for algorithm Top down (Search): only satisfiability.

name	satisfiable?	found	type	size	ms (avg)	ms (std)	tested (avg)	tested (std)
N_1T	FALSE	0	T	1	1.7	0.67	12	0
N_2J	FALSE	0	J	2	0	0	0	0
N_2P_R	FALSE	0	P	2	71	20.11	1957	0
N_2T	FALSE	0	T	2	478.8	84.63	16364	0
N_3J	FALSE	0	J	3	0.1	0.32	0	0
N_3P_C	FALSE	0	P	3	2446.7	211.66	109601	0
N_3P_R	FALSE	0	P	3	266.7	16.77	12330	0
N_3T	FALSE	-1	T	3	-1	0	-1	0
N_4J	FALSE	0	J	4	0	0	0.9	0.74
N_4P_C	FALSE	-1	P	4	-1	0	-1	0
N_4T	FALSE	-1	T	4	-1	0	-1	0
N_5J	FALSE	0	J	5	0	0	5.4	1.65
N_5P_C	FALSE	-1	P	5	-1	0	-1	0
N_5T	FALSE	-1	T	5	-1	0	-1	0
S_1T	TRUE	1	T	1	0.6	0.52	31	3.2
S_2J	TRUE	1	J	2	0.5	0.53	16.3	1.49
S_2P_R	TRUE	1	P	2	17.9	1.73	636.3	15.81
S_2T	TRUE	1	T	2	0.3	0.48	14.8	2.49
S_3J	TRUE	1	J	3	1146.4	68.34	193106	1468.36
S_3P_C	TRUE	1	P	3	73.4	7.35	13808.7	250.37
S_3T	TRUE	1	T	3	0.2	0.42	57.9	5.74
S_4J	TRUE	1	J	4	2671.5	196.85	439371.1	3103.24
S_4P_C	TRUE	-1	P	4	-1	0	-1	0
S_4T	TRUE	1	T	4	0.3	0.48	90.4	9.58
S_5P_C	TRUE	-1	P	5	-1	0	-1	0
S_5T	TRUE	1	T	5	1	0	173.3	16.75

8.2.1 Data

We generated 27 test BGPs with different characteristics. These are listed in Table 13. BGP labels are informative:

- S: the bgp is satisfiable
- N: the bgp is not satisfiable
- *number*: the number of triples in the bgp
- T: the bgp contains only variables without joins
- J: the bgp contains one or more joins
- P: the bgp contains a join on an object and must satisfy the single-path-to-root condition
- R: the bgp contains a join on an object that is fx:root
- C: the bgp contains a join on an object that is a container

8.2.2 Results

Results are shown in tables 14-17. Table 14 show results for the Top down (Search) algorithm when doing satisfiability check, ie. the algorithm stops when the first satisfiable annotation is found. We report the speed in average milliseconds and the standard deviation. Furthermore, we show the

Table 15 Results for algorithm Top down (Search): generate all satisfiable annotations.

name	satisfiable?	found	type	size	ms (avg)	ms (std)	tested (avg)	tested (std)
S_1T	TRUE	6	T	1	6	0.67	187.4	17.88
S_2J	TRUE	36	J	2	541.9	62.94	72895.7	795.27
S_2P_R	TRUE	1	P	2	33.8	9.38	6445.1	291.25
S_2T	TRUE	32.3	T	2	2917.4	1055.36	603611	212133.29
S_3J	TRUE	-1	J	3	-1	0	-1	0
S_3P_C	TRUE	-1	P	3	-1	0	-1	0
S_3T	TRUE	-1	T	3	-1	0	-1	0
S_4J	TRUE	-1	J	4	-1	0	-1	0
S_4P_C	TRUE	-1	P	4	-1	0	-1	0
S_4T	TRUE	-1	T	4	-1	0	-1	0
S_5P_C	TRUE	-1	P	5	-1	0	-1	0
S_5T	TRUE	-1	T	5	-1	0	-1	0

Table 16 Results for algorithm Bottom up (CSP): only satisfiability.

