# smart-KG: Partition-Based Linked Data Fragments for Querying Knowledge Graphs

Amr Azzam [a,*], Axel Polleres [a,b], Javier D. Fernández [c], and Maribel Acosta [d,e]

[a] *Department of Informations Systems and Operations, Vienna University of Economics and Business, Austria*
*E-mails: aazzam@wu.ac.at, axel.polleres@wu.ac.at*
[b] *Complexity Science Hub Vienna, Austria*
[c] *Data Science Acceleration (DSX), F. Hoffmann-La Roche, Basel,Switzerland*
*E-mail: javier_d.fernandez@roche.com*
[d] *Department of Computer Science, Technical University of Munich, Germany*
*E-mail: maribel.acosta@tum.de*
[e] *Faculty of Computer Science , Ruhr University Bochum, Germany*

**Abstract.** RDF and SPARQL provide a uniform way to publish and query billions of triples in open knowledge graphs (KGs) on the Web. Yet, provisioning of a fast, reliable, and responsive live querying solution for open KGs is still hardly possible through SPARQL endpoints alone: while such endpoints provide a remarkable performance for single queries, they typically can not cope with highly concurrent query workloads by multiple clients. To mitigate this, the Linked Data Fragments (LDF) framework sparked the design of different alternative low-cost interfaces such as Triple Pattern Fragments (TPF), that partially offload the query processing workload to the client side. On the downside, such interfaces still come with the expense of unnecessarily high network load due to the necessary transfer of intermediate results to the client, leading to query performance degradation compared with endpoints. To address this problem, in the present work, we investigate alternative interfaces, refining and extending the original TPF idea, which also aims at reducing server-resource consumption, by shipping query-relevant partitions of KGs from the server to the client. To this end, first, we align formal definitions and notations of the original LDF framework to uniformly present existing LDF implements and such "partition-based" LDF approaches. These novel LDF interfaces retrieve, instead of the exact triples matching a particular query pattern, a subset of pre-materialized, compressed, partitions of the original graph, containing all answers to a query pattern, to be further evaluated on the client side. As a concrete representative of partition-based LDF, we present `smart-KG`$^+$, extending and refining our prior work [1] in several respects. Our proposed approach is a step forward towards a better-balanced share of the query processing load between clients and servers by shipping graph partitions driven by the structure of RDF graphs to group entities described with the same sets of properties and classes, resulting in significant data transfer reduction. Our experiments demonstrate that the `smart-KG`$^+$ significantly outperforms existing Web SPARQL interfaces on both pre-existing benchmarks for highly concurrent query execution as well as an accustomed query workload inspired by query logs of existing SPARQL endpoints.

Keywords: Knowledge Graph, SPARQL, Linked Data Fragments, Graph Partitioning, Availability

## 1. Introduction

Knowledge Graphs (KGs) have emerged as a promising foundation of data management to enable the construction of scalable data models that represent a collection of interlinked, diverse, and heterogeneous facts about entities

*Corresponding author. E-mail: aazzam@wu.ac.at.

and the relations between these diverse entities [2]. While the adoption of KGs has tangible benefits for commercial applications including Google, Microsoft, and Facebook, to name a few industry-led efforts, these mainly aim at building mostly centralized large-scale knowledge repositories empowering their providers' services. In addition to these efforts, several research fields have recognized the potential of KGs for scalable, *decentralized* data integration through the provision and inter-linking of diverse knowledge bases on the Web. Indeed, driven by the Linked Data principles, the amount of *open* KGs published on the Web has seen continuous growth over the past decade, constructing thousands of interconnected KGs connected as Linked Data, many of which comprise billions of edges [2]. Examples of such openly available interlinked KGs include DBpedia [3], Freebase[1] [4], Yago [5], and Wikidata [6]. These KGs are based on the semi-structured RDF data model and SPARQL query language to allow users to perform queries on the published KG following the Linked Data principles. These open KGs are queryable via Web interfaces such as public SPARQL endpoints or downloadable data dumps.

With the continuous growth of open KGs on the Web in both sizes and numbers, providing reliable queryable access to RDF graphs [7] encounters a serious challenge. Data publishers typically provide server-side access through public SPARQL endpoints; yet, whereas RDF triple stores offer impressive performance with single queries, they are resource-hungry (e.g., expensive to host) and hard to maintain in case of serving complex queries on large KGs to concurrent clients [8, 9]. This imposes the main threat to the progression of open KGs, since availability under high-demanding queries and limited server resources cannot be guaranteed [10]. For instance, SPARQLES [11], an online service for tracking the availability of $557$ SPARQL endpoints, shows that only $29.56\%$ of these endpoints were available (e.g. uptime last 7 days) as of Dec. 2020, and getting down to as few as $14.34\%$ by August 2023.[2]

To alleviate SPARQL endpoints limitations, Linked Data Fragments (LDF) has introduced a foundational framework to explore a spectrum of potential Web querying interfaces over KGs which distribute the query processing load between servers and clients [12]. Current LDF proposals enable data providers to publish large KGs based on a low-cost solution for evaluating low-expressive queries and that enhances servers availability. Examples of these interfaces are Triple Pattern Fragments (TPF) [13] and Bindings-Restricted TPF (brTPF) [14]. Yet, the evaluation of full SPARQL queries over these interfaces sometimes suffers from drastic performance degradation, due to local joins of query fragments on the client combined with a potential transfer of large unnecessary intermediate results, leading additionally to high network traffic between clients and servers.

In order to address the limitations of low-expressive LDF interfaces, in our work, we explore an alternative approach that rather will ship a set of KG partitions that can be locally queried on the client-side to retrieve the exact answer of the query. We call this approach *Partition-Based Linked Data Fragments*: partition-based LDF generalizes LDF interfaces, which returns compressed and queryable partitions that can be used to answer several triple patterns in a single request. In this context, serving KG partitions drastically reduces the need for unnecessary data transfers and a high number of requests in shared server- and client-side query processing. Hence, in this work, we first align formal definitions to uniformly present a variety of different existing LDF interfaces, and then we present a formalization of existing KG partitioning techniques (e.g., horizontal, vertical, etc.) to as partition-based LDF under the umbrella of the framework.

As a concrete implementation of a partition-based LDF, we initially proposed `smart-KG` [1], which is based on *family partitioning*. Family partitioning is inspired by Characteristic Sets [15, 16], which captures the entities (subjects) represented with the same set of predicates and groups them into star-shape KG partitions on the server-side. Family partitioning in its original form is optimized to serve star queries with unbound objects, that is, without restrictions on the object values. Our initial results in [1], which we herein significantly extend, show that shipping compressed and queryable family partitions, increases the server availability while achieving competitive query performance. As we will show in the present paper, there is still room for reducing the shipped KG partitions by further developing the partitioning mechanism. For instance, in practice, many star-shaped sub-queries include at least some bound objects for the *rdf:type* predicate. To verify this claim, we have analyzed the real-world LSQ query log [17], and found that $88\%$ of the queries contain star-shaped patterns with at least a type predicate with a bound object value (i.e. to a class).

[1] In fact, freebase, after being one of the first openly available KGs, has been discontinued and commercially been acquired and subsumed in Google's KG, cf. https://developers.google.com/freebase, last accessed August 2023

[2] https://sparqles.ai.wu.ac.at/availability, last accessed August 2023

Motivated by these findings, in this paper, we propose, formalize, and extend smart−KG towards smart−KG$^+$, where we additionally introduce a graph partitioning technique named *typed family-partitioning* that benefits from this phenomenon by horizontally partitioning the families based on the classes of the entities. In addition, we propose a new smart−KG$^+$ server-side partition-aware query planner, not present in smart−KG, to create an optimized query plan where the subqueries within a query are ordered based on the pre-computed cardinality estimations (i.e. characteristic sets): while minimally increasing server-side overhead for query planning, compared with the original smart−KG, based on the received query plan from the server, we can show increased effectiveness of the approach; we perform joins on the client-side locally based on implementing an asynchronous pipeline of iterators executing first the most selective iterator in order to produce the join results in an incremental fashion.

Finally, we evaluate smart−KG$^+$ and existing approaches using synthetic and real-world KGs ranging from 10 million up to 1 Billion triples. Overall, our results show that smart−KG$^+$ is on average 10 times faster, uses 5 times less network traffic, sends 20 times fewer requests, and requires 5 times less server CPU usage in an extensive comparison with not only the original, preliminary version of smart−KG, but also with other state-of-the-art approaches that fall within our generalised notion of (single-partition as well as partition-based) LDF.

**Contributions.** In summary, the novel contributions of this work are as follows:

- **C1** We extend the LDF specification and align formal definitions to uniformly present different LDF interfaces, including our introduced *partition-based LDF* approaches.
- **C2** We analyze existing partitioning techniques for RDF graphs and discuss their applicability to serve as partitioning mechanisms for partition-based LDF interfaces.
- **C3** We present a concrete implementation of partition-based LDF, smart−KG$^+$, that ships compressed, queryable KG partitions to distribute the processing of SPARQL queries between clients and servers. Our approach employs a new RDF graph partitioning technique named *(typed) family-partitioning* which extends family partitioning technique introduced in [1] to consider both predicates and classes specified in a query.
- **C4** We prove smart−KG$^+$'s correctness, in terms of soundness and completeness.
- **C5** Finally, we conduct an extensive empirical evaluation of concurrent query processing using smart−KG$^+$ and state-of-the-art LDF approaches on synthetic and real-world KGs.

**Paper Organization.** The remainder of this paper is structured as follows. The background of this work is in Section 2. In Section 3, we introduce possible concrete implementations of LDF APIs based on partition shipping. We present a design overview of the proposed approach smart−KG$^+$ in Section 4. In Section 4.2, we detail the KG partition creation process. Section 4.3 elaborates the query processing of smart−KG$^+$ and the dynamicity between clients and the server. An empirical evaluation and results are discussed in Section 5. We conclude in Section 6, where we also highlight future work directions.

## 2. Background

In this section, we first present basic notions on the RDF data model and the SPARQL query language. Then, we provide definitions alignment with the well-known Linked Data Fragment framework.

### 2.1. RDF and SPARQL

The Resource Description Framework (RDF) [18] is a graph-based data model to represent information about web resources (e.g. documents, people, sensors, etc.) and their relationships in the form of triples $(subject, predicate, object) \in (U \cup B) \times U \times (U \cup B \cup L)$, where $U$, $B$, $L$ are infinite, mutually disjoint sets of IRIs, blank nodes, and literals, respectively [19].

Let $t$ be a single *RDF triple* belonging to the *RDF knowledge graph* $G$. We use $subj(t)$, $pred(t)$, and $obj(t)$ to denote the components of $t$, which represent RDF terms including URIs/IRIs, blank nodes, and literals. The RDF knowledge graph $G$ is a finite set of such triples, where $subj(G)$, $pred(G)$, and $obj(G)$ represent the subjects, predicates, and objects contained within $G$.

RDF graphs can be queried using the SPARQL [20] query language, which relies on graph pattern matching. The atomic expression of SPARQL is a *triple pattern tp* from $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$ where elements from a set $V$ of variables are permitted, which are disjoint with the previously mentioned RDF terms $U$, $B$ and $L$.

Note that, in the context of our introduced work, without loss of generality – and similar to [13] – (i) we do not consider explicitly blank nodes in query patterns, which are just synonyms for (non-distinguished) variables in query patterns, and also (ii) assume blank nodes in the graphs $G$ just as constants like IRIs, leaving out the intricacies of blank node matching in the definition of the SPARQL specification [20].

A *Basic Graph Pattern (BGP)* consists of a conjunction of triple patterns also represented as a set $\{tp_1, \ldots, tp_n\}$. We denote (left-linear join order) query execution plans using sequences $(\ldots)$, i.e., for instance, $(tp_1, \ldots, tp_n)$ denotes a left-linear query execution plan $(\ldots (tp_1 \bowtie tp_2) \bowtie \ldots) \bowtie tp_n)$ where $\bowtie$ denotes a join operator.

Apart from BGPs, we will also consider in some discussions in this paper FILTER patterns, i.e., if $P$ is a BGP and $\phi$ is a FILTER condition, then $P$ FILTER $\phi$ is called a filtered graph pattern (FGP). For the sake of this paper, we restrict ourselves to simple filter conditions of the form $\phi = (term\ \theta\ const)$, where *term* is a subject, predicate, or object of triples, and $\theta$ is a comparison operator such as $<, >, =, \leqslant, and \geqslant$. In the general case, FILTER conditions could possibly contain multiple terms connected by the logical connectors such as $\neg$, $\vee$, and $\wedge$, etc. For any (BGP or FGP) query pattern $Q$ over an RDF graph $G$, we denote by $var(Q)$ its variables. Additionally, subj(Q), pred(Q), and obj(Q) denote the subjects, predicates, and objects extracted from the query, respectively.

The solutions of a query pattern $Q$ over a KG $G$ are denoted as $[\![Q]\!]_G = \Omega$, where each $\omega \in \Omega$ denotes a substitution from the variables in $Q$ matching triples in $G$. The solutions are given as sets $\Omega$ of *bindings*, i.e., mappings of the form, $\omega : var(Q) \rightarrow R$ to the set $R$ of RDF terms appearing in $G$, such that $G \models \omega(Q)$, i.e., $\omega(Q)$ forms a (sub)graph entailed by $G$. If there are no result mappings, then $[\![Q]\!]_G = \emptyset$, whereas a solution to a variable-free pattern is indicated by a single solution, i.e., $[\![Q]\!]_G = \{\omega_\emptyset\}$, with $\omega_\emptyset = \{\}$ being the empty mapping.

For BGPs and FGPs, $[\![\cdot]\!]_G$ can be defined as follows, cf. [21]. Let $P$ be a BGP and $\phi$ be a FILTER condition, then:

$$[\![P]\!]_G = \{\omega \mid dom(\omega) = vars(P) \text{ and } \omega(P) \in G\}$$
$$[\![P \text{ FILTER } \phi]\!]_G = \{\omega \in [\![P]\!]_G \mid \omega(\phi) = \top\}$$

where by $\omega(\phi)$ we refer to the truth evaluation of a simple FILTER condition $\phi$ (as defined above) with variables substituted according to $\omega$.

Two mappings $\omega_1, \omega_2$ are *compatible* [21], denoted as $\omega_1 || \omega_2$, if for any $v \in dom(\omega_1) \cap dom(\omega_2), \omega_1(v) = \omega_2(v)$.

Since the main focus of this paper is on BGPs, we omit further details about the other patterns which in essence can be constructed on top of the base retrieval functionality of BGPs (cf. [21]). Note that FGPs and also other patterns e.g. `UNION`, `OPTIONAL`, etc. are mostly expected to be executed on the client-side in the approaches we discuss further. In future work, we look into the direction of distributing the execution of other patterns between server and client to further increase the efficiency of our approach.

### 2.2. RDF HDT Compression

For efficient storage and querying of RDF graphs, we will herein particularly rely on HDT [22], a well-known compressed format for RDF graphs that permits efficient triple pattern retrieval over the compressed data. HDT has three main components: (i) a *dictionary* maps RDF terms to IDs, such that (ii) the *triples* component encodes the resulting *ID-graph* (i.e. a graph of ID-triples after replacing RDF terms by their corresponding dictionary IDs) as a set of *adjacency lists*, one per different subject in the graph. In addition, (iii) the *header* provides descriptive metadata (publishing information, basic statistics, etc.) about the RDF graph.That is, an HDT file of a graph $G$ consists of a header $H$, a dictionary $D$ and triples $T$, i.e., $HDT(G) = (H, D, T)$, stored in compressed, queryable encodings: Both the HDT dictionary and triples are self-indexed to support efficient retrieval operations. The dictionary implements prefix-based Front-Coding compression [23], which allows for high compression ratios and efficient *string-to-id* and *id-to-string* operations. Triples are indexed by subject (in *SPO* order) using bitmaps [22].

RDF graphs compressed with HDT can be queried loaded in memory or mapped from disk without prior decompression. HDT exhibits competitive performance for scan queries as well as triple pattern execution when the subject is provided. In addition, HDT compressed graphs are typically enriched with a companion HDT index file [24]. This

additional file includes two inverted indexes on the ID-triples (in $OPS$ and $PSO$ order) to achieve high performance for resolving all SPARQL triple patterns.

