chameleon-db: a Workload-Aware Robust RDF Data Management System

University of Waterloo Technical Report CS-2013-10

Güneş Aluç, M. Tamer Özsu, Khuzaima Daudjee, Olaf Hartig

David R. Cheriton School of Computer Science, University of Waterloo

{galuc,tozsu,kdaudjee,ohartig}@uwaterloo.ca

ABSTRACT

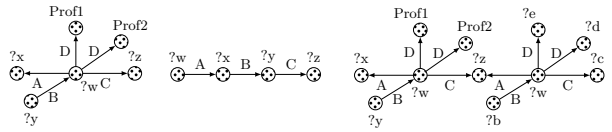
The Resource Description Framework (RDF) is a World Wide Web Consortium (W3C) standard for the conceptual modeling of web resources, and SPARQL is the standard query language for RDF. As RDF is becoming more widely utilized, RDF data management systems are being exposed to workloads that are much more diverse and dynamic than they were designed to support, for which they are unable to provide consistently good performance. The problem arises because these systems are workload-agnostic; that is, they rely on a database structure and types of indexes that are fixed a priori, which cannot be modified at runtime.

In this paper, we introduce *chameleon-db*, which is a workload-aware RDF data management system that we have developed. *chameleon-db* automatically and periodically adjusts its layout of the RDF database to optimize for queries so that they can be executed efficiently. Since one cannot afford to stop processing queries, we propose a novel design that enables partitions to be concurrently updated. We demonstrate that *chameleon-db* can achieve robust performance across a diverse spectrum of query workloads, outperforming its competitors by up to 2 orders of magnitude, and that it can easily adapt to changing workloads.

1. INTRODUCTION

RDF and SPARQL are the building blocks of the semantic web, and they are important tools in web data integration. RDF is composed of subject-predicate-object (s, p, o) statements called *triples* [17]. Each triple describes an aspect of a web resource. The subject of the triple denotes the resource that is described, the predicate denotes a feature of that resource, and the object stores the value for that feature.

With the proliferation of very large, web-scale distributed RDF datasets such as the Linked Open Data (LOD) cloud [6], the demand for high-performance RDF data management systems has increased. The wide variety of applications that RDF data management systems support mean that queries submitted to these systems are becoming more diverse [4, 11, 16]. This diversity has a number of reasons. First, a single *triple pattern*, which is the atomic unit in a SPARQL query, can be composed in eight different ways, and in real applications, each composition appears with increasing regularity [4]. To address this type of diversity, some systems index triples on all binary permutations of attributes (i.e., *s-p*, *s-o*, *p-s*, *p-o*, *o-s*, *o-p*) [19, 29]. Second, triple patterns can be combined in numerous ways (Figure 1), giving rise to *structural diversity*—query structures can be star-shaped, linear, snowflake-shaped, or an even more complex structure. Our objective in this paper is to build an RDF data management system whose performance is robust across structurally diverse queries even when the frequently queried structures are changing.



(a) Star (b) Linear (c) Snowflake

Figure 1: Structural diversity in SPARQL.

1.1 Motivation

As SPARQL queries in typical workloads become more structurally diverse [4, 11, 16], state-of-the-art systems are unable to provide consistently good performance for such workloads. The problem is that most systems are workload-agnostic; that is, they rely on a database structure and types of indexes that are fixed a priori, which cannot be modified at runtime. If the logical and physical schema of a database are not designed for answering a particular type of query, that system is likely to incur significant performance penalty.

Consider the following experiment that demonstrates our argument. We generated 100 million triples using the Waterloo SPARQL Diversity Test Suite (WSDTS)[1] data generator and measured the performance of three state-of-the-art RDF data management systems, namely, *x-RDF-3x* [22], *Virtuoso* [12] and *gStore* [34], on three query mixes, each consisting of only one of (i) star-shaped, (ii) linear, (iii) snowflake-shaped, or (iv) complex queries. Section 7 contains more information regarding our experimental setup.

A snapshot of our results on a subset of the queries that we used in our experiments (Table 1) highlight two important

[1] https://cs.uwaterloo.ca/~galuc/wsdts/

	linear:L4	star:S3	snowflake:F5	complex:C3
x-RDF-3x	**7.6**ms	8.9ms	56.4ms	timeout
gStore	53.9ms	**6.2**ms	timeout	**162.7**ms
Virtuoso	282.8ms	347.2ms	**9.4**ms	103623.7ms

Table 1: Snapshot from our experimental results (timeout is issued if query evaluation does not terminate in 15 minutes).

issues. First, while performing very well on one type of query mix, the performance of each system is consistently worse on the remaining queries. Second, there is a different winner by a large margin for each query type. Therefore, deciding which system to deploy for a particular workload is difficult. One can choose a system that efficiently handles the "most frequent" query types in the workload. However, since that system can be very inefficient in executing the remaining types of queries, even though they appear less frequently, the overall performance of the system for that workload could be far less than optimal. Furthermore, the system is not robust to even the slightest changes in the workload; that is, when the frequently queried structures change, performance will significantly degrade.

1.2 Overview of Our Approach

In this paper, we introduce *chameleon-db*, which is a workload-aware RDF data management system that we have developed. Workload-awareness means that *chameleon-db* will automatically and periodically adjust the layout of the RDF database to optimize for queries so that they can be executed efficiently. In *chameleon-db*, RDF data and SPARQL queries are represented as graphs, and queries are evaluated using a subgraph matching algorithm [28]. In this sense, there are similarities with *gStore* [34], but instead of evaluating queries over the entire RDF graph, we partition it. Then, during query evaluation, irrelevant partitions are pruned out using *partition indexes* that are created across the partitions based on the contents of each partition (Figure 2). Our focus in this paper is on techniques and algorithms for the proper partitioning of the RDF graph and efficient query execution over this partitioned data. Therefore, we omit system features and details; we only note that *chameleon-db* is a fully operational system involving 30K lines of C++ code.

The way the graph is partitioned impacts performance, and *chameleon-db* uses it as a knob to periodically auto-tune its performance. We express query performance as a function of the number of intermediate tuples processed during query evaluation. There may be some intermediate tuples from which no final result is generated, which is natural because, for instance, not all intermediate tuples need to be joined with another tuple. We call these *dormant* tuples. The issue with dormant tuples is that even though resources are spent on constructing and processing them, neither their presence nor their absence alters the final query results. However, their abundance is bad for performance. By carefully isolating the parts of the graph that truly contribute to the final query results from those that do not, it is possible to reduce the number of dormant tuples that need to be processed during query evaluation, thereby improving system performance for that workload. For this purpose, we periodically re-partition the graph. Since we cannot afford to stop processing queries when the partitions are being updated, we propose a novel design that enables partitions to be concurrently updated while queries are being evaluated.

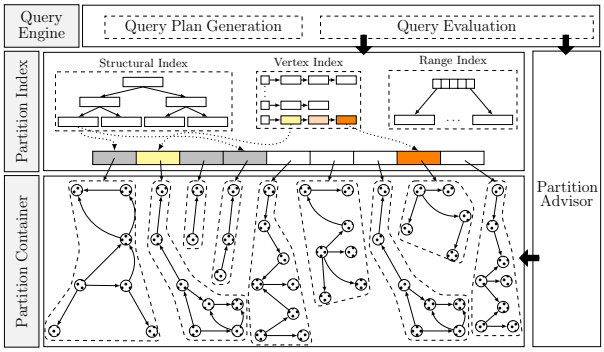


Figure 2: The architectural layout of *chameleon-db*.

Consider evaluating the linear query $Q = ?w \xrightarrow{A} ?x \xrightarrow{B} ?y \xrightarrow{C} ?z$ against the graph in Figure 3b. Any triple that lies outside the shaded region is not part of the query results. Ideally, we would like to find a partitioning that clearly separates these irrelevant triples from the actual query results such that as few dormant tuples as possible are produced during query evaluation. It turns out that Partitioning A in Figure 3c will serve this purpose. The important point is that the irrelevant triples (marked with dashed lines) are completely bypassed in this evaluation because query evaluation can be localized to P_2.

In contrast, consider Partitioning B (Figure 3d) that mixes irrelevant triples with triples that are actually part of the query results. In this case, the query needs to be decomposed into three sub-queries: $Q_1 = ?w \xrightarrow{A} ?x$, $Q_2 = ?x \xrightarrow{B} ?y$ and $Q_3 = ?y \xrightarrow{C} ?z$ (we will postpone the discussion of "why" to Section 4, where we describe our query evaluation model). Then, each subquery is evaluated over partitions P_3–P_6, producing three sets of tuples T_1, T_2 and T_3, respectively. Finally, these tuples are joined as shown in Figures 3e–3i. This evaluation produces dormant tuples. In this case, reordering the join operations or applying sideways information passing [20] to early-prune some of the tuples in T_1–T_3 does not eliminate the problem. For example, while tuples (v_1, v_5) and (v_4, v_{11}) can be eliminated from T_2 once v_2 is identified as the only join value in $T_1 \bowtie T_2$, (v_2, v_8) remains as a dormant tuple until all join operations are completed.

When a query is evaluated, *chameleon-db* collects statistics to estimate whether the current partitioning produces dormant tuples, and if so, how many. Using this information, a better partitioning is computed next time database re-organization takes place, which includes partitioning the database and updating the partition indexes. Consequently, *chameleon-db* achieves more robust performance than existing systems across queries with different properties (that is, its performance is consistently good across different query structures), and it can adapt to changing workloads.

1.3 Challenges and Our Contributions

One of the major challenges we address in this paper is the problem of partitioning the RDF graph. The question is how to use the information collected during query evaluation (i.e., regarding dormant tuples) to quantify the "goodness" of a partitioning such that an "optimal" one can be selected. Since it is also desirable to partition the graph in multiple ways so as to provide support for a wide range of workloads, there is a need for a generic query evaluation model that

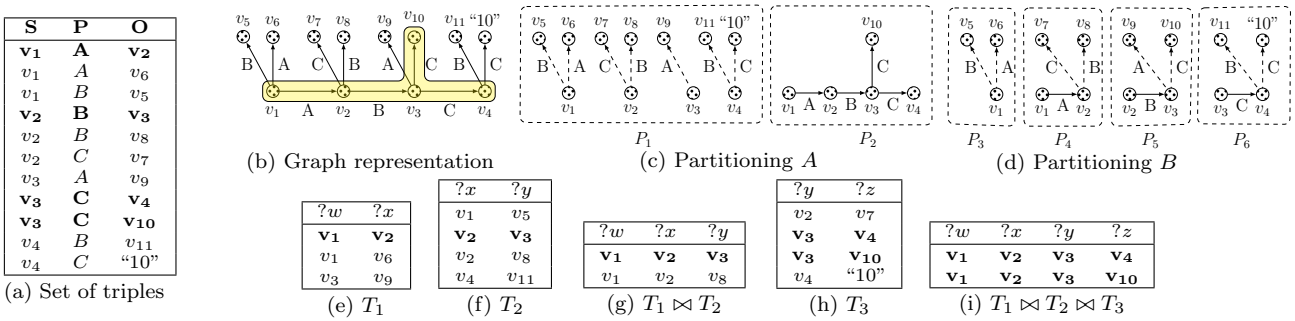


Figure 3: Sample dataset, their graph representation and sample partitionings.

produces correct results under all potential partitionings of the graph. In order to ensure correctness, a query may need to be decomposed, and therefore, finding an "optimal" decomposition is another challenge.

A related issue is that, unlike in relational databases, there is no schema that describes a partitioning in a well-defined manner that one can utilize to generate valid query plans. Arguably, a schema can be generated from a given partitioning; however, this requires expensive computational and maintenance overhead. Alternatively, a schema can be determined first, according to which the partitioning may be computed. However, that limits the type of partitionings, and therefore, the types of queries that can be efficiently supported by the system, which is contradictory to the idea of workload-awareness. Therefore, we propose efficient techniques for generating query plans without having to know the complete structure of the underlying partitioning.

Creating the partition indexes is another challenge, because partitions can be indexed on different (potentially exponentially many) combinations of attributes, some of which will never be relevant to the queries in the workload. For space considerations, we provide only an overview of our indexing approach in this paper (Section 6); it will be treated more fully elsewhere.

The paper makes the following contributions:

- We introduce a set of operations and a query evaluation model, namely, partition-restricted evaluation (PRE). The operations are carefully designed to accommodate a repartitioning of the RDF graph that does not disrupt query evaluation (Section 4).
- We prove that (i) for any partitioning of the graph, there exists a PRE expression that correctly evaluates the query and that (ii) this expression can be generated deterministically (Section 4). We extend our query evaluation model with a set of equivalence rules to generate more efficient expressions.
- We introduce methods to quantify the "goodness" of a partitioning with respect to a workload and develop a strategy in which the partitioning can be concurrently updated while queries are being evaluated (Section 5).
- We experimentally quantify the benefits of our workload-aware partitioning over workload-agnostic techniques employed by other RDF data management systems (Section 7), and we show that *chameleon-db* is more robust across all query structures and that it can adapt to changing workloads.

2. RELATED WORK

Research on RDF data management systems can be classified into two categories: (i) *single-node* approaches that aim to improve query performance by using alternative storage layouts, logical representations of data and indexing methods [1, 7, 12, 21, 30, 32, 34], and (ii) *distributed* approaches that focus on improving scalability through techniques for distributing RDF data on multiple machines and for answering queries over the distributed system [8,13,18,31,33]. These two are orthogonal research directions. In fact, the underlying query engine in most distributed approaches is a single-node system such as *RDF-3x* [21] or *gStore* [34], which is responsible for processing queries within each distributed node. *chameleon-db* falls into the first category of systems, therefore, we continue our discussion with single-node RDF data management systems.

Existing RDF data management systems are workload-agnostic by "design", that is, their choice of storage layout, logical representation of data and indexing had already been fixed at the time the system was implemented, and their design cannot be changed at runtime. Specifically, they

- use (i) a row-oriented [7,12,30], or (ii) a column-oriented store [1,12], (iii) a native store [21,32], or (iv) a graph store [34];
- represent RDF data as (i) a single table of triples [12, 21], (ii) a single large RDF graph [34], or (iii) a partitioned database in which each partition corresponds to a two-attribute table (one for each predicate in the dataset) [1], or (iv) same as (iii) but each partition corresponds to groups of triples that describe similar resources (i.e., resources that share common set of attributes are considered "similar") [7, 30];
- index (i) only a predefined (sub)set of attributes [1, 7, 12, 34], or (ii) exhaustively, all possible combinations of attributes [21].

These design choices affect the types of queries that can be efficiently supported by each system, and as our experiments show, these systems are unable to support certain types of queries efficiently. For example, gStore [34] is optimized to handle star-shaped queries and it needs to decompose more general queries into stars before executing them. In SW-Store [1], data are fully partitioned into two-attribute tables (one for each predicate), hence, queries need to be decomposed even further. RDF-3x [21] is designed to evaluate triple patterns efficiently. However, when it comes to star-shaped, snowflake-shaped or linear queries, they always need to be decomposed into triple patterns, which can result in a very large number of join operations. Although *DB2-RDF* [7] clusters triples that belong to the same type in an effort to reduce the number of joins, clustering relies

on the characteristics of the data, but *not* the workload. *chameleon-db* uses a graph-store, but in contrast to existing systems, it dynamically and periodically partitions the RDF graph and updates the indexes based on the current workload. Therefore, it is far more robust in the type of query structures it can handle, and it is capable of adapting to changing workloads.

3. BACKGROUND AND PRELIMINARIES

This section formally describes the parts of RDF and SPARQL that are relevant to this paper. We represent RDF data and the conjunctive fragment of SPARQL queries as graphs, as the example in Figure 3b, because a graph-based representation is suitable for describing how queries are evaluated in *chameleon-db*, where query evaluation is modeled as a subgraph isomorphism problem [28].

Assume two disjoint, countably infinite sets $\mathcal{U}$ (URIs) and $\mathcal{L}$ (literals)[2]. URIs uniquely denote Web resources or features of Web resources. Literals denote values such as strings, natural numbers and booleans. Then, an *RDF triple* is a 3-tuple from the set $\mathcal{T} = \mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$.

DEFINITION 1. *An* RDF graph *is a directed, labeled multi-graph $G = (V, E)$ where: (i) the vertices (V) are URIs or literals such that $V \subset (\mathcal{U} \cup \mathcal{L})$; (ii) the directed, labeled edges (E) are RDF triples such that $E \subset (V \times \mathcal{U} \times V) \cap \mathcal{T}$; and (iii) each vertex $v \in V$ appears in at least one edge, where for each edge $(s, p, o) \in E$, s is the source of the edge, p is the label, and o is the target of the edge. Hereafter, we use $V(G)$ and $E(G)$ to denote the set of vertices and the set of edges of an RDF graph, respectively.*

To define queries, we assume a countably infinite set of variables $\mathcal{V}$ that is disjoint from both $\mathcal{U}$ and $\mathcal{L}$. It is possible to restrict the values that can be bound to a variable using a filter expression.

DEFINITION 2. *A* filter expression *is an expression of the form $?x \circ c$ where $?x \in \mathcal{V}$, $c \in (\mathcal{U} \cup \mathcal{L})$, and $\circ \in \{=, <, \leq, >, \geq\}$.*

Next, we define what we consider as the most basic class of SPARQL queries within our framework. Similar to RDF graphs, we use a graph-based representation.

DEFINITION 3. *A* constrained pattern graph (CPG) *is a directed, labeled multi-graph $Q = (\hat{V}, \hat{E}, R)$ where: (i) the vertices ($\hat{V}$) are variables, URIs, or literals such that $\hat{V} \subset \mathcal{V} \cup \mathcal{U} \cup \mathcal{L}$; (ii) the directed, labeled edges ($\hat{E}$) are 3-tuples such that $\hat{E} \subset \hat{V} \times (\mathcal{V} \cup \mathcal{U}) \times \hat{V}$, where for each edge $(\hat{s}, \hat{p}, \hat{o}) \in \hat{E}$, $\hat{s}$ is the source of the edge, $\hat{p}$ is the label, and $\hat{o}$ is the target of the edge; (iii) each vertex $\hat{v} \in \hat{V}$ appears in at least one edge; and (iv) the graph is augmented with a finite set of filter expressions R.*

Figure 1 depicts three CPGs. Note that these simple examples do not contain any filter expressions. CPGs correspond to the AND-FILTER fragment of SPARQL as defined by Pérez et al. [24], where each edge in a CPG represents a triple pattern. CPGs can be combined using operators AND, UNION, and OPT [24]. Thus, our notion of a SPARQL query is defined recursively as follows.

[2] For simplicity, we ignore blank nodes in our discussions; however, our formalization can be extended to support them.

DEFINITION 4. *Any CPG is a* SPARQL query. *If S_1 and S_2 are SPARQL queries, and F is a filter expression, then expressions $(S_1$ AND $S_2)$, $(S_1$ UNION $S_2)$, $(S_1$ OPT $S_2)$, and $(S_1$ FILTER $F)$ are SPARQL queries.*

We now define the *semantics* of these queries. As a basis for this definition we use standard SPARQL solution mappings [24].

DEFINITION 5. *A* solution mapping *is a mapping $\mu : \mathcal{V}' \to (\mathcal{U} \cup \mathcal{L})$, where $\mathcal{V}'$ is a finite subset of $\mathcal{V}$. Two solution mappings μ_1 and μ_2 are* compatible, *denoted by $\mu_1 \sim \mu_2$, if $\mu_1(?x) = \mu_2(?x)$ for all variables $?x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.*

Figures 3e to 3i depict sets of solution mappings in a tabular form. Sets of solution mappings can be combined based on the following standard SPARQL algebra operations [24].