name	satisfiable?	found	type	size	ms (avg)	ms (std)	tested
N_1T	FALSE	0	T	1	17.5	1.84	2
N_2J	FALSE	0	J	2	0.3	0.48	2
N_2P_R	FALSE	0	P	2	6.2	0.79	36
N_2T	FALSE	0	T	2	3.4	0.7	24
N_3J	FALSE	0	J	3	0	0	24
N_3P_C	FALSE	0	P	3	2.9	0.88	18
N_3P_R	FALSE	0	P	3	8	3.16	36
N_3T	FALSE	0	T	3	16.6	2.46	288
N_4J	FALSE	0	J	4	0	0	288
N_4P_C	FALSE	0	P	4	2.8	0.92	54
N_4T	FALSE	0	T	4	93.8	9.35	3456
N_5J	FALSE	0	J	5	0.1	0.32	3456
N_5P_C	FALSE	0	P	5	7.4	1.58	162
N_5T	FALSE	0	T	5	536.1	78.69	41472
S_1T	TRUE	1	T	1	0.5	0.53	12
S_2J	TRUE	1	J	2	0.1	0.32	144
S_2P_R	TRUE	1	P	2	0.6	0.52	36
S_2T	TRUE	1	T	2	0.4	0.52	144
S_3J	TRUE	1	J	3	1.4	0.7	432
S_3P_C	TRUE	1	P	3	0.8	0.42	18
S_3T	TRUE	1	T	3	0.9	0.32	1728
S_4J	TRUE	1	J	4	12.1	2.42	5184
S_4P_C	TRUE	1	P	4	0.5	0.53	54
S_4T	TRUE	1	T	4	5.1	0.99	20736
S_5P_C	TRUE	1	P	5	0.3	0.48	162
S_5T	TRUE	1	T	5	361.4	46.9	248832

Table 17 Results for algorithm Bottom up (CSP): generate all satisfiable annotations.

name	satisfiable	found	type	size	ms (avg)	ms (std)	gen.
S_1T	TRUE	6	T	1	1	0.47	12
S_2J	TRUE	36	J	2	1.1	0.32	144
S_2P_R	TRUE	1	P	2	0.4	0.52	36
S_2T	TRUE	36	T	2	1.1	0.32	144
S_3J	TRUE	60	J	3	3	0.47	432
S_3P_C	TRUE	4	P	3	0.5	0.53	18
S_3T	TRUE	216	T	3	15.1	1.85	1728
S_4J	TRUE	300	J	4	36.7	5.48	5184
S_4P_C	TRUE	8	P	4	0.9	0.32	54
S_4T	TRUE	1296	T	4	171.9	31.8	20736
S_5P_C	TRUE	16	P	5	2.2	0.63	162
S_5T	TRUE	7776	T	5	1754.5	270.5	248832

number of solution patterns that are evaluated. When reporting intermediate solutions evaluated, we include both average and standard deviation, as the algorithm is not fully deterministic, as the temporary search states are not ordered and the process stops when the first solution is found (without completing the search). It can be seen that the algorithm failed to return a result in five seconds in several cases. Table 15 shows results for the Top-down (Search) algorithm when generating all satisfiable annotations, ie. the algorithm proceeds to find all solution patterns. BGPs with more than two triples required more than five seconds. Results show how with just 2 triples (S_2T), the algorithm generated 36 viable solutions after evaluating an average of 585926.3 ± 8903.79 possible annotations. We can conclude that this algorithm is inefficient.

Table 16 shows results for the Bottom up (CSP) algorithm when doing satisfiability check, ie. the algorithm stops when the first satisfiable annotation is found. Average and standard deviation for the number of solutions generated is fixed as the algorithm is fully deterministic (although the process stops when the first viable solution pattern is found). Table 17 shows results for the Bottom-up (CSP) algorithm when generating all satisfiable annotations, ie. the algorithm proceeds to find all solution patterns. With this algorithm, we managed to evaluate all BGPs for satisfiability and, in addition, to generate all possible solution patterns, in the worst case, in less than two seconds. However, the worst results (N_5T) goes still above the second, which opens the question whether it would be possible to perform further optimisations. We can conclude that the Bottom-up (CSP) algorithm is more efficient and practically sustainable.

8.3 Evaluation with real world queries

We evaluate the practical feasibility with basic graph patterns derived from real world queries. We only perform experiments with the most efficient, bottom up algorithm (Listing 6). Here, we summarise the data and results. We refer the reader to Appendix A for extensive details on data, results, and reproducibility.

8.3.1 Data

We extract 1,309 BGPs from 449 queries collected from public GitHub repositories. 228 of those queries have only a single BGP (50.7%). As illustrated in figures 3 and 4, 83.7% of the queries have less than 5 BGPs, 44.6% of the collected BGPs have a single triple, while 75.9% of them

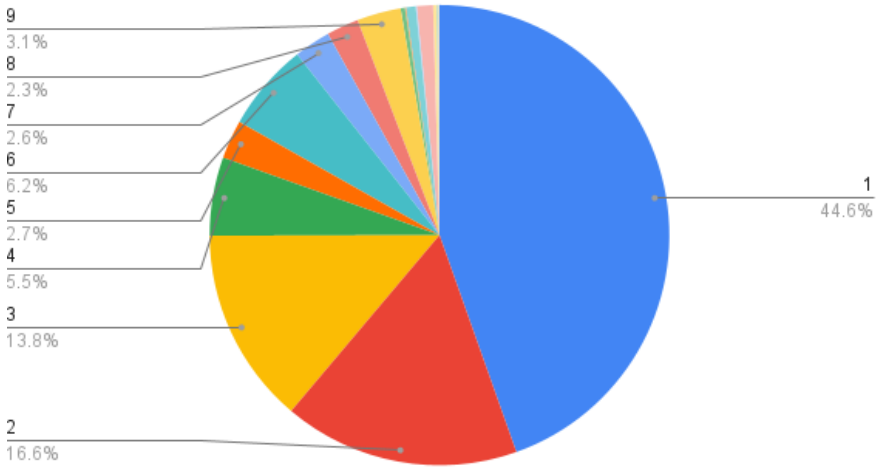

Figure 3 Dimension of BGPs from real-world queries: number of triples.

have up to 5 variables. We can conclude that the majority of real world queries and related BGPs have few triples and variables, and have similar sizes to the curated BGPs designed to evaluate the two algorithms. See further details in Appendix A.