In fact, HDT has been used as an efficient backend for many implementations of Linked Data Fragments query interfaces, which we will discuss next.

### 2.3. Linked Data Fragments (LDF)

In this section, we characterize existing Web KGs querying interfaces following the foundations set by the Linked Data Fragments framework (LDF) [13]. LDF has been designed to abstractly model Web KG querying interfaces with a higher or lesser degree of expressivity and balance for distributing the query processing load between clients and servers. In essence, LDF characterizes interfaces that enable live querying to fragments of a KG $G$ based on a limited range of query patterns (e.g, single triple patterns or star patterns) that a client is allowed to request from the server. Generally, the aim of different LDF interfaces is to mitigate the expensive server-side computation load and to enable efficient reusable caching for these limited patterns, while shifting the processing of more complex patterns to the client. Several LDF interfaces also support additional controls. For instance, a control parameter to transfer intermediate bindings together with query patterns, or a control to specify the page size to determine the "chunk size" of results batched in each server response.

In the following, we slightly adapt the original specification of the LDF framework [12, 13] to align formal definitions and notations to uniformly describe the current KG APIs, while we leave out herein details in LDF such as metadata sent along with query results and hypermedia controls:

**Definition 2.1** (LDF API, adapted from [13, 25])**.** *An LDF API of a KG $G$ accessible at an endpoint IRI $u$[3] is a tuple $f = \langle s, \Phi \rangle$ with*

- *a selector function $\sigma(G, Q, \Omega)$ that defines how a fragment $\Gamma \subseteq G$, or alternatively a set of fragments*[4] *$\Gamma^* \subseteq 2^G$ is constructed upon calls to the API. The selector function $\sigma$ has as parameters an RDF graph $G$, a SPARQL pattern $Q$, and a set of bindings $\Omega$,*[5]*,*
- *a paging mechanism $\Phi(n, l, o)$ parameterized by $n, l, o \in \mathbb{N}_0$ denoting maximum page size, limit, and offset.*

For BGP queries $Q$, we define two specific variants of selector functions, $s(\cdot)$ and $s^*(\cdot)$, which differ essentially in terms of returning either a single graph containing all triples relevant to *any* solution or one subgraph *per* solution $\omega \in [\![Q]\!]_G$:

**Definition 2.2** (Standard Selector Function)**.** $s(G, Q, \Omega) = \{t \in \omega(Q) \mid \omega \in [\![Q]\!]_G : G \models \omega(Q) \wedge (\Omega \neq \emptyset \implies \exists \omega' \in \Omega : \omega' || \omega)\}$

**Definition 2.3** (Overloaded Standard Selector Function)**.** $s^*(G, Q, \Omega) = \{\omega(Q) \mid \omega \in [\![Q]\!]_G : G \models \omega(Q) \wedge (\Omega \neq \emptyset \implies \exists \omega' \in \Omega : \omega' || \omega)\}$

Note that, whenever the set of bindings $\Omega$ is not considered (i.e. only empty binding sets $\Omega = \emptyset$ are expected) in a particular selector function, we will conveniently also just write short $\sigma(G, Q)$ (or $s(G, Q)$, $s^*(G, Q)$, resp.) instead of $\sigma(G, Q, \Omega)$ (or $s(G, Q, \Omega)$, $s^*(G, Q, \Omega)$, resp.) in the following.[6] As we will see, all existing LDF APIs considered in this paper and summarised in Table 1 can be expressed in terms of one of the two standard selector functions $s(\cdot)$ and $s^*(\cdot)$, whereas we will extend and modify those – in terms of partition-based LDF – in the following.

The general *paging mechanism* $\Phi$ we use in this paper enables the ability to retrieve the result in batches e.g., for the cases where $\Gamma$ (or, resp., $\Gamma^*$) is overly large, or, resp. when only partial results are required or to enable incremental results. Hence, we assume that $\Phi(n, l, o)$ simply defines a mechanism to divide $\Gamma$ into a set of partitions

[3] Via this base IRI the API can be accessed and queried as well as additional controls can be submitted.

[4] We note that this is a generalization from the original LDF proposal, which – technically – could be realized, for instance, by returning RDF datasets in the sense of SPARQL (consisting of a default graph and optionally a set of (named) graphs), or resp. a set of quads instead of triples.

[5] We note that this strict definition of allowed parameters for $\sigma$ is not made in [13], but we will rather use those here to describe the considered APIs uniformly.

[6] Observe that, in the spirit of brTPF, in Definitions 2.2+2.3, the existence of a "compatible" binding in $\Omega$ is only checked for non-empty $\Omega$.

(or *pages*) $\Gamma^* = \{\Gamma_0, \ldots, \Gamma_{k-1}\}$, where for each page $\Gamma_i$ it is guaranteed that $|\Gamma_i| < n$ (i.e. $\Gamma_i$ does not contain more than $n$ triples), and $l$ and $o$, resp. would allow to request the pages from $\Gamma_o$ to $\Gamma_{o+l-1}$. We assume $l$ to default to $l = 1$, $o$ to default to $o = 0$, and finally $n = \infty$ signifying that whole graph $\Gamma$ (or, resp., $\Gamma^*$) should be returned. "Pagination", i.e., retrieving $\Gamma^*$ in chunks of $l$ pages could then be achieved in terms of iteratively calling the LDF API with $\Phi(n, l, o)$ with increasing $o$, starting from $o = 0$ in steps of $o := o + l$. As such $l$, $o$ could be viewed analogous to the SPARQL LIMIT and OFFSET modifiers but applied to pages instead of individual solution mapping; we note here, that we define $\Phi$ and likewise $\Gamma^*$ in a more general way than the original LDF paper [13],[7] allowing parameters to both specify a maximum page size $n$, and the number of pages $l$ retrieved per request as *separate* parameters, starting from a page index offset $o$, in order to allow flexible interpretations of the LDF "metaphor", fitting various interfaces and partitionings. We also note that in the following – as opposed to and generalizing [13] – we do not necessarily consider $\Gamma_i$ and $\Gamma_j$ disjoint for $i \neq j$.

*2.3.1. Characterization of Existing KG Interfaces as Linked Data Fragments (LDF)*

In the following, we will describe existing LDF interfaces from the literature, summarizing their respective characterizations in terms of the pre-described definitions in Table 1.

**Data Dump.** This approach is a client-side solution where data publishers enable clients to access a data dump of an entire KG, at best, in an RDF serialization. To perform a SPARQL query, clients request an entire KG from the server and deploy an RDF triple store to locally process their queries. A use case where data dumps can be a very valuable solution is when the clients have powerful processing resources while demanding resource-hungry query workload tasks. However, in general, the data dumps solution puts the processing cost on the clients, plus incurs potentially high network traffic on both client and server sides in the case of frequently evolving KGs.

**Triple Pattern Fragments (TPF).** The TPF [13] interface enables reliable querying over KGs by limiting the server functionality to only answer single triple patterns and delegating the processing of more complex patterns – and particularly joins – to the client-side engines [26–28]. TPF clients receive paginated intermediate results of each triple pattern in the query and incrementally combine the intermediate results to compute the complete results on the client. The experimental evaluations [13] show that TPF, powered by an HDT backend, increases the server availability compared to a traditional query shipping approach (i.e. SPARQL endpoints). However, this comes at the cost of a significant increase in network traffic including the number of HTTP requests and the shipped data. In particular, non-selective queries (i.e. queries with high cardinality triple patterns) can suffer from poor performance as a consequence of the potentially high number of useless shipped intermediate results (i.e. transferred data that does not contribute to the final query answer).

**Binding-Restricted Triple Pattern Fragments (brTPF).** The brTPF [14] interface is an extension of TPF that strives to reduce the network traffic by additionally permitting arbitrary $\Omega \neq \emptyset$. The attached solution mappings $\Omega$ from the previously evaluated triple patterns potentially reduce the number of requests to the server, plus provide a higher query performance than TPF. However, brTPF still potentially encounters serious delays with increasing the number of concurrent clients or with queries that require shipping a large number of intermediate results.

**Star Pattern Fragments (SPF).** The SPF [29] interface proposes to generalize brTPF from evaluating single triple patterns to evaluating star-shaped subqueries as well on the *server*. Similar to TPF, more complex queries involving joins over stars or single triples are processed on the client. Still, evaluating star-shaped subqueries directly on the server may drastically reduce the number of requests made during query processing while still maintaining a relatively low server load since star patterns can be answered in a relatively efficient manner by the server [21]. For processing joins efficiently, analogously to brTPF, bindings can be shipped along with each star-shaped subquery. SPF, as an instance of LDF, differs from brTPF with respect to the restriction of the selector function and allowed patterns as defined in Table 1. Experiments [29] show that SPF (compared to brTPF) can decrease the number of requests made to the server and intermediate result sizes transferred to the client, maintaining a comparable low network load.

[7] We note our slightly deviating notation: our $\Gamma^*$ is denoted as $\Phi$ in [13], originally, whereas we use $\Phi(n, l, o)$ to refer to parameterising a request to the selector function, requesting particular pages.

Table 1
Aligned formal definitions and notations with LDF original specifications to uniformly present different existing LDF APIs

| LDF Interface | Definition |
|---|---|
| Data Dump | The selector function is $s(\cdot)$<br>The only admissible form of $Q$ and $\Omega$ are $Q = \{(?s, ?p, ?o)\}$ and $\Omega = \emptyset$<br>The only admissible parameter for $\Phi(n, l, o)$ is $\Phi(\infty, 1, 0) = \{\Gamma_0\} = \{\Gamma\}$ |
| TPF | The selector function is $s(\cdot)$<br>The only admissible form of $Q$ are triple patterns and $\Omega = \emptyset$<br>$\Phi(n, l, o)$ allows results to be "batched" into chunks of $n$ triples,i.e., in TPF the publisher can set $n$ as a parameter, whereas limit $l = 1$ as it is possible to only retrieve one page at a time, and offset $o$ is the page number requested by the client |
| brTPF | The selector function is $s(\cdot)$<br>The only admissible form of $Q$ are triple patterns<br>$\Omega$ can be any set of bindings<br>$\Phi(n, l, o)$ as defined in TPF |
| SPF | The selector function is $s^*(G, Q, \Omega)$, i.e., $s^*(\cdot)$ is used to return results per pattern solution<br>the only admissible form of $Q$ are star-shaped BGPs<br>$\Omega$ can be any set of bindings<br>$\Phi(n, l, o)$: as solutions are returned per pattern solution, $n$ is fixed to the star pattern of size $k$ but SPF also allows paginating over solutions, i.e., retrieving results in chunks of $l$ solutions. |
| SPARQL Endpoints | A variant of $s^*(\cdot)$ by returning subgraphs of the form $\omega(Q)$. In practice, SPARQL endpoints return solution mappings, yet, it is possible to devise a correspondence between these and $s^*(\cdot)$<br>Any pattern $Q$ is admissible<br>$\Omega = \emptyset$, unless `VALUES` patterns are considered. In this case, $\Omega$ is encoded in the `VALUES` clause of $Q$<br>$\Phi$: the standard `LIMIT` and `OFFSET` operators for BPGs could be considered as `LIMIT` $l$ and `OFFSET` $o$ such that $n$ is fixed to the cardinality of the BGP $Q$, i.e. $n = \|Q\|$. |
| SAGE | A variant of $s^*(\cdot)$ by returning subgraphs of the form $\omega(Q)$, analogous to SPARQL endpoints.<br>Any pattern $Q$ is admissible<br>$\Omega = \emptyset$, unless `VALUES` patterns are considered. In this case, $\Omega$ is encoded in the `VALUES` clause of $Q$<br>As for the interpretation of $\Phi$, we distinguish two cases:<br>$\Phi(n, \infty, o)$: assuming that $Q$ *does not* include the keywords `LIMIT` and `OFFSET`. $o$ may be thought of as being used to indicate that $\{\Gamma_0, \ldots, \Gamma_{o-1}\}$ has been received by the client. In practice, the SAGE client sends the last solution mapping $\omega$ that has been produced in $\Gamma_{o-1}$. Still, it is possible to devise a correspondence between $\omega$ and $o$.<br>$\Phi(n, l, o + o')$: assuming that $Q$ *does* include the keywords `LIMIT` $l$ and `OFFSET` $o'$. $o$ is defined as in the previous case;<br>$n$, in both cases, is again, as in SPARQL endpoints, fixed to $n = \|Q\|$, i.e., each page corresponds to a solution. |

**SPARQL Endpoint.** A SPARQL endpoint provides a purely server-side efficient solution for SPARQL queries. However, as we show in Table 1, we can also understand any SPARQL endpoint as an LDF interface in our introduced terminology. SPARQL endpoints minimize load on the client-side which only receives the final results of the submitted query. To date, hundreds of public SPARQL endpoints have been published [8], serving arbitrary SPARQL queries from remote clients. In this client-server scenario, clients are limited to sending queries and receiving results, whereas servers are in charge of the full query planning, execution, and shipping of results. SPARQL endpoints provide outstanding performance under low query loads. However, with increasing the number of concurrent clients and the complexity of the submitted queries, SPARQL endpoints potentially overload server resources and the submitted queries struggle from excessive delays that lead to the acknowledged scalability issues (i.e. low availability and poor performance) on concurrent query workloads [13]. That is, SPARQL endpoints are expensive to host and maintain from the data publishers' perspective. Furthermore, several recent studies on public SPARQL

endpoints [8, 10] show that far more than $50\%$ of the SPARQL endpoints are not responding to the requests. In practice, public SPARQL endpoints impose restrictions to ensure a balanced distribution of server resources among clients: e.g., public SPARQL endpoints regulate the number of submitted queries from each IP address. Such restrictions largely defeat the vision of developing live applications based on public endpoints [30].

**SAGE.** SaGe [31] can be considered a variant of general SPARQL endpoints that supports Web preemption in order to guarantee a more fair distribution of server resources amongst concurrent clients. Under Web preemption, the server suspends a running query $Q$ after a predefined time quantum $\tau$ and returns partial results $\{\Gamma_0, \ldots, \Gamma_{o-1}\}$ to the client. A SPARQL query $Q$ is resumed based upon the client's request; this process is repeated until all results are produced. This ability enables SaGe engine to prevent long-running queries from exploiting the server resources, especially under high concurrent load [31]. The experimental evaluation [31] demonstrates that SaGe improves the average time required to receive the first result and the average workload completion time per client. In general, SaGe has impressive performance for most of the query shapes. However, SaGe suffers from excessive delay in the case of high concurrent clients with complex queries due to frequent query context switching.

### *2.3.2. Partition-based LDF*

Unlike the previous LDF approaches, which – given a particular admissible query $Q$ – would return a graph or partition that exactly contains the query results, in this paper, we will focus on an alternative approach that rather will ship an overestimate from a set of hosted partitions that potentially can be used to answer the query, which we will call *partition-based LDF* approaches: partition-based LDF can be seen as a generalization of the aforementioned, existing LDF interfaces, which - instead of the exact triples matching a particular admissible query pattern $Q$, rather returns a subset of partitions from a pre-computed partitioning $\mathcal{G} = \{G_1, \ldots, G_n\}$ of $G$, such that $G = G_1 \cup G_2 \cup \cdots \cup G_n$, where $\forall i \neq j\ G_i \cap G_j = \emptyset$. I.e,. $\mathcal{G}$ is a cover of $G$.

That is, the idea here is that a *partition-based LDF server* serves $\mathcal{G}$ such that upon an LDF API call with a query pattern $Q$ the selector function $\sigma(G, Q) \subseteq \mathcal{G}$ returns a subset of matching partitions that contain all query answers for $Q$. That is, it is ensured that

$$[\![Q]\!]_G \equiv [\![Q]\!]_{\bigcup_{G_i \in \sigma(Q,G)} G_i} \tag{1}$$

hence, the client can therefore compute the complete result of the actual query $Q$ from just calling and retrieving $\sigma(G, Q)$ from the server.