DEFINITION 6. *Let Ω_1 and Ω_2 be two sets of solution mappings and let F be a filter expression. Union ($\cup$), join ($\bowtie$), difference ($\setminus$) and selection (Θ) are defined as follows:*

$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},$$
$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\},$$
$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2 : \mu \not\sim \mu'\},$$
$$\Theta_F(\Omega_1) = \{\mu \in \Omega_1 \mid \mu \text{ satisfies } F\},$$

where a solution mapping μ satisfies a filter expression $?x \circ c$, if $?x \in \text{dom}(\mu)$ and $\mu(?x) \circ c$[3].

For our most basic type of SPARQL query, that is, a CPG, solution mappings can be computed from subgraphs of a queried RDF graph that match the CPG. This is very similar to the notion of a "match" in the context of subgraph isomorphism [28] except for the presence of variables in SPARQL. To accommodate this difference, we first introduce *compatibility* between an edge in an RDF graph and an edge in a CPG (Definition 7). Informally, two edges are compatible if they have the potential to match. Formally:

DEFINITION 7. *Let $e = (s, p, o) \in E$ be an edge in an RDF graph $G = (V, E)$, and let $\hat{e} = (\hat{s}, \hat{p}, \hat{o}) \in \hat{E}$ be an edge in a CPG $Q = (\hat{V}, \hat{E}, R)$. Edges e and $\hat{e}$ are* compatible *if either (i) $p = \hat{p}$, or (ii) $\hat{p} \in \mathcal{V}$.*

Using the notion of edge compatibility, we define a match between a CPG and an RDF graph as surjection from the edges (and vertices) of a CPG onto the edges (and vertices) of an RDF graph (possibly a subgraph of the queried RDF graph) such that corresponding edges are compatible and the source (and the target) vertices of a pair of corresponding edges are also mapped onto.

DEFINITION 8. *Let $G = (V, E)$ be an RDF graph, and let $Q = (\hat{V}, \hat{E}, R)$ be a CPG. Given a solution mapping μ, G μ-matches Q if (i) $\text{dom}(\mu)$ is the set of variables mentioned in Q, (ii) μ satisfies all filter expressions in R, and (iii) there exist two surjective functions $M_V : \hat{V} \to V$ and $M_E : \hat{E} \to E$ such that:*

- *for each $(\hat{v}_1, v_2) \in \hat{V} \times V$ with $M_V(\hat{v}_1) = v_2$: if $\hat{v}_1 \in \mathcal{V}$, then $\mu(\hat{v}_1) = v_2$, else $\hat{v}_1 = v_2$;*

[3] We assume that each binary relation $\circ \in \{=, <, \leq, >, \geq\}$ over sets $\mathcal{U}$ and $\mathcal{L}$ is defined in the obvious way according to the SPARQL specification [25].

- *for each* $(\hat{e}_1, e_2) \in \hat{E} \times E$ *with* $M_E(\hat{e}_1) = e_2$*:* $\hat{e}_1 = (\hat{s}_1, \hat{p}_1, \hat{o}_1)$ *and* $e_2 = (s_2, p_2, o_2)$*, (a)* $\hat{e}_1$ *and* e_2 *are compatible and if* $\hat{p}_1 \in \mathcal{V}$*, then* $p_2 = \mu(\hat{p}_1)$*, and (b) if* $M_V(\hat{s}_1) = s_2$*, then* $M_V(\hat{o}_1) = o_2$*.*

G matches Q *if there exists a solution mapping* μ *such that* G μ*-matches* Q*.*

Putting it all together, we define the expected result of evaluating a SPARQL query over an RDF graph as follows.

DEFINITION 9. *The* result *of a SPARQL query* S *over an RDF graph* $G = (V, E)$*, denoted by* $[\![S]\!]^*_G$*, is defined recursively as follows:*

1. *If* S *is a CPG* Q*, then* $[\![S]\!]^*_G = \{\mu \mid G'$ *is a subgraph of* G *and* G' μ*-matches* $Q\}$*;*
2. *If* S *is* S_1 AND S_2*, then* $[\![S]\!]^*_G = [\![S_1]\!]^*_G \bowtie [\![S_2]\!]^*_G$*;*
3. *If* S *is* S_1 UNION S_2*, then* $[\![S]\!]^*_G = [\![S_1]\!]^*_G \cup [\![S_2]\!]^*_G$*;*
4. *If* S *is* S_1 OPT S_2*, then* $[\![S]\!]^*_G = ([\![S_1]\!]^*_D \bowtie [\![S_2]\!]^*_G) \cup ([\![S_1]\!]^*_G \setminus [\![S_2]\!]^*_G)$*.*
5. *If* S *is* S_1 FILTER F*, then* $[\![S]\!]^*_G = \Theta_F([\![S]\!]^*_G)$*.*

The query model that we support, as codified in Definition 9, corresponds to the query model in SPARQL 1.0 specification except for complex filter expressions involving built-in functions.

4. QUERY EVALUATION

In this section, we present how *chameleon-db* computes the results of a SPARQL query as specified in Definition 9. For the implementation of joins ($\bowtie$), unions ($\cup$) and set difference ($\setminus$), we use existing techniques [21]. Therefore we primarily focus on Step-1 of Definition 9, which defines query results over all subgraphs of an RDF graph that match a CPG. With only subtle differences, this is the subgraph matching (or isomorphism) problem, therefore we adapt an existing algorithm [28] in our implementation.

We propose a novel framework for evaluating CPGs, which we call *partition-restricted evaluation* (PRE). Given a CPG Q and an RDF graph G, instead of evaluating the CPG over the whole graph, we partition the graph and describe a method that produces the expected result, that is, $[\![Q]\!]^*_G$, over the partitioned graph. For several reasons, this method is much more suitable for a workload-aware system such as *chameleon-db*. First, query processing can be more easily localized since irrelevant partitions can be pruned out early in the evaluation of a CPG. Second, for partitions that are considered relevant, indexes can be built on more than one attribute and even combinations of attributes, thus, enabling better pruning, whereas, an index on a single attribute will be sufficient for the remaining partitions. Third, PRE enables the results to be computed in isolation on each partition, thereby enabling the system to concurrently update the partitioning while queries are being executed on other partitions of the RDF graph.

4.1 Partition-Restricted Evaluation

In this section, we define what a partitioning of an RDF graph is, and then we introduce two new operations: *partitioned-match* (Definition 11) and *prune* (Definition 12), which are the fundamental building blocks of PRE. Finally, we discuss how CPGs are evaluated using PRE.

DEFINITION 10. *Given an RDF graph* $G = (V, E)$*, a* partitioning *of* G *is a set of RDF graphs* $\mathbb{P} = \{P_1, \ldots, P_m\}$ *such that (i) each* P_i *is a subgraph of* G*, (ii)* P_i*'s are edge disjoint, (iii)* $E(G) = \bigcup_{P_i \in \mathbb{P}} E(P_i)$*, and (iv)* $V(G) = \bigcup_{P_i \in \mathbb{P}} V(P_i)$*.*

In a partitioning of an RDF graph, vertices can be shared between partitions whereas each edge always belongs to a single partition, as shown in Figures 3c and 3d. Next, we define the operations *partitioned-match* and *prune*.

DEFINITION 11. *Given a CPG* Q *and a partitioning* $\mathbb{P}$ *of an RDF graph, the* partitioned-match *of* Q *over* $\mathbb{P}$*, denoted as* $[\![Q]\!](\mathbb{P})$*, is defined as* $[\![Q]\!](\mathbb{P}) = \bigcup_{P \in \mathbb{P}} [\![Q]\!]^*_P$*.*

DEFINITION 12. *Given a partitioning* $\mathbb{P}$ *of an RDF graph and a CPG* Q*, a* prune *of* $\mathbb{P}$ *with respect to* Q*, which is denoted by* $\sigma_Q(\mathbb{P})$*, is defined as a subset* $\mathbb{P}_r$ *of* $\mathbb{P}$ *such that* $[\![Q]\!]^*_P = \emptyset$ *for all* $P \in (\mathbb{P} \setminus \mathbb{P}_r)$*.*

Now, we expand Step-1 of Definition 9 and discuss how we apply our partition-restricted evaluation method to compute the results of a CPG. In other words, we aim to replace $[\![Q]\!]^*_G$ with a semantically equivalent expression that operates on a partitioning of an RDF graph (or subsets of partitions thereof), instead of the entire graph. We call these type of expressions PRE expressions. Naturally, if Q is a CPG, and $\mathbb{P}$ is a partitioning of an RDF graph, $[\![Q]\!](\mathbb{P})$ and $[\![Q]\!](\sigma_Q(\mathbb{P}))$ are PRE expressions because they both operate on a set of partitions. Furthermore, if M_1 and M_2 are PRE expressions, then we define $(M_1 \bowtie M_2)$ and $(M_1 \cup M_2)$ to be PRE expressions.

For performance reasons, our partitioning algorithm, which will be introduced in Section 5, aims to find a partitioning $\mathbb{P}$ of an RDF graph G such that $[\![Q]\!]^*_G = [\![Q]\!](\mathbb{P})$ for most queries in the workload. However, this condition may not hold for every CPG Q. The reason is that the partitioned-match operation ignores subgraphs of G (i.e., the queried RDF graph) that match the CPG but that span more than one partition in a partitioning $\mathbb{P}$ of G. In that case, we may need to decompose Q into a set of smaller CPGs, and then use this set to construct a more complex PRE expression M such that $[\![Q]\!]^*_G = M$. A decomposition that we are particularly interested in is the *trivial decomposition* of a CPG, in which the CPG is decomposed into single-edge CPGs (Definition 13), because with this type of decomposition we can guarantee the construction of a PRE expression M such that $[\![Q]\!]^*_G = M$ for any CPG Q and any partitioning $\mathbb{P}$ of an RDF graph G (Theorem 1). We call this expression the *baseline* PRE *expression* (Definition 14).

DEFINITION 13. *Given a CPG* $Q = (\hat{V}, \hat{E}, R)$*, if the filter expressions in* R *mention only the variables in* Q*, then the* trivial decomposition *of* Q *is defined as the set* $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$ *of CPGs, where each* $Q_i \in \mathcal{Q}$ *contains exactly one edge, the set of edges in each* Q_i *are disjoint,* $\hat{V} = \bigcup_{i=1}^{k} V(Q_i)$*,* $\hat{E} = \bigcup_{i=1}^{k} E(Q_i)$*,* $R = \bigcup_{i=1}^{k} R(Q_i)$*, and for each* $i \in \{1, \ldots, k\}$*, filter expressions in* $R(Q_i)$ *mention only the variables in* Q_i*.*

DEFINITION 14. *Let* Q *be a CPG, let* G *be an RDF graph, and let* $\mathbb{P}$ *be a partitioning of* G*. If* $\{Q_1, \ldots, Q_k\}$ *is the trivial decomposition of* Q*, then the* baseline PRE expression *for* Q *over* $\mathbb{P}$ *is* $[\![Q_1]\!](\mathbb{P}) \bowtie \cdots \bowtie [\![Q_k]\!](\mathbb{P})$*.*

THEOREM 1. *Given a CPG Q, an RDF graph G and a partitioning $\mathbb{P}$ of G, if the baseline expression is defined for Q over $\mathbb{P}$[4], then $[\![Q]\!]^*_G = M$, where M is the baseline* PRE *expression for Q over $\mathbb{P}$[5].*

From the baseline PRE expression, it is possible to generate a collection of equivalent expressions that all produce the expected result of a CPG. Consequently, there is an opportunity for cost-based optimization. Deferring that discussion to Section 4.2, we describe how *chameleon-db* computes the results of a CPG. First, for a given CPG, a PRE expression is generated (which of the equivalent expressions the system chooses is the topic of Section 4.2). Figure 4a illustrates a tree representation of such a PRE expression. Then, each sub-expression of the form $\sigma_{Q_i}(\mathbb{P})$ is evaluated by pruning out the irrelevant partitions using the partition indexes (Figure 2). Consequently, each sub-expression of the form $[\![Q_i]\!](\sigma_{Q_i}(\mathbb{P}))$ is simplified to $[\![Q_i]\!](\mathbb{P}_i)$, where $\mathbb{P}_i \subseteq \mathbb{P}$. Then, for each resulting sub-expression $[\![Q_i]\!](\mathbb{P}_i)$, (i) the sub-expression is evaluated in isolation on each partition using a standard subgraph matching technique [28], and (ii) the results from each evaluation are unioned. In the subsequent steps, intermediate tuples from the evaluation of each sub-expression are joined or unioned according to the standard definitions in SPARQL algebra (cf., Definition 6).

To perform subgraph matching within each partition, we represent each RDF graph in the partitioning as an adjacency list and use a variation of Ullmann's algorithm [28]. For each vertex $\hat{v}$ in the CPG, we compute candidate matching vertices in the RDF graph. If $\hat{v}$ is a URI or literal, one can directly lookup the vertex in the adjacency list. Otherwise, if $\hat{v}$ is a variable, we rely on the labels of the edges that are incident on $\hat{v}$ to prune the search space. Of course, it is possible to build an index (other than adjacency list) over each partition to facilitate subgraph matching (cf., *gStore* [34]), but that is outside the scope of this paper.

4.2 Query Rewrite

We use the baseline PRE expression to generate a set of equivalent expressions that all produce the expected result of a CPG. Then, we use a cost model to choose the one that has the best estimated cost. In our model, we define cost as a function of the number of *dormant* tuples that are processed during query evaluation. For simplicity, we currently ignore disk related I/O costs. Therefore, in this paper, we rely on a heuristic which states that with increasing number of join operations within a PRE expression, one expects the number of dormant tuples to increase. Later in this section, we use an example to illustrate the reasoning behind our heuristic and in Section 7, we experimentally validate our observation. Future work will improve the cost function.

To realize the aforementioned logical query optimization scheme, we introduce equivalence rules that are in two categories: generic and conditional. Generic rules are applicable irrespective of how the RDF graph is partitioned, whereas the applicability of a conditional rule depends on whether the partitioning satisfies certain conditions.

Table 2 lists our equivalence rules. Assuming Q_A and Q_B are two CPGs, let $\mathbb{P}$ be a partitioning of an RDF graph, and let $\mathbb{P}_1, \ldots, \mathbb{P}_m$ be subsets thereof ($\mathbb{P}_i \subseteq \mathbb{P}$ for all $i \in \{1, \ldots, m\}$). Rules 1–3 are specific to the partition-match operation, whereas rules 4–9 are derived from SPARQL algebra [3]. Rules that are marked with an asterisk (*) are conditional. Observe that the expansion rule relies on a condition that is independent of the way the graph is partitioned. In other words, for any partitioning $\mathbb{P}$ of an RDF graph, one may generate some $\mathbb{P}_1, \ldots, \mathbb{P}_m$, such that the condition is satisfied; hence, its classification as a generic rule. On the contrary, we shall see that the conditions in join elimination and join reduction are directly related to the way the graph is partitioned; therefore, they need to be checked every time a query is evaluated. The following theorems formalize these conditions.

	Name	Equivalence Rules	Condition
1	Expansion	$[\![Q_A]\!](\mathbb{P}) = \bigcup_{i=1}^{m} [\![Q_A]\!](\mathbb{P}_i)$	$\bigcup_{i=1}^{m} \mathbb{P}_i = \mathbb{P}$
2	Join elimination*	$[\![Q_A]\!](\mathbb{P}_1) \bowtie [\![Q_B]\!](\mathbb{P}_2) = \emptyset$	Thm. 2
3	Join reduction*	$[\![Q_A]\!](\mathbb{P}_1) \bowtie [\![Q_B]\!](\mathbb{P}_1) = [\![Q_A \oplus Q_B]\!](\mathbb{P}_1)$	Thm. 3
4	Identity ($\bowtie$)	$\Omega_1 \bowtie \emptyset = \emptyset \bowtie \Omega_1 = \emptyset$	$\Omega_1, \Omega_2, \Omega_3$
5	Identity ($\cup$)	$\Omega_1 \cup \emptyset = \emptyset \cup \Omega_1 = \Omega_1$	are sets of
6	Associativity ($\bowtie$)	$\Omega_1 \bowtie (\Omega_2 \bowtie \Omega_3) = (\Omega_1 \bowtie \Omega_2) \bowtie \Omega_3$	solution mappings
7	Associativity ($\cup$)	$\Omega_1 \cup (\Omega_2 \cup \Omega_3) = (\Omega_1 \cup \Omega_2) \cup \Omega_3$	
8	Distributivity ($\bowtie$ over $\cup$)	$\Omega_1 \bowtie (\Omega_2 \cup \Omega_3) = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \bowtie \Omega_3)$	
9	Reflexivity	$\Omega_1 \bowtie \Omega_2 = \Omega_2 \bowtie \Omega_1$	

Table 2: Equivalence rules that are applicable to PRE expressions ($\mathbb{P}_1, \ldots, \mathbb{P}_m$ represent sets of RDF graphs).

THEOREM 2. *Given a partitioning $\mathbb{P}$ of an RDF graph and two CPGs Q_A and Q_B with $V(Q_A) \cap V(Q_B) \neq \emptyset$, let $I = \bigcup_{(P_i,P_j)\in\mathbb{P}_1\times\mathbb{P}_2} V(P_i) \cap V(P_j)$, where $\mathbb{P}_1, \mathbb{P}_2 \subseteq \mathbb{P}$. Then, $[\![Q_A]\!](\mathbb{P}_1) \bowtie [\![Q_B]\!](\mathbb{P}_2) = \emptyset$ if for each vertex $v \in I$, there exists a vertex $\hat{v} \in V(Q_A) \cap V(Q_B)$ and an edge $\hat{e} \in inc(Q_A, \hat{v}) \cup inc(Q_B, \hat{v})$ such that $\hat{e}$ is not compatible (cf., Definition 7) with any edge from $\bigcup_{P\in\mathbb{P}_1\cup\mathbb{P}_2} inc(P, v)$, where $inc(G, v)$ denotes the set of edges that are incident on a vertex v.*

DEFINITION 15. *Given two CPGs Q_A and Q_B, we define the* concatenation *of Q_A and Q_B, denoted by $Q_A \oplus Q_B$, as a CPG $Q = (\hat{V}, \hat{E}, R)$ such that (i) $\hat{V} = V(Q_A) \cup V(Q_B)$, (ii) $\hat{E} = E(Q_A) \cup E(Q_B)$ and (iii) $R = R(Q_A) \cup R(Q_B)$.*

THEOREM 3. *Given a partitioning $\mathbb{P}$ and two CPGs Q_A and Q_B with $V(Q_A) \cap V(Q_B) \neq \emptyset$, $[\![Q_A]\!](\mathbb{P}) \bowtie [\![Q_B]\!](\mathbb{P}) = [\![Q_A \oplus Q_B]\!](\mathbb{P})$ if for each vertex v, where $\left|cont(\mathbb{P}, v)\right| > 1$, (where $cont(\mathbb{P}, v)$ denotes the subset of partitions in $\mathbb{P}$ that contain v), either*

- *(i) there exists a vertex $\hat{v} \in V(Q_A) \cap V(Q_B)$ and an edge $\hat{e} \in inc(Q_A, \hat{v}) \cup inc(Q_B, \hat{v})$ such that $\hat{e}$ is not compatible with any edge from $\bigcup_{P\in\mathbb{P}} inc(P, v)$, or*
- *(ii) there exists a single partition $\bar{P} \in \mathbb{P}$ such that for every edge $e \in \bigcup_{P\in\mathbb{P}} inc(P, v)$ and for every vertex $\hat{v} \in V(Q_A) \cap V(Q_B)$, if e is compatible with an edge from $inc(Q_A, \hat{v}) \cup inc(Q_B, \hat{v})$, then $cont(\mathbb{P}, e) = \{\bar{P}\}$.*

Let us revisit the example from Section 1 and demonstrate how join reduction can be applied to the partition-restricted evaluation of query $?w \xrightarrow{A} ?x \xrightarrow{B} ?y \xrightarrow{C} ?z$ over the partitioning in Figure 3c. The baseline PRE expression for this CPG is $[\![Q_1]\!](\mathbb{P}) \bowtie [\![Q_2]\!](\mathbb{P}) \bowtie [\![Q_3]\!](\mathbb{P})$, where

[4]Note that if the baseline expression is not defined (because Q contains filter expressions that mention variables not in Q), then $[\![Q]\!]^*_G$ is unsatisfiable, hence, query result is always empty.