8.3.2 Results

Satisfiability (finding the first valid solution pattern) can be verified in less than 100ms for all BGPs with up to 10 triples or 14 variables. The only problematic BGP is one with 19 triples and 19 variables ("query-77.rq" in the supplemental material), which cannot be verified in less than 5 seconds. Similarly, all solution patterns can be generated efficiently in all but two cases (below half second). The only two problematic BGPs are one with 19 triples and 19 variables ("query-77.rq" in the supplemental material), and another with 20 triples and 23 variables ("query-244.rq" in the supplemental material), for which it is not possible to generate all solution patterns in less than 5 seconds. See further details in Appendix A.

8.4 Evaluation of benefits with benchmark queries

We evaluate the advantages of assessing the satisfiability of a query before its evaluation. We perform experiments with the most efficient, i.e. the bottom up algorithm.

8.4.1 Data

The evaluation relied on the of the GTFS-Madrid-Bench benchmark [10], a virtual knowledge graph access benchmark aimed at assessing the performance of Ontology-based Data Access (OBDA) engines. More specifically, we use a specialisation of the benchmark [6] for façade-based data access engines.

The benchmark comprises a set of 18 SPARQL queries that interrogate a collection of data sources. Table 24 (see Appendix B) reports the characteristics of the queries of the benchmark in terms of the number of BGPs, the number of variables per BGP, and the number of triple patterns per BGP.

Furthermore, the benchmark offers a data generator that can increase the size of the original data and distribute the datasets across various formats.

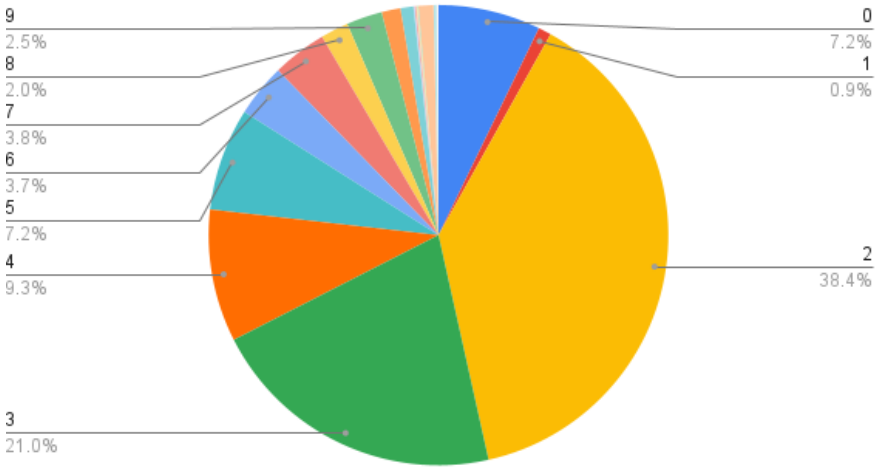

Figure 4 Dimension of BGPs from real-world queries: number of variables.

In this experiment, we use JSON to represent hierarchical data formats and CSV for tabular data sources. We generate datasets of sizes 1 and 10 and execute the 18 queries on them using the SPARQL Anything engine (version v1.1.0)[24]. We execute each query three times and record the time taken to construct and load the Façade-X RDF graph in memory for each execution. Finally, we calculate the average of these three runs. Moreover, for each query of the benchmark, we run the bottom-up algorithm for assessing the query satisfiability. Table 24 (see Appendix B) reports construction and loading time and time for assessing the satisfiability of the queries.

8.4.2 Results

The satisfiability of the GTFS-Madrid-Bench queries can be assessed in less than 15 ms. Excluding the first query reduces the maximum time to 2.4. In contrast, construction and loading times are at least two orders of magnitude longer (the shortest recorded time was 161.5 ms for query 6, size 1, with CSV input). This demonstrates the benefit of assessing satisfiability prior to query execution, since construction and loading time can be avoided for unsatisfiable queries, thus saving computational resources. Naturally, the larger the input, the greater the benefit: while construction and loading times increase with the size of the input, assessment of satisfiability is independent from it.