As we will see, different partitioning techniques lend themselves to this overall idea better or worse: the tricky part is to find a partitioning $\mathcal{G}$ such that (1) $\sigma(G, Q)$ can be easily computed from $Q$, and (2) $\sigma(G, Q)$ provides a "close estimate" minimizing the number of partitions to be shipped and where the union of these partitions does contain all necessary, but not too many unnecessary triples for computing the actual query $Q$, and finally (3) all possible partitions for any admissible queries $Q$ in the range of $\sigma(Q, G)$ can be efficiently served (and, ideally, pre-computed) on and LDF server.

In this context, we note that shipping a full **Data Dump** could be considered as a "trivial" partitioning-shipping technique, where

- $\mathcal{G} = \{G\}$
- any query $Q$ is admissible
- $\sigma(G, Q) = \{s(G, \{?s, ?p, ?o\})\} = \mathcal{G}$

As such, non-trivial partition-based LDF methods could be considered as shipping only a "necessary subset of partial dumps per query". Further, for partition-based LDF interfaces, in general, we herein will assume

- $\Omega = \emptyset$ is the only admissible binding set, i.e., we do not consider binding restrictions,
- $\Phi$: only $n = \infty$ is admissible, i.e., no paging is supported since the union of all relevant partitions will be typically needed to compute the query results.

In the following section, we will review several existing RDF partitioning techniques to assess their applicability in a Web querying environment fitting within this framework.

Table 2
An overview of the exiting graph partitioning mechanisms utilized in RDF engines

| Partitioning Mechanisms | RDF Systems |
|---|---|
| Vertical Partitioning | SW-Store [32], PRoST [33], SPARQLGX [34], Sempala [35], S2RDF [36], SparkRDF [37], SANSA [38], CliqueSquare [39],PigSPARQL [40], Jena-HBase [41], and HadoopRDF [42] |
| Horizontal Partitioning | AllegroGraph[8], Blazegraph[9], Akhter et. al [43], SHARD [44], DiStRDF [45], and Partout [46] |
| Hash Partitioning | YARS2 [47], TriAD [48], AdPart [49], PigSPARQL [40], CliqueSquare [39], Koral [50], CumulusRDF [51], SHAPE [52],and SHARD [44] |
| Workload-aware Partitioning | Partout [46], chameleon-db [53], WARP [54], and WORQ [55]. |
| K-way Partitioning | Akhter et. al [43], EAGRE [56], H-RDF-3X [57], TriAD-SG [48] |

## 3. Concrete Implementations of Partition-based LDF

In this section, we analyze various partitioning techniques utilized in prior works (both centralized and distributed RDF processing) and their applicability to efficient Web querying and as a basis for partition-based LDF, which we have introduced – on an abstract level – above. In our analysis, we first analyze the advantages and limitations of existing partitioning techniques [58–60]. A summary is presented in Table 2.

### *3.1. Vertical Partitioning (VP)*

VP [61] creates a partition for each unique predicate in $pred(G)$, i.e., in our terms,

$$\mathcal{G} = \{G_p \mid p \in pred(G) \wedge G_p = \{\omega((?s, p, ?o)) \mid \omega \in [\![(?s, p, ?o)]\!]_G\}\} \tag{2}$$

Next, *admissible queries* are any single triple pattern queries $Q = \{tp\}$ where

$$\sigma(G, Q) = \{G_p \in \mathcal{G} \mid \{p\} = pred(Q) \cap pred(G) \vee pred(Q) \text{ is a variable}\} \tag{3}$$

That is, for any triple pattern query $Q$, either a single predicate partition corresponding to the query predicate, or all predicate partitions would be returned.

Many RDF processing systems (cf. Table 2) report achieving a high query performance using vertical partitioning. Yet, as a partitioning mechanism for partition-based LDFs interfaces, this approach only works well for triple pattern queries with bounded predicates, whereas other triple patterns require shipping all predicate partitions. Along these lines, assuming all predicates in $Q$ are bound, a strict lower bound for the number of shipped partitions is $|pred(Q) \cap pred(G)|$, because only the partitions for predicates mentioned in the query that also occur in $G$ are shipped.

A second drawback of using vertical partitioning in the context of partition-based LDF interfaces is that it only supports single triple queries while any joins or more complex patterns would need probably to be fully evaluated on the client side. Also, full vertical partition shipping has potential downsides compared with TPF or brTPF, which solves any binding in triple patterns directly on the server side. For all these reasons, we will in our proposed approach rather use (br)TPF directly for single triple queries.

### 3.2. Horizontal/Range/Sharding Partitioning

In the context of distributed relational databases, horizontal partitioning involves splitting a relation horizontally (i.e. row-wise) into sub-relations based on selections to enhance the load balancing. Analogously, RDF management systems have adopted horizontal partitioning strategies to distribute the triples of an RDF graph into multiple partitions based on certain selection criteria. In these strategies, the selection is typically used to generate horizontal subsets of the RDF triples for very common predicates (such as e.g. `rdf:type`, which often does not lend itself well to vertical partitioning techniques), where each subset consists of all the triples that satisfy a predetermined selection condition on the objects or subjects. Herein, we exemplify horizontal partitioning based on object ranges; that is, we assume partitions per $n$ object ranges (e.g. from a histogram) can be split into a set of ordered values $\{v_0, \ldots, v_n\}$. Given the RDF model, this is not an unreasonable assumption, indeed, both literals and likewise URIs could be assumed to be ordered with respect to their string representations, and – even if many real-world RDF graphs do not contain blank nodes – also blank nodes could, while not ordered in the RDF model itself, be canonicalised [62] and ordered, respectively. Accordingly, we can define

$$\mathcal{G} = \{G_i \mid 1 \leqslant i \leqslant n \wedge G_i = \{\omega((?s, ?p, ?o)) \mid \omega \in [\![(?s, ?p, ?o)\ \texttt{FILTER}\ (v_{i-1} <?o \wedge ?o \leqslant v_i)]\!]_G\}\} \tag{4}$$

Object-based horizontal partitioning could be used for partition shipping, where any BGP query $Q$ is admissible that consists of triples with the same object, i.e., $obj(Q) = \{o\}$ (which of course includes single triple queries with bounded object), but, again, for unbounded objects, the entire partitioning $\mathcal{G}$ would need to be shipped:

$$\sigma(G, Q) = \{G_i \in \mathcal{G} \mid (v_{i-1} < o \wedge o \leqslant v_i) \vee o \text{ is a variable}\} \tag{5}$$

Horizontal partitioning could be analogously defined for bound subjects, or be combined with vertical partitioning (i.e. be used to further subdivide vertical partitions); in fact, vertical partitioning as defined above could be viewed as a "special form" of horizontal partitioning on the predicate position, with "predicate ranges" corresponding to the single predicates in $pred(G)$.

Variations of horizontal partitioning have been used successfully by several RDF systems, especially in distributed environments (cf. Table 2), where partitions are allocated to different nodes while minimizing the communication cost among the nodes (by placing jointly queried data together) and balancing the node workload (by placing highly requested partitions in different nodes). In general, horizontal partitioning supports efficient querying for queries that require shipping a single partition based on the FILTER condition that defines the shipped partitions. As such, there are similar (dis-)advantages as for vertical partitioning: for our example of horizontal partitioning on the object, whenever the object is unbound, all partitions would need to be retrieved. Likewise, depending on the choice of ranges ($v_1$ to $v_n$) to "split" the partitions and data distribution, the matching partitions could contain potentially large amounts of irrelevant data or different horizontal partitions could contain a prohibitively large superset of the answers of the query, e.g., by including further predicates which are not requested in the query. The latter could be remedied by combining more sophisticated forms of vertical and horizontal partitioning. For example, family-based partitioning techniques described in Sect. 3.6 and Sect. 3.7 can be seen in a sense as vertical partitioning and a combination of vertical and horizontal partitioning, respectively.

### 3.3. Hash Partitioning (HP)

Hash-based partitioning is a common partitioning strategy among RDF distributed systems. For instance, *position-based hashing* is a lightweight partitioning strategy that applies a hash function to a particular position (e.g. subject-based hashing) in triples, distributing the RDF triples according to their hash values into a fixed number of $n$ bins. Thus, all the triples with the same value in this position (e.g. same subject) are allocated to one partition. Hash partitioning is computationally inexpensive plus the hash operation can be efficiently computed in parallel. However, as usual with hashing, hash collisions may cause skewed partition sizes. Hash-based partitioning could be

defined in a very similar manner as above, exemplified here for subject-based hashing with $n$ partitions. Assuming a suitable hash function $h(\cdot)$:

$$\mathcal{G} = \{G_i \mid 1 \leqslant i \leqslant n \wedge G_i = \{\omega((?s,?p,?o)) \mid \omega \in [\![(?s,?p,?o)\ \texttt{FILTER}\ (h(?s) = i)]\!]_G\}\} \tag{6}$$

For position-based hashing (analogously to position-based horizontal partitioning explained above), any basic graph patterns sharing the same value in the respective position, e.g. subjects, would be admissible patterns. For such admissible queries $\sigma(G,Q)$ could again be analogously defined based on the hash function $h(\cdot)$ of the resp. position, that is e.g. based on $h(subj(Q))$, as above, with the same problems of retrieving all $\mathcal{G}$ whenever the subject is unbound. Likewise, these definitions can easily be extended to object, predicate, or even triple-based hashing (based on a "ternary" hash function $h(s,p,o)$).

Position-based hashing can be extended by specific hash functions, e.g. prefix-hashing [50], to ensure that subjects (or other position terms) with the same prefix end up in the same partition, which can be exploited in range queries. Another extension is k-hop hashing which could cater for certain path queries, by creating (potentially overlapping) partitions that extend simple hash-based partitions with the k-hop neighborhoods of the hashed triples [52].

### *3.4. Workload-aware partitioning*

Workload-aware partitioning makes use of query workloads in order to partition RDF graphs. Ideally, the query workload includes representative queries extracted from a real-world or a synthetic/simulated query log.

Several RDF distributed systems rely on workload-aware partitioning such as Partout [46], chameleon-db [53], WARP [54], and WORQ [55]. Bonifati et. al [63] has conducted an analytical study of end users' queries harvested from real-world query logs of SPARQL endpoints. According to the analysis of the graph structure of queries, tree-like shapes such as single triple patterns, chains, stars, trees, and forests are the most observed shapes. We consider the aforementioned observation especially star queries in family partitioning technique introduced in Sec. 3.6.

In our context, workload-aware partitioning could be seen as a form of "caching", where subgraphs containing a superset of or exactly the results of particularly common sub-queries could be stored as separate partitions. However, in order to make use of such caching, complex queries would need to be analyzed whether they contain any of these "cached" subqueries or respectively subqueries subsumed by the cached queries. Since such a form of partitioning is rather related to index-learning from query logs, a concrete formalization depends on formalizing/extractable common query patterns from such query logs. We see various options here and consider them as somewhat complementary and orthogonal to our current work. For instance, the answers of repetitive queries can be precomputed and stored within dedicated partitions, that can be shipped directly to clients. This approach serves to mitigate the computational load imposed by recurrent queries [64]. Bonifati et al. [65] observed that robotic queries are frequently duplicated and the server computation can be reduced by materializing partitions for such queries. In the present paper, we restrict the scope to partitioning definable by the (characteristics of the) graph only. We therefore leave a concrete formalization/implementation of partition-based LDF following this idea to future work.

### *3.5. K-way Partitioning (KP)*

Similarly, K-way partitioning is not directly amenable to our framework: K-way partitioning algorithms, such as [66] strive to partition the graph into roughly equal-sized smaller graphs with the intention of minimizing the number of edges linking vertices from different partitions and thus could be viewed rather as a "clustering" technique for RDF graphs than partitioning based on/or specifically used for evaluating particular query patterns. As such, we also leave it open for future work on how/whether such techniques could be used for computing a partitioning $\mathcal{G}$ that allows deriving an easy-to-compute selector function $\sigma$.

### *3.6. Family-Based Partitioning of RDF Graphs*

After having discussed various existing partitioning techniques, primarily in the context of single triple queries, we herein would like to focus on a novel partitioning technique, that we previously introduced [1]. The overall idea

of this partitioning technique is to serve partitions that cater for efficient evaluation of star-shaped (sub-)queries on the *client-side*, somewhat orthogonal to the above-mentioned SPF LDF interface on the server-side.

The intuition here is that real-world RDF graphs commonly exhibit an inherent structure due to the recurring occurrence of identical subject descriptions forming such common star-structures, i.e., many subjects of the same type share the same combinations of predicates. The assumption here is that in real-world RDF graphs, subjects with similar characteristics are described in the same fashion forming such common star-structures, i.e., many subjects of the same type share the same combinations of predicates. For instance, predicates describing *Films* (e.g., director, starring, launchDate, language, etc.) are different than those describing *Persons* (e.g., birthday, nationality, etc.) in DBpedia. In the literature, so-called characteristic sets [15, 16] have been defined to capture these latent structures that eventually construct a "soft schema" from the entities that are semantically similar in a graph.

The structures described by Neumann and Moerkotte [15, 16] are effectively represented using the concept of characteristic sets, commonly referred to as predicate families [67] (or simply families). We define the *predicate family* of a subject $s$, $F(s)$, as the set of predicates related to the subject $s$, that is:

$$F(s) = \{p \mid \exists o \in obj(G) : (s, p, o) \in G\} \tag{7}$$

Analogously, we denote as $F(G)$ or just $F$, to the set of all different predicate families occurring in $G$, as follows:

$$F(G) = \{F(x) \mid x \in subj(G)\} \tag{8}$$

Indeed, predicate families imply a partitioning

$$\mathcal{G} = \{G_{F_i} \mid F_i \in F(G)\} \tag{9}$$

usable for partition-based LDF as defined above, where each partition $G_{F_i}$ is defined by a corresponding respective predicate familiy $F_i \in F(G)$ as follows:

$$G_{F_i} = \{(s, p, o) \in G \mid F(s) = F_i\} \tag{10}$$

We will refer to this partitioning as *family-partitioning*; slightly abusing notation we will simply write $G_i$ for $G_{F_i}$ in the following. Next, the admissible queries for family-partitioning are star-shaped query patterns, i.e., BGPs composed of $k$ triple patterns form $Q = \{(s, p_i, o_i) \mid 1 \leqslant i \leqslant k,\ s \in V \cup U, p_i \in U, o_i \in V \cup U \cup L\}$ with a single common subject $s$, where

$$\sigma(G, Q) = \{G_i \in \mathcal{G} \mid pred(Q) \subseteq F_i\} \tag{11}$$

Obviously, for any star-shaped query, $\sigma(G, Q)$ contains all relevant triples from $G$ to compute the answers.

To illustrate the previous definitions consider the KG $G$ shown in Fig. 1, and the predicate families shown in Fig. 2. Following the definition of predicate family in Eq. (8), the subjects *s1* and *s2* belong to the same family $F_1$, as they have the same predicates. The subject *s2* belongs to family $F_2$. For the KG $G$, there are two families denoted $F(G)$, i.e., $F_1$ and $F_2$. Lastly, each of these families induces a partition over $G$. For example, $G_{F_2}$ contains all the triples of subjects that belong to family $F_2$, which in this case is triples *t8* and *t9*. Lastly, the set of partitions computed for $G$, denoted $\mathcal{G}$ are $G_{F_1}$ and $G_{F_2}$.