[5] The proofs are in the Appendix.

$inc(\mathbb{P}, v_1)$	$inc(\mathbb{P}, v_2)$	$inc(\mathbb{P}, v_3)$	in	$inc(\mathbb{P}, v_4)$
$v_1 \xrightarrow{A} v_2$	$v_2 \xleftarrow{A} v_1$	$v_3 \xrightarrow{A} v_9$	P_1	$v_4 \xrightarrow{B} v_{11}$
$v_1 \xrightarrow{A} v_6$	$v_2 \xrightarrow{B} v_3$	$v_3 \xleftarrow{B} v_2$	P_2	$v_4 \xleftarrow{C} v_3$
$v_1 \xrightarrow{B} v_5$	$v_2 \xrightarrow{B} v_8$	$v_3 \xrightarrow{C} v_4$	P_2	$v_4 \xrightarrow{C}$ "10"
	$v_2 \xrightarrow{C} v_7$	$v_3 \xrightarrow{C} v_{10}$	P_2	

Table 3: Incident edges on v_1–v_4 in Partitioning-A (Fig. 3c)

$?w \xrightarrow{A} ?x$, $?x \xrightarrow{B} ?y$, $?y \xrightarrow{C} ?z$ are the three CPGs Q_1, Q_2 and Q_3 in the trivial decomposition of the query, and $\mathbb{P}$ consists of P_1 and P_2 in Figure 3c. For simplicity, we ignore prune operations for now. This baseline PRE expression can be rewritten as $[\![Q_1]\!](\mathbb{P}) \bowtie [\![Q_2 \oplus Q_3]\!](\mathbb{P})$ if $[\![Q_2]\!](\mathbb{P}) \bowtie [\![Q_3]\!](\mathbb{P}) = [\![Q_2 \oplus Q_3]\!](\mathbb{P})$. For the given partitioning, the four vertices v_1, v_2, v_3 and v_4 exist in multiple partitions. Therefore, we need to check the conditions in Theorem 3. Note that $inc(Q_2, ?y) \cup inc(Q_3, ?y)$ consists of two edges, namely, $(?x, B, ?y)$ and $(?y, C, ?z)$. Condition (i) holds for v_1 because $(?y, C, ?z)$ is not compatible with any of the edges in $inc(\mathbb{P}, v_1)$, which is illustrated in Table 3. For v_2, $(?y, B, ?z)$ is not compatible with any of the edges in $inc(\mathbb{P}, v_2)$ due to the direction of edges. The same argument holds for v_4. As for v_3, both $(?x, B, ?y)$ and $(?y, C, ?z)$ have at least one compatible edge, therefore, we also need to check condition (ii) for v_3. Since all compatible edges are from the same partition, namely, P_2, the baseline PRE expression can be simplified to $[\![Q_1]\!](\mathbb{P}) \bowtie [\![Q_2 \oplus Q_3]\!](\mathbb{P})$. Continuing with the process, the expression can be further simplified to $[\![Q_1 \oplus Q_2 \oplus Q_3]\!](\mathbb{P})$, which may be more efficient because in the latter case three predicate patterns, namely, $\langle A \rangle$, $\langle B \rangle$ and $\langle C \rangle$, can be collectively used to prune out partitions, making the evaluation more selective. In fact, as Figure 5c in Section 7 suggests, when we analyzed all of the execution logs generated during our experimentation with *chameleon-db*, we found out that as the number of join operations in a query plan increases, it generally causes an increase in the number of triples that are processed.

The algorithm for rewriting a baseline PRE expression proceeds in three phases. We describe this algorithm using the example illustrated in Figure 4. Consider a baseline PRE expression: $[\![Q_1]\!](\sigma_{Q_1}(\mathbb{P})) \bowtie [\![Q_2]\!](\sigma_{Q_2}(\mathbb{P})) \bowtie [\![Q_3]\!](\sigma_{Q_3}(\mathbb{P}))$ (Figure 4a). First, the joins in the baseline expression are reordered according to their estimated selectivities [27]. Second, by using generic equivalence rules, the expression is transformed into a canonical form. A PRE expression is in canonical form if it consists of the union of a set of sub-expressions $T_1 \cup \cdots \cup T_m$, where each sub-expression is made up of the exact same set of partitioned-match operations, which differ only in the partitions they operate on. For this purpose, each prune operation is evaluated, producing multiple sets of partitions with one set for each prune operation (Figure 4b). As illustrated in Figure 4b, these sets of partitions are factorized into maximal common subsets such that factorization produces as few segments as possible. Then, each partitioned-match operation is expanded across the corresponding subsets of partitions using Rule 1. Rules 4–9 are applied to the nodes of the expression-tree in a bottom-up fashion, which is repeated until no further rewriting is possible. At this stage, the canonical expression is produced (Figure 4c).

In the third phase, each sub-expression in the canonical form is optimized independently using conditional rules (i.e., Rules 2–3) as well as Rules 4–9. For query optimization,

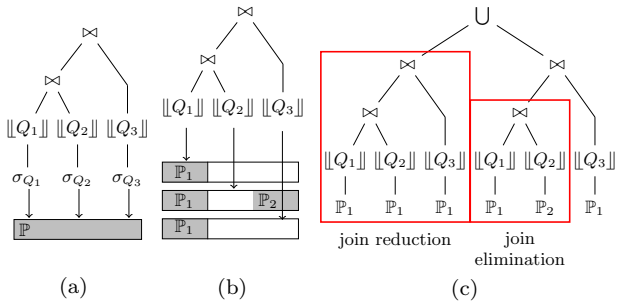


Figure 4: Illustration of query rewriting and optimization.

we rely on the aforementioned heuristic that decomposing a CPG is likely to increase the number of dormant tuples that need to be processed. For instance, considering our earlier example about join reduction, we prefer $[\![Q_1 \oplus Q_2 \oplus Q_3]\!](\mathbb{P})$ over $[\![Q_1]\!](\mathbb{P}) \bowtie [\![Q_2]\!](\mathbb{P}) \bowtie [\![Q_3]\!](\mathbb{P})$. In this regard, join reduction and join elimination are applied recursively to the nodes of each sub-expression until the number of join operations are reduced as much as possible. The right-hand side of union is eliminated using join elimination (Figure 4c) and the remaining expression is simplified to $[\![Q_1 \oplus Q_2 \oplus Q_3]\!](S_1)$ using join reduction.

4.3 Discussion

Partition-restricted evaluation offers important advantages. First of all, it is possible to compute a partitioning such that most of the queries in a workload do not require join operations across partitions. We demonstrate that this has significant impact on performance because it helps reduce the number of dormant tuples. Even in the worst case when a query cannot be rewritten as any other expression, the baseline PRE expression still guarantees correct results, which provides the flexibility to compute a partitioning that favors the most frequent query patterns in a workload, and update the partitioning as these frequencies change.

Second, by enforcing PRE, we limit the scope of subgraph matching to contents within each partition. Although this introduces query rewriting and optimization overhead, it provides isolation, whose reward is much higher (as our experiments will show). Since partitions are now truly isolated from each other, new partitions can be added and existing partitions can be split or merged without affecting the integrity of query evaluation on other parts of the graph. Consequently, query evaluation can be more easily interleaved with re-partitioning of the graph, which is one of the key objectives of our work. There is also an opportunity for parallelization—subgraph matching can be performed concurrently on multiple partitions.

Third, when determining whether or not a partition contains a subgraph that matches a query, the index needs to consider only the subgraphs that reside within a single partition. For this reason, by manipulating how the graph is partitioned, *chameleon-db* can execute linear queries or more complex queries efficiently without the heavier indexing overhead of other systems. Needless to say, having this update-friendly and memory-efficient index is a requirement for a system like *chameleon-db* that constantly needs to reorganize its layout for changing workloads.

5. WORKLOAD-AWARE PARTITIONING

Our partitioning algorithm is workload-driven: without making any *a-priori* assumptions about how the partitioning should be, *chameleon-db* partitions the graph based on the workload. Initially, every edge of the RDF graph is placed in its own partition, which is not necessarily an optimal partitioning for any workload. However, as soon as some queries are executed, the system gains a better understanding of the workload, and periodically computes a better partitioning of the RDF graph.

To facilitate the computation of a suitable partitioning, upon the execution of a CPG, we annotate each distinct subgraph that matches the CPG with a unique label and a timestamp. Each annotation is of the form $\langle qid, sid, t\rangle$, where (i) *qid* is a unique identifier generated by *chameleon-db* for every CPG that comes in, (ii) *sid* is a unique identifier for the corresponding subgraph, and (iii) t is a timestamp. Since matching subgraphs can be overlapping, for each annotated edge of the RDF graph we maintain a (FIFO) queue. At every repartitioning, annotations that are too old to fit within a user-defined window are deleted from the queue, after which the partitioning is computed based on the remaining annotations.

In a query workload, the same query structure may have multiple instantiations. For example, both $Prof_1 \xrightarrow{A} ?x \xrightarrow{B} ?y \xrightarrow{C} VLDB14$ and $Prof_2 \xrightarrow{A} ?x \xrightarrow{B} ?y \xrightarrow{C} SIGMOD14$ are instantiations of the same linear query $?w \xrightarrow{A} ?x \xrightarrow{B} ?y \xrightarrow{C} ?z$. Our partitioning algorithm detects if a particular query structure occurs frequently, and if so, it tunes the partitioning to better support instantiations of that query structure. For this purpose, we generalize every CPG that is evaluated on the system to its structural form, which means that every URI or literal vertex in the CPG is replaced with a distinct variable as shown above. Second, we determine the frequency of occurence of each query structure. Finally, before periodic repartitioning takes place, we actually execute the most frequent structures and annotate the RDF graph based on their results—just as we would do for any other query in the workload—to generate the labels also for the generalized CPGs.

In the remainder of this section, we describe our partitioning algorithm. First, we formulate an objective function (Section 5.1), and then we explain the algorithm (Section 5.2). Finally, we discuss how partitions are updated following each partitioning (Section 5.3).

5.1 Partitioning Objectives

Given an RDF graph G and a CPG Q (with a specific *qid*), let G^* be the subgraph of G that consists of all edges in G labeled with $\langle qid, *, *\rangle$, where wildcard (*) can take any value. Since $[\![Q]\!]^*_G = [\![Q]\!]^*_{G^*}$, we call the edges in G^* as *relevant*, and the rest of the edges in G as *irrelevant* to the evaluation of Q. For example, for the CPG in Figure 1b, the relevant edges are indicated by the shaded region in Figure 3b. Recall that from a performance standpoint, our objective is to eliminate all dormant tuples from query evaluation. How well this can be accomplished depends on two conditions: (i) every partition that is not pruned (see Section 6) should contain as few irrelevant edges as possible, and (ii) every subgraph that matches a CPG should be distributed over as few partitions as possible.

The first condition is intuitive. The earlier irrelevant triples are pruned out in query evaluation, the less likely it is for the dormant tuples to accumulate as partial results are joined. The second condition has to do with the way partition-restricted evaluation works. If a subgraph that matches a CPG is distributed across more than one partition, the CPG needs to be decomposed to obtain the correct results. A side-effect of decomposition is that each subquery generally contains fewer selection predicates than the original query. Therefore, evaluation of each subquery is likely to produce more dormant tuples (e.g., Figure 3e–Figure 3i). Using these two conditions as guidelines, we quantify the "goodness" of a partitioning as a combination of two measures: *segmentation* and *minimality*. Segmentation is a measure of how distributed subgraphs that match a query across the partitions of an RDF graph are. Minimality indicates how minimal partitions are with respect to those subgraphs that match a CPG.

DEFINITION 16. *Given a partitioning $\mathbb{P}$ of an RDF graph G, let Γ^Q_G denote all distinct subgraphs of G that match a CPG Q, and let $E^* = \bigcup_{G' \in \Gamma^Q_G} E(G')$. Then,* segmentation *and* minimality *of $\mathbb{P}$ with respect to Q are defined as:*

$$segm^Q_{\mathbb{P}} = \left|\left\{(G', P) \in \Gamma^Q_G \times \mathbb{P} \mid E(G') \cap E(P) \neq \emptyset\right\}\right| - \left|\Gamma^Q_G\right|$$

$$minim^Q_{\mathbb{P}} = \left|E^*\right| \;/\; \left|\left\{E(P) \mid P \in \mathbb{P} \text{ and } E(P) \cap E^* \neq \emptyset\right\}\right|$$

The definitions of segmentation and minimality can be easily extended to a query workload $\mathbb{W} = \{Q^1, \ldots, Q^n\}$: $segm^{\mathbb{W}}_{\mathbb{P}} = \sum_{i=1}^{n} segm^{Q^i}_{\mathbb{P}}/|\mathbb{W}|$ and $minim^{\mathbb{W}}_{\mathbb{P}} = \sum_{i=1}^{n} minim^{Q^i}_{\mathbb{P}}/|\mathbb{W}|$.

Segmentation can take any positive real value, while minimality is always between [0, 1]. An ideal partitioning is one whose segmentation is minimal (0) and minimality takes the highest possible value (1) with respect to a query workload. We say that a partitioning is *completely segmented* if its segmentation is maximal with respect to a query workload.

5.2 Partitioning Algorithm

Given a query workload, we address the problem of computing a suitable partitioning of an RDF graph using a hierarchical clustering algorithm, which starts from a completely segmented partitioning, and successively merges partitions until the clustering objective is achieved. The clustering algorithm operates as follows:

1. Initially, each edge of the RDF graph resides in its own partition, which corresponds to a completely segmented partitioning for any workload;
2. The pair of partitions, whose merging improves segmentation the most, while causing the least trade-off in minimality, is identified;
3. Partitions found in Step 2 are merged, which results in a potential decrease in segmentation and/or minimality;
4. Steps 2–3 are repeated as long as the aggregate minimality of the partitioning is greater than a threshold.

It is important to note that segmentation and minimality measures are monotonically non-increasing within this algorithm. That is, whenever two partitions are merged, segmentation will potentially decrease because edges having the same *qid* and *sid* labels may be brought together. However, at the same time, edges having different *qid* labels may also be placed in the same partition, which does not affect segmentation, but reduces minimality. While we would like to

reduce segmentation, we would like to increase minimality. When partitions contain too many edges that are individually irrelevant to the execution of most of the queries, the overhead of subgraph matching within each partition can mask the benefits of reduced segmentation. Specific to our implementation, we observed that if the partitions contain on average more than 10 times as many irrelevant triples as there are relevant ones, performance of query evaluation starts to degrade. For this reason, we set the lower threshold on minimality as 0.1.

There are two reasons for choosing a hierarchical clustering algorithm. First, the way hierarchical clustering works is aligned with our clustering objectives. That is, partitions are merged one pair-at-a-time until a global objective is achieved. This is not true for centroid-based clustering [15] or spectral clustering [15]. Second, other algorithms such as k-means [15] require the number of clusters to be known in advance, which is not possible in our case.

We expect the final partitioning to be fine-grained since subgraphs that match the queries in the workload are likely to be comparable in size to the query graphs, which are relatively small. Furthermore, the final partitions are not likely to be much larger than these subgraphs due to the minimality threshold. Therefore, a bottom-up (agglomerative) approach can reach the clustering objective in fewer number of iterations than a top-down (divisive) approach (hence, the reason why we start with a completely segmented partitioning and employ agglomerative clustering).

A critical issue is to decide which pair of partitions to merge in each iteration. We define a distance function $\delta : \mathbb{P} \times \mathbb{P} \rightarrow [0, 1]$ over the partitions such that: (i) $\delta = 1$ is reserved for partitions that should not be merged; (ii) a smaller distance between two partitions implies that the decrease in segmentation is higher (with a lower trade-off in minimality) if these two partitions are merged.

To compute the pairwise distances between partitions, we rely on the annotations of edges in each pair of partitions, namely, the set of $\langle qid, sid, t\rangle$ 3-tuples. We define the distance between a pair of partitions as a combination of two Jaccard distances: δ_S is defined over the sets of subgraph identifiers (sid), and δ_Q is defined over the sets of query identifiers (qid). For any partition $P \in \mathbb{P}$, let $\pi_s(P)$ and $\pi_q(P)$ denote the set of subgraph identifiers and the set of query identifiers with which P is annotated, respectively. Given two partitions P_1 and P_2, the distances $\delta_S(P_1, P_2)$ and $\delta_Q(P_1, P_2)$ are respectively defined as:

$$\delta_S = 1 - \frac{|\pi_s(P_1) \cap \pi_s(P_2)|}{|\pi_s(P_1) \cup \pi_s(P_2)|}, \delta_Q = 1 - \frac{|\pi_q(P_1) \cap \pi_q(P_2)|}{|\pi_q(P_1) \cup \pi_q(P_2)|}.$$

The two distance functions are complementary. By merging P_1 with P_2, segmentation decreases by at least $|\pi_s(P_1) \cap \pi_s(P_2)|$, therefore, δ_S is more sensitive to predicting the expected change in segmentation. Likewise, $|\pi_q(P_1) \cup \pi_q(P_2)| - |\pi_q(P_1) \cap \pi_q(P_2)|$ is a more accurate approximation of the expected decrease in minimality, thus, δ_Q is more sensitive to changes in minimality, hence our reliance on a combination of both distances. However, in doing so, we pay particular attention to some race conditions. Specifically, the distance function is designed such that the following order, in which partitions are merged, is always preserved: (i) a pair of partitions whose $\delta_S = 0$ (which also implies that $\delta_Q = 0$) are merged before any other pair of partitions; (ii) partitions whose $\delta_S \neq 0$, but $\delta_Q = 0$, are merged next; (iii) finally, partitions whose $\delta_S \neq 0$ and $\delta_Q \neq 0$ are merged according to a combined distance $\delta = \alpha\delta_S + (1 - \alpha)\delta_Q$, where $\alpha = 0.5$.

Note that in the first two cases, minimality will not decrease because the two partitions that are merged have subgraphs that match only a single query. Hence, they are preferred over the third case, whose minimality is expected to decrease. Furthermore, even though the first and second cases are both guaranteed to reduce segmentation (without a compromise in minimality), the first case can achieve the same objective with smaller partitions, hence, it is preferred over the other. When two partitions P_1 and P_2 are merged, all distances between the new partition and any other existing partition P_x for which $\delta(P_1, P_x) < 1$ or $\delta(P_2, P_x) < 1$ need to be updated.

5.3 Updating the Partitions

Once a suitable partitioning is computed, the system realizes the transformation from the current partitioning to the desired one as a set of atomic update (i.e., deletion and insertion) operations on the set of partitions. Each operation has the property that before and after the operation, the database represents exactly the same RDF graph—only using a different partitioning.

The update operations are executed concurrently with the queries. The trick lies in the fact that according to PRE, once results are computed within a partition, query evaluation does not need to access that partition anymore, thus allowing that partition to be updated even though the query may be processing other partitions. In order to ensure that updates do not take place before a query has completely "consumed" the contents of a partition, we use a two-level locking scheme, whose details we omit in this paper.