9 Conclusions

In this article, we studied the satisfiability of SPARQL basic graph patterns in Façade-X. We introduced a consolidated version of Façade-X including formalised mappings to RDF. We studied the conditions under which a BGPs are satisfiable as the basis for an algorithm to verify that efficiently. Extensive experiments demonstrate the feasibility of satisfiability check. Two extreme cases of large BGPs motivate future work on further improving the performance, focusing on reducing the space of potential Façade-X annotations before verifying the constraints. In the future, we will extend this study to consider specific Façade-X profiles, studying how those can

[24] https://github.com/SPARQL-Anything/sparql.anything/releases/tag/v1.1.0

bring additional constraints to verify (e.g. Façade-X graphs from CSVs will never have types). Crucially, identifying solution patterns is a preliminary step for investigating classes of queries (e.g. star queries) whose solutions can be streamed directly from the file sources, without needing to load intermediate results in-memory. Finally, our work paves the way for studying query execution strategies for Façade-X data access with SPARQL and supporting developers in building more efficient data integration systems for knowledge graphs.

References

1 Serge Abiteboul, Sophie Cluet, Vassilis Christophides, Tova Milo, Guido Moerkotte, and Jérôme Siméon. Querying documents in object databases. *International Journal on Digital Libraries*, 1:5–19, 1997. doi:10.1007/S007990050001.

2 Renzo Angles, Georg Gottlob, Aleksandar Pavlović, Reinhard Pichler, and Emanuel Sallinger. Sparqlog: A system for efficient evaluation of sparql 1.1 queries via datalog. *Proceedings of the VLDB Endowment*, 16(13):4240–4253, 2023. doi:10.14778/3625054.3625061.

3 Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. On the semantics of sparql. In *Semantic web information management: A model-based perspective*, pages 281–307. Springer, 2009. doi:10.1007/978-3-642-04329-1_13.

4 Julián Arenas-Guerrero, David Chaves-Fraga, Jhon Toledo, María S Pérez, and Oscar Corcho. Morph-kgc: Scalable knowledge graph materialization with mapping partitions. *Semantic Web*, 15(1):1–20, 2024. doi:10.3233/SW-223135.

5 Luigi Asprino and Miguel Ceriani. How is your knowledge graph used: Content-centric analysis of SPARQL query logs. In *Proceedings of the 22nd International Semantic Web Conference, Part I*, pages 197–215, 2023. doi:10.1007/978-3-031-47240-4_11.

6 Luigi Asprino, Enrico Daga, Justin Dowdy, Aldo Gangemi, and Paul Mulholland. Materialisation approaches for Façade-based data access with SPARQL. *Semantic Web -– Interoperability, Usability, Applicability, IOS Press*, 2022. URL: https://www.semantic-web-journal.net/content/materialisation-approaches-fa%C3%A7ade-based-data-access-sparql-0.

7 Luigi Asprino, Enrico Daga, Justin Dowdy, Paul Mulholland, Aldo Gangemi, and Marco Ratta. Streamlining Knowledge Graph Construction with a façade: The SPARQL Anything project. In Publications Office of the European Union and European Commission. Directorate General for Informatics., editor, *ENDORSE, 2nd EuropeaN Data conference On Reference data and SEmantics ENDORSE 2023: virtual event, 14 16 March 2023 : proceedings*. Publications Office, LU, 2023. doi:10.2830/343811.

8 Luigi Asprino, Enrico Daga, Aldo Gangemi, and Paul Mulholland. Knowledge graph construction with a façade: A unified method to access heterogeneous data sources on the web. *ACM Transactions on Internet Technology*, 23(1):1–31, 2023. doi:10.1145/3555312.

9 Sarra Bouhenni, Said Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. A survey on distributed graph pattern matching in massive graphs. *ACM Computing Surveys (CSUR)*, 54(2):1–35, 2021. doi:10.1145/3439724.

10 David Chaves-Fraga, Freddy Priyatna, Andrea Cimmino, Jhon Toledo, Edna Ruckhaus, and Oscar Corcho. GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain. *Journal of Web Semantics*, 65:100596, 2020. doi:10.1016/J.WEBSEM.2020.100596.

11 David Chaves-Fraga, Edna Ruckhaus, Freddy Priyatna, Maria-Esther Vidal, and Oscar Corcho. Enhancing virtual ontology based access over tabular data with morph-csv. *Semantic Web*, 12(6):869–902, 2021. doi:10.3233/SW-210432.

12 Chris Clifton, Hector Garcia-Molina, and David Bloom. Hyperfile: A data and query model for documents. *The VLDB Journal*, 4:45–86, 1995. URL: http://www.vldb.org/journal/VLDBJ4/P045.pdf.

13 Richard Cyganiak. Tarql (sparql for tables): Turn csv into rdf using sparql syntax. Technical report, Technical Report, 2015. Available at: http://tarql.github.io, 2015.