In our concrete implementation in Section 4, we will also exploit the fact that predicate families or *characteristic sets* as the basis for family partitioning have been used successfully for query execution and join evaluation: the ability of characteristic sets to provide an inherent partitioning of an RDF graph has been

:s1 rdf:type :Film. #t1
:s1 rdf:type :Work. #t2
:s1 :starring :o1. #t3
:s1 :director :o2.#t4

:s2 rdf:type :Work. #t5
:s2 :starring :o1. #t6
:s2 :director o3. #t7

:s3 rdf:type :Work. #t8
:s3 :director :o4. #t9

Fig. 1.: KG example

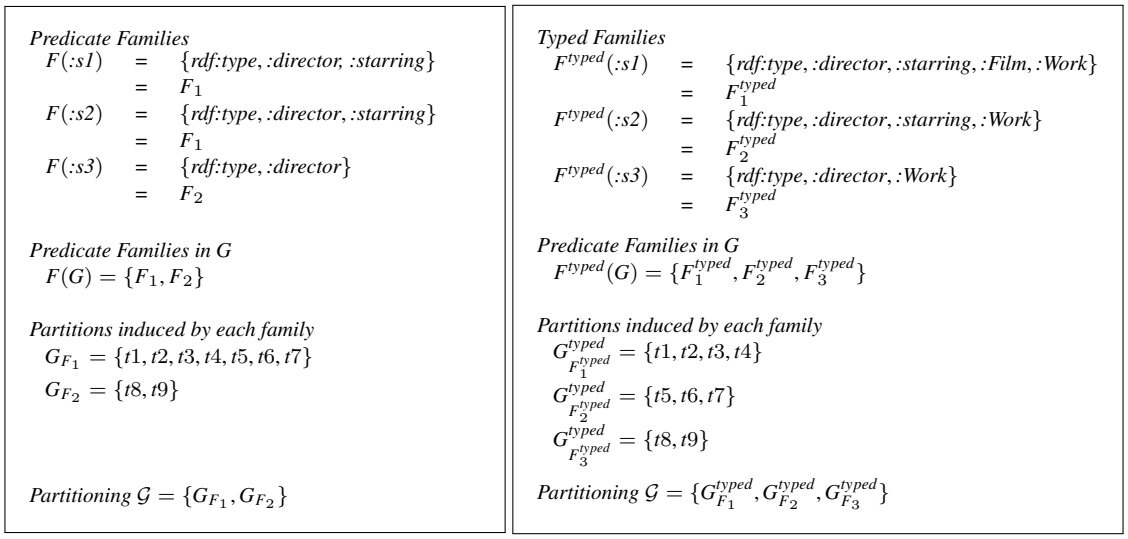



Fig. 2. Predicate families and typed families for the KG shown in Fig. 1

utilized for (i) cardinality estimation [15, 16] for SPARQL join optimization, (ii) improving RDF graph compressibility [68], and (iii) building an indexing scheme such as in AxonDB [69] which extends the notion of characteristic sets also to object nodes to speed up SPARQL query performance of AxonDB.

### *3.7. Typed Family-Partitioning*

While – as we will see – family-partitioning provides a solid basis for partition-based LDF, unfortunately, family partition sizes can be significantly skewed for very popular classes (with a large number of instances), or, respectively, very large partitions could be further subdivided by the different (sub-)classes occurring for subjects. For instance, common attributes $\{$*rdf:type*, *:director*, *:starring*$\}$ for subjects of the class *Film*, would also occur for each of the subclasses of *Film*. Intuitively, one can further subdivide each family partition "horizontally", by the different *rdf:type*s per subject. Further, we note that, based on observations of query logs for common public SPARQL query services, a large number of user queries include *rdf:type* predicates with a bound object: to back up this claim, we analyzed the real-world DBpedia LSQ [17] query log and found out that the percentage of queries with at least one star query with a *rdf:type* predicate with a bound object is $88\%$ (excluding single triple queries).

Based on these observations, we propose an extension of family-based partitioning, called *typed family-partitioning*. Assuming (without loss of generality) that the set of class URIs and predicate URIs are disjoint,[10] we can extend the concept of (predicate) families to *typed families* as follows:

$$F^{typed}(s) = F(s) \cup \{c \mid (s, \textit{rdf:type}, c) \in G\} \tag{12}$$

Analogously, we extend the other notions from above, i.e., the set of typed families for a graph $G$:

$$F^{typed}(G) = \{F^{typed}(x) \mid x \in subj(G)\} \tag{13}$$

and again the notion of *typed partitions* $G^{typed}_{F_i}$ corresponding to a famlily $F_i \in F^{typed}(G)$ implies a partioning of $\mathcal{G}$ as follows:

$$\mathcal{G} = \{G^{typed}_{F_i} \mid F_i \in F^{typed}(G)\} \tag{14}$$

[10] Of course this does not generally hold in RDF, but we make this assumption merely to simplify notation.

where $G^{typed}_{F_i}$ can be defined for each typed family $F_i \in F^{typed}(G)$ as

$$G^{typed}_{F_i} = \{(s,p,o) \in G \mid F^{typed}(s) = F_i\} \tag{15}$$

Again, we simply write $G_i$ for $G^{typed}_{F_i}$, and finally, analogously can define

$$\sigma(G,Q) = \{G_i \in \mathcal{G} \mid pred(Q) \cup types(Q) \subseteq F_i\} \tag{16}$$

for again star-shaped admissible queries $Q$, where by $types(Q)$ we denote all (non-variable) objects of *rdf:type* triple patterns in $Q$.

To illustrate the previous definitions consider the KG $G$ shown in Fig. 1, and the typed families shown in Fig. 2 Following the definition of typed-family in Eq. (12), the subject *s1* belongs to family $F^{typed}_1$, the subject *s2* belongs to family $F^{typed}_2$, and the subject *s3* belongs to family $F^{typed}_3$. Note that in predicate families, the subjects *s1* and *s2* were in the same family; this is no longer the case, as their set of classes is different. For the KG $G$, there are three typed families denoted $F^{typed}(G)$, i.e., $F^{typed}_1$, $F^{typed}_2$ and $F^{typed}_3$. Lastly, each of these families induces a partition over $G$. For example, $F^{typed}_2$ contains all the triples of the subject *s2*, which in this case is triples *t8* and *t9*. Lastly, the set of partitions computed for $G$, denoted $\mathcal{G}$ are $G^{typed}_{F_1}$, $G^{typed}_{F_2}$, and $G^{typed}_{F_3}$.

## 4. Our Approach: smart-KG$^+$

### 4.1. Design and Overview

smart-KG$^+$ (cf. Fig. 3), which extends the original approach presented in [1], combines shipping HDT compressed family partitions with the shipping of intermediate results from evaluating a given sub-(query) over the existing LDF interfaces. As such, smart-KG$^+$ relies on both shipping intermediate results from executing single-triple patterns using a brTPF LDF interface on the server, as well as using a (typed) family-partition-based LDF interface for star-shaped subqueries (which will be evaluated on the client side, based on the shipped partition). The rest of SPARQL complex patterns other than triple or star-patterns will be evaluated on the client side.

Initially, the smart-KG$^+$ server constructs the family-based partitions (cf. Sec. 4.2 see the practical partition generator) for a given knowledge graph. The generated KG partitions are materialized as HDT files in the storage module together with *family catalog* that summarizes metadata information about the KG partitions including structural and statistical metadata. In addition, smart-KG$^+$ API offers access to the KG based on two operators: one to execute a single triple pattern and the other to ship the requested partition to smart-KG$^+$ client.

Upon receiving a BGP Q from smart-KG$^+$ clients, the smart-KG$^+$ server decomposes the input query into a set of o star-shaped subqueries where the server query planner devises an annotated query plan $\Pi$ that decides for each pattern whether to be executed using brTPF or partition shipping. The client then evaluates the annotated query plan received from the server based on the specified subquery ordering.

As a side note, getting back to our original formalization of LDF and the fact that we do not consider "paging" ($\Phi$) in relation to partition-based LDF: note that it would not make sense to decompose family-based partitions into chunks since chunking up the HDT-compressed partitions would require decompression.

### 4.2. smart-KG$^+$ Partition Generator

In this section, we detail how the smart-KG$^+$ server, upon loading an RDF KG, processes it into partitions $G_1, \ldots, G_m$ per family, as described in Eq. (15) and stores those partitions as HDT compact files format. In practice, however, real-world RDF graphs such as DBpedia and Yago can potentially generate a relatively large number of partitions due to the high semi-structured nature of these RDF graphs. Thus, we introduce the concept of *predicate-restricted families*, where we control some particular predicates during the process of generating families.

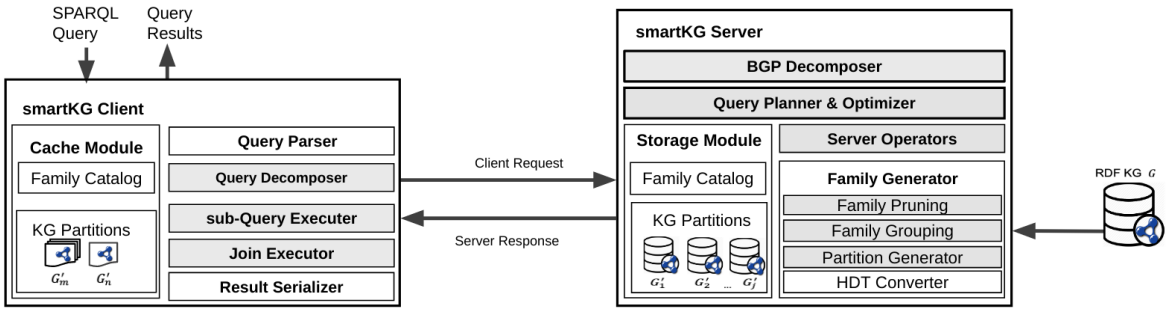



Fig. 3. Overall architecture for the smart-KG$^+$ client and server.

**Predicate-restricted families.** Let us consider a restricted set of predicates, $P'_G \subseteq pred(G)$. The predicate-restricted family of a subject $s$ w.r.t. $P'_G$, denoted $F'(s)$, is defined as $F'(s) = P'_G \cap F(s)$.

Analogously, we denote as $F'(G) = \{F'_1, F'_2, \cdots, F'_{m'}\}$, or just $F'$, the set of different predicate-restricted families for $G$, where $m' = |F'(G)|$. These families correspond to a set $\mathcal{G}' = \{G'_1, G'_2, \ldots, G'_{m'}\}$ of partitions of a subgraph of $G$ based on the $P'_G$-restricted families, with

$$G'_i = \{(s,p,o) \in G \mid F'(s) = F'_i\} \tag{17}$$

The prior definitions carry over to *predicate-restricted typed-family partitions and* typed-partitions, i.e., $F'^{typed}$ and $G'^{typed}_{F_i}$ can be defined analogously, where we additionally restrict the classes by a set $C'_G$. Note that, however $\mathcal{G}'$ is no longer a full cover of $G$, but the graph $G' = \bigcup G'_i$ only contains the "projection" of $G$ to $P'_G$, with the intention that predicates other than $P'_G$ (or, resp. classes other than $C'_G$) are delegated to brTPF.

Serving predicate-restricted families allows a smart-KG$^+$ publisher to select $P'_G$ (and $C'_G$) depending on (i) the cardinality of the predicates (i.e. the number of occurrences in the graph), and (ii) the importance of predicates (and combinations) in actual query workloads. We will describe a concrete method to pick $P'_G$ (and $C'_G$) based on the cardinality of predicates and classes in Sec. 4.2.2.

### *4.2.1. Family Grouping*

Relying on restricted families enables the publisher to control the number of generated families and reduce the generation of infrequently queried families to some degree, however, the number and the size of partitions are still driven by entities' distribution in the graph. In practice, many RDF graphs are skewed in the sense that there exist "dominant" families with large corresponding partitions, as opposed to several small, very similar families of much smaller sizes. This phenomenon arises due to the semi-structured nature of RDF, where entities of the same type could potentially have different attributes representing diverse relationships. Thus, alongside predicate-restricted families, the introduced partition shipping strategy merges (i.e. groups) similar families into a single family. For instance, all disjoint families containing a certain set of predicates e.g. $F_1$={`foaf:name`, `dbo:birthPlace`, `dbo:almaMater`} and $F_2$={`foaf:name`,`dbo:birthPlace`, `dbo:occupation`} can be merged into a single family $F_{\{1,2\}}$={`foaf:name`, `dbo:birthPlace`}. The intuition behind family grouping is to materialize families representing overlapping predicate subsets which may be frequently present in query patterns as predicate families. Therefore, smart-KG$^+$ server can send a single compact partition representing the smallest merged families required to resolve a star query pattern rather than serving the union of partitions which have two main downsides: first, it requires on average extra data transfer due to shipping needless predicates to the star query pattern; second, it requires to locally union the shipped partitions on the client-side. While typed-based families can in theory also be grouped, a preliminary study of ours showed that considering different combinations or even hierarchies of classes would drastically increase the number of materialized partitions, which might benefit only a set of specific queries. For this reason, in the remainder of this work, we focus on only grouping predicated-restricted families.

Note that, in order to define the notion of a merge of families and respective (predicate-restricted) partitions we refer to particular families in $F'(G) = \{F'_1, F'_2, \cdots, F'_{m'}\}$, by their index $\{1, \ldots, m'\}$. Using this notation, formally,

for an index set $I \in 2^{\{1,\cdots,m'\}}$, we define the merge $F'_I$ of the set of families $\{F'_j \mid j \in I\}$ as follows[11]:

$$F'_I = \bigcap\nolimits_{i \in I} F'_i \tag{18}$$

Analogously, the corresponding merged partition $G'_I \subseteq \bigcup_{i \in I} G'_i$can also be defined as:

$$G'_I = \{(s,p,o) \in G \mid F'_I \subseteq F(s)\} \tag{19}$$

if $G'_1$ and $G'_2$ are merged into $G'_{\{1,2\}}$, then to evaluate a query pattern that involves the predicates `foaf:name` and `dbo:birthPlace`, we only transfer $G'_{\{1,2\}}$ rather than $G'_1 \cup G'_2$. Note that the most important consideration is that all the subjects are a matching result for those queries only involving the predicates in $G'_{1,2}$.

Following similar premises, Gubichev and Neumann [15] establish a hierarchy of characteristic sets, in each step removing one element of the set and keeping only the one that minimizes the query costs (i.e. cost can be understood as cardinality, in this context). For instance, in the previous example, the approach by Gubichev and Neumann will inspect all combinations of two predicates, $F^1_{\{1,2\}}$={`foaf:name`, `foaf:birthPlace`}, $F^2_{\{1,2\}}$={`foaf:name`, `dct:title`}, etc., to select the one with smallest cardinality, e.g. $F^2_{\{1,2\}}$, for query planning.

We use a similar idea, but the main differences with the previous work are that (i) we do not compute all predicate subsets of a given family (this is mainly to estimate the cost of join operations [15]) but only those subsets that represent merges, corresponding to non-empty intersections with other families, and (ii) we keep all these intersections in a map, irrespective of their cardinality.

To create the map of merged families for all potentially non-empty intersections of sub-families, we start from $F'(G) = \{F'_1, \ldots F'_m\}$, and iteratively construct a partial map $\mu$ such that, given a set of predicates $f$, $\mu(f)$ returns (whenever $f$ corresponds to a non-empty intersection) a set of indexes of all original families that contain subjects contributing to $f$, as shown in Alg. 1. We initialize $\mu$ with $F'(G)$ (lines 2–3), and then, iteratively, until $\mu$ does not change anymore (lines 4–13), create mappings (corresponding to a merged family) collecting all indexes, for each non-empty intersection of families (lines 9–12). If there already is a (merged) family corresponding to the intersection found, i.e., $f \cup g$ appears already in the domain of $\mu$ (line 9), then also the corresponding index(es) are considered (line 10) and the mapping is updated, otherwise, a new mapping is created (line 12). Note that, as opposed to this pseudo-code, our actual implementation is using a hashmap for the (merged) families and avoids revisiting the same intersections repeatedly.

Then, $\mu(\cdot)$ is used to compute the partitions served by the `smart-KG`$^+$ server, denoted $\mathcal{G}_{serv}$, where $\mathcal{G}'$ is replaced with a set of partitions obtained from the merged families:

$$\mathcal{G}_{serv} = \{G'_{\mu(f)} \mid f \in dom(\mu)\} \tag{20}$$

Note that the elements in $\mathcal{G}_{serv}$ are no longer non-overlapping, i.e., formally, they are not partitions anymore but fragments of the graph $G$. However, for the sake of readability, we abuse notation and refer to these fragments as *merged partitions* (or simply *partitions*). The advantage of serving these merged partitions is that the client can determine a unique minimal matching partition among $\mathcal{G}_{serv}$ to answer a query using the mapping $\mu$.

#### *4.2.2. Family Pruning*

Note that, in practice, the cost of fully materializing the partitions generated from *all* potential merges (intersections) of all families in $G$ could be prohibitive. For instance, as we will show in our evaluation, in the DBpedia graph, a naive merge would create +600k partially very large families, which are unfeasible to serve.