6. INDEXING THE PARTITIONS

To prune the partitions irrelevant to a query, we employ a workload-aware, incremental indexing technique, which is similar to work on *database cracking* by Idreos et al. [14]. This specific indexing technique is orthogonal to the focus of this paper, therefore we restrict the discussion to a summary of our technique, which will be described fully elsewhere.

Before any query is evaluated, the partition index consists of only a doubly-linked list of pointers to all of the partitions. Initially, the index does not assume anything about the contents within each partition. However, as queries are evaluated, it uncovers more information about the partitions and indexes them as follows: When the first CPG, say Q^1, is evaluated, the list is divided into two segments, namely, $\mathbb{P}_l$ and $\mathbb{P}_r$ such that every partition that lies to the left of a pivot has matching subgraphs for the CPG, whereas those to the right do not. The overhead of restructuring the list is small. Note that the list has to be traversed anyway to compute the matching subgraphs; while doing so, partitions can be reordered in-place and in one-pass over the list—just like the partitioning step of the quicksort algorithm [9]. For the next CPG (Q^2), there are three possible scenarios:

- If Q^2 is the same as Q^1, query results can be computed directly from the partitions in $\mathbb{P}_l$, which does not need to be divided any further.
- If Q^2 is a strict supergraph of Q^1, a subset of partitions in $\mathbb{P}_l$ are relevant. Therefore, $\mathbb{P}_l$ needs to be traversed and divided into two segments: $\mathbb{P}_{l.l}$ and $\mathbb{P}_{l.r}$ like previously.

- For all other cases, potentially some partitions in both $\mathbb{P}_l$ and $\mathbb{P}_r$ have matching subgraphs of Q^2. Therefore, both lists need to be traversed and divided further.

As the list of partition pointers gets divided into multiple segments, we keep track of which segments are relevant to which queries using a decision tree. For every segment of partitions, which contain at least one matching subgraph for any CPG in the decision tree, we also maintain two second-tier indexes. The *vertex-index* is a hash table that maps URIs to the subset of partitions that contain vertices of that URI. The *range-index* keeps track of the minimum and maximum literal values within each partition for each distinct predicate, and it works as a filter when partitions are traversed. Consequently, as more queries are evaluated, the index becomes more effective in pruning partitions.

7. EXPERIMENTAL EVALUATION

chameleon-db is implemented in C++, and it consists of more than 30K lines of native source code. The system supports SPARQL 1.0 except for complex filter expressions that involve built-in functions. Join operations are currently implemented using the hash-join algorithm [10], which we extended with an adaptation of sideways information passing [20], which is a technique that is also used by x-RDF-3x. We use integer encodings to compress URIs and order-preserving compression to reduce the size of the literals [2]. The dictionary is stored in Berkeley DB [23]. In main memory, each partition is represented as an adjacency list and it is serialized on disk as a consecutive sequence of RDF triples that are sorted on their subject attributes.

7.1 Experimental Setup

In our evaluations, we used the Waterloo SPARQL Diversity Test Suite (WSDTS)[6], which we developed to measure how an RDF data management system performs across a wide spectrum of SPARQL queries with varying (i) structural characteristics, and (ii) selectivity classes. WSDTS differs from existing benchmarks in two aspects. First, existing benchmarks are designed to evaluate other features of systems such as how fast a system can do semantic inferencing, or how well different SPARQL features such as FILTER expressions or OPTIONAL patterns can be handled by a system, but they do not test directly (and in detail) how well a system performs across different query structures. Second, it has been argued [5, 32] that the query workloads existing benchmarks are too simple, repetitive and contain unrealistically selective triple patterns (in some cases retrieving a single triple). Third, as pointed out by Duan et al. [11], existing SPARQL benchmarks do not accurately represent the true structural diversity of the RDF data on the Web.

In WSDTS, data are generated such that some resources are more structured, while some are less structured, thereby providing the opportunity to generate queries that can have very different selectivities. Our query workload consists of 4 structural categories: LINEAR (L), STAR (S), SNOWFLAKE (F) and COMPLEX (C) queries. We generated our test query templates randomly while making sure that each query structure is sufficiently represented, and that the selectivities of the queries within each query structure vary. The full description of the WSDTS dataset and the 20 test queries that we used in our experiments are in the Appendix as well as at the aforementioned link.

[6] `https://cs.uwaterloo.ca/~galuc/wsdts/`

In our experiments, we use WSDTS with scale factor 100 and 1000, which generates 10 million and 100 million triples, respectively. The raw size of the largest dataset is 16 GBs. We use a commodity machine with AMD Phenom II $X4$ 955 3.20 GHz processor, 16 GB of main memory, and 100 GB of available hard disk space. We configured *chameleon-db*'s buffer pool to store a maximum of 100 thousand partitions, which corresponds to less than 1% of the partitions that can be generated using the test datasets. The underlying operating system is Ubuntu 12.04 LTS.

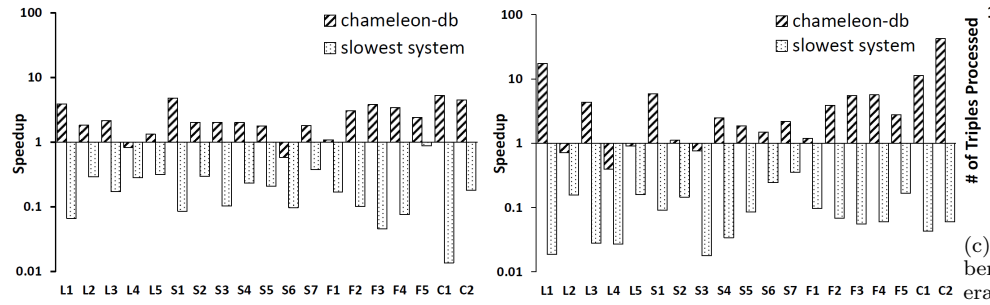
7.2 Static Performance

The purpose of our first experiment is to determine whether *chameleon-db* can achieve consistently good performance across a diverse set of SPARQL queries. We compare *chameleon-db* against a system that relies on exhaustive indexing (*x-RDF-3x* 0.3.7) [21], a graph-based RDF engine (*gStore*, 0.2) [34], and a relational-backed system (*Virtuoso* 6.1.7) [12]. In contrast to *chameleon-db*, none of these systems are workload-aware. For this reason, for each query template, first, we train *chameleon-db* with 3 training workloads (executed in a sequence), and until the end of the training phase, we let *chameleon-db* re-partition its database. Once training is over, we turn-off re-partitioning and execute the test workloads. For fairness, we also execute the training workloads on other systems prior to evaluating them. Moreover, as a baseline, we disable *chameleon-db*'s partitioning advisor and repeat our experiments.

Both the *test* or *training* workloads are randomly generated from 250 instantiations of a query template. In all of our experiments, we take measurements over 10 independent executions and report the average.

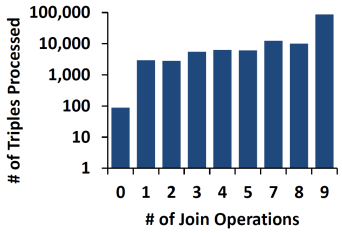
Figures 5a and 5b show the speedup for each query achieved by *chameleon-db* relative to the best system among others (Table 4). A speedup of x implies that *chameleon-db* performs x times as fast as the best system. For reference, we also display the speedup of the slowest system, which is less than or equal to 1 by definition. In these figures, we do not include those systems that have timed-out.

Our results highlight several important points. First, by employing a workload-aware approach, *chameleon-db* addresses some of the weaknesses of existing systems, and it can achieve consistently better performance across most WSDTS queries with a speedup of up to 2 orders in magnitude. The only major exceptions are $S6$ at 10M triples and $L4$ at 100M triples, which we believe can be improved by using a more sophisticated query optimization strategy. Nevertheless, even in those cases, *chameleon-db* performs much better than the slowest system. In fact, the difference between the performance of the fastest and the slowest system can be very large. There are several reasons for this. For example, regardless of how complex a query is, *x-RDF-3x* needs to decompose it into triple patterns because of the way data are organized, evaluate each triple pattern in some sequence, and join the results. If the first triple pattern that is evaluated is selective, but the others are not, then *x-RDF-3x* does not encounter any problems, and it outperforms other systems. On the other hand, if all triple patterns are selective, *x-RDF-3x* is not able to get rid of dormant tuples until all join operations are completed, hence, it displays poor performance. In contrast, in that second

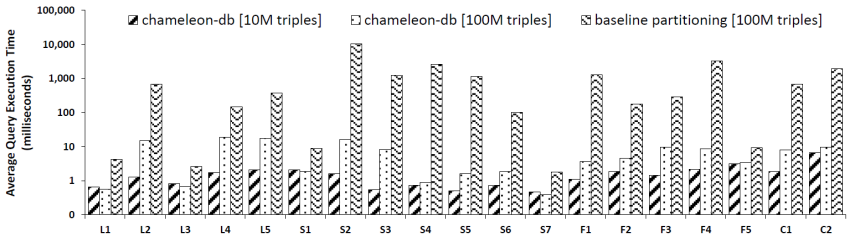


(a) Relative speedup at 10M triples

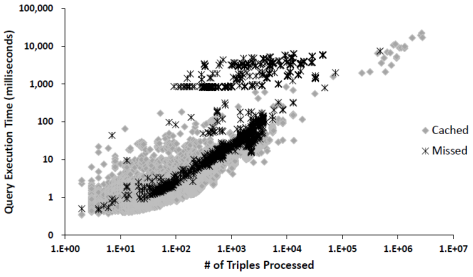
(b) Relative speedup at 100M triples



(c) Correlation between the number of join operations and the average number of triples that are processed



(d) Static performance of *chameleon-db*



(e) Correlation between the number of triples processed during query evaluation and the time to execute the query

Figure 5: Performance evaluation of *chameleon-db*

		L1	L2	L3	L4	L5	S1	S2	S3	S4	S5	S6	S7	F1	F2	F3	F4	F5	C1	C2
10M	RDF-3x	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	✓		✓	✓
	gStore										✓	✓		✓						
	Virtuoso																	✓		
	chameleon-db	♠	♠	♠		♠	♠	♠	♠	♠	♠		♠	♠	♠	♠	♠	♠	♠	♠
100M	RDF-3x		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
	gStore								✓											
	Virtuoso	✓																✓		
	chameleon-db	♠		♠			♠	♠		♠	♠	♠	♠	♠	♠	♠	♠	♠	♠	♠

Table 4: Best system with and without *chameleon-db*: ✓ marks the best system excluding *chameleon-db*, and ♠ indicates *chameleon-db* has outperformed the best system.

Figure 6: Adaptive behaviour of *chameleon-db*

case, if the queries are star-shaped, *gStore* is able to process them much more efficiently because it is optimized for pruning star-shaped patterns all at once as opposed to performing joins on the fly. However, it cannot perform as well on other types of queries (e.g., linear and complex ones). These observations strengthen the motivation of our paper and highlight potential areas of improvement.

Now, consider Figure 5d, which shows the absolute query execution times for *chameleon-db*. Note that *chameleon-db* (i) is robust, with little fluctuation in mean query execution time across different WSDTS queries, (ii) it scales well when going from 10 to 100 million triples (with sub-linear deterioration in performance), and (iii) owes most of its performance improvements to its workload-aware partitioning. In fact, without workload-aware partitioning (i.e., baseline case in Figure 5d), *chameleon-db* is generally slower and its performance fluctuates more across the queries. We attribute the improvement after re-partitioning to the reduced number of join operations, which in turn reduces the number of dormant tuples. To validate the latter hypothesis, we analyzed all of the execution logs generated during our experiments. Figure 5e illustrates that there is a strong positive correlation between the number of triples that are processed in executing a query and the execution time of that query. The only exception is when there are cache-misses, which highlights an area of potential fine-tuning.

7.3 Adaptive Behaviour

Next, we demonstrate how the current version of *chameleon-db* adapts to changing workloads. For this reason, we defined two query mixes over WSDTS. The first one consists of 4200 instantiations of $L1$ and $L3$, which are linear queries. The second query mix consists of 4200 instantiations of $S1$, $S4$ and $S6$, which are star-shaped queries. Again, we randomly picked these queries. In this experiment, initially, *chameleon-db* starts with a partitioning in which each partition contains a single triple. We configure *chameleon-db* so that re-partitioning takes place only twice—just before executing the 600^{th} and 4500^{th} queries. Note that re-partitioning is a process in which (i) a suitable partitioning is computed, and (ii) the underlying partitions are updated *in parallel with query processing*. For the experiments, we instrumented *chameleon-db* to record when a re-partitioning process has been completed and found out that this phase lasts for a very short time, often within the lifetime of the subsequent 5–10 queries.

Figure 6 depicts how throughput changes as our test workload is executed. Note that before re-partitioning takes place, throughput is low, which is normal because the underlying partitioning is not suitable for the given workload. Furthermore, the partition-index has not yet fully adjusted to the recently introduced CPGs. After partitions are fully updated, throughput is improved because the system is using a much better partitioning, which validates our previous observations. During the re-partitioning phase throughput

recovers fast (i.e, very soon after the 600^{th} query).

Note that after the 4200^{th} query throughput drops. At this stage, the workload has changed, but the system has not yet been triggered to re-partition. Hence, queries are not fully optimized, which reflects poorly on performance until re-partitioning kicks in at the 4500^{th} query. This points to an area of future work, which is discussed next.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented *chameleon-db*, an RDF store that can automatically adapt its physical layout for different SPARQL workloads to achieve efficient query execution. Evaluation of *chameleon-db* using a diverse set of query structures with varying selectivities shows that it is more robust with predictable performance than its competitors and can achieve speedups of up to 2 orders of magnitude over them. Our experiments show that *chameleon-db* can adapt well to changing workloads, a desirable feature that is not present in other systems. These advantages open up three areas that can be exploited in the future to further optimize the performance of *chameleon-db*. First, we would like to support parallelization so that (i) results of a *single* query can be computed in parallel across partitions where there is a query match, (ii) system throughput can be increased by parallelizing the execution of *multiple* queries. Second, we want to integrate a more sophisticated cost-model into our query rewriting and optimization. Finally, our ultimate objective is to make *chameleon-db* a fully autonomous, self-tuning system that can decide when to re-partition and that can continuously adapt to changing workloads.

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18:385–406, 2009.

[2] G. Antoshenkov. Dictionary-based order-preserving string compression. *VLDB J.*, 6(1):26–39, 1997.

[3] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *Proc. 30th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 305–316, 2011.

[4] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *Proc. 1st Int. Workshop on Usage Analysis and the Web of Data*, 2011.

[5] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *Proc. 19th Int. World Wide Web Conf.*, pages 41–50, 2010.

[6] C. Bizer. Web of linked data - a global public data space on the Web. In *Proc. 13th Int. Workshop on the World Wide Web and Databases*, 2010.

[7] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 121–132, 2013.

[8] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung. Spider: a system for scalable, parallel / distributed evaluation of large-scale RDF data. In *Proc. 18th ACM Int. Conf. on Information and Knowledge Management*, pages 2087–2088, 2009.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms, 2nd edition.* McGraw-Hill, 2001.

[10] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *ACM SIGMOD Rec.*, 14(2):1–8, 1984.

[11] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD Conference*, pages 145–156, 2011.

[12] O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[13] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB*, 4(11):1123–1134, 2011.

[14] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 297–308, 2009.

[15] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31:264–323, 1999.

[16] M. Kirchberg, R. K. L. Ko, and B. S. Lee. From linked data to relevant data – time is the essence. In *Proc. 1st Int. Workshop on Usage Analysis and the Web of Data*, 2011.

[17] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, 2004.

[18] A. Matono, S. Pahlevi, and I. Kojima. RDFCube: A P2P-based three-dimensional index for structural joins on distributed triple stores. In *Proc. Databases, Int. Workshops on Information Systems, and Peer-to-Peer Computing*, pages 323–330, 2006.

[19] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB*, 1(1):647–659, 2008.

[20] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 627–640, 2009.

[21] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[22] T. Neumann and G. Weikum. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB*, 3(1):256–263, 2010.

[23] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. pages 43–43, 1999.

[24] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):1–45, 2009.

[25] E. Prudhommeaux and A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2008.

[26] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proc. 13th Int. Conf. on Database Theory*, pages 4–33, 2010.

[27] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proc. 17th Int. World Wide Web Conf.*, pages 595–604, 2008.

[28] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[29] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB*, 1(1):1008–1019, 2008.

[30] K. Wilkinson. Jena property table implementation. Technical Report HPL-2006-140, HP-Labs, 2006.

[31] Y. Yan, C. Wang, A. Zhou, W. Qian, L. Ma, and Y. Pan. Efficient indices using graph partitioning in RDF triple stores. In *Proc. 25th Int. Conf. on Data Engineering*, pages 1263–1266, 2009.

[32] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: a fast and compact system for large scale RDF data. *Proc. VLDB*, 6(7):517–528, 2013.

[33] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *Proc. VLDB*, 6(4):265–276, 2013.

[34] L. Zou, J. Mo, D. Zhao, L. Chen, and M. T. Özsu. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB*, 4(1):482–493, 2011.

APPENDIX

A. RELATED PROOFS

In this section we provide the proofs of the theorems introduced in the paper. However, first we introduce two lemmas, which we use in the subsequent proofs.

DEFINITION 17. *Given two RDF graphs G_A and G_B, we define the concatenation of G_A and G_B, denoted by $G_A \oplus G_B$, as an RDF graph $G = (V, E)$ such that (i) $V = V(G_A) \cup V(G_B)$ and (ii) $E = E(G_A) \cup E(G_B)$.*

LEMMA 4. *Let μ be a solution mapping; and let Q_A and Q_B be two CPGs. Given an RDF graph G, G μ-matches the (concatenated) CPG $Q_A \oplus Q_B$ iff there exist two solution mappings μ_A and μ_B and two RDF graphs G_A and G_B such that $G = G_A \oplus G_B$, G_A μ_A-matches Q_A, G_B μ_B-matches Q_B, $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$.*

PROOF OF LEMMA 4. In order to prove Lemma 4, we need to show both of the following statements are true:

C1. Given two RDF graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, and two CPGs $Q_A = (\hat{V}_A, \hat{E}_A, R_A)$ and $Q_B = (\hat{V}_B, \hat{E}_B, R_B)$, if G_A μ_A-matches Q_A and G_B μ_B-matches Q_B, where μ_A and μ_B are two solution mappings such that $\mu_A \sim \mu_B$, then G μ-matches $Q = (\hat{V}, \hat{E}, R)$, where G is an RDF graph with $G = G_A \cup G_B$, $\mu = \mu_A \cup \mu_B$ and $Q = Q_A \oplus Q_B$.

C2. Given an RDF graph $G = (V, E)$, and two CPGs $Q_A = (\hat{V}_A, \hat{E}_A, R_A)$ and $Q_B = (\hat{V}_B, \hat{E}_B, R_B)$, if G μ-matches $Q_A \oplus Q_B$, where μ is a solution mapping, then there exists two RDF graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, and two solution mappings μ_A and μ_B such that G_A μ_A-matches Q_A, G_B μ_B-matches Q_B, $\mu_A \sim \mu_B$, and $\mu = \mu_A \cup \mu_B$.

We use proof by construction for both of the above statements. To prove that $C1$ is true, first we show that μ satisfies conditions (i) and (ii) in Definition 8 and then we construct two surjective functions M_V and M_E that satisfy condition (iii) in Definition 8. Let us prove each of the following one by one.