14 Enrico Daga. Sparql-anything/facade-x-queries-on-github: v1, January 2026. doi:10.5281/zenodo.18133036.

15 Enrico Daga. Supplementary material - sparql-anything/fxbgp: v4, January 2026. doi:10.5281/zenodo.18196574.

16 Enrico Daga, Luigi Asprino, Paul Mulholland, and Aldo Gangemi. Facade-X: an opinionated approach to SPARQL anything. *Studies on the Semantic Web*, 53:58–73, 2021.

17 Enrico Daga, Luca Panziera, and Carlos Pedrinaci. A basilar approach for building web apis on top of sparql endpoints. In *CEUR Workshop Proceedings*, volume 1359, pages 22–32, 2015. URL: https://ceur-ws.org/Vol-1359/paper4.pdf.

18 Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2246–2258, 2022.

19 George HL Fletcher. An algebra for basic graph patterns. In *Proceedings of the Workshop on Logic in Databases*, pages 38–46. Citeseer, 2008.

20 Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, volume 45, pages 43–53, 2006.

21 Herminio García-González, Iovka Boneva, Sławek Staworko, José Emilio Labra-Gayo, and Juan Manuel Cueva Lovelle. Shexml: improving the usability of heterogeneous data mapping languages for first-time users. *PeerJ Computer Science*, 6:e318, 2020. doi:10.7717/PEERJ-CS.318.

22 Ramanathan Guha and Dan Brickley. RDF schema 1.1. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

23 Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

24 Olaf Hartig and Ralf Heese. The sparql query graph model for query optimization. In *European Semantic Web Conference*, pages 564–578. Springer, 2007. doi:10.1007/978-3-540-72667-8_40.

25 Pieter Heyvaert, Ben De Meester, Anastasia Dimou, and Ruben Verborgh. Declarative rules for linked data generation at your fingertips! In *European Semantic Web Conference*, pages 213–217. Springer, 2018. doi:10.1007/978-3-319-98192-5_40.

26 Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. Knowledge graphs. *ACM Computing Surveys (Csur)*, 54(4):1–37, 2021. doi:10.1145/3447772.

27 Enrique Iglesias, Samaneh Jozashoori, David Chaves-Fraga, Diego Collarana, and Maria-Esther Vidal. Sdm-rdfizer: An rml interpreter for the efficient creation of rdf knowledge graphs. In *Proceedings of the 29th ACM international conference on Information & Knowledge Management*, pages 3039–3046, 2020. doi:10.1145/3340531.3412881.

28 Enrique Iglesias, Samaneh Jozashoori, and Maria-Esther Vidal. Scaling up knowledge graph creation to large and heterogeneous data sources. *Journal of Web Semantics*, 75:100755, 2023. doi:10.1016/J.WEBSEM.2022.100755.

29 Ana Iglesias-Molina, Dylan Van Assche, Julián Arenas-Guerrero, Ben De Meester, Christophe Debruyne, Samaneh Jozashoori, Pano Maria, Franck Michel, David Chaves-Fraga, and Anastasia Dimou. The rml ontology: a community-driven modular redesign after a decade of experience in mapping heterogeneous data to rdf. In *International Semantic Web Conference*, pages 152–175. Springer, 2023. doi:10.1007/978-3-031-47243-5_9.

30 Kostis Kyzirakos, Ioannis Vlachopoulos, Dimitrianos Savva, Stefan Manegold, and Manolis Koubarakis. GeoTriples: a Tool for Publishing Geospatial Data as RDF Graphs Using R2RML Mappings. In *TC/SSN@ ISWC*, pages 33–44, 2014. URL: https://ceur-ws.org/Vol-1401/paper-03.pdf.

31 Laks VS Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng Zhao. On testing satisfiability of tree pattern queries. In *VLDB*, volume 4, pages 120–131, 2004.

32 Markus Lanthaler, Richard Cyganiak, and David Wood. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

33 Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. A sparql extension for generating rdf from heterogeneous formats. In *Proc of ESWC*, pages 35–50. Springer, 2017. doi:10.1007/978-3-319-58068-5_3.

34 Lorenzo Livi and Antonello Rizzi. The graph matching problem. *Pattern Analysis and Applications*, 16:253–283, 2013. doi:10.1007/S10044-012-0284-8.

35 Albert Meroño-Peñuela and Rinke Hoekstra. grlc makes github taste like linked data apis. In *The Semantic Web: ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29–June 2, 2016, Revised Selected Papers 13*, pages 342–353. Springer, 2016. doi:10.1007/978-3-319-47602-5_48.

36 Franck Michel, Catherine Faron-Zucker, and Fabien Gandon. SPARQL micro-services: lightweight integration of web APIs and linked data. In *LDOW@ WWW*, 2018.

37 Sitt Min Oo and Olaf Hartig. An algebraic foundation for knowledge graph construction. In Edward Curry, Maribel Acosta, María Poveda-Villalón, Marieke van Erp, Adegboyega K. Ojo, Katja Hose, Cogan Shimizu, and Pasquale Lisena, editors, *The Semantic Web - 22nd European Semantic Web Conference, ESWC 2025, Portoroz, Slovenia, June 1-5, 2025, Proceedings, Part I*, volume 15718 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2025. doi:10.1007/978-3-031-94575-5_1.