To this end, we present a family pruning strategy for restricting the number of materialized partitions, where we (i) restrict considered predicates in $P'_G$ based on their cardinality, (ii) restrict considered classes based on their

[11] Note that we consider the identity merge, i.e., $F'_{\{j\}} = F'_j$

**Algorithm 1:** Family Grouping

**Input :** $F'(G) = \{F'_1, \dots F'_m\}$, the set of different (restricted) families.
**Output:** $\mu(\cdot)$ a partial mapping from sets of predicates to index sets $I \in 2^{\{1,\cdots,m\}}$

```
Initialize μ with the original families:
foreach f ∈ F'(G) do
    μ(F'_i) ← {i}
repeat
    μ'(·) ← μ(·)
    foreach f ∈ dom(μ) do
        foreach g ∈ dom(μ) do
            if g ∩ f ≠ ∅ then
                if g ∩ f ∈ dom(μ) then
                    μ(g ∩ f) ← μ(g ∩ f) ∪ μ(g) ∪ μ(f)
                else
                    μ(g ∩ f) ← μ(g) ∪ μ(f)
until μ ≠ μ';
return μ
```

cardinality in generating typed families, (iii) avoid the creation of small families that deviate only slightly from other overlapping, "core" families, and (iv) avoid materialization of families over a certain size.

**(i) Restrict predicates based on cardinality.** The cardinality of predicates is a key factor in determining the number and size of shipped partitions. Therefore we distinguish between infrequent and frequent predicates in the KG.

*Infrequent predicates* are those that occur rarely in the KG compared to the most commonly occurring predicates. Infrequent predicates may be scattered across various subjects in the KG, leading to the creation of multiple small families. In this case, a TPF/brTPF call efficiently evaluates a single triple pattern with an infrequent predicate without the need to transfer large intermediate results (i.e. unnecessary materialization of small family partitions).

*Frequent predicates* can be part of almost all families such as `dbo:wikiPageExternalLink` in DBpedia leading to an undesirable increase in the size of each family, especially if they are rarely mentioned in queries.[12]

Note specifically that although `rdf:type` is a typically frequent predicate, we do not exclude it at this point, as we will tackle this issue separately in (ii), in the handling of typed partitions.

To control predicates cardinalities, we use thresholds $\tau_{p_{low}}, \tau_{p_{high}}$ with $0 \leqslant \tau_{p_{low}} < \tau_{p_{high}} \leqslant 1$, to delimit the minimum and maximum percentage of triples per predicate, and define $P'_G$ accordingly based on these two thresholds:

$$P'_G = \{p' \in pred(G) \mid \tau_{p_{low}} \leqslant \frac{|(s, p', o) \in G|}{|G|} \leqslant \tau_{p_{high}}\} \tag{21}$$

**(ii) Restrict classes cardinality based on cardinality to generate typed-families.** As discussed when introducing typed partitions, `rdf:type` is a natural horizontal partitioner for predicate families, which plays an essential role in reducing the size of the shipped families; plus, as also mentioned above, `rdf:type` is a heavy hitter in real-world queries, since it is a frequently used predicate in log queries as shown in Table 4.

Therefore, similar to issue (i), the cardinality of classes contributes to the number and size of the shipped partitions: firstly, rare classes occurring as `rdf:type` objects in triple patterns are by nature selective: such triple patterns are better handled through a TPF/brTPF/SPF call without shipping a *typed* family; secondly, frequent classes can be potentially present in many of the families (for instance `owl:Thing`) and, in practice, are rarely used in queries.[13]

[12] For example, `dbo:wikiPageExternalLink` appears merely 59 times in the LSQ query log, or when they remain entirely unqueried, as `dbo:wikiPageLength`, which is not mentioned in the LSQ query log.

[13] For instance `foaf:Document` is a large class but mentioned in only 68 queries in LSQ query log

We address the aforementioned issues, similar to issue (i), by excluding these classes, and maintaining minimum ($\tau_{class_{low}}$) and maximum ($\tau_{class_{high}}$) thresholds for the percentage of triples per class. We rely on these two thresholds to define the set of classes $C'_G$ for restricting the created typed partitions:

$$C'_G = \{c \in types(G) \mid \tau_{class_{low}} \leqslant \frac{|(s, rdf{:}type, c) \in G|}{|G|} \leqslant \tau_{class_{high}}\} \tag{22}$$

**(iii) Avoid the creation of small families.** To address this issue, we aim at considering only "core" families for the partition merging process, i.e., we select predicate combinations (i.e. families) that are used by a proportionally large number of subjects, above a threshold $\alpha_s$. That is, we define these core families as

$$F'_{core} = \{F'_i \in F' \mid \frac{|subj(G'_i)|}{|subj(G)|} \geqslant \alpha_s\} \tag{23}$$

with the respective index set $I_{core} = \{i \mid F'_i \in F'_{core}\}$ and predicate set $P'_{core} = \{p \in F'_i \mid F'_i \in F'_{core}\}$. Intuitively, these *core families* represent the structured parts of the graph, i.e., star-shaped sub-graphs where entities are described with the same attributes.

**(iv) Avoid the creation of large families.** Finally, we avoid the materialization of overly large (e.g. hundreds of millions of triples in DBpedia) merged partitions $G_I$ with size above a threshold $\alpha_t$. In order to only take core families into account for the creation of partitions, and limit merged families to sizes below $\alpha_t$, it is sufficient to modify Equation (20) as follows:

$$\mathcal{G}_{serv} = \underbrace{\left\{ G'_{\mu(f)} \;\middle|\; \begin{array}{l} f \in dom(\mu) \wedge \\ \mu(f) \cap I_{core} \neq \emptyset \wedge \\ \sum_{i \in \mu(f)} |G'_i| \leqslant \alpha_t \end{array} \right\}}_{\text{Merged partitions}} \cup \underbrace{\{G_{\{i\}} | F'_i \in F'\}}_{\text{Non-merged partitions}} \cup \underbrace{\{G_{i,class} | class \in C'_G, F'_i \cup \{class\} \in F^{typed}\}}_{\text{Typed partitions}} \tag{24}$$

In the conditions in the first part of Equation (24), line 2 addresses issue (iii)[14] and line 3 addresses issue (iv)[15]. The second part ensures that, despite pruning, the non-merged partitions of families in $F'$ remain being served.

Due to these pruning steps, no longer all the partitions corresponding to families in $dom(\mu)$ will be materialized in $\mathcal{G}_{serv}$. Therefore, in practice, we define another mapping function, $\mu_G$, that allows us to directly map families from $dom(\mu)$ to "minimal" sets of matching partitions in $\mathcal{G}_{serv}$. In practice, we compute the partitions $\mathcal{G}_{serv}$ along with $\mu_G$ in one go. That is, we build a mapping $\mu_G : dom(\mu) \mapsto 2^{2^{\{1,\ldots,m\}}}$ that maps a family $f$ to a set of index sets $\{I_1, \ldots I_k\}$ representing (lists of) materialized matching partitions, i.e., where $\mu_G(f) = \{I \mid G'_I \in \mathcal{G}^{\prec}_{serv}(f)\}$. For $G'_{\mu(f)} \in \mathcal{G}_{serv}$, i.e., if the respective partition is materialized, then $\mu_G(f) = \{\mu(f)\}$. In this case, $\mathcal{G}^{\prec}_{serv}(f)$ is defined as: let $\mathcal{G}_{serv}(f) = \{G'_I \in \mathcal{G}_{serv} \mid f \subseteq \mu^{-1}(I)\}$ be all materialized partitions matching a family $f$, then $\mathcal{G}^{\prec}_{serv}(f)$ is the $\prec$-*minimal* subset of $\mathcal{G}_{serv}(f)$ with $\prec$ defined as: $G'_{I_1} \prec G'_{I_2}$ iff $\mu^{-1}(I_1) \subset \mu^{-1}(I_2)$. That is, as the partition merging can result in no longer disjoint partitions in $\mathcal{G}_{serv}$, the intuition is to pick, at query time, the partitions that are "subset-minimal with respect to their corresponding families". In practice, `smart-KG`$^+$ materializes the partitions in $\mathcal{G}_{serv}$ as HDT files.

### 4.3. smart-KG$^+$ Query Processing

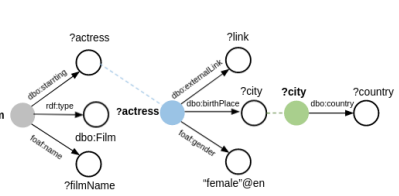
In this section, we detail how the `smart-KG`$^+$ server and client work together to process SPARQL queries. In particular, we describe how the query processing is performed on the server-side and on the client-side.

[14] since $Subj(G'_i) \cap Subj(G'_j) = \emptyset$ for all original families $F'_i, F'_j \in F'$, by construction it holds that $|Subj(G'_I)| = \sum_{i \in I} |Subj(G'_i)|$

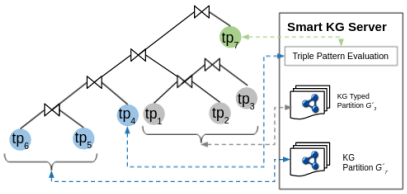
[15] since $|G'_I| = \sum_{i \in I} |G'_i|$

```
SELECT * WHERE {
 ?film  dbo:starring  ?actress .  # tp1  :  544,735
 ?film  foaf:name ?filmName .  # tp2    3,926,490
 ?film  rdf:type dbo:Film.          # tp3  405,579
 ?actress dbo:wikiPageExternalLink ?link.# tp4  9,28
 ?actress dbo:birthPlace ?city  .  #tp5 1,410,336
 ?actress foaf:gender "female"@en . # tp6 25,609
 ?city  dbo:country ?country . }     # tp7   784,823
```

(a) Select all actresses, their movies, and birthplace information




(b) Star-shaped query decomposition




(c) Server Query Plan

Fig. 4. Example of processing a SPARQL query with the smart-KG$^+$ client.

#### 4.3.1. *smart-KG$^+$ Server*

In this section, we describe how smart-KG$^+$ server query planner creates a query execution plan for a submitted BGP. In addition, we detail how smart-KG$^+$ server enables the clients to access the partitions constructed by the partition generator described in Sec. 4.2 and to evaluate single triple patterns using brTPF.

*Query Decomposer.* First, smart-KG$^+$ splits parsed Basic Graph Patterns (BGPs) into *stars* as follows: given a BGP $Q$, with subjects $subj(Q)$, a decomposition $\mathcal{Q} = \{Q_s \mid s \in subj(Q)\}$ of $Q$ is a set of star-shaped BGPs $Q_s$ such that $Q = \bigcup_{s \in subj(Q)} Q_s$:

$$Q_s = \{tp \in Q \mid tp = (s, p, o)\} \tag{25}$$

Analogous to graphs, we can also associate a family to each star query $Q_s$:

$$F(Q_s) = \{p \mid \exists o : (s, p, o) \in Q_s, p \in U\} \tag{26}$$

Typed-families for a query $F^{typed}(Q_s)$ can be computed analogously to graphs.

Given the SPARQL query in Fig. 4a, the BGP is decomposed into $\mathcal{Q} = \{Q_{?\texttt{film}}, Q_{?\texttt{actress}}, Q_{?\texttt{city}}\}$ around the three subjects (cf. Fig. 4b). Each of the star families $F(Q_s)$ that can be mapped to existing predicate families in $dom(\mu_G)$ on the server that has a non-empty answer. For example, $Q_{?\texttt{film}} = \{(?\texttt{film}, \texttt{dbo:starring}, ?\texttt{actress}), (?\texttt{film}, \texttt{foaf:name}, ?\texttt{filmName}), (?\texttt{film}, \texttt{rdf:type}, \texttt{dbo:Film})\}$ has $F(Q_{?\texttt{film}}) = \{\texttt{dbo:starring}, \texttt{foaf:name}, \texttt{rdf:type}\}$. Note that $Q_{?\texttt{film}}$ contains rdf:type which will be distinguished by the query planner and optimizer while devising the query plan.

*Shipping-based Query Planner & Optimizer* In smart-KG$^+$, the query planner and the optimizer are executed at the server-side to provide more efficient query plans based on pre-computed characteristic set cardinality estimations and the server's partition metadata. When the smart-KG$^+$ server receives a request $Plan(Q)$ for a BGP $Q$ from a client, the server query planner devises an *annotated query plan* to specify which interfaces are used per subquery. The interfaces are denoted with superscripts to represent triple pattern shipping using *brTPF* and partition shipping using *SKG* to describe the interfaces as $Q_s^{interface}$.

Given $\mathcal{Q}$, $P'_G$, and $C'_G$ as input, the query optimizer devises a query plan $\Pi_Q$ where the resp. algorithm to compute $Plan(Q, P'_G, C'_G)$ is shown in Alg. 2. If the estimated cardinality of a given star-shaped subquery $Q_s$ is zero (i.e., empty result set), then the result set of the query $Q$ will be empty, therefore, the query planner returns an empty plan (lines 4-5). Then, the optimizer finds the star-subquery $Q_{s_i}$ with the lowest cardinality estimation using the function $card(Q_s, \Pi_Q)$ (line 6); in our running example, this would order $Q_{?actress}$, followed by $Q_{?films}$, and lastly the triple pattern in $Q_{?city}$.

Next, $Q_{s_i}$ is annotated with a label that corresponds to the interface that will be utilized to evaluate the subquery: *SKG* label and *brTPF* label. Therefore, the optimizer characterizes each $Q_{s_i}$ to decide whether to use partition shipping or triple pattern (for parts) of $Q_{s_i}$, as follows:

**Algorithm 2:** Query Optimizer and Planner: *optimizePlan*

**Input:** Star-shaped query decomposition $\mathcal{Q}$ , $P'_G, C'_G$
**Output:** $\Pi_Q$ an annotated query plan for $\mathcal{Q}$

1 $\Pi_Q \leftarrow ()$
2 **while** $\mathcal{Q} \neq \emptyset$ **do**
3  **for** $Q_s \in \mathcal{Q}$ **do**
4  **if** $card(Q_s, \Pi_Q) = 0$ **then**
5  **return** ()
6  $Q_{s_i} \leftarrow Q_s \in \mathcal{Q}$ such that $card(Q_s, \Pi_Q) \leqslant card(Q_{s_j}, \Pi_Q)$ for all $Q_{s_j} \in \mathcal{Q}$
7  $Q'_{s_i} \leftarrow \{(s_i, p, o) \in Q_{s_i} \mid p \in P'_G\}$
8  $Q''_{s_i} \leftarrow Q_{s_i} \setminus Q'_{s_i}$
 // Check if there exists family-typed partitions for classes in $Q'_{s_i}$
9  **for** $tp = (s_i, rdf{:}type, o) \in Q'_{s_i}$ **do**
10  **if** $o \notin C'_G$ **then**
11  $Q''_{s_i} \leftarrow Q''_{s_i} \cup \{tp\}$
12  $Q'_{s_i} \leftarrow Q'_{s_i} \setminus \{tp\}$
 // Re-order triple patterns within subqueries
13  $Q'_{s_i} \leftarrow reorder(Q'_{s_i})$
14  $Q''_{s_i} \leftarrow reorder(Q''_{s_i})$
 // Annotate plan with interface
15  **if** $|Q'_{s_i}| > 1$ **then**
16  $\Pi_Q \leftarrow append(\Pi_Q, Q'^{SKG}_{s_i})$// Evaluate plan using partitions
17  **else**
18  $\Pi_Q \leftarrow append(\Pi_Q, Q'^{brTPF}_{s_i})$// Evaluate plan using brTPF
 // Evaluate tps with no materialized partitions using brTPF
19  $\Pi_Q \leftarrow append(\Pi_Q, Q''^{brTPF}_{s_i})$
20  $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{Q_{s_i}\}$
21 **return** $(\Pi_Q)$

**Partition Shipping.** Shipping relevant partitions to evaluate a star $Q_s \in \mathcal{Q}$ requires considering the materialized partitions at the server. Since graph partitions are generated based on the pruned families (cf. Sec. 4.2), only stars with $F(Q_s) \subseteq P'_G$ can be fully evaluated by served partitions. Therefore, the optimizer partitions each $Q_{s_i} \in \mathcal{Q}$ into the disjoint sets $Q'_{s_i}$ and $Q''_{s_i}$, where $Q'_{s_i}$ is the part of the star that can potentially be evaluated over the served partitions (line 7), whereas the remaining triple patterns in $Q''_{s_i}$ are delegated to brTPF requests (line 8). Note that $Q''_{s_i}$ also includes triple patterns with predicate variables, i.e., $p \in V$. Then, optimizer checks (lines 9–12) for each triple pattern $tp$ with `rdf:type` predicate whether the class $o$ belongs to the restricted set of classes $C'_G$. Note that this also captures the case with $o \in V$. In the negative case, there is no materialized typed-family partition to serve $Q_{s_i}$, therefore, this $tp$ will be pushed to $Q''_s$ to be evaluated using brTPF (line 11). Then, the optimizer considers heuristics that partition shipping is only followed if $|Q'_{s_i}| > 1$ where $Q'_{s_i}$ is annotated with the respective label for partition shipping $SKG$ (lines 15–16). As in practice, the transfer of graph partitions to resolve a single triple pattern usually takes longer than delegating to a brTPF request directly (line 18).