- Is $\text{dom}(\mu)$ the set of variables mentioned in Q?
  - Since $\mu = \mu_A \cup \mu_B$,
$$\begin{aligned} \text{dom}(\mu) &= \text{dom}(\mu_A \cup \mu_B) \\ &= \text{dom}(\mu_A) \cup \text{dom}(\mu_B) \end{aligned} \quad (1)$$
  - Furthermore, $Q = Q_A \oplus Q_B$ implies that the set of variables mentioned in Q is the union of the set of variables mentioned in Q_A and the set of variables mentioned in Q_B.
  - Consequently, according to Equation 1, $\text{dom}(\mu)$ denotes the set of variables mentioned in Q.
- Does μ satisfy all filter expressions in R?
  - Since $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$, it is not difficult to see that $\mu \sim \mu_A$ and $\mu \sim \mu_B$.
  - Based on Definition 8, we also know that μ_A satisfies all filter expressions in R_A and μ_B satisfies all filter expressions in R_B.
  - Observe that given any two solution mappings μ_1 and μ_2 such that $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ and $\mu_1 \sim \mu_2$, if μ_1 satisfies a filter expression F, then μ_2 also satisfies F.
  - Since $\text{dom}(\mu_A) \subseteq \text{dom}(\mu)$ and $\mu \sim \mu \sim \mu_A$, μ satisfies all filter expressions in R_A.
  - Likewise, since $\text{dom}(\mu_B) \subseteq \text{dom}(\mu)$ and $\mu \sim \mu \sim \mu_B$, μ satisfies all filter expressions in R_B.
  - Consequently, μ satisfies all filter expressions in R because $R = R_A \cup R_B$.
- Does there exist two surjective functions M_V and M_E such that condition (iii) in Definition 8 is satisfied?
  - Let us define $M_V : (\hat{V}_A \cup \hat{V}_B) \to (V_A \cup V_B)$ such that
    - $M_V(\hat{v}) = M_V^A$ for every $\hat{v} \in (\hat{V}_A \setminus \hat{V}_B)$,
    - $M_V(\hat{v}) = M_V^A$ for every $\hat{v} \in (\hat{V}_A \cap \hat{V}_B)$ and
    - $M_V(\hat{v}) = M_V^B$ for every $\hat{v} \in (\hat{V}_B \setminus \hat{V}_A)$, where

    $M_V^A : \hat{V}_A \to V_A$ and $M_V^B : \hat{V}_B \to V_B$ denote the two surjective functions implied by G_A μ_A-matches Q_A and G_B μ_B-matches Q_B, respectively.
  - Let us define $M_E : (\hat{E}_A \cup \hat{E}_B) \to (E_A \cup E_B)$ such that
    - $M_E(\hat{e}) = M_E^A$ for every $\hat{e} \in (\hat{E}_A \setminus \hat{E}_B)$,
    - $M_E(\hat{e}) = M_E^A$ for every $\hat{e} \in (\hat{E}_A \cap \hat{E}_B)$ and
    - $M_E(\hat{e}) = M_E^B$ for every $\hat{e} \in (\hat{E}_B \setminus \hat{E}_A)$, where

    $M_E^A : \hat{E}_A \to E_A$ and $M_E^B : \hat{E}_B \to E_B$ denote the two surjective functions implied by G_A μ_A-matches Q_A and G_B μ_B-matches Q_B, respectively.
  - Note that as long as
    - $M_V^A(\hat{v}) = M_V^B(\hat{v})$ for every $\hat{v} \in (\hat{V}_A \cap \hat{V}_B)$ and
    - $M_E^A(\hat{e}) = M_E^B(\hat{e})$ for every $\hat{e} \in (\hat{E}_A \cap \hat{E}_B)$,

    it can be shown that both M_V and M_E satisfy condition (iii) in Definition 8 because of the way M_V and M_E are defined with respect to M_V^A, M_V^B, M_E^A and M_E^B, which already satisfy condition (iii) in Definition 8.
    - We know that if $\hat{v}$ is a constant, then $M_V^A(\hat{v}) = \hat{v}$ and $M_V^B(\hat{v}) = \hat{v}$, therefore $M_V^A(\hat{v}) = M_V^B(\hat{v})$.
    - If $\hat{v} \in \mathcal{V}$, then $M_V^A(\hat{v}) = \mu_A(\hat{v})$ and $M_V^B(\hat{v}) = \mu_B(\hat{v})$.
    - Since $\mu_A \sim \mu_B$, it holds that $\mu_A(?x) = \mu_B(?x)$ for all variables $?x \in \text{dom}(\mu_A) \cap \text{dom}(\mu_B)$, where $\text{dom}(\mu_A) \cap \text{dom}(\mu_B) = (\hat{V}_A \cap \hat{V}_B) \cap \mathcal{V}$.
    - Consequently, $M_V^A(\hat{v}) = M_V^B(\hat{v})$ for every $\hat{v} \in (\hat{V}_A \cap \hat{V}_B)$ is true.
    - The proof of $M_E^A(\hat{e}) = M_E^B(\hat{e})$ for every $\hat{e} \in (\hat{E}_A \cap \hat{E}_B)$ follows almost the same approach, therefore, we omit it.

For C2, we only provide a sketch of our proof.

- It is possible to construct a surjective function $M_E^A : \hat{E}_A \to E_A$ by projecting M_E onto the domain $\hat{E}_A$.
- Likewise, we can construct a surjective function $M_V^A : \hat{V}_A \to V_A$ by projecting M_V onto the domain $\hat{V}_A$.
- As we are taking projections, it is not difficult to see that there exists a solution mapping μ_A, such that $\mu_A \sim \mu$ and $\text{dom}(\mu_A)$ corresponds to the set of variables mentioned in Q_A, for which conditions (ii) and (iii) in Definition 8 are satisfied.
  - Perhaps the more difficult part is to prove that for each $(\hat{e}_1, e_2) \in \hat{E}_A \times E_A$ with $M_E^A(\hat{e}_1) = e_2$: $\hat{e}_1 = (\hat{s}_1, \hat{p}_1, \hat{o}_1)$ and $e_2 = (s_2, p_2, o_2)$, if $M_V^A(\hat{s}_1) = s_2$, then $M_V^A(\hat{o}_1) = o_2$.
  - However, since $\hat{V}_A$ and $\hat{E}_A$ define a CPG (i.e., Q_A), we also know that both the source and the target of each edge $\hat{e} \in \hat{E}_A$ have to be elements of $\hat{V}_A$, which makes the proof possible.
  - The observation above can also be utilized in order to show that $G_A = (V_A, E_A)$ is an RDF graph.
- Similar arguments can be made about the existence of an RDF graph G_B and a solution mapping μ_B such that G_B μ_B-matches Q_B.
- Since (i) $\mu_A \sim \mu$ and $\mu_B \sim \mu$, and (ii) $\text{dom}(\mu_A) \subseteq \text{dom}(\mu)$ and $\text{dom}(\mu_B) \subseteq \text{dom}(\mu)$, it also holds that $\mu_A \sim \mu_B$.
- Furthermore, $Q = Q_A \oplus Q_B$ implies that $\text{dom}(\mu) = \text{dom}(\mu_A) \cup \text{dom}(\mu_B)$, hence, it also holds that $\mu = \mu_A \cup \mu_B$.
- Consequently, Statement C2 is true.

□

LEMMA 5. *Given an RDF graph G and two CPGs Q_A and Q_B, $[\![Q_A \oplus Q_B]\!]^*_G = [\![Q_A]\!]^*_G \bowtie [\![Q_B]\!]^*_G$.*

PROOF OF LEMMA 5. In order to prove Lemma 5, we need to show that both of the following statements hold:

L1. If μ is a solution mapping such that $\mu \in [\![Q_A \oplus Q_B]\!]^*_G$, then $\mu \in \big([\![Q_A]\!]^*_G \bowtie [\![Q_B]\!]^*_G\big)$; and

L2. If μ is a solution mapping such that $\mu \in \left([\![Q_A]\!]^*_G \bowtie [\![Q_B]\!]^*_G\right)$, then $\mu \in [\![Q_A \oplus Q_B]\!]^*_G$.

We prove both of these statements by contradiction. Let us start with statement L1.

- Assume there exists a solution mapping μ such that $\mu \in [\![Q_A \oplus Q_B]\!]^*_G$, but $\mu \notin \left([\![Q_A]\!]^*_G \bowtie [\![Q_B]\!]^*_G\right)$.
- By Definition 9(1), there exists an RDF graph G' that is a subgraph of G such that G' μ-matches $Q_A \oplus Q_B$ where $\mu \notin [\![Q_A]\!]^*_G \bowtie [\![Q_B]\!]^*_G$.
- Then, according to Lemma 4, there exists two RDF graphs G_A and G_B that are subgraphs of G' such that G_A μ_A-matches Q_A and G_B μ_B-matches Q_B, where μ_A and μ_B are two solution mappings such that $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$.
- Since G_A and G_B are also subgraphs of G (transitively), according to Definition 9(1), $\mu_A \in [\![Q_A]\!]^*_G$ and $\mu_B \in [\![Q_B]\!]^*_G$.
- However, since $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$, according to Definition 6, μ must also be an element of $[\![Q_A]\!]^*_G \bowtie [\![Q_B]\!]^*_G$, which is a contradiction.
- Consequently, Statement L1 must hold.

Now, let us prove statement L2.

- Assume there exists a solution mapping μ such that $\mu \in \left([\![Q_A]\!]^*_G \bowtie [\![Q_B]\!]^*_G\right)$, but $\mu \notin [\![Q_A \oplus Q_B]\!]^*_G$.
- Based on Definition 6, there must exist two solution mappings $\mu_A \in [\![Q_A]\!]^*_G$ and $\mu_B \in [\![Q_B]\!]^*_G$ such that $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$.
- According to Definition 9(1), there must exist two RDF graphs G_A and G_B such that (i) G_A and G_B are both subgraphs of G, (ii) G_A μ_A-matches Q_A, and (iii) G_B μ_B-matches Q_B.
- Since $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$, Lemma 4 suggests that there also exists an RDF graph G' such that $G' = G_A \cup G_B$ and G' μ-matches $Q_A \oplus Q_B$.
- Since both G_A and G_B are subgraphs of G, so is G', and according to Definition 9(1), $\mu \in [\![Q_A \oplus Q_B]\!]^*_G$, however, this is a contradiction.
- Consequently, Statement L2 must hold.

□

Proof of Theorem 1. We need to show that given a CPG Q, an RDF graph G and a partitioning $\mathbb{P}$ of G, $[\![Q]\!]^*_G = M$ holds, where M is the baseline PRE expression for Q over $\mathbb{P}$. Let $\text{TD}(Q) = \{Q_1, \ldots, Q_k\}$ be the trivial decomposition of Q, and let $\mathbb{P} = \{P_1, \ldots, P_m\}$. Note that according to Definition 11, for each $Q_i \in \text{TD}(Q)$,

$$\lfloor\!\lfloor Q_i \rfloor\!\rfloor(\mathbb{P}) = [\![Q_i]\!]^*_{P_1} \cup \cdots \cup [\![Q_i]\!]^*_{P_m}. \tag{2}$$

Based on Definition 13, we also know that each $Q_i \in \text{TD}(Q)$ contains exactly one edge, therefore

$$[\![Q_i]\!]^*_{P_1} \cup \cdots \cup [\![Q_i]\!]^*_{P_m} = [\![Q_i]\!]^*_{P_1 \cup \cdots \cup P_m}$$

and

$$\lfloor\!\lfloor Q_i \rfloor\!\rfloor(\mathbb{P}) = [\![Q_i]\!]^*_{P_1 \cup \cdots \cup P_m}. \tag{3}$$

Since $\mathbb{P} = \{P_1, \ldots, P_m\}$ is a partitioning of G, by Definition 10, $P_1 \cup \cdots \cup P_m = G$, hence,

$$\lfloor\!\lfloor Q_i \rfloor\!\rfloor(\mathbb{P}) = [\![Q_i]\!]^*_G. \tag{4}$$

Substituting values in the baseline PRE expression with the right hand side of Equation 4, we get

$$M = [\![Q_1]\!]^*_G \bowtie \cdots \bowtie [\![Q_k]\!]^*_G. \tag{5}$$

By recursively applying Lemma 5, we get

$$M = [\![Q_1 \oplus \cdots \oplus Q_k]\!]^*_G. \tag{6}$$

Since $Q = Q_1 \oplus \cdots \oplus Q_k$ by definition of trivial decomposition, $[\![Q]\!]^*_G = M$ holds. □

Proof of Theorem 2. Assuming the conditions in Theorem 2, we want to show that $\lfloor\!\lfloor Q_A \rfloor\!\rfloor(\mathbb{P}_1) \bowtie \lfloor\!\lfloor Q_B \rfloor\!\rfloor(\mathbb{P}_2) = \emptyset$ holds. Note that according to Definition 11, a semantically equivalent expression that we can prove is $\bigcup_{(P_i,P_j)\in\mathbb{P}_1\times\mathbb{P}_2} [\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} = \emptyset$, which we do by using proof-by-contradiction.

A. Let us assume that there exist two partitions $P_i \in \mathbb{P}_1$ and $P_j \in \mathbb{P}_2$ such that $[\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} \neq \emptyset$.
B. Based on Definition 6, since the join in the above expression returns a non-empty set of solution mappings, there must exist a solution mapping $\mu \in [\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j}$ and two solution mappings $\mu_A \in [\![Q_A]\!]^*_{P_i}$ and $\mu_B \in [\![Q_B]\!]^*_{P_j}$ such that $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$.
C. Then, according to Definition 9(1), there must exist two RDF graphs G_A and G_B such that (i) G_A is a subgraph of P_i and G_A μ_A-matches Q_A, and (ii) G_B is a subgraph of P_j and G_B μ_B-matches Q_B.
D. Lemma 4 suggests that there also exists an RDF graph G' such that $G' = G_A \cup G_B$ and G' μ-matches $Q_A \oplus Q_B$.
E. Statements (C) and (D), together with Definitions 3 and 8 imply that for every vertex $\hat{v} \in V(Q_A) \cap V(Q_B)$
   - $inc(Q_A, \hat{v}) \neq \emptyset$ and $inc(Q_B, \hat{v}) \neq \emptyset$ (because according to Definition 3, a vertex in a CPG cannot exist without an edge incident on it); and
   - there exists a vertex $v \in V(G_A) \cap V(G_B)$ to which $\hat{v}$ is mapped (because the three surjective functions $M_V^{G_A}$, $M_V^{G_B}$ and $M_V^{G'}$ implied by G_A μ_A-matches Q_A, G_B μ_B-matches Q_B and G' μ-matches $Q_A \oplus Q_B$ must agree on all common vertices) such that
     - every edge $\hat{e} \in inc(Q_A, \hat{v})$ is compatible with an edge from $e \in inc(G_A, v)$;
     - likewise, every edge $\hat{e} \in inc(Q_B, \hat{v})$ is compatible with an edge from $e \in inc(G_B, v)$ (which follows the previous statement and Definition 8); and
F. In other words, for every vertex $\hat{v} \in V(Q_A) \cap V(Q_B)$, there exists a vertex $v \in V(G_A) \cap V(G_B)$ such that every edge $\hat{e} \in inc(Q_A, \hat{v}) \cup inc(Q_B, \hat{v})$ is compatible with an edge from $inc(G_A, v) \cup inc(G_B, v)$.
G. Recall that $V(Q_A) \cap V(Q_B) \neq \emptyset$, therefore, Statement (F) also implies that there exists a vertex $v \in V(G_A) \cap V(G_B)$ and a vertex $\hat{v} \in V(Q_A) \cap V(Q_B)$ such that every edge $\hat{e} \in inc(Q_A, \hat{v}) \cup inc(Q_B, \hat{v})$ is compatible with an edge from $inc(G_A, v) \cup inc(G_B, v)$.
H. However, Statement (G) contradicts the conditions in Theorem 2, therefore proof-by-contradiction suggests Statement (A) does not hold under these conditions.
I. Consequently, $\bigcup_{(P_i,P_j)\in\mathbb{P}_1\times\mathbb{P}_2} [\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} = \emptyset$ and Theorem 2 holds.

□

Proof of Theorem 3. Assuming the conditions in Theorem 3, we want to show that

$$\lfloor\!\lfloor Q_A \rfloor\!\rfloor(\mathbb{P}) \bowtie \lfloor\!\lfloor Q_B \rfloor\!\rfloor(\mathbb{P}) = \lfloor\!\lfloor Q_A \oplus Q_B \rfloor\!\rfloor(\mathbb{P}) \tag{7}$$

holds. According to Definition 11, Equation 7 is equivalent to

$$\bigcup_{(P_i,P_j)\in(\mathbb{P}\times\mathbb{P})} [\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} = \bigcup_{P\in\mathbb{P}} [\![Q_A \oplus Q_B]\!]^*_P. \tag{8}$$

Furthermore, according to Lemma 5, the right hand side of Equation 8 can be rewritten as follows:

$$\bigcup_{(P_i,P_j)\in(\mathbb{P}\times\mathbb{P})} [\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} = \bigcup_{P\in\mathbb{P}} [\![Q_A]\!]^*_P \bowtie [\![Q_B]\!]^*_P, \tag{9}$$

Given the equivalence of Equation 7 and Equation 9, we prove Theorem 3 by showing that Equation 9 holds. Note that Equation 9 holds iff

$$[\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} = \emptyset \tag{10}$$

for all $(P_i, P_j) \in \mathbb{P} \times \mathbb{P}$ with $P_i \neq P_j$. We need to prove that Equation 10 holds both for sub-condition (i) and sub-condition (ii) in Theorem 3. The proof of sub-condition (i) follows the same steps as the proof of Theorem 2, therefore, we omit it. We prove that Theorem 3 holds for sub-condition (ii) using proof-by-contradiction.

A. Let us assume that there exist two partitions $P_i, P_j \in \mathbb{P}$, where $P_i \neq P_j$, such that $[\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} \neq \emptyset$.
B. Same as Statements (B)–(E) in the proof of Theorem 2
C. Furthermore, according to Definition 8, for every $\hat{v} \in V(Q_A) \cap V(Q_B)$, there exists an edge e_A in $inc(G_A, v)$, which is compatible with an edge in $inc(Q_A, \hat{v})$, such that $cont(\mathbb{P}, e_A) = P_i$ and an edge e_B in $inc(G_B, v)$, which is compatible with an edge in $inc(Q_B, \hat{v})$, such that $cont(\mathbb{P}, e_B) = P_j$.
D. However, $P_i \neq P_j$, which contradicts condition (ii) in Theorem 3.
E. Proof-by-contradiction suggests Statement (A) does not hold under any of the above conditions.
F. Consequently, $\bigcup_{(P_i,P_j)\in\mathbb{P}_1\times\mathbb{P}_2} [\![Q_A]\!]^*_{P_i} \bowtie [\![Q_B]\!]^*_{P_j} = \emptyset$ and Theorem 3 holds.

□

B. EXPRESSIVENESS

In this section, we show that the framework we use in this paper has the same expressive power as the standard formalization of SPARQL [24] except for complex filter expressions involving built-in functions, which can be integrated into our framework if desired. To this end, let us use FILTER$^\bigstar$ to denote those filter expressions in the standard formalization that are consistent with our definition of filter expressions (Definition 2).