38 Yue Pang, Linglin Yang, Lei Zou, and M Tamer Özsu. gfov: A full-stack sparql query optimizer & plan visualizer. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pages 5081–5085, 2023. doi:10.1145/3583780.3614741.

39 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006. doi:10.1007/11926078_3.

40 Marco Ratta, Enrico Daga, and Luigi Asprino. Pysparql anything showcase. In *European Semantic Web Conference*, pages 301–305. Springer, 2024. doi:10.1007/978-3-031-78952-6_46.

41 Mariano Rodriguez-Muro and Martin Rezk. Efficient sparql-to-sql with r2rml mappings. *Journal of Web Semantics*, 33:141–169, 2015. doi:10.1016/J.WEBSEM.2015.03.001.

42 Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

43 Jaime Salas and Aidan Hogan. A taxonomy of basic graph pattern motifs for understanding SPARQL query logs. In Benny Kimelfeld, Maria Vanina Martinez, and Renzo Angles, editors, *Proceedings of the 15th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2023)*, volume 3409, 2023. URL: https://ceur-ws.org/Vol-3409/paper15.pdf.

44 Jason Slepicka, Chengye Yin, Pedro A Szekely, and Craig A Knoblock. Kr2rml: An alternative interpretation of r2rml for heterogenous sources. In *Proc of Cold*, 2015.

45 Jean-Marc Steyaert and Philippe Flajolet. Patterns and pattern-matching in trees: An analysis. *Information and Control*, 58(1):19–58, 1983. doi:10.1016/S0019-9958(83)80056-4.

46 Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, pages 595–604, 2008. doi:10.1145/1367497.1367578.

47 Dylan Van Assche, Thomas Delva, Gerald Haesendonck, Pieter Heyvaert, Ben De Meester, and Anastasia Dimou. Declarative rdf graph generation from heterogeneous (semi-) structured data: A systematic literature review. *Journal of Web Semantics*, 75:100753, 2023. doi:10.1016/J.WEBSEM.2022.100753.

48 Anne-Marie Vercoustre and François Paradis. A descriptive language for information object reuse through virtual documents. In Maria E. Orlowska and Roberto Zicari, editors, *OOIS'97*, pages 299–311, London, 1998. Springer London.

49 Maria-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently joining group patterns in sparql queries. In *Extended Semantic Web Conference*, pages 228–242. Springer, 2010. doi:10.1007/978-3-642-13486-9_16.

50 Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. A short survey of recent advances in graph matching. In *Proceedings of the 2016 ACM on international conference on multimedia retrieval*, pages 167–174, 2016. doi:10.1145/2911996.2912035.

51 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

52 Xiaowang Zhang, Jan Van den Bussche, and François Picalausa. On the satisfiability problem for sparql patterns. *Journal of Artificial Intelligence Research*, 56:403–428, 2016. doi:10.1613/JAIR.5028.

53 Antoine Zimmermann. RDF 1.1: On semantics of RDF datasets. W3C note, W3C, February 2014. https://www.w3.org/TR/2014/NOTE-rdf11-datasets-20140225/.

A Experiments with real-world queries

We report on experiments with real world SPARQL queries tailored to facade based data access. The queries were collected from the GitHub API on 18 December 2025 and include all files published in public repositories with extensions ".rq" and ".sparql", which include the string "x-sparql-anything" in the text. The code and data are published[25] and archived [14]. The analysis was performed on Google drive and can be viewed at `https://docs.google.com/spreadsheets/d/11n7c101WLQBL8aApZs-fXNtHAYTIBIP9nZUDxD6TUa8/edit`.

We perform experiments for both satisfiability and to generate all possible solution patterns on 449 collected queries. We extract 1430 BGPs from the 449 queries. Table 18 shows a summary of the queries grouped by number of BGPs. Of these queries, 58 cannot be parsed because they include syntax errors. 228 of the remaining queries have a single BGP. The majority of queries have less than five basic graph patterns. Table 19 summarizes the BGPs extracted from the queries, to which we apply our satisfiability problem. 103 BGPs with 0 triples refer to BGPs that included SPARQL Anything configuration options, that we exclude. Figure 5 shows the number of BGPs by the number of variables included. Clearly, the more the variables, the more complex it is to verify satisfiability, since variables are annotated with more potential annotations. The majority of BGPs have between 1 and 5 variables. In what follows, we show the average values of 10 executions.

A.1 Satisfiability check

We perform the same experiment setting as with the designed queries (Section 8), focusing on the most performing bottom-up algorithm only, with the 449 real world queries.