**Triple Pattern Shipping.** $Q_s$ with a single triple pattern, triple patterns with infrequent predicates, and triple patterns with predicate variables will be annotated with $brTPF$ label (lines 17). These subqueries will be eventually evaluated using a brTPF request to the server.

Lastly, the optimizer reorders the triple patterns with the function *reorder* in the sub-plans based on their cardinality estimations; this allows for an efficient evaluation at the client side. Then, the optimizer attaches the sub-plans with the *append* function to build the final plan $\Pi_Q$. The resulting query plan $\Pi_Q$ comprises sub-plans annotated with the corresponding shipping strategy. We describe a full annotated query plan as sequences of pat-

terns being interpreted as left-linear query plans, that is, we write query plans that evaluate patterns as permutations of the decomposed stars in $\mathcal{Q}$. For our example shown in Fig. 4a, the annotated plan could be written as $\Pi = (Q'^{SKG}_{?actress}, Q''^{brTPF}_{?actress}, Q^{SKG}_{?film}, Q^{brTPF}_{?city})$, describing an execution plan at the level of joining star patterns as follows: $(((Q'_{?actress} \bowtie Q''_{?actress}) \bowtie Q_{?film}) \bowtie Q_{?city})$. Fig. 4c shows the shipping strategies for each sub-plan from our example. The query optimizer first starts with the subquery $Q_{?actress}$ which is the most selective subquery. For $Q_{?actress}$, the optimizer creates $Q'^{SKG}_{?actress} \bowtie Q''^{TPF}_{?actress}$, as the triple pattern $tp_4$ in $Q_{?actress}$ is evaluated using triple pattern shipping as the optimizer determined that the predicate `dbo:wikiPageExternalLink` is not in $P'_G$. Next, the query optimizer evaluates $Q_{?film}$ via partition shipping on the client based on a family-based partition. Finally, $Q_{?city}$ is added to be executed as a single triple pattern using brTPF.

*Server Operators* The smart-KG$^+$ server provides operators to ship partitions and their metadata, or to respond to brTPF requests. These operators are defined in the following interface calls to access a KG $G$:

- A brTPF LDF API interface $brTPF(Q_s, \Omega)$ which returns $\sigma_{brTPF}(G, Q_s, \Omega) = s(G, Q_s, \Omega)$, that retrieves the answers for a single triple pattern $Q_s$ while taking into consideration the attached bindings $\Omega$, i.e., the smart-KG$^+$ server returns the triples from $G$ that match $Q_s$ based on brTPF requests.
- A SKG LDF API interface $SKG(Q_s, \emptyset)$ that handles star-shaped queries $Q_s$ and returns $\sigma_{SKG}(G, Q_s, \emptyset) = \sigma(G, Q_s)$, i.e., the set of typed-family partitions if exists, otherwise the family partitions, of which $[\![Q_s]\!]_G$ can be computed and joined on the client-side.
- *Plan*$(Q)$ to create a query plan for the received BGP $Q$. Unlike the initial smart-KG$^+$ prototype [1], where the query plan was inaccurately created on the client-side, we propose shifting the query execution planning from the client to the server to compute better query plans as the server has access to more accurate cardinality estimations to determine the order of stars and triple patterns.

#### 4.3.2. smart-KG$^+$ Client

The primary focus of this work is on evaluating BGPs as the essential retrieval functionality of the SPARQL query language. However, our introduced interface is able to process a full SPARQL query including operators such as `UNION` and `OPTIONAL`, `FILTER`, etc., which are all evaluated locally on the client-side. Herein, we introduce the general approach for processing a SPARQL query, as follows:

1. Upon receiving a SPARQL query, the *query parser* translates the input query string into the corresponding SPARQL algebra expressions.
2. Initially, the client sends a request *Plan*$(Q)$ to retrieve from the server an optimized query execution plan $\Pi_Q$ for the extracted BGP $Q$.
3. The *query executor* evaluates the received plan and iteratively combines the results using a dynamic pipeline of iterators, following brTPF [14], where each iterator deals with a certain annotated subquery $Q^c_s$ that request a partition or performs a brTPF request.
4. The *results serializer* translates the locally joined results into the specified format. Note that the downloaded partitions from the smart-KG$^+$ server during query evaluation can be locally stored in the *family cache* to be reused in the upcoming queries.

We describe in detail the algorithms that implement the query executor in a recursive manner in Alg. 3 and Alg. 4. The function *evalPlan* recursively evaluates the received plan $\Pi$ by traversing the left-tree of sub-plans (cf. Alg. 3). The query executor initially evaluates the first and most selective subquery $Q^{c1}_{s1}$ (line 1). Then, the algorithm traverses the rest of the plan (lines 2-6). The base case is when the plan is associated with a single star pattern $Q^c_s$ (line 2). In this case, the executor evaluates the star using $eval_c(Q^c_s, \Omega')$ while considering the intermediate results from earlier subqueries $\Omega'$ (for details, cf. Alg. 4). Otherwise, the query executor will recursively call the evaluation of the remaining subtree and join them with the set of bindings retrieved from the previous calls (line 5). The final output of the query executor is the query result set $\Omega$ of a given plan (line 6). In practice, the executor implements an iterator to push intermediate results of evaluating one sub-plan to the next operator in the plan. This allows the smart-KG$^+$ client to incrementally stream query results once computed.

Alg. 4 presents the function $eval_c(Q^c_s, \Omega')$, which calls the corresponding interface (i.e. the respective smart-KG$^+$ server operators for the shipping strategy determined by the server query plan). The first case $c = SKG$ is to evalu-

**Algorithm 3:** Query Executor: *evalPlan*

**Input:**
$\Pi = (Q_{s_1}^{c_1}, \ldots, Q_{s_n}^{c_n})$, // an execution plan for a BGP query with stars $Q_{s_1}^{c_1}, \ldots, Q_{s_n}^{c_n}$;
$\Omega$ // a set of bindings; the initial recursive call will set $\Omega = \emptyset$
**Output:** $\Omega$ // set of solution mappings

1 $\Omega \leftarrow eval_c(Q_{s_1}^{c_1}, \Omega)$
2 **if** $|(Q_{s_2}^{c_2}, \ldots, Q_{s_n}^{c_n})| = 1$ **then**
3 $\quad \Omega \leftarrow \Omega \bowtie eval_c(Q_{s_n}^{c_n}, \Omega)$
4 **else if** $|(Q_{s_2}^{c_2}, \ldots, Q_{s_n}^{c_n})| > 1$ **then**
5 $\quad \Omega \leftarrow \Omega \bowtie evalPlan((Q_{s_2}^{c_2}, \ldots, Q_{s_n}^{c_n}), \Omega)$
6 **return** $\Omega$

**Algorithm 4:** Query Executor: $eval_c$

**Input:**
$Q_s^c$ // A decomposed pattern and it is annotated execution plan;
$\Omega'$ // a set of binding if available
**Output:** $\Omega$ a set of solution mappings

1 **if** $c = SKG$ **then**
2 $\quad G^* = SKG(Q_s^c, \emptyset)$
3 $\quad \Omega \leftarrow \{\omega_\emptyset\}$
4 $\quad$ **for** $tp \in Q_s^c$ **do**
5 $\quad\quad \Omega \leftarrow \Omega \bowtie \bigcup_{G_j \in G^*} [\![tp]\!]_{G_j}$
6 **else if** $c = brTPF$ **then**
7 $\quad \Omega \leftarrow brTPF(Q_s^c, \Omega')$
8 **return** $\Omega$

ate a star pattern using a shipped partition on the client-side. The second case $c = TPF$ involves calling the brTPF interface which directly returns the result bindings. In the following, we explain the two cases in detail:

**Case SKG.** Each sub-plan $Q_s^{SKG}$ is evaluated (cf. Alg. 4, lines 2–5) by retrieving HDT partitions using the server operator for partition shipping $SKG(Q_s, \emptyset)$. This operation returns a set of partitions $G^*$ which is either a set of family-based partitions or a set of typed-family partitions (cf. Alg. 4, line 2). The query executor evaluates each triple pattern $tp$ of the star pattern $Q_s^{SKG}$ (lines 4–5) over the partitions (using the SPARQL algebra union operator when several partitions are retrieved). The results of each $tp$ are joined to produce the final results of the star pattern.

**Case: brTPF.** Each sub-plan $Q_s^{brTPF}$ involves a single triple pattern which is executed by calling the server operator *brTPF(tp, Ω′)* which is a brTPF interface that directly returns the result bindings (cf. Alg. 4, lines 6-7).

The evaluation of SPARQL BGP queries with smart-KG$^+$ is correct, as stated in the following proposition. The proof can be found in Appendix A.

**Proposition 1.** *The result of evaluating a BGP Q over an RDF graph G with* smart-KG$^+$*, denoted smart-eval*$(Q, G)$*, is correct w.r.t. the semantics of the SPARQL language, i.e., smart-eval*$(Q, G) = [\![Q]\!]_G$.

## 5. Experimental Evaluation

We report the performance of smart-KG$^+$ in comparison to state-of-the-art SPARQL engines over Linked Data Fragments. All datasets, queries, and results, including additional experiments, details on the implementation and

configurations used in the experiments are available online[16]. We organize the conducted experiments as follows: First, in Sect. 5.1, we present the details of our experimental setup. Next, in Sect. 5.2, we present the results of the partition generation. We perform an ablation study to assess the impact of each contribution in Sect. 5.3. Subsequently, in Sect. 5.4, we conduct a performance evaluation of our approach, comparing it to the state of the art. Further, we extend this evaluation in Sect. 5.5 to assess the query performance under different query shapes. The resource consumption of our introduced interface is compared to other existing interfaces in Sect. 5.6. Finally, in Sect. 5.7, we evaluate typed-family partitioning using multiple datasets.

### *5.1. Experimental Setup*

In this section, we present the experimental setup, including the characteristics of the compared systems, the benchmark KGs, the query workloads, the hardware and software configurations, and the evaluation metrics.

#### *5.1.1. Compared Systems*

- **smart-KG**: We use the Java implementation of `smart-KG` [1], extending the TPF implementations[17]. HDT indexes and data are stored on the server's disk, with no client-side family caching. This implementation includes:
  - Query Planner: The `smart-KG` client-side query planner generates left-linear plans. This planner relies on the server's partition metadata to determine whether to use the triple pattern or partition shipping. The metadata is transferred to the client-side once before evaluating queries, requiring additional data transfer.
  - Client-side Joining: We implement a joining strategy, following TPF implementation [13]. The join processing is performed on the client-side based on the client-side query plan.
- **smart-KG$^+$**: We implement both client and server in Java[16], extending `smart-KG`, which includes:
  - Query Planner: We implement a server-side query planner to re-order the star-subqueries and triple patterns, which relies on cardinality estimations computed at the server. Details are presented in Sect. 4.3.1, Alg. 2.
  - Client-side and Server-side Joining: We implement a joining strategy, following the brTPF implementation [14]. We enable the clients to attach intermediate results to brTPF requests.This enables a distributed join execution between the client and server using the bind join strategy [70].
- **Triple Pattern Fragments (TPF)**: We use the Java TPF client along with the TPF server [13].
- **SaGe**: We use the Python implementation of both the SaGe server and client. We follow the recommended a time quantum of 75 milliseconds as recommended by the authors [31]. Specifically, we configure SaGe to operate with 65 workers following SPF experiements [29] and gunicorn recommendation[18]
- **WISEKG**: We utilize the WISEKG client and server Java implementation, extending the TPF implementations. The WISEKG server employs Star Pattern Fragments (SPF) for efficient server-side processing of star-subqueries and uses the family generator from `smart-KG` to manage and store HDT files for family-based partitions. The WISEKG client implements a bind join strategy similar to brTPF and SPF, `smart-KG`$^+$ client implementations.

In our experiments, we do not consider SPARQL endpoints, since several previous studies [13, 29, 31] including ours [1] have already shown that endpoints suffer from scalability problems when increasing the number of clients.

#### *5.1.2. Knowledge Graphs*

We use various RDF graph datasets including synthetic and real-world datasets. We construct three different dataset sizes including 10M, 100M, and 1B triples from the synthetic dataset Waterloo SPARQL Diversity Benchmark (WatDiv) [71]. We design these KG sizes according to the size of open KGs on the LOD Cloud[19], with an average of 183M RDF triples. In addition, we evaluate the compared systems based on a real-world dataset such

[16] https://github.com/smartkgplus/smartkgplus/tree/master
[17] Linked Data Fragments: http://linkeddatafragments.org/software/.
[18] https://docs.gunicorn.org/en/stable/design.html#how-many-workers
[19] The Linked Open Data Cloud. https://lod-cloud.net/

Table 3
Characteristics of the evaluated knowledge graphs

| RDF Graph $G$ | # triples $\|G\|$ | # subjects $\|S_G\|$ | # predicates $\|P_G\|$ | # objects $\|O_G\|$ |
|---|---|---|---|---|
| WatDiv-10M | 10,916,457 | 521,585 | 86 | 1,005,832 |
| WatDiv-100M | 108,997,714 | 5,212,385 | 86 | 9,753,266 |
| WatDiv-1B | 1,092,155,948 | 52,120,385 | 86 | 92,220,397 |
| DBpedia | 837,257,959 | 113,986,155 | 60,264 | 221,623,898 |

as DBpedia (v.2015A) [72]. We report the characteristics of the evaluated RDF KGs in Table 3. In addition, in Appendix B, we report statistics on computing family partitioning over other real-world RDF KGs such as WordNet [73], Yago2 [74], DBLP [75], Freebase [4]. However, these KGs are not used for assessing the performance of the query engines, as there are no well-known benchmark queries for these datasets

*5.1.3. Queries and Workloads*

We consider two different query workloads for the synthetic WatDiv datasets:

- A *basic testing* workload denoted as `watdiv-btt` that includes a set of queries extracted from WatDiv basic testing templates[20]. We generate for each client a set of 20 queries with the following shapes: *linear (L)*, which represents simple path queries; *star (S)*, which includes star queries with at least one instantiated object; *snowflake (F)*, which combines multiple star shapes connected with short paths; and *complex (C)*, which provides challenging queries composed of typically low-selective stars and path queries. Various clients may exhibit query overlap among themselves, but within an individual client, there are no instances of query repetition.
- A *stress testing* workload denoted as `watdiv-sts` comprises a collection of queries sourced from the WatDiv stress-testing suite[21]. Each client workload encompasses a total of 156 non-overlapping queries[22]. These queries were generated using the Waterloo SPARQL Diversity Test Suite (WatDiv), which provides stress testing tools [71], allowing us to randomly select queries from the WatDiv stress test query workload in a uniform manner. This workload offers a diverse range of structural and data-driven features [71].