Assuming that $\mathcal{S}$ denotes the set of all possible SPARQL expressions that can be expressed using the standard syntax [24] (except those involving complex filter expressions with built-in functions) and that $\mathcal{S}^*$ denotes the set of all possible queries that can be expressed using our notation, we introduce two functions

- $M : \mathcal{S} \to \mathcal{S}^*$ that maps SPARQL expressions in the standard syntax to a query expression in our notation (Definition 18), and
- $M^* : \mathcal{S}^* \to \mathcal{S}$ that maps queries in our notation to a canonical SPARQL expression in the standard syntax (Definitions 19 and 20)—such canonicalization is necessary because as Definition 19 suggests, a CPG can be mapped to multiple SPARQL expressions.

DEFINITION 18. *$M : \mathcal{S} \to \mathcal{S}^*$ is defined recursively as follows:*

- *If S is a triple pattern $(\hat{s}, \hat{p}, \hat{o})$, then $M(S)$ is the CPG $Q = (\hat{V}, \hat{E}, R)$ where $\hat{V} = \{\hat{s}, \hat{o}\}$, $\hat{E} = \{(\hat{s}, \hat{p}, \hat{o})\}$ and $R = \emptyset$;*
- *If S is S_1 AND S_2, and*
  - *(a) if S_1 and S_2 are both in the AND-FILTER$^\bigstar$ fragment of SPARQL, then $M(S)$ is the (concatenated) CPG $M(S_1) \oplus M(S_2)$,*
  - *(b) else $M(S)$ is $M(S_1)$ AND $M(S_2)$;*
- *If S is S_1 FILTER F, where F is a filter expression that is consistent with Definition 2, and*
  - *(a) if S_1 is in the AND-FILTER$^\bigstar$ fragment of SPARQL, assuming $M(S_1) = Q_1$, then $M(S)$ is the CPG $Q = (\hat{V}, \hat{E}, R)$ where $\hat{V} = V(Q_1)$, $\hat{E} = E(Q_1)$ and $R = R(Q_1) \cup F$,*
  - *(b) else $M(S)$ is $M(S_1)$ FILTER F;*
- *If S is S_1 UNION S_2, then $M(S)$ is $M(S_1)$ UNION $M(S_2)$; and*
- *If S is S_1 OPT S_2, then $M(S)$ is $M(S_1)$ OPT $M(S_2)$.*

DEFINITION 19. *Assume $\preceq_T$ and $\preceq_F$ are total orderings of (i) all possible triple patterns, and (ii) all possible filter expressions, respectively. Then, the canonical SPARQL representation of a CPG Q is defined as*

$$\text{CR}(Q) = \big(\cdots((tp_1 \text{ AND} \cdots \text{ AND } tp_m) \text{ FILTER } F_1) \cdots \text{ FILTER } F_n\big),$$

where $tp_i \in E(Q)$ for all $i \in \{1, \ldots, m\}$, $F_j \in R(Q)$ for all $j \in \{1, \ldots, n\}$, $tp_i \preceq_T tp_j$ for all $i < j$ and $F_i \preceq_F F_j$ for all $i < j$.

DEFINITION 20. *$M^* : \mathcal{S}^* \to \mathcal{S}$ is defined recursively as follows:*

- *If Q is a CPG, $M^*(Q) = \text{CR}(Q)$;*
- *If Q is a Q_1 AND Q_2, $M^*(Q)$ is $M^*(Q_1)$ AND $M^*(Q_2)$;*
- *If Q is a Q_1 FILTER F, $M^*(Q)$ is $M^*(Q_1)$ FILTER F;*
- *If Q is a Q_1 UNION Q_2, $M^*(Q)$ is $M^*(Q_1)$ UNION $M^*(Q_2)$; and*
- *If Q is a Q_1 OPT Q_2, $M^*(Q)$ is $M^*(Q_1)$ OPT $M^*(Q_2)$.*

To show that our formalism is as expressive as the standard, we must prove that for any RDF graph G, whose edges correspond to a set of triples D, it holds that

I. for any SPARQL query $S \in \mathcal{S}$, $[\![M(S)]\!]^*_G = [\![S]\!]_D$ (Theorem 6), and
II. for any query expression $S^* \in \mathcal{S}^*$, $[\![S^*]\!]^*_G = [\![M^*(S^*)]\!]_D$ (Theorems 7 and 8),

where $[\![\cdot]\!]^*_G$ denotes the evaluation function introduced in this paper, and $[\![\cdot]\!]_D$ denotes the standard evaluation.

THEOREM 6. *Given any RDF graph G, whose edges correspond to a set of triples D, for any SPARQL expression $S \in \mathcal{S}$, it holds that $[\![M(S)]\!]^*_G = [\![S]\!]_D$.*

PROOF OF THEOREM 6. We use proof by induction on the construction of S, which is formally described in Definition 18.
Base case (S is a single triple pattern): We need to show that both of the following statements hold:

B1. If μ is a solution mapping such that $\mu \in [\![M(S)]\!]^*_G$, then $\mu \in [\![S]\!]_D$; and
B2. If μ is a solution mapping such that $\mu \in [\![S]\!]_D$, then $\mu \in [\![M(S)]\!]^*_G$.

We prove both of these statements by contradiction. Recall that for any SPARQL expression with a single triple pattern such that $S = (\hat{s}, \hat{p}, \hat{o})$, then $M(S)$ is a CPG $Q(\hat{V}, \hat{E}, R)$ where $\hat{V} = \{\hat{s}, \hat{o}\}$, $\hat{E} = \{(\hat{s}, \hat{p}, \hat{o})\}$ and $R = \emptyset$. Then, the proof of Statement B1 proceeds as follows:

- Assume there exists a solution mapping μ such that $\mu \in [\![M(S)]\!]^*_G$, but $\mu \notin [\![S]\!]_D$.
- By Definition 9(1), there exists an RDF graph G' that is a subgraph of G such that G' μ-matches $M(S)$.
- By Definition 8, G' consists of a single edge, and since G' is a subgraph of G (which is defined over the set of triples D), there must exist a triple $t \in D$ such that $\mu[tp] = t$.
- If such a triple exists, then μ must also be an element of $[\![tp]\!]_D$.
- However, then $\mu \in [\![S]\!]_D$, which is a contradiction.
- Consequently, statement B1 must hold.

We prove statement B2 using a similar reasoning.

- Assume there exists a solution mapping μ such that $\mu \in [\![S]\!]_D$, but $\mu \notin [\![M(S)]\!]^*_G$.
- Therefore, there must exist a triple $t \in D$ such that $\mu[tp] = t$.
- Hence, there exists an RDF graph G' that is a subgraph of G which consists of a single edge that represents t.
- According to the definition of μ-match (Definition 8) and our earlier statement about t, G' must μ-match Q[7].
- Then, according to Definition 9(1), $\mu \in [\![Q]\!]^*_G$, which is a contradiction.
- Consequently, statement B2 must hold.

Inductive step: There are multiple cases to consider, which are outlined shortly. For each of the cases, we need to show that both of the following statements hold:

I1. If μ is a solution mapping such that $\mu \in [\![M(S)]\!]^*_G$, then $\mu \in [\![S]\!]_D$; and
I2. If μ is a solution mapping such that $\mu \in [\![S]\!]_D$, then $\mu \in [\![M(S)]\!]^*_G$.

[7]It must be noted that μ can be defined over an empty domain of variables.

Case-I (S is S_1 AND S_2, where S_1 and S_2 are both in the AND-FILTER$^\bigstar$ fragment of SPARQL): We assume by induction that $[\![M(S_1)]\!]^*_G = [\![S_1]\!]_D$ and $[\![M(S_2)]\!]^*_G = [\![S_2]\!]_D$, and we prove I1 and I2 by contradiction. Let us start with the first statement.

- Assume there exists a solution mapping μ such that $\mu \in [\![M(S_1 \text{ AND } S_2)]\!]^*_G$, but $\mu \notin [\![S_1 \text{ AND } S_2]\!]_D$.
- According to Definition 18, since S_1 and S_2 are both in the AND-FILTER$^\bigstar$ fragment of SPARQL, $\mu \in [\![M(S_1) \oplus M(S_2)]\!]^*_G$.
- Then, according to Lemma 5, $\mu \in [\![M(S_1)]\!]^*_G \bowtie [\![M(S_2)]\!]^*_G$.
- Therefore, by induction, $\mu \in [\![S_1]\!]_D \bowtie [\![S_2]\!]_D$.
- However, $[\![S_1]\!]_D \bowtie [\![S_2]\!]_D = [\![S_1 \text{ AND } S_2]\!]_D$, therefore $\mu \in [\![S_1 \text{ AND } S_2]\!]_D$, which is a contradiction.
- Consequently, statement I1 holds if S is (S_1 AND S_2).

To prove statement I2, we use a similar method.

- Assume there exists a solution mapping μ such that $\mu \in [\![S_1 \text{ AND } S_2]\!]_D$, but $\mu \notin [\![M(S_1 \text{ AND } S_2)]\!]^*_G$.
- Therefore, $\mu \in [\![S_1]\!]_D \bowtie [\![S_2]\!]_D$, and by induction, $\mu \in [\![M(S_1)]\!]^*_G \bowtie [\![M(S_2)]\!]^*_G$.
- According to Lemma 5, $\mu \in [\![M(S_1) \oplus M(S_2)]\!]^*_G$.
- However, this is a contradiction because S_1 and S_2 are both in the AND-FILTER$^\bigstar$ fragment of SPARQL, and according to Definition 18, $M(S_1 \text{ AND } S_2) = M(S_1) \oplus M(S_2)$.
- Consequently, statement I2 holds if $S = S_1$ AND S_2.

Case-II (S is S_1 FILTER F where S_1 is in the AND-FILTER$^\bigstar$ fragment of SPARQL, and F is a filter expression that is consistent with Definition 2): We assume by induction that $[\![M(S_1)]\!]^*_G = [\![S_1]\!]_D$. Recall that according to Definition 18, assuming $M(S_1) = Q_1$, then $M(S) = Q(\hat{V}, \hat{E}, R)$ where $\hat{V} = V(Q_1)$, $\hat{E} = E(Q_1)$ and $R = R(Q_1) \cup F$. Then, we can prove I1 and I2 by contradiction. Let us start with I1.

- Assume there exists a solution mapping μ such that $\mu \in [\![M(S)]\!]^*_G$, but $\mu \notin [\![S]\!]_D$.
- By Definition 9(1), there exists an RDF graph G' that is a subgraph of G such that G' μ-matches Q.
- According to Definition 8, μ satisfies F.
- According to Definition 8, it also holds that G' μ-matches Q_1; therefore, $\mu \in [\![M(S_1)]\!]^*_G$.
- Then, by induction, $\mu \in [\![S_1]\!]_D$.
- However, we already know that μ satisfies F, and according to the algebraic formalism by Schmidt et al. [26], it can be easily shown that $\mu \in [\![S]\!]_D$, which is a contradiction.
- Consequently, statement I1 holds.

For statement I2, we use a similar method.

- Assume there exists a solution mapping μ such that $\mu \in [\![S]\!]_D$, but $\mu \notin [\![M(S)]\!]^*_G$.
- Since F is consistent with Definition 2, it can be concluded that μ satisfies F.
- Furthermore, $\mu \in [\![S_1]\!]_D$.
- Then, by induction, $\mu \in [\![M(S_1)]\!]^*_G$.
- According to Definition 18, $\mu \in [\![M(S_1)]\!]^*_G$ because S_1 is in the AND-FILTER$^\bigstar$ fragment of SPARQL.
- Let $Q_1 = (\hat{V}, \hat{E}, R)$ be the CPG that $M(S_1)$ denotes.
- By Definition 9(1), there exists an RDF graph G' that is a subgraph of G such that G' μ-matches Q_1.
- Note that according to Definition 18, $M(S) = Q'$, where $Q' = (\hat{V}, \hat{E}, R')$ with $R' = R \cup \{F\}$.
- Then, according to Definition 8, it also holds that G' μ-matches $M(S)$ because $M(S) = Q'$ and μ satisfies F.
- However, last statement implies that $\mu \in [\![M(S)]\!]^*_G$, which is a contradiction.
- Consequently, I2 holds.

Case-III (S is S_1 OP S_2, where OP $\in \{$AND, UNION, OPT$\}$ such that if OP is AND, then at least one of S_1 or S_2 is not in the AND-FILTER$^\bigstar$ fragment of SPARQL): We need to show that

$$[\![M(S)]\!]^*_G = [\![S]\!]_D. \tag{11}$$

We will show the proof steps only for OP = UNION, but omit the others as they are very similar.

A. Without loss of generality, Equation 11 can be restated as:

$$[\![M(S_1 \text{ UNION } S_2)]\!]^*_G = [\![S_1 \text{ UNION } S_2]\!]_D \tag{12}$$

B. According to Definition 18, Equation 12 becomes

$$[\![M(S_1) \text{ UNION } M(S_2)]\!]^*_G = [\![S_1 \text{ UNION } S_2]\!]_D.$$

C. Then, according to Definition 9, equation becomes

$$[\![M(S_1)]\!]^*_G \cup [\![M(S_2)]\!]^*_G = [\![S_1 \text{ UNION } S_2]\!]_D.$$

D. By induction it holds that

$$[\![M(S_1)]\!]^*_G = [\![S_1]\!]_D \text{ and}$$
$$[\![M(S_2)]\!]^*_G = [\![S_2]\!]_D.$$

E. Therefore, equation becomes

$$[\![S_1]\!]_D \cup [\![S_2]\!]_D = [\![S_1 \text{ UNION } S_2]\!]_D.$$

F. However, based on the standard formalism,

$$[\![S_1 \text{ UNION } S_2]\!]_D = [\![S_1]\!]_D \cup [\![S_2]\!]_D,$$

which proves Equation 11.

Case-IV (S is S_1 FILTER F, where S_1 is not in the AND-FILTER$^\bigstar$ fragment of SPARQL): The proof is very similar to the proof of Case-III, therefore, we omit it.

□

To prove that for any query expression $S^* \in \mathcal{S}^*$, $[\![S^*]\!]^*_G = [\![M^*(S^*)]\!]_D$, first, we show that

- for any CPG Q and any RDF graph G, $[\![Q]\!]^*_G = [\![M^*(Q)]\!]_D$, where $D = E(G)$ (Theorem 7).

Then, we extend our proof to each sub-part of Definition 9 (Theorem 8).

THEOREM 7. *Given any RDF graph G, whose edges correspond to a set of triples D, for any CPG Q, it holds that $[\![Q]\!]^*_G = [\![M^*(Q)]\!]_D$.*

PROOF OF THEOREM 7. We use proof by induction on the number of edges in Q.

Base case ($|E_Q| = 1$): We need to show that both of the following statements hold:

B1. If μ is a solution mapping such that $\mu \in [\![Q]\!]^*_G$, then $\mu \in [\![M^*(Q)]\!]_D$; and

B2. If μ is a solution mapping such that $\mu \in [\![M^*(Q)]\!]_D$, then $\mu \in [\![Q]\!]^*_G$.

We prove both of these statements by contradiction. Recall that according to Definitions 19 and 20 if $E(Q) = \{tp\}$ and $R(Q) = \{F_1, \cdots, F_n\}$, then

$$M^*(Q) = \big(\cdots (tp \text{ FILTER } F_1) \cdots F_n\big).$$

Then, the proof of Statement B1 proceeds as follows:

- Assume there exists a solution mapping μ such that $\mu \in [\![Q]\!]^*_G$, but $\mu \notin [\![M^*(Q)]\!]_D$.
- By Definition 9(1), there exists an RDF graph G' that is a subgraph of G such that G' μ-matches Q.

- By Definition 8, G' consists of a single edge, and since G' is a subgraph of G (which is defined over the set of triples D), there must exist a triple $t \in D$ such that $\mu[tp] = t$ and μ satisfies $F_1, \cdots, F_n$.
- If such a triple exists, then μ must also be an element of $[\![M^*(Q)]\!]_D$, which is a contradiction.
- Consequently, statement B1 must hold.

We prove statement B2 using a similar reasoning.

- Assume there exists a solution mapping μ such that $\mu \in [\![M^*(Q)]\!]_D$, but $\mu \notin [\![Q]\!]^*_G$.
- Therefore, there must exist a triple $t \in D$ such that $\mu[tp] = t$, where μ also satisfies all filter expressions $F_1 \dots F_n$.
- Hence, there exists an RDF graph G' that is a subgraph of G which consists of the edge that represents t.
- According to the definition of μ-match (Definition 8) and our earlier statement about t, G' must μ-match Q.
- Then, according to Definition 9(1), $\mu \in [\![Q]\!]^*_G$, which is a contradiction.
- Consequently, statement B2 must hold.

Inductive step ($|E(Q)| = n+1$): In order to show that $[\![Q]\!]^*_G = [\![M^*(Q)]\!]_D$, we need to show that both of the following statements hold:

I1. If μ is a solution mapping such that $\mu \in [\![Q]\!]^*_G$, then $\mu \in [\![M^*(Q)]\!]_D$; and
I2. If μ is a solution mapping such that $\mu \in [\![M^*(Q)]\!]_D$, then $\mu \in [\![Q]\!]^*_G$.

We prove statements I1 and I2 by contradiction. Without loss of generality, we assume two CPGs Q_A and Q_B such that $Q = Q_A \oplus Q_B$. Given $|E(Q_A)| \leq n$ and $|E(Q_B)| \leq n$, by induction, $[\![Q_A]\!]^*_G = [\![M^*(Q_A)]\!]_D$ and $[\![Q_B]\!]^*_G = [\![M^*(Q_B)]\!]_D$. Let us start with the first statement.

- Assume there exists a solution mapping μ such that $\mu \in [\![Q_A \oplus Q_B]\!]^*_G$, but $\mu \notin [\![M^*(Q_A \oplus Q_B)]\!]_D$.
- By Definition 9(1), there exists an RDF graph G' that is a subgraph of G such that G' μ-matches $Q_A \oplus Q_B$.
- Then, according to Lemma 4, there exists two RDF graphs G_A and G_B that are subgraphs of G' such that G_A μ_A-matches Q_A and G_B μ_B-matches Q_B, where μ_A and μ_B are two solution mappings such that $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$.
- Since G_A and G_B are also subgraphs of G (transitively), according to Definition 9(1), $\mu_A \in [\![Q_A]\!]^*_G$ and $\mu_B \in [\![Q_B]\!]^*_G$.
- Due to the inductive hypothesis, it also holds that $\mu_A \in [\![M^*(Q_A)]\!]_D$ and $\mu_B \in [\![M^*(Q_B)]\!]_D$.
- Note that $[\![M^*(Q_A \oplus Q_B)]\!]_D = [\![M^*(Q_A) \text{ AND } M^*(Q_B)]\!]_D$ (which is easily verified by using the algebraic equivalence shown by Schmidt et al. [26]).
- According to the standard interpretation [24], it holds that $[\![M^*(Q_A) \text{ AND } M^*(Q_B)]\!]_D = [\![M^*(Q_A)]\!]_D \bowtie [\![M^*(Q_B)]\!]_D$.
- Since $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$, according to the definition of join ($\bowtie$) in the standard formalization [24], $\mu \in [\![M^*(Q_A)]\!]_D \bowtie [\![M^*(Q_B)]\!]_D$.
- Thus, $\mu \in [\![M^*(Q_A \oplus Q_B)]\!]_D$, which is a contradiction.
- Consequently, statement I1 holds.

To prove statement I2, we use a similar method.