Table 20 shows the average duration of the more efficient bottom-up satisfiability check algorithm presented in Section 7. Data is grouped by BGPs with the same number of triples. BGPs with 0 triples are the ones that only included triples with SPARQL Anything configuration,

[25] `https://github.com/SPARQL-Anything/facade-x-queries-on-github/`

Table 18 Number of queries by BGPs.

Queries	BGPs
1	40
1	31
1	29
2	27
2	26
1	25
1	24
2	22
1	20
1	17
2	13
1	12
11	11
9	10
4	9
1	8
6	7
4	6
22	5
32	4
37	3
79	2
228	1

and can be ignored. Table 21 shows the average duration grouped by BGPs with the same number of variables. We can see how for BGPs with up to 16 triples and 18 variables (except one with 15 variables), satisfiability can be verified in less than 100 ms. The only problematic BGP is one with 19 triples and 19 variables ("query-77.rq" in the supplemental material), which cannot be verified in less than 5 seconds. We ignore errors from two other BGPs, one within a DBPedia service clause ("query-66.rq"), and another coming from a negative test within a SPARQL Anything code base ("query-199.rq").

A.2 Generating all solutions

We perform the same experiment setting as with the designed queries (Section 8), focusing on the most performing bottom-up alogorithm only, with the 449 real world queries.

Table 22 shows the average duration of the more efficient bottom-up algorithm presented in Section 7 to generate all possible solution patterns. Data is grouped by BGPs with the same number of triples. In addition, Table 23 shows the average duration grouped by BGPs with the same number of variables. Solutions can be generated efficiently in all but two cases (below half second). The only two problematic BGPs are one with 19 triples and 19 variables ("query-77.rq" in the supplemental material), and another with 20 triples and 23 variables ("query-244.rq" in the supplemental material), for which it is not possible to generate all solution patterns in less than 5 seconds.

Table 19 Basic graph patterns grouped by number of triples (that are not SPARQL Anything configuration options).

Number of triples	Number of BGPs
0	103
1	591
2	220
3	183
4	73
5	36
6	82
7	34
8	30
9	41
10	4
11	1
12	9
13	1
14	15
16	3
18	1
19	1
20	1

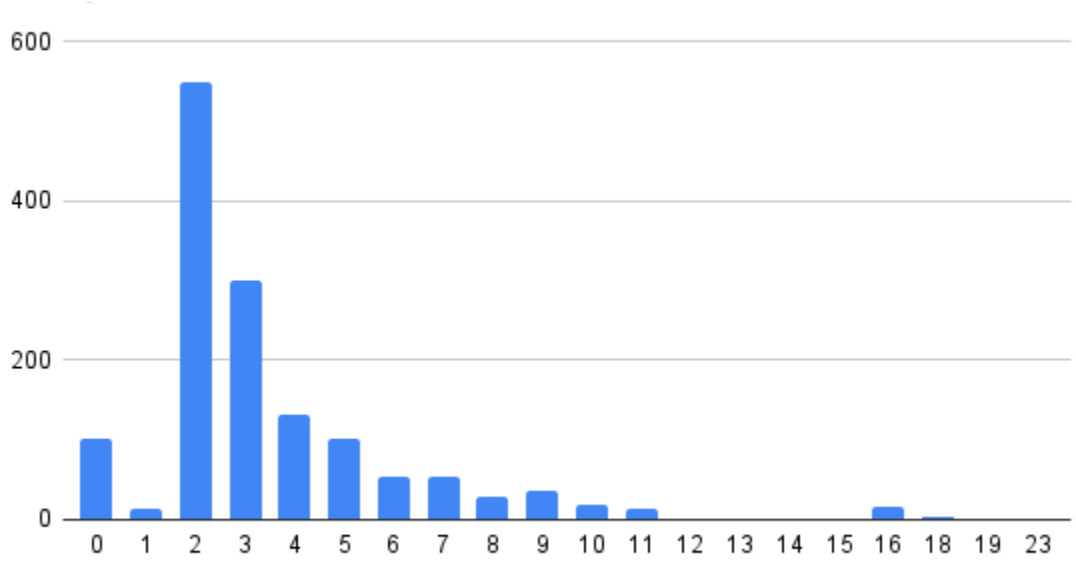

Figure 5 BGPs by number of variables.

Table 20 Performance of the bottom-up satisfiability check algorithm on real world queries. We group the data by size of BGPs (number of triples). There is only 1 BGP with 11,13,18,19, and 20 triples. One query with 19 triples cannot be evaluated in less the 5 seconds (our limit).

Triples	AVG	STD
1	0.10	0.3
2	0.11	0.37
3	0.14	0.27
4	0.14	0.36
5	0.41	0.29
6	0.28	0.17
7	0.48	0.15
8	1.36	0.67
9	0.48	0.25
10	1.15	0.58
11	0.20	–
12	6.73	1.81
13	8.30	–
14	0.33	0.07
16	0.43	0.02
18	218.30	–
19	>5s	–
20	8.20	–

Table 21 Performance of the bottom-up satisfiability check algorithm on real world queries. We group the data by the size of BGPs - number of variables. There are only single BGPs with 12,14,15,19, and 23 variables. One query with 19 variables cannot be evaluated in less the 5 seconds (our limit).