In addition, we consider a DBpedia real-world query workload:

- A *real-world testing* workload, named `DBpedia-lsq`, consists of 30 `SELECT` queries per client obtained from the FEASIBLE framework [76]. These queries are derived from real user interactions and were executed on the DBpedia 3.5.1 dataset. FEASIBLE is a benchmark generation framework that receives a query log (LSQ [17] in our case) and produces a representative set of queries from the log considering both data-driven and structural query features. Since we are interested in highly-demanding queries, we randomly selected 30 BGP queries (out of 259) from FEASIBLE with runtime higher than 1s. We include the results of this workload in our online repository

In order to evaluate the proposed typed-family partitioning, as shown in details in Table 4, we derive the following testing workloads from *basic* testing and *stress* testing workloads on Watdiv dataset and a *real-world* testing workload extracted from LSQ query logs based on FEASBLE benchmark framework:

- A *typed-family partitioning testing* workload, named as `watdiv-btf`, includes 8 queries derived for each client from `watdiv-btt`. Each query contains at least one star-shaped subquery $Q_s$ with a triple pattern that has a `rdf:type` predicate. We divide this workload into two workloads: the first workload named $\texttt{watdiv-btf}_{bounded}$ includes 4 queries for each client where the object of the triple pattern with `rdf:type` predicate is bounded to a value, the second workload, named $\texttt{watdiv-btf}_{unbounded}$, where the object of the triple pattern with the `rdf:type` predicate is unbounded (i.e. variable).
- A *typed-family partitioning testing* workload derived from `watdiv-sts` named `watdiv-stf`. We include a set of queries that contain at least one star-shaped subquery $Q_s$ with the `rdf:type` predicate. We divide the obtained queries into three different workloads. The first workload, named $\texttt{watdiv-stf}_{bounded}$, includes 23

[20] https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml
[21] Waterloo SPARQL Diversity Benchmark. https://dsg.uwaterloo.ca/watdiv/
[22] brTPF: http://olafhartig.de/brTPF-ODBASE2016/

Table 4
Evaluation Workloads Statistics. We provide the total numbers for all the 128 clients

| Query Workload | Number of queries | Number of stars | Number of stars with type predicate | Number of stars with *bounded* type predicate |
|---|---|---|---|---|
| watdiv-sts | 19968 | 35683 | 6283 | 3886 |
| watdiv-btt | 2560 | 5248 | 1152 | 512 |
| watdiv-btf | 1024 | 1664 | 1152 | 512 |
| watdiv-btf$_{bounded}$ | 512 | 640 | 640 | 512 |
| watdiv-btf$_{unbounded}$ | 512 | 1024 | 512 | 0 |
| watdiv-stf$_{bounded}$ | 2944 | 6144 | 2944 | 2944 |
| watdiv-stf$_{unbounded}$ | 1792 | 3072 | 1792 | 0 |
| watdiv-stf$_{both}$ | 768 | 1792 | 1536 | 768 |
| DBpedia-lsq | 3840 | 5632 | 896 | 768 |
| DBpedia-btt$_{bounded}$ | 2432 | 4352 | 3200 | 3200 |
| DBpedia-btt$_{unbounded}$ | 768 | 768 | 0 | 0 |

queries for each client where the object of the triple pattern with type predicate is bounded. The second workload, `watdiv-stf`$_{unbounded}$, includes 14 queries for each client where the object of the triple pattern with `rdf:type` predicate is a variable. The third workload, `watdiv-stf`$_{both}$, contains 6 queries per client, where one star-subquery involves a bounded object in the triple pattern with the type predicate, while another star-subquery includes an unbounded object in the triple pattern with the type predicate.

- A *real-world typed-family partitioning testing* workload. We extract 25 real-users SELECT queries for each client from FEASIBLE [76] benchmark on the DBpedia 3.5.1 dataset. Note that we make sure that the queries are compatible with our DBpedia dataset version, v.2015A. We selected queries that contain at least one-star pattern with at least one triple pattern with the `rdf:type` predicate. We divide the selected queries into two workloads. The first workload, named as `DBpedia-btt`$_{bounded}$, consists of 19 queries with at least one-star query with a bounded type predicate. The second workload, named as `DBpedia-btt`$_{unbounded}$, consists of 6 queries for each client with at least one-star query with an unbounded `rdf:type` predicate.

### 5.1.4. Hardware Setup

- **Client specifications:** We design experiments with an increasing number of clients following eight configurations with $2^i$ clients ($0 \leqslant i \leqslant 7$) issuing concurrent queries to the server. Each client executes one query at a time, i.e., the server receives at most 128 queries simultaneously. We ran all eight configurations 1, 2, 4, 8, 16, 32, 64, and 128 clients concurrently on a virtual machine with 128 vCPU cores of 2.5GHz, 64KB L1 cache, 512KB L2 cache, 8192KB L3 cache, and 2TB main memory. To ensure equal resource allocation among the clients, we bound each client (for the compared systems) to a single vCPU core and 15 GB of main memory.
- **Server specifications:** The compared systems servers run on a virtual machine (VM) hosted on a machine with 32 3GHz vCPU cores, 64KB L1 cache, 4096KB L2 cache, 16384KB L3 cache, and 128GB main memory. To ensure that enough resources are left for the VM, it was made sure that the hypervisor was not over-committing resources. Furthermore, KVM processor affinity was configured so that each VM would be only using a set of explicitly defined CPU cores, ensuring that other VMs running on the hyper-visor are not using the resources of the VM running the SPARQL servers.
- **Network configuration:** While clients and servers are connected over a 1 GBit Ethernet network, we bound the network speed of each client to 20MBit/sec to emulate a practical bandwidth offered by internet service providers.

### 5.1.5. Evaluation Metrics

- **Throughput**: Number of workload queries completed per minute.
- **Timeouts (TO)**: Number of workload queries that exceed the timeout. We set timeout thresholds of 5 and 30 minutes for WatDiv and DBpedia queries, respectively.
- **Workload Completion Time**: Total elapsed time required by a client to execute an entire query workload.
- **Query Execution Time (ET)**: Average elapsed time to execute a single query in a query workload.
- **First Result of a Query**: Elapsed time to retrieve the first result of a query in a query workload.
- **Server CPU load**: The average percentage of server CPU used during the execution of a query workload.
- **Number of Requests (Req)**: Total number of requests received by the server from a client.
- **Number of Transferred Bytes (DT)**: Total number of bytes transferred on the network between the server and clients.

Table 5
Family-based Partitions Parameter Settings in our Experiment

| RDF Graph $G$ | $\alpha_s$ | $\alpha_t$ | $\tau_{p_{low}}$ | $\tau_{p_{high}}$ | $\tau_{class_{low}}$ | $\tau_{class_{high}}$ | $\lvert P'_G\rvert$ | $\lvert P'_{core}\rvert$ | $\lvert F'_{core}\rvert$ | $\lvert\mathcal{G}_{serv}\rvert$ | C.Time (h) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| WatDiv-10M | 0 | $\lvert G\rvert$ | 0 | 1 | 0 | 1 | 85 | 85 | 13,002 | 38,400 | 2 |
| WatDiv-10M | 0 | $0.05\lvert G\rvert$ | 0.01/100 | 1 | 0 | 1 | 59 | 59 | 10,106 | 21,210 | 1 |
| WatDiv-100M | 0 | $0.05\lvert G\rvert$ | 0.01/100 | 1 | 0 | 1 | 59 | 59 | 22,855 | 37,392 | 7 |
| WatDiv-1B | 0 | $0.05\lvert G\rvert$ | 0.01/100 | 1 | 0 | 1 | 59 | 59 | 39,046 | 52,885 | 12 |
| DBpedia | 0.01/100 | $0.05\lvert G\rvert$ | 0.01/100 | 0.1/100 | 0.01/100 | 0.1/100 | 218 | 84 | 35 | 29,965 | 23 |

### 5.2. Creation of Family-based Partitions

Table 5 presents the thresholds used for creating the family-based partitions for each KG $G$. Note that the configuration $(\alpha_s, \alpha_t, \tau_{p_{low}}, \tau_{p_{high}}, \tau_{class_{low}}, \tau_{class_{high}}) = (0, |G|, 0, 1, 0, 1)$ corresponds to full materialization of all families. For the smallest dataset WatDiv-10M, we tested this configuration. Then, we assess the impact of the smart-KG$^+$ family pruning strategies with the following set up. We empirically set $\alpha_t = 0.05|G|$ for all datasets, to avoid large families containing more than $5\%$ of the triples in $G$. Then, we use $\alpha_s = 0$ for WatDiv to allow all families (even small ones), but $\alpha_s = 0.01/100$ for DBpedia to create families where the predicates appear in at least $0.01$ of the subjects in the graph. Likewise, we fixed $\tau_{p_{low}} = 0.01/100$ for all $G$, while we set $\tau_{p_{high}} = 0.1/100$ for DBpedia, as we empirically tested that the resultant predicate set filters out both infrequent and heavy hitters. We refer to [1] for a study on DBpedia on the number of families with different values of our parameters. Lastly, for typed families, we tested the parameters $\tau_{class_{low}} = 0.01/100$ and $\tau_{class_{high}} = 0.1/100$ for DBpedia, applied to 376 classes selected based on an empirical test that the resultant class set filters out heavy hitter classes as well as infrequent classes.

For each graph $G$, Table 3 also shows the number of restricted and core predicates ($|P'_G|$, $|P'_{core}|$), core families, $|F'_{core}|$, and the materialized partitions after grouping/pruning, $|\mathcal{G}_{serv}|$, as well as the total computation time (including family computation, pruning, and partition generation). Table 3 also shows that $|F'_{core}|$, $|\mathcal{G}_{serv}|$, and the computation time are sub-linearly increasing with the graph sizes. In WatDiv, $F'_{core} = F'(G)$, whereas in DBpedia, the initial number of $P'_G$-restricted[23] families $|F'(G)|$ is >600K: the family pruning strategy allows smart-KG$^+$ to identify $|F'_{core}| = 35$ core families, which are merged into ∼30K materialized partitions. We provide an analysis of the impact/coverage of different parameter values for the case of DBpedia in our online repository[16]. Lastly, Appendix B presents the results of family creation in further real-world KGs, i.e., Yago2, WordNet, and DBLP.

### 5.3. Ablation Study: Assessing the Impact of the smart-KG$^+$ Components

In this section, we conduct an ablation study to evaluate the performance of each individual contribution made to smart-KG$^+$. The goal is to gain insights into the significance of each change introduced w.r.t. the earlier version. For this purpose, we developed three configurations of the interface:

- **TPF+OP:** This configuration represents the early version of smart-KG, combining TPF with client-side query planning (OP).
- **TPF+NP:** This configuration is a variant of our smartKG interface that allows us to observe the impact of the new server-side query planning (NP) while using TPF.
- **brTPF+NP:** This configuration represents our proposed solution smart-KG$^+$, which combines brTPF with server-side query planning (NP).

To assess the performance of these configurations, we conduct a performance evaluation using watdiv-btt and watdiv-sts on watdiv10M; results are presented in Table 6 and Table 7.

In Table 6, we observe that the smart-KG outperforms other approaches in handling simple linear queries (L1 - L5) and highly selective star queries (S1 - S3). This performance superiority is attributed to the comparatively lower average execution time of 82 milliseconds for these queries, while the query planning process in our proposed

[23]The 218 restricted DBpedia predicates cover over 40% of the predicates occurring in highly-demanding BGPs (>1s of execution time) in the real-world LSQ query log [17].

Table 6

An ablation study to assess the performance of each individual contribution over watdiv10M using watdiv-btt workload. (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in ms, TO: Timeouts). GM-T = Total Geometric mean for all query classes

| | smart-KG$^+$ (brTPF +NP) | | | smart-KG (TPF +NP) | | | smart-KG (TPF +OP) | | |
|---|---|---|---|---|---|---|---|---|---|
| Query | Req | DT | ET | Req | DT | ET | Req | DT | ET |
| L1 | **4** | **0.54** | **206** | 28 | 1.1 | 333 | 60 | 0.54 | 218 |
| L2 | 3 | 0.34 | 175 | **3** | 0.34 | 188 | **2** | 0.34 | **51** |
| L3 | **2** | 0.5 | 579 | **2** | 0.5 | 566 | **2** | 0.5 | **28** |
| L4 | **2** | 0.5 | 188 | **2** | 0.5 | 169 | **2** | **0.48** | **69** |
| L5 | 3 | 0.16 | 192 | 3 | 0.16 | 221 | **2** | 0.16 | **52** |
| S1 | 3 | 0.13 | 221 | 3 | 0.13 | 206 | **2** | **0.12** | **66** |
| S2 | **2** | 0.22 | 191 | **2** | 0.22 | 204 | **2** | 0.22 | **117** |
| S3 | **2** | 0.59 | 209 | **2** | 0.59 | 181 | **2** | **0.55** | **61** |
| S4 | **10** | **0.48** | **226** | 16 | 0.66 | 575 | 216 | 0.72 | 1476 |
| S5 | **2** | 0.42 | **161** | **2** | 0.42 | 163 | **2** | **0.39** | 3863 |
| S6 | **2** | **0.01** | 170 | **2** | **0.01** | **141** | 699 | 3.8 | 7508 |
| S7 | 2 | 0.003 | **221** | 2 | 0.003 | 224 | 2 | 0.003 | **52** |
| F1 | **4** | **0.9** | 240 | **4** | 0.98 | **228** | 15 | 2.95 | 452 |
| F2 | 3 | **0.9** | 184 | 3 | 0.91 | **179** | **2** | 1.2 | 361 |
| F3 | **5** | 1.5 | 311 | 1503 | **0.16** | **268** | 2541 | 1.4 | 298 |
| F4 | **5** | **0.8** | **217** | 2029 | 30.56 | 31858 | 24000 | 0.81 | 76073 |
| F5 | 5 | 5.8 | **247** | 5 | 5.8 | 282 | **3** | 5.8 | 2478 |
| C1 | **4** | **6.9** | **372** | **4** | **6.9** | 384 | **6** | 7.4 | 683 |
| C2 | 39218 | 52.1 | 300071 | 3181 | 52.1 | **171848** | 102441 | 3.1 | 208721 |
| C3 | **2** | 0.8 | **21635** | **2** | 0.81 | 22200 | **2** | 0.8 | 28316 |
| GM-T | **4.9** | **0.5** | **407.7** | 8.76 | 0.57 | 539.03 | 18.82 | 0.64 | 605.12 |

Table 7

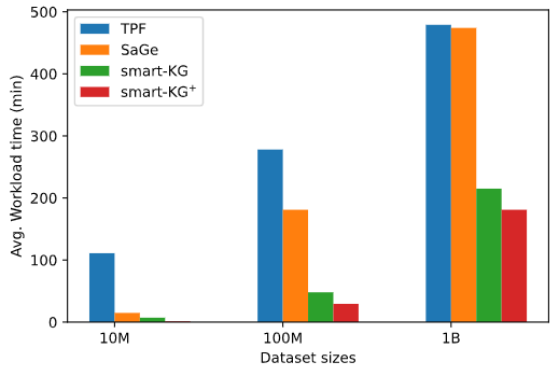
An ablation study to assess the performance of each individual contribution over watdiv10M using watdiv-sts workload. (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in ms, TO: Timeouts). GM-T = Total Geometric mean for all query classes

| Workload | smart-KG$^+$ (brTPF + NP) | | | | smart-KG (TPF + NP) | | | | smart-KG (TPF + OP) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Req | DT | ET | TO | Req | DT | ET | TO | Req | DT | ET | TO |
| watdiv-sts | **554** | **203.41** | **24.405** | 6 | 24722 | 546.26 | 382.236 | **0** | 3768 | 387.953458 | 55.876 | **0** |

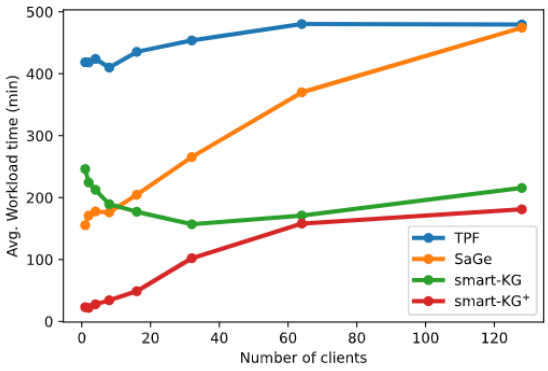
solution, smart-KG$^+$, requires an average of 70 milliseconds. However, it is essential to consider that client-side query planning requires an initial data transfer of 1.75MB on average, comprising metadata that represents the family partitioning of the queried knowledge graph. This metadata is crucial for identifying the required partition for each query. Shipping the metadata file demands an average of 700 milliseconds, but it can be cached locally and subsequently utilized for multiple queries. It is important to note that the performance results presented for smart-KG presume that the metadata file is already stored on the client-side.