- Assume there exists a solution mapping μ such that $\mu \in [\![M^*(Q_A \oplus Q_B)]\!]_D$, but $\mu \notin [\![Q_A \oplus Q_B]\!]^*_G$.
- Note that $[\![M^*(Q_A \oplus Q_B)]\!]_D = [\![M^*(Q_A) \text{ AND } M^*(Q_B)]\!]_D$ (which is easily verified by using the algebraic equivalence shown by Schmidt et al. [26]).
- According to the standard interpretation [24], it holds that $[\![M^*(Q_A) \text{ AND } M^*(Q_B)]\!]_D = [\![M^*(Q_A)]\!]_D \bowtie [\![M^*(Q_B)]\!]_D$.
- Then, according to the definition of join ($\bowtie$) in the standard formalization [24], there must exist two solution mappings μ_A and μ_B such that $\mu_A \in [\![M^*(Q_A)]\!]_D$, $\mu_B \in [\![M^*(Q_B)]\!]_D$, $\mu = \mu_A \cup \mu_B$ and $\mu_A \sim \mu_B$.
- Due to the inductive hypothesis, it also holds that $\mu_A \in [\![Q_A]\!]^*_G$.
- Furthermore, since Q_B consists of a single edge, the base case of Theorem 7 suggests that $\mu_B \in [\![Q_B]\!]^*_G$.
- Then, according to Lemma 4, there exists an RDF graph $G' = G_A \cup G_B$ such that G' μ-matches $Q_A \oplus Q_B$.
- Since both G_A and G_B are subgraphs of G and $G' = G_A \cup G_B$, G' is also a subgraph of G.
- By Definition 9(1), $\mu \in [\![Q_A \oplus Q_B]\!]^*_G$, which is a contradiction.
- Consequently, by proof-by-contradiction, statement I2 holds.

□

Now, we are ready to extend our proof to the other steps in Definition 9.

THEOREM 8. *Given any RDF graph G, whose edges correspond to a set of triples D, for any query expression $Q \in \mathcal{S}^*$ it holds that $[\![Q]\!]^*_G = [\![M^*(Q)]\!]_D$.*

PROOF OF THEOREM 8. We prove Theorem 8 by induction on the number of AND, UNION, OPT and FILTER operations in a query expression $Q \in \mathcal{S}^*$. In the base case, we show that

- If Q_1 and Q_2 are two CPGs, then

$$[\![Q_1 \text{ AND } Q_2]\!]^*_G = [\![M^*(Q_1 \text{ AND } Q_2)]\!]_D, \quad (13)$$
$$[\![Q_1 \text{ UNION } Q_2]\!]^*_G = [\![M^*(Q_1 \text{ UNION } Q_2)]\!]_D \text{ and} \quad (14)$$
$$[\![Q_1 \text{ OPT } Q_2]\!]^*_G = [\![M^*(Q_1 \text{ OPT } Q_2)]\!]_D; \text{ and if} \quad (15)$$

- If Q_1 is a CPG and F is a filter expression, then

$$[\![Q_1 \text{ FILTER } F]\!]^*_G = [\![M^*(Q_1 \text{ FILTER } F)]\!]_D. \quad (16)$$

In the inductive step, we assume $[\![Q_1]\!]^*_G = [\![M^*(Q_1)]\!]_D$ and $[\![Q_2]\!]^*_G = [\![M^*(Q_2)]\!]_D$ hold for $Q_1, Q_2 \in \mathcal{S}^*$, where Q_1 and Q_2 contain less than or equal to n number of AND, UNION, OPT and FILTER operations. Then, we prove Equations 13–16 are true for this general case as well.

Base case: We will show the proof steps for Equation 13, but omit the others as they are very similar.

A. According to Definition 9, the left hand side of Equation 13 becomes

$$[\![Q_1 \text{ AND } Q_2]\!]^*_G = [\![Q_1]\!]^*_G \bowtie [\![Q_2]\!]^*_G.$$

B. Since Q_1 and Q_2 are CPGs, Theorem 7 suggests that

$$[\![Q_1]\!]^*_G = [\![M^*(Q_1)]\!]_D \text{ and}$$
$$[\![Q_2]\!]^*_G = [\![M^*(Q_2)]\!]_D.$$

C. Therefore,

$$[\![Q_1 \text{ AND } Q_2]\!]^*_G = [\![M^*(Q_1)]\!]_D \bowtie [\![M^*(Q_2)]\!]_D.$$

D. According to Definition 20,

$$M^*(Q_1 \text{ AND } Q_2) = M^*(Q_1) \text{ AND } M^*(Q_2).$$

E. Then, based on the standard formalism [24], the right hand side of Equation 13 becomes

$$[\![M^*(Q_1 \text{ AND } Q_2)]\!]_D = [\![M^*(Q_1)]\!]_D \bowtie [\![M^*(Q_2)]\!]_D.$$

F. Hence,

$$[\![Q_1 \text{ AND } Q_2]\!]^*_G = [\![M^*(Q_1 \text{ AND } Q_2)]\!]_D,$$

which proves Equation 13 for the base case.

Inductive step: The proof of the inductive step is very similar to that of the base case except for Step (B) above, which needs to be adjusted as follows:

- Based on the inductive hypothesis,

$$[\![Q_1]\!]^*_G = [\![M^*(Q_1)]\!]_D \text{ and}$$
$$[\![Q_2]\!]^*_G = [\![M^*(Q_2)]\!]_D.$$

□

	scale factor = 1
triples	105257
distinct subjects	5597
distinct predicates	85
distinct objects	13258
URIs	5947
literals	14286
distinct literals	8018

Table 5: Characteristics of the WSDTS test dataset.

C. DETAILS OF WSDTS

WSDTS consists of two components: the data generator and the query generator. The WSDTS data generator allows users to define their own dataset through a dataset description language. This way, users can control (i) which entities to include in their dataset, (ii) how coherent each entity is (for details please refer to a research paper by Duan et al. [11]), (iii) how different entities are associated, (iv) the probability that an entity of type X is associated with an entity of type Y, and (v) the cardinality of such associations.

Using these features, we designed the WSDTS test dataset (see the associated dataset description model [8]). By executing the data generator with different scale factors, it is possible generate test datasets with different sizes. Table 5 lists the properties of the dataset at scale factor = 1.

An important characteristic that distinguishes the WSDTS test dataset from existing benchmarks is that instances of the same entity do not necessarily have the same set of attributes. Table 6 lists all the different entities used in WSDTS. Take the *Product* entity for instance. *Product* instances may be associated with different Product Categories (e.g., Book, Movie, Classical Music Concert, etc.), but depending on which category a product belongs to, it will have a different set of attributes. For example, products that belong to the category "Classical Music Concert" have the attributes *mo:opus*, *mo:movement*, *wsdbm:composer*, *mo:performer* (in addition to the attributes that is common to every product), whereas products that belong to the category "Book" have the attributes *sorg:isbn*, *sorg:bookEdition* and *sorg:numberOfPages*. Furthermore, even within a single product category, not all instances share the same set of attributes. For example, while *sorg:isbn* is a required attribute for a book, *sorg:bookEdition* ($Pr = 0.5$) and *sorg:numberOfPages* ($Pr = 0.25$) are optional attributes, where Pr indicates the probability that an instance will be generated with that attribute. It must be also noted that some attributes are correlated, which means that either all or none of the correlated attributes will be present in an instance (the PGROUP construct in the WSDTS dataset description language allows the grouping of such correlated attributes). For a complete list of probabilities, please refer to Tables 7 and 8.

In short, we designed the WSDTS test dataset such that

- some entities are more structured (meaning that they contain few optional attributes) while the others are less structured;
- entities are associated in complex ways that mimic the real types of distributions on the Web;
- cardinalities of these associations are varied.

At this point, you may be wondering how these differentiating aspects of WSDTS affect system evaluation, and why they are important at all. The answer is trivial: by relying on a more diverse dataset as such (which is typical for data on the Web), we were able to generate test queries that focus on much wider aspects of query evaluation, which cannot be easily captured by other benchmarks. Consider the two SPARQL query templates $C3$ and $S7$ (Appendix D). $C3$ is a star query that retrieves certain information about users such as the products they like, their friends and some demographics information. On our website [9], for each triple pattern in the query template, we also display its selectivity (the reported selectivities are estimations based on the probability distributions specified in the WSDTS dataset description model). Note that while individually triple patterns in $C3$ are not that selective, this query as a whole, is very selective. Now, consider $S7$, which (as a whole) is also very selective, but unlike $C3$, its selectivity is largely due to only a single triple pattern. It turns out that different systems behave very differently for these queries. Systems like *RDF-3x* [21], which (i) decompose queries into triple patterns, (ii) find a suitable ordering of the join operations and then (iii) execute the joins in that order, perform very well on queries like $S7$ because the first triple pattern they execute is very selective. On the other hand, they do not do as well on queries like $C3$ because the decomposed evaluation produces many irrelevant intermediate tuples. In contrast, *gStore* [34] treats the star-shaped query as a whole and it can pinpoint the relevant vertices in the RDF graph without performing joins; hence, it is much more efficient in executing $C3$.

Based on the above observations, in WSDTS we tried to generate as diverse a test workload as possible. WSDTS test workloads consist of queries in four categories, namely, linear queries (L), star queries (S), snowflake-shaped queries (F) and complex queries (C) with a total of 20 query templates. These query templates were randomly selected from a pool of queries generated by performing a random walk on the dataset description model (which can be represented as a graph), while making sure that (i) the selected queries sufficiently represent each category, (ii) the selectivities of the queries within each category vary, and (iii) in some queries selectivity originates from a single (or few) triple patterns while in the others, it originates as a combination of multiple somewhat less selective triple patterns.

Entity Type	Instance Count [per scale factor if applicable]
wsdbm:Purchase	1500
wsdbm:User	1000
wsdbm:Offer	900
wsdbm:Topic*	250
wsdbm:Product	250
wsdbm:City*	240
wsdbm:SubGenre*	145
wsdbm:Website	50
wsdbm:Language*	25
wsdbm:Country*	25
wsdbm:Genre*	21
wsdbm:ProductCategory*	15
wsdbm:Retailer	12
wsdbm:AgeGroup*	9
wsdbm:Role*	3
wsdbm:Gender*	2

Table 6: Entities generated according to WSDTS data description model. Entities marked with an asterisk * do not scale.

[8] `https://cs.uwaterloo.ca/~galuc/wsdts/wsdts-data-model.txt`

[9] `https://cs.uwaterloo.ca/~galuc/wsdts/`

Table 7: Probability that a randomly picked instance of a given entity (based on a normal/uniform/Zipfian distribution) has a given attribute such that instance appears as the subject and attribute appears as the predicate of a triple.

Subject[@TypeRestriction]	**Predicate**	***Pr* (normal)**	***Pr* (uniform)**	***Pr* (Zipfian)**	**Cardinality**
wsdbm:City	gn:parentCountry	1	1	1	1
wsdbm:Offer	gr:includes	1	1	1	1
wsdbm:Offer	gr:price	1	1	1	1
wsdbm:Offer	gr:serialNumber	1	1	1	1
wsdbm:Offer	gr:validFrom	0.4	0.4	0.5	1
wsdbm:Offer	gr:validThrough	0.4	0.4	0.4	1
wsdbm:Offer	sorg:eligibleQuantity	1	1	1	1
wsdbm:Offer	sorg:eligibleRegion	0.5	0.5	0.5	4
wsdbm:Offer	sorg:priceValidUntil	0.2	0.2	0.2	1
wsdbm:Product	foaf:homepage	0.2	0.3	0.3	1
wsdbm:Product	og:tag	0.6	0.6	0.7	9.8
wsdbm:Product	og:title	1	1	1	1
wsdbm:Product	rdf:type	1	1	1	1
wsdbm:Product	rev:hasReview	0.2	0.2	0.2	32.8
wsdbm:Product	sorg:caption	0.2	0.2	0.1	1
wsdbm:Product	sorg:contentRating	0.4	0.4	0.3	1
wsdbm:Product	sorg:contentSize	0.1	0.1	0	1
wsdbm:Product	sorg:description	0.6	0.6	0.7	1
wsdbm:Product	sorg:expires	0.1	0.1	0.1	1
wsdbm:Product	sorg:keywords	0.3	0.3	0.4	1
wsdbm:Product	sorg:text	0.3	0.3	0.3	1
wsdbm:Product	wsdbm:hasGenre	1	1	1	2.4
wsdbm:Product@wsdbm:ProductCategory0	foaf:homepage	0	0.1	0.5	0
wsdbm:Product@wsdbm:ProductCategory0	mo:conductor	0.6	0.3	0.1	1
wsdbm:Product@wsdbm:ProductCategory0	mo:movement	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory0	mo:opus	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory0	mo:performed_in	0.2	0.5	0.7	1
wsdbm:Product@wsdbm:ProductCategory0	mo:performer	0.5	0.7	0.9	1
wsdbm:Product@wsdbm:ProductCategory0	og:tag	0.2	0.2	0.3	11
wsdbm:Product@wsdbm:ProductCategory0	og:title	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory0	rdf:type	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory0	rev:hasReview	0.1	0.1	0.6	21.3
wsdbm:Product@wsdbm:ProductCategory0	sorg:caption	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory0	sorg:contentRating	0.5	0.6	0.3	1
wsdbm:Product@wsdbm:ProductCategory0	sorg:contentSize	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory0	sorg:description	0.6	0.3	0.6	1
wsdbm:Product@wsdbm:ProductCategory0	sorg:expires	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory0	sorg:keywords	0.4	0.6	0.3	1
wsdbm:Product@wsdbm:ProductCategory0	sorg:text	0.2	0.5	0.7	1
wsdbm:Product@wsdbm:ProductCategory0	wsdbm:composer	1	0.9	0.5	1
wsdbm:Product@wsdbm:ProductCategory0	wsdbm:hasGenre	1	1	1	2.6
wsdbm:Product@wsdbm:ProductCategory1	foaf:homepage	0.1	0.2	0.4	1
wsdbm:Product@wsdbm:ProductCategory1	mo:artist	0.9	0.9	0.7	1
wsdbm:Product@wsdbm:ProductCategory1	mo:producer	0.5	0.4	0.3	1
wsdbm:Product@wsdbm:ProductCategory1	mo:record_number	0.8	0.7	0.9	1
wsdbm:Product@wsdbm:ProductCategory1	mo:release	0.7	0.4	0.3	1
wsdbm:Product@wsdbm:ProductCategory1	og:tag	0.4	0.5	0.7	11.4
wsdbm:Product@wsdbm:ProductCategory1	og:title	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory1	rdf:type	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory1	rev:hasReview	0.3	0.2	0.1	18.9
wsdbm:Product@wsdbm:ProductCategory1	sorg:caption	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory1	sorg:contentRating	0.4	0.4	0.4	1
wsdbm:Product@wsdbm:ProductCategory1	sorg:contentSize	0.1	0.2	0.1	1
wsdbm:Product@wsdbm:ProductCategory1	sorg:description	0.4	0.5	0.7	1
wsdbm:Product@wsdbm:ProductCategory1	sorg:expires	0.2	0.1	0.1	1
wsdbm:Product@wsdbm:ProductCategory1	sorg:keywords	0.3	0.3	0.2	1
wsdbm:Product@wsdbm:ProductCategory1	sorg:text	0	0.1	0.3	1
wsdbm:Product@wsdbm:ProductCategory1	wsdbm:hasGenre	1	1	1	2.5
wsdbm:Product@wsdbm:ProductCategory2	foaf:homepage	0.2	0.2	0.1	1
wsdbm:Product@wsdbm:ProductCategory2	og:tag	0.5	0.6	0.7	10.1
wsdbm:Product@wsdbm:ProductCategory2	og:title	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory2	rdf:type	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory2	rev:hasReview	0.1	0.1	0.1	40.3
wsdbm:Product@wsdbm:ProductCategory2	sorg:actor	0.8	0.8	0.9	11.5
wsdbm:Product@wsdbm:ProductCategory2	sorg:award	0.1	0.1	0	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:caption	0.3	0.3	0.1	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:contentRating	0.3	0.4	0.2	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:contentSize	0.1	0	0	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:description	0.8	0.8	1	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:director	0.8	0.8	0.6	1

wsdbm:Product@wsdbm:ProductCategory2	sorg:duration	0.4	0.4	0.3	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:expires	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory2	sorg:keywords	0.2	0.2	0.4	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:language	0.3	0.2	0.2	1.6
wsdbm:Product@wsdbm:ProductCategory2	sorg:producer	0.4	0.4	0.6	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:text	0.1	0.2	0.1	1
wsdbm:Product@wsdbm:ProductCategory2	sorg:trailer	0.1	0.1	0.1	2
wsdbm:Product@wsdbm:ProductCategory2	wsdbm:hasGenre	1	1	1	2.1
wsdbm:Product@wsdbm:ProductCategory3	foaf:homepage	0.4	0.4	0.5	1
wsdbm:Product@wsdbm:ProductCategory3	og:tag	0.7	0.7	0.8	11.7
wsdbm:Product@wsdbm:ProductCategory3	og:title	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory3	rdf:type	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory3	rev:hasReview	0.1	0.1	0.1	39.5
wsdbm:Product@wsdbm:ProductCategory3	sorg:author	0.7	0.7	0.8	1.9
wsdbm:Product@wsdbm:ProductCategory3	sorg:bookEdition	0.4	0.4	0.2	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:caption	0.1	0.1	0.3	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:contentRating	0.5	0.7	0.8	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:contentSize	0	0.1	0	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:description	0.4	0.3	0.3	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:editor	0.4	0.4	0.4	2.4
wsdbm:Product@wsdbm:ProductCategory3	sorg:expires	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory3	sorg:isbn	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:keywords	0.3	0.3	0.3	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:language	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory3	sorg:numberOfPages	0.5	0.5	0.5	1
wsdbm:Product@wsdbm:ProductCategory3	sorg:text	0.4	0.5	0.3	1
wsdbm:Product@wsdbm:ProductCategory3	wsdbm:hasGenre	1	1	1	2.4
wsdbm:Product@wsdbm:ProductCategory4	foaf:homepage	0.4	0.5	0.4	1
wsdbm:Product@wsdbm:ProductCategory4	og:tag	0.5	0.5	0.8	12.7
wsdbm:Product@wsdbm:ProductCategory4	og:title	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory4	rdf:type	1	1	1	1
wsdbm:Product@wsdbm:ProductCategory4	rev:hasReview	0	0.1	0.1	6.3
wsdbm:Product@wsdbm:ProductCategory4	sorg:author	0.8	0.8	0.9	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:caption	0.2	0.2	0.2	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:contentRating	0.1	0.1	0.1	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:contentSize	0	0	0	0
wsdbm:Product@wsdbm:ProductCategory4	sorg:datePublished	0.6	0.5	0.8	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:description	0.8	0.7	0.6	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:editor	0.2	0.1	0	1.3
wsdbm:Product@wsdbm:ProductCategory4	sorg:expires	0.1	0.1	0	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:keywords	0.1	0.1	0.2	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:printColumn	0	0.2	0.1	0.7
wsdbm:Product@wsdbm:ProductCategory4	sorg:printEdition	0	0.1	0.1	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:printPage	0.3	0.3	0.1	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:printSection	0.3	0.4	0.1	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:publisher	1	0.8	0.5	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:text	0.2	0.4	0.3	1
wsdbm:Product@wsdbm:ProductCategory4	sorg:wordCount	0.1	0.1	0	1
wsdbm:Product@wsdbm:ProductCategory4	wsdbm:hasGenre	1	1	1	2.1
wsdbm:ProductCategory	rdf:type	0	0	0	0
wsdbm:Purchase	gr:price	1	1	1	1
wsdbm:Purchase	wsdbm:purchaseDate	1	1	1	1
wsdbm:Purchase	wsdbm:purchaseFor	1	1	1	1
wsdbm:Retailer	gr:description	1	1	1	1
wsdbm:Retailer	gr:name	1	1	1	1
wsdbm:Retailer	gr:offers	1	1	1	90.3
wsdbm:Retailer	sorg:aggregateRating	0.4	0.6	0.7	1
wsdbm:Retailer	sorg:contactPoint	1	0.9	1	1
wsdbm:Retailer	sorg:email	0.8	0.9	1	1
wsdbm:Retailer	sorg:employee	0	0.1	0	3
wsdbm:Retailer	sorg:faxNumber	0	0.1	0	1
wsdbm:Retailer	sorg:legalName	0	0	0	0
wsdbm:Retailer	sorg:openingHours	0.8	0.8	0.9	1
wsdbm:Retailer	sorg:paymentAccepted	0.6	0.6	0.5	1
wsdbm:Retailer	sorg:telephone	0.8	0.8	0.8	1
wsdbm:Review	rev:rating	1	1	1	1
wsdbm:Review	rev:reviewer	1	1	1	1
wsdbm:Review	rev:text	0.7	0.7	0.6	1
wsdbm:Review	rev:title	0.3	0.3	0.3	1
wsdbm:Review	rev:totalVotes	0.1	0	0.1	1
wsdbm:SubGenre	og:tag	1	0.9	1	3
wsdbm:SubGenre	rdf:type	1	1	1	1
wsdbm:User	dc:Location	0.4	0.4	0.4	1