Variables	AVG	STD
1	0.05	0.18
2	0.11	0.31
3	0.11	0.26
4	0.15	0.42
5	0.25	0.22
6	0.27	0.41
7	0.37	0.13
8	0.22	0.18
9	1.65	0.69
10	0.42	0.26
11	4.09	1.61
12	0.20	–
13	4.30	0.77
14	0.20	–
15	218.30	–
16	0.33	0.07
18	0.43	0.02
19	>5s	–
23	8.20	–

Table 22 Performance of the bottom-up algorithm on real world queries, when generating all solution patterns. We group the data by size of BGPs (number of triples). There is only 1 BGP with 11,13,18,19, and 20 triples. For two queries with 19 and 20 triples, the complete set of solution patterns cannot be evaluated in less the 5 seconds (our limit).

BGP Size (number of triples)	AVG	STD
1	0.05	0.37
2	0.08	0.19
3	0.13	0.25
4	0.23	0.16
5	8.05	6.27
6	1.36	0.82
7	2.23	1.01
8	7.79	1.75
9	5.51	1.19
10	38.58	3.21
11	21.50	–
12	58.10	20.53
13	163.70	–
14	191.97	12.58
16	1,250.00	40.42
18	1,127.40	–
19	>5s	–
20	>5s	–

Table 23 Performance of the bottom-up algorithm on real world queries. We group the data by the size of BGPs – number of variables. There are only single BGPs with 12,14,15,19, and 23 variables. For two queries with 19 and 23 variables the set of solution patterns cannot be generated in less the 5 seconds (our limit).

BGP Size (number of variables)	AVG	STD
1	0.02	0.14
2	0.04	0.36
3	0.08	0.29
4	0.14	0.32
5	0.24	0.14
6	0.46	0.16
7	1.21	0.58
8	1.19	0.23
9	8.47	2.01
10	25.58	8.36
11	21.15	3.1
12	21.50	–
13	256.55	35.91
14	67.40	–
15	1,127.40	–
16	191.97	12.58
18	1,250.00	40.42
19	-1.00	–
23	-1.00	–

B Experiments with benchmark queries

Table 24 describes the characteristics of the queries of the GTFS-Madrid-Bench benchmark in terms of number of BGPs for each query, number of variables and number of triple patterns per BGP.

Table 24 Characteristics of the queries contained in the GTFS-Madrid-Bench benchmark.

Query ID	Number of BGPs	Number of variables per BGP	Number of Triple Patterns per BGP
1	1	5	4
2	4	2,2,2,3	1,1,1,2
3	5	2,2,2,3,2	1,1,1,2,1
4	7	2,2,2,2,2,4,2	1,1,1,1,1,3,1
5	1	4	3
6	1	3	2
7	10	2,2,2,3,2,2,3,2,2,3	1,1,1,2,1,1,2,2,1,2
8	11	2,2,2,3,2,2,4,2,2,2,3	1,1,1,2,1,1,3,1,1,1,2
9	3	5,2,4	4,1,3
10	2	2,3	1,2
11	4	3,4,1,2	2,3,2,1
12	4	3,3,3,3	2,2,2,3
13	3	3,2,4	2,1,3
14	4	5,3,2,2	4,2,1,1
15	1	4	2
16	2	4,3	3,3
17	4	2,2,4,4	1,1,3,3
18	5	2,4,2,2,2	2,3,1,1,1

Table 25 shows how long it takes to construct and load the RDF graph required to answer each query into memory. Additionally, the time required to compute the satisfiability of all BGPs in each query is reported.

Table 25 Construction Loading time and time for assessing the satisfiability of the queries in the GTFS-Madrid-Bench benchmark.

Query	Construction + Loading time (ms)				Time for computing the satisfiability of all BGPs (ms)
	CSV		JSON		
	Size 1	Size 10	Size 1	Size 10	
1	1049.5	7476.5	1221.5	6252	14.40
2	310.5	607.5	284	603.5	0.80
3	402.5	595	301	673.5	0.70
4	283	251	218	241.5	1.70
5	166.5	230	215.5	222	0.20
6	161.5	177	166	218	0.10
7	390	1031.5	390	1151.5	2.40
8	449	1598.5	410	1627	1.10
9	1390	6078.5	1260	5609.5	0.70
10	292	656.5	292	609.5	0.90
11	240.5	331.5	218.5	306	1.80
12	379.5	851	305	841	1.70
13	275.5	566.5	246.5	515	1.60
14	327	1107	348.5	1173	1.20
15	270	1340	289	1291	0.40
16	202.5	299.5	213	294.5	0.40
17	233	438	264.5	422.5	0.80
18	214	309	237	298	0.80