wsdbm:User	foaf:age	0.5	0.5	0.5	1
wsdbm:User	foaf:familyName	0.7	0.7	0.6	1
wsdbm:User	foaf:givenName	0.7	0.7	0.6	1
wsdbm:User	foaf:homepage	0.1	0	0	1
wsdbm:User	rdf:type	1	1	1	1.1
wsdbm:User	sorg:birthDate	0.2	0.2	0.3	1
wsdbm:User	sorg:email	0.9	0.9	0.7	1
wsdbm:User	sorg:jobTitle	0.1	0.1	0	1
wsdbm:User	sorg:nationality	0.2	0.2	0.2	1
wsdbm:User	sorg:telephone	0	0	0.1	1
wsdbm:User	wsdbm:follows	0.8	0.8	0.8	40.7
wsdbm:User	wsdbm:friendOf	0.4	0.4	0.6	106.7
wsdbm:User	wsdbm:gender	0.6	0.6	0.7	1
wsdbm:User	wsdbm:likes	0.3	0.3	0.2	5.3
wsdbm:User	wsdbm:subscribes	0.2	0.2	0.1	7
wsdbm:User	wsdbm:userId	1	1	1	1
wsdbm:User@wsdbm:Role0	dc:Location	0.4	0.3	0.3	1
wsdbm:User@wsdbm:Role0	foaf:age	0.5	0.5	0.4	1
wsdbm:User@wsdbm:Role0	foaf:familyName	0.7	0.7	0.7	1
wsdbm:User@wsdbm:Role0	foaf:givenName	0.7	0.7	0.7	1
wsdbm:User@wsdbm:Role0	foaf:homepage	0	0	0	1
wsdbm:User@wsdbm:Role0	rdf:type	1	1	1	1.1
wsdbm:User@wsdbm:Role0	sorg:birthDate	0.2	0.2	0.3	1
wsdbm:User@wsdbm:Role0	sorg:email	0.9	0.9	0.8	1
wsdbm:User@wsdbm:Role0	sorg:jobTitle	0.1	0	0.1	1
wsdbm:User@wsdbm:Role0	sorg:nationality	0.2	0.2	0.1	1
wsdbm:User@wsdbm:Role0	sorg:telephone	0	0	0	1
wsdbm:User@wsdbm:Role0	wsdbm:follows	0.8	0.8	0.8	42.9
wsdbm:User@wsdbm:Role0	wsdbm:friendOf	0.4	0.4	0.5	105.5
wsdbm:User@wsdbm:Role0	wsdbm:gender	0.6	0.6	0.7	1
wsdbm:User@wsdbm:Role0	wsdbm:likes	0.3	0.2	0.2	5.1
wsdbm:User@wsdbm:Role0	wsdbm:makesPurchase	0.3	0.3	0.4	9.9
wsdbm:User@wsdbm:Role0	wsdbm:subscribes	0.2	0.2	0.1	6.9
wsdbm:User@wsdbm:Role0	wsdbm:userId	1	1	1	1
wsdbm:User@wsdbm:Role1	dc:Location	0.4	0.4	0.5	1
wsdbm:User@wsdbm:Role1	foaf:age	0.4	0.4	0.5	1
wsdbm:User@wsdbm:Role1	foaf:familyName	0.7	0.7	0.6	1
wsdbm:User@wsdbm:Role1	foaf:givenName	0.7	0.7	0.6	1
wsdbm:User@wsdbm:Role1	foaf:homepage	0.1	0.1	0	1
wsdbm:User@wsdbm:Role1	rdf:type	1	1	1	1.2
wsdbm:User@wsdbm:Role1	sorg:birthDate	0.2	0.2	0.1	1
wsdbm:User@wsdbm:Role1	sorg:email	0.9	0.9	0.8	1
wsdbm:User@wsdbm:Role1	sorg:jobTitle	0.1	0.1	0.2	1
wsdbm:User@wsdbm:Role1	sorg:nationality	0.2	0.2	0.2	1
wsdbm:User@wsdbm:Role1	sorg:telephone	0.1	0.1	0	1
wsdbm:User@wsdbm:Role1	wsdbm:follows	0.8	0.8	0.7	40.3
wsdbm:User@wsdbm:Role1	wsdbm:friendOf	0.4	0.4	0.5	107.6
wsdbm:User@wsdbm:Role1	wsdbm:gender	0.7	0.6	0.4	1
wsdbm:User@wsdbm:Role1	wsdbm:likes	0.3	0.3	0.2	6.9
wsdbm:User@wsdbm:Role1	wsdbm:subscribes	0.2	0.2	0.2	6.8
wsdbm:User@wsdbm:Role1	wsdbm:userId	1	1	1	1
wsdbm:User@wsdbm:Role2	dc:Location	0.4	0.4	0.4	1
wsdbm:User@wsdbm:Role2	foaf:age	0.5	0.5	0.4	1
wsdbm:User@wsdbm:Role2	foaf:familyName	0.7	0.7	0.8	1
wsdbm:User@wsdbm:Role2	foaf:givenName	0.7	0.7	0.8	1
wsdbm:User@wsdbm:Role2	foaf:homepage	0.1	0.1	0.2	1
wsdbm:User@wsdbm:Role2	rdf:type	1	1	1	1.2
wsdbm:User@wsdbm:Role2	sorg:birthDate	0.2	0.2	0.2	1
wsdbm:User@wsdbm:Role2	sorg:email	0.9	0.9	0.9	1
wsdbm:User@wsdbm:Role2	sorg:jobTitle	0.1	0.1	0	1
wsdbm:User@wsdbm:Role2	sorg:nationality	0.2	0.2	0.1	1
wsdbm:User@wsdbm:Role2	sorg:telephone	0.1	0.1	0.1	1
wsdbm:User@wsdbm:Role2	wsdbm:follows	0.8	0.8	0.8	39.1
wsdbm:User@wsdbm:Role2	wsdbm:friendOf	0.4	0.4	0.3	101.8
wsdbm:User@wsdbm:Role2	wsdbm:gender	0.7	0.6	0.4	1
wsdbm:User@wsdbm:Role2	wsdbm:likes	0.3	0.3	0.1	4.6
wsdbm:User@wsdbm:Role2	wsdbm:subscribes	0.2	0.2	0.1	7.5
wsdbm:User@wsdbm:Role2	wsdbm:userId	1	1	1	1
wsdbm:Website	sorg:url	1	1	1	1
wsdbm:Website	wsdbm:hits	1	1	1	1

Table 8: Probability that a randomly picked instance of a given entity (based on a normal/uniform/Zipfian distribution) has a given attribute such that instance appears as the object and attribute appears as the predicate of a triple.

Object[@TypeRestriction]	**Predicate**	***Pr* (normal)**	***Pr* (uniform)**	***Pr* (Zipfian)**	**Cardinality**
wsdbm:AgeGroup	foaf:age	1	1	1	90
wsdbm:City	dc:Location	0.4	0.5	0.8	1.7
wsdbm:City	mo:performed_in	0	0	0	1
wsdbm:Country	gn:parentCountry	1	1	1	5.7
wsdbm:Country	sorg:eligibleRegion	1	1	1	51.9
wsdbm:Country	sorg:nationality	0.9	0.9	1	4.8
wsdbm:Gender	wsdbm:gender	1	1	1	436.5
wsdbm:Genre	rdf:type	1	1	1	6.4
wsdbm:Language	sorg:language	0	0.2	0.6	1.5
wsdbm:Offer	gr:offers	0.6	0.6	0.9	1.6
wsdbm:Product	gr:includes	1	1	1	3.8
wsdbm:Product	wsdbm:likes	1	1	1	4.3
wsdbm:Product	wsdbm:purchaseFor	0.9	0.8	1	3
wsdbm:Product@wsdbm:ProductCategory0	gr:includes	1	1	1	2.5
wsdbm:Product@wsdbm:ProductCategory0	wsdbm:likes	1	1	1	4.4
wsdbm:Product@wsdbm:ProductCategory0	wsdbm:purchaseFor	1	1	1	2.2
wsdbm:Product@wsdbm:ProductCategory1	gr:includes	0.8	0.9	0.9	3.6
wsdbm:Product@wsdbm:ProductCategory1	wsdbm:likes	1	1	1	5.3
wsdbm:Product@wsdbm:ProductCategory1	wsdbm:purchaseFor	0.6	0.7	0.8	7.9
wsdbm:Product@wsdbm:ProductCategory2	gr:includes	1	0.9	1	5.2
wsdbm:Product@wsdbm:ProductCategory2	wsdbm:likes	0.8	1	1	3.9
wsdbm:Product@wsdbm:ProductCategory2	wsdbm:purchaseFor	0.8	0.9	0.9	4.9
wsdbm:Product@wsdbm:ProductCategory3	gr:includes	1	1	1	3.5
wsdbm:Product@wsdbm:ProductCategory3	wsdbm:likes	1	1	1	4
wsdbm:Product@wsdbm:ProductCategory3	wsdbm:purchaseFor	0.5	0.8	0.9	4.5
wsdbm:Product@wsdbm:ProductCategory4	gr:includes	1	1	1	4.5
wsdbm:Product@wsdbm:ProductCategory4	wsdbm:likes	1	1	1	8.5
wsdbm:Product@wsdbm:ProductCategory4	wsdbm:purchaseFor	0.9	0.9	0.9	18.4
wsdbm:Purchase	wsdbm:makesPurchase	1	1	1	1
wsdbm:Review	rev:hasReview	1	1	1	1
wsdbm:Role	rdf:type	1	1	1	257.6
wsdbm:SubGenre	wsdbm:hasGenre	1	0.8	0.5	7
wsdbm:Topic	og:tag	1	1	1	7.8
wsdbm:User	sorg:author	0	0	0	1.1
wsdbm:User	sorg:contactPoint	0	0	0	1
wsdbm:User	sorg:editor	0	0	0	1
wsdbm:User	sorg:employee	0	0	0	1
wsdbm:User@wsdbm:Role0	sorg:author	0	0	0	1
wsdbm:User@wsdbm:Role0	sorg:contactPoint	0	0	0	1
wsdbm:User@wsdbm:Role0	sorg:editor	0	0	0	1
wsdbm:User@wsdbm:Role0	sorg:employee	0	0	0	1
wsdbm:User@wsdbm:Role1	rev:reviewer	1	1	1	5.4
wsdbm:User@wsdbm:Role1	sorg:author	0	0	0	1
wsdbm:User@wsdbm:Role1	sorg:contactPoint	0	0	0	1
wsdbm:User@wsdbm:Role1	sorg:editor	0	0	0	1
wsdbm:User@wsdbm:Role1	sorg:employee	0	0	0	0
wsdbm:User@wsdbm:Role2	mo:artist	0	0	0.3	1.3
wsdbm:User@wsdbm:Role2	mo:conductor	0	0	0	0
wsdbm:User@wsdbm:Role2	sorg:actor	0.5	0.5	0.7	1.5
wsdbm:User@wsdbm:Role2	sorg:author	0.1	0.1	0.1	1.2
wsdbm:User@wsdbm:Role2	sorg:contactPoint	0	0	0	1
wsdbm:User@wsdbm:Role2	sorg:director	0	0.1	0.4	1
wsdbm:User@wsdbm:Role2	sorg:editor	0.1	0.1	0	1
wsdbm:User@wsdbm:Role2	sorg:employee	0	0	0	0
wsdbm:Website	foaf:homepage	1	1	1	2.6
wsdbm:Website	sorg:trailer	0	0.1	0.1	1
wsdbm:Website	wsdbm:subscribes	1	1	1	27

D. WSDTS QUERY TEMPLATES

L1
```
#mapping v1 wsdbm:Website uniform
SELECT ?v0 ?v2 ?v3 WHERE {
        ?v0 wsdbm:subscribes %v1% .
        ?v2 sorg:caption ?v3 .
        ?v0 wsdbm:likes ?v2 .
}
```

L2
```
#mapping v0 wsdbm:City uniform
SELECT ?v1 ?v2 WHERE {
        %v0% gn:parentCountry ?v1 .
        ?v2 wsdbm:likes wsdbm:Product0 .
        ?v2 sorg:nationality ?v1 .
}
```

L3
```
#mapping v2 wsdbm:Website uniform
SELECT ?v0 ?v1 WHERE {
        ?v0 wsdbm:likes ?v1 .
        ?v0 wsdbm:subscribes %v2% .
}
```

L4
```
#mapping v1 wsdbm:Topic uniform
SELECT ?v0 ?v2 WHERE {
        ?v0 og:tag %v1% .
        ?v0 sorg:caption ?v2 .
}
```

L5
```
#mapping v2 wsdbm:City uniform
SELECT ?v0 ?v1 ?v3 WHERE {
        ?v0 sorg:jobTitle ?v1 .
        %v2% gn:parentCountry ?v3 .
        ?v0 sorg:nationality ?v3 .
}
```

S1
```
#mapping v2 wsdbm:Retailer uniform
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE {
        ?v0 gr:includes ?v1 .
        %v2% gr:offers ?v0 .
        ?v0 gr:price ?v3 .
        ?v0 gr:serialNumber ?v4 .
        ?v0 gr:validFrom ?v5 .
        ?v0 gr:validThrough ?v6 .
        ?v0 sorg:eligibleQuantity ?v7 .
        ?v0 sorg:eligibleRegion ?v8 .
        ?v0 sorg:priceValidUntil ?v9 .
}
```

S2
```
#mapping v2 wsdbm:Country uniform
SELECT ?v0 ?v1 ?v3 WHERE {
        ?v0 dc:Location ?v1 .
        ?v0 sorg:nationality %v2% .
        ?v0 wsdbm:gender ?v3 .
        ?v0 rdf:type wsdbm:Role2 .
}
```

S3
```
#mapping v1 wsdbm:ProductCategory uniform
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
        ?v0 rdf:type %v1% .
        ?v0 sorg:caption ?v2 .
        ?v0 wsdbm:hasGenre ?v3 .
        ?v0 sorg:publisher ?v4 .
}
```

S4
```
#mapping v1 wsdbm:AgeGroup uniform
SELECT ?v0 ?v2 ?v3 WHERE {
        ?v0 foaf:age %v1% .
        ?v0 foaf:familyName ?v2 .
        ?v3 mo:artist ?v0 .
        ?v0 sorg:nationality wsdbm:Country1 .
}
```

S5
```
#mapping v1 wsdbm:ProductCategory uniform
SELECT ?v0 ?v2 ?v3 WHERE {
        ?v0 rdf:type %v1% .
        ?v0 sorg:description ?v2 .
        ?v0 sorg:keywords ?v3 .
        ?v0 sorg:language wsdbm:Language0 .
}
```

S6
```
#mapping v3 wsdbm:SubGenre uniform
SELECT ?v0 ?v1 ?v2 WHERE {
        ?v0 mo:conductor ?v1 .
        ?v0 rdf:type ?v2 .
        ?v0 wsdbm:hasGenre %v3% .
}
```

S7
```
#mapping v3 wsdbm:User uniform
SELECT ?v0 ?v1 ?v2 WHERE {
        ?v0 rdf:type ?v1 .
        ?v0 sorg:text ?v2 .
        %v3% wsdbm:likes ?v0 .
}
```

F1
```
#mapping v1 wsdbm:Topic uniform
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
        ?v0 og:tag %v1% .
        ?v0 rdf:type ?v2 .
        ?v3 sorg:trailer ?v4 .
        ?v3 sorg:keywords ?v5 .
        ?v3 wsdbm:hasGenre ?v0 .
        ?v3 rdf:type wsdbm:ProductCategory2 .
}
```

F2
```
#mapping v8 wsdbm:SubGenre uniform
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 WHERE {
        ?v0 foaf:homepage ?v1 .
        ?v0 og:title ?v2 .
        ?v0 rdf:type ?v3 .
        ?v0 sorg:caption ?v4 .
        ?v0 sorg:description ?v5 .
        ?v1 sorg:url ?v6 .
        ?v1 wsdbm:hits ?v7 .
        ?v0 wsdbm:hasGenre %v8% .
}
```

F3
```
#mapping v3 wsdbm:SubGenre uniform
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
        ?v0 sorg:contentRating ?v1 .
        ?v0 sorg:contentSize ?v2 .
        ?v0 wsdbm:hasGenre %v3% .
        ?v4 wsdbm:makesPurchase ?v5 .
        ?v5 wsdbm:purchaseDate ?v6 .
        ?v5 wsdbm:purchaseFor ?v0 .
}
```

F4
```
#mapping v3 wsdbm:Topic uniform
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
        ?v0 foaf:homepage ?v1 .
        ?v2 gr:includes ?v0 .
        ?v0 og:tag %v3% .
        ?v0 sorg:description ?v4 .
```

?v0 sorg:contentSize ?v8 .
?v1 sorg:url ?v5 .
?v1 wsdbm:hits ?v6 .
?v1 sorg:language wsdbm:Language0 .
?v7 wsdbm:likes ?v0 .
}

F5

#mapping v2 wsdbm:Retailer uniform
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
?v0 gr:includes ?v1 .
%v2% gr:offers ?v0 .
?v0 gr:price ?v3 .
?v0 gr:validThrough ?v4 .
?v1 og:title ?v5 .
?v1 rdf:type ?v6 .
}

C1

SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
?v0 sorg:caption ?v1 .
?v0 sorg:text ?v2 .
?v0 sorg:contentRating ?v3 .
?v0 rev:hasReview ?v4 .
?v4 rev:title ?v5 .
?v4 rev:reviewer ?v6 .
?v7 sorg:actor ?v6 .
?v7 sorg:language ?v8 .
}

C2

SELECT ?v0 ?v3 ?v4 ?v8 WHERE {
?v0 sorg:legalName ?v1 .
?v0 gr:offers ?v2 .
?v2 sorg:eligibleRegion wsbm:Country5 .
?v2 gr:includes ?v3 .
?v4 sorg:jobTitle ?v5 .
?v4 foaf:homepage ?v6 .
?v4 wsdbm:makesPurchase ?v7 .
?v7 wsdbm:purchaseFor ?v3 .
?v3 rev:hasReview ?v8 .
?v8 rev:totalVotes ?v9 .
}

C3

SELECT ?v0 WHERE {
?v0 wsdbm:likes ?v1 .
?v0 wsdbm:friendOf ?v2 .
?v0 dc:Location ?v3 .
?v0 foaf:age ?v4 .
?v0 wsdbm:gender ?v5 .
?v0 foaf:givenName ?v6 .
}