Table 6 also shows a significant improvement in performance, with up to a 50% reduction in execution time observed, for both systems reliant on the server-side query planner for F queries comprising 2-3 stars per query. This improvement can be attributed to our server-side query planner's utilization of star reordering based on characteristic sets, which offers a better reordering compared to the one achieved by smart-KG in the case of snowflake queries.

In the case of C queries, C1 demonstrates performance enhancement through the adoption of the star-reordering technique provided by the query planner of smart-KG$^+$. However, for C2, brTPF+NP exhibits slightly lower performance compared to other systems. This is attributed to the query execution strategy of smart-KG$^+$, which always pushes intermediate results to brTPF, instead of joining the intermediate results entirely at the client-side. This lead to unnecessary requests in C2, resulting in a longer runtime. Still, brTPF+NP provides the best total geometric mean for the number of requests, data transfer, and execution time compared to the other two versions.

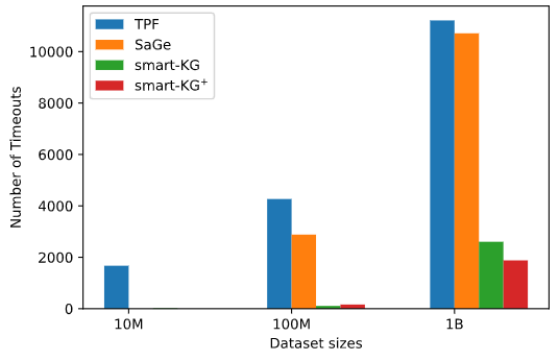



(a) Average Workload Completion Time of 128 concurrent clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets on `watdiv-sts` workload

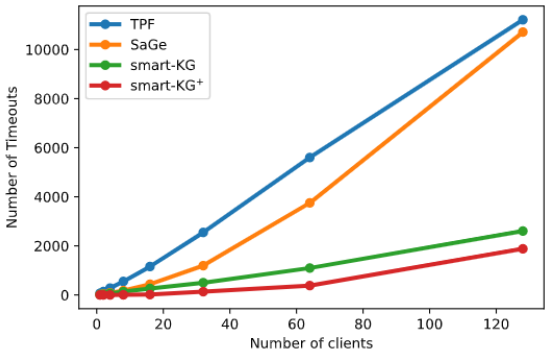



(b) Average Workload Completion Time for increasing numbers of clients over `watdiv1B` on `watdiv-sts` workload

Fig. 5. Workload completion time (lower is better)




(a) Number of timeouts of 128 concurrent clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets on `watdiv-sts` workload




(b) Number of timeouts for increasing numbers of clients over `watdiv1B` on `watdiv-sts` workload

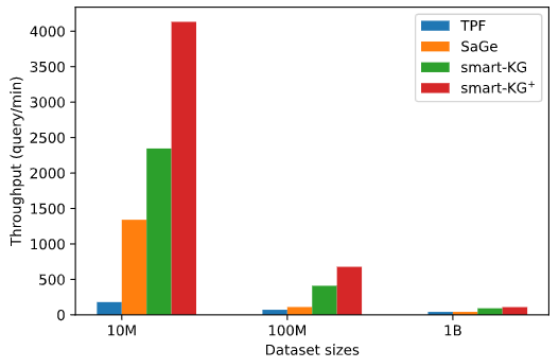
Fig. 6. Number of timeouts (lower is better)

In Table 7, we present the performance analysis of three different configurations applied to the stress workload `watdiv-sts`. The results demonstrate a significant improvement in the performance of `smart-KG+` (brTPF +NP) when compared to the other two versions. We note that the `smart-KG+` (brTPF +NP) version experienced 6 timeouts, whereas the remaining versions did not encounter any timeouts. These timeouts results from the following reasons. First, low selective queries may time out due to the strategy of attaching large intermediate results back to the server. Second, the process of attaching the intermediate results to the `brTPF` request incurs higher costs compared to a regular `TPF` request. This increased cost further affects the overall performance and contributes to the occurrence of timeouts. Third, a mismatch in query planning can potentially lead to longer execution times. This was observed in two queries in `TPF+NP`, which took more than a minute to execute, as well as in the case of `C2`.

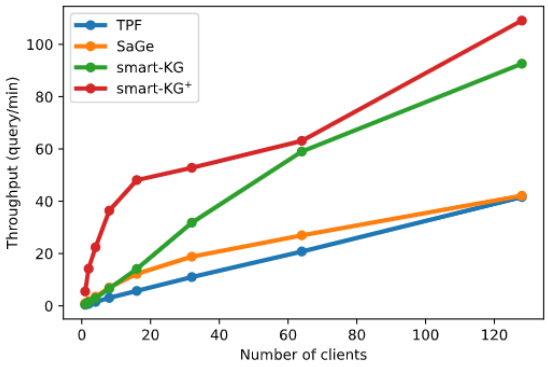
To conclude, the new query planner finds better query plans but with the cost of a server request. In addition, our strategy to query brTPF achieves better performance in queries that require shipping a small number of intermediate result, while querying TPF achieves better performance for queries that require shipping many intermediate results.

### *5.4. System Performance Evaluation*

In this section, we evaluate the performance of an increasing number of concurrent clients (up to 128 clients) on three different graph sizes including `watdiv10M`, `watdiv100M`, and `watdiv1B` using `watdiv-sts` workload.

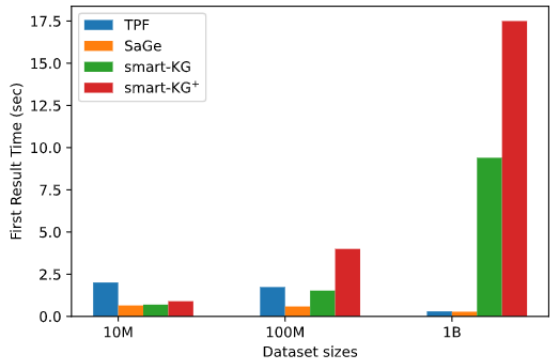



(a) Query throughput of 128 concurrent clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets on `watdiv-sts` workload

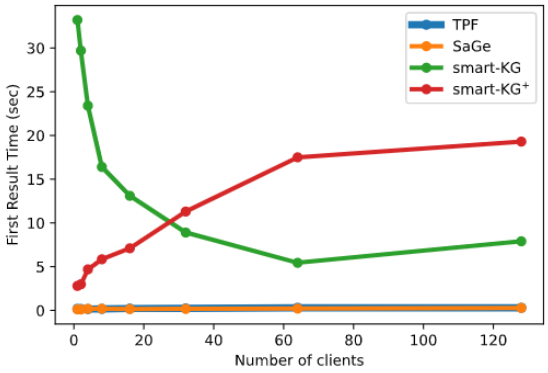



(b) Query throughput for increasing numbers of clients over `watdiv1B` on `watdiv-sts` workload

Fig. 7. Throughput (higher is better)




(a) Average first result time for 128 clients over increasing sizes datasets on `watdiv-sts` workload




(b) Average first result time for increasing number of clients over `watdiv1B` dataset on `watdiv-sts` workload

Fig. 8. Average first result time (lower is better)

For these experiments, and the ones presented in Sec. 5.5 and Sec. 5.6, `smart-KG+` does not use the typed-partitions. This allows for measuring the impact of the new planning and pipelined join strategies implemented and comparing them to the previous techniques implemented in `smart-KG`.

**Workload Completion Time Analysis.** Fig. 5 shows the average workload completion time results of executing the `watdiv-sts` workload including the queries that have timed out. Fig. 5a shows the scenario of increasing KG size with the highest number of concurrent clients (128 clients) on using `watdiv-sts`. `smart-KG+` is up to 7, 2, and 1.3 times faster than `smart-KG` on `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets, respectively. This improvement in performance is due to performing query planning on the server-side, which results in fewer intermediate results transferred over the network.

As shown in Fig. 5b, `smart-KG+` provides a significant performance improvement compared to all systems; `smart-KG+` has an outstanding performance over `watdiv1B` dataset from 1 up to 32 clients compared to `smart-KG` since `smart-KG+` utilizes brTPF which significantly reduces the number of HTTP requests. Note that `smart-KG` performance slightly improves with an increasing number of concurrent clients since TPF requests sent by `smart-KG` clients have a higher potential for a cache hit than brTPF request sent by `smart-KG+` client since

the HTTP caching is designed to serve the identical requests to earlier ones without the need to access the server to recompute the response over again.

Overall, smart-KG$^+$ provides a faster workload completion time on using watdiv-sts than TPF and SaGe in all experiment setups from 1 up to 128 clients over watdiv1B dataset. smart-KG$^+$ is up to 18 and 7 times faster in the case of 1 client workload, and 3 and 2.6 times with 128 concurrent client workloads than TPF and SaGe, respectively. For less than 16 concurrent clients, SaGe provides a slightly faster workload completion time than smart-KG. From this point forth, SaGe suffers from performance degradation due to the excessive waiting queue time of the round-robin policy.

**Timeout Analysis.** Fig. 6a illustrates that smart-KG and smart-KG$^+$ produces relatively low timeouts compared to the state-of-the-art system TPF and SaGe. That is, with 128 concurrent clients, smart-KG$^+$ and smart-KG have approximately a percentage of $9\%$ and $13\%$ of watdiv-sts workload queries timed out over watdiv1B dataset. In contrast, as shown in Fig. 6b, on watdiv1B, the percentage of timeouts increases for TPF with an increasing number of clients, from $44\%$ in 1-client workload to $56\%$ with 128 clients. Similarly, the percentage of timeouts rises rapidly from $10\%$ with 1 client up to $54\%$ with 128 concurrent clients.

As expected, the number of timeouts of TPF and SaGe has excessively increased with the size of the RDF KG size. On watdiv10M, SaGe produces no timeouts while TPF has a percentage of $10\%$ timeouts. In turn, SaGe timeouts increase substantially with increasing KG sizes, with a trend similar to TPF over watdiv100M and watdiv1B.

**Throughput Analysis.** We consider throughput as a metric to explore the performance of the systems under high load, i.e., an increasing number of concurrent clients and the sizes of the KGs. We measure the throughput as the total number of queries executed per minute from all concurrent clients. Note that we consider the queries that have terminated successfully and provided complete results within the predetermined timeout limit.

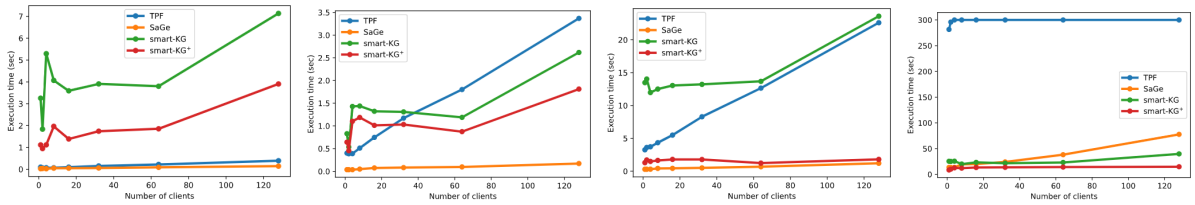
Fig. 7a shows that smart-KG$^+$ achieves higher throughput values than all the compared systems over different KG sizes reaching 4132, 678, 109 query/min over watdiv10M, watdiv100M and watdiv1B, respectively. In Fig. 7b, we report two main findings. First, smart-KG$^+$ scales better than the other approaches since it has a higher query throughput with an increasing number of clients. Second, all compared systems are able to achieve higher throughput with an increasing number of clients, which shows that the systems can scale well but at a different rate.

**First Result of a Query Analysis.** Fig. 8a shows that SaGe provides the best response time to all systems over different sizes of KGs. This is not surprising since SaGe is, in principle, a SPARQL endpoint that adopts a Web preemption technique to avoid the convoy effect phenomenon caused by the long-running queries. smart-KG$^+$ provides a comparable query response time to smart-KG and SaGe and even slightly better than TPF on watdiv10M dataset. However, as the KG size increases, the average response time also increases for watdiv100M and watdiv1B. This can be attributed to two factors. First, the larger sizes of the shipped KG partitions result in longer download times. Second, smart-KG$^+$ relies on brTPF for handling single triple pattern fragments, unlike the previous version smart-KG which used TPF. While brTPF potentially requires fewer requests compared to TPF, it introduces additional time for attaching and parsing solution mappings. Moreover, brTPF utilizes the binding join strategy to distribute the workload between clients and the server. Consequently, with an increasing number of clients, brTPF puts more load on the server compared to TPF, resulting in slightly slower query response times as shown in Fig. 8a.

In Fig. 8b, we observe that TPF and SaGe have an almost constant curve (i.e. negligible response time increase) with an increasing number of clients. In turn, smart-KG$^+$ has on average a longer response time between 2 seconds on 1 client workload and 17 seconds on 128 client workload. As a final noteworthy observation regarding the response time metric, smart-KG response time is actually decreasing with an increasing number of concurrent clients, as we discussed earlier, the likelihood of a cache hit for identical TPF requests from different query execution is higher than brTPF requests. In other words, with an increasing number of clients, TPF requests issued by smart-KG clients are more frequently answered from an HTTP cache that acts as a proxy server than brTPF requests issued by smart-KG$^+$. This is consistent with the results reported by Hartig and Buil-Aranda [14].

### *5.5. Performance evaluation on different query shapes*

In this section, we investigate the query performance of smart-KG$^+$ compered to the state-of-the-art interfaces on four different query shapes previously introduced by the *WatDiv Basic Testing* [71]. In the following, we provide




(a) L1 (b) S1 (c) F1 (d) C1

Fig. 9. Avg. execution time per client on WatDiv-100M, for the first query of each category L, S, F, and C and the rest in Appendix C

Table 8

Avg. execution time per client (in sec.) for 128 clients over `watdiv100M` for the `watdiv-btt` workload. GM=Geometric Mean per query class. GM-T = Total Geometric mean for all query classes.

| Query | L1 | L2 | L3 | L4 | L5 | GM-L | F1 | F2 | F3 | F4 | F5 | GM-F | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TPF | **0.39** | 268.7 | 0.16 | 35.9 | 117.18 | 9.3 | 22.60 | 44.35 | 41.8 | 50.27 | 2.21 | 21.5 | |
| SaGe | 0.141 | 11.27 | **0.26** | 6.86 | 7.47 | **1.84** | **1.21** | **0.93** | **1.67** | **2.33** | **0.37** | **1.1** | |
| `smart-KG` | 7.13 | 20.66 | 0.88 | 2.89 | 0.94 | 3.23 | 23.58 | 7.19 | 28.19 | 7.17 | 7.83 | 12.18 | |
| `smart-KG+` | 3.90 | **5.9** | 0.92 | **1.99** | **0.875** | 2.05 | 1.8 | 1.832 | 3.34 | 2.75 | 2.27 | 2.39 | |
| Query | S1 | S2 | S3 | S4 | S5 | S6 | S7 | GM-S | C1 | C2 | C3 | GM-C | GM-T |
| TPF | 3.36 | 59.2 | 38.063 | 36.91 | 92.39 | 9.58 | 0.034 | 9.75 | 300.0 | 300.0 | 510.37 | 358.13 | 20.19 |
| SaGe | **0.17** | 2.79 | 10.85 | **2.9** | 5.70 | **0.77** | **0.09** | **1.28** | 77.74 | **74.18** | 480.10 | 140.41 | **2.73** |
| `smart-KG` | 2.61 | 0.99 | 0.91 | 43.02 | 3.62 | 67.41 | 0.97 | 4.22 | 39.85 | 200.0 | 363.35 | 163.16 | 4.58 |
| `smart-KG+` | 1.81 | **0.83** | **0.402** | 23.02 | **2.8** | 3.6 | 0.43 | 1.79 | 14.85 | 310.0 | **260.11** | **106.18** | 3.65 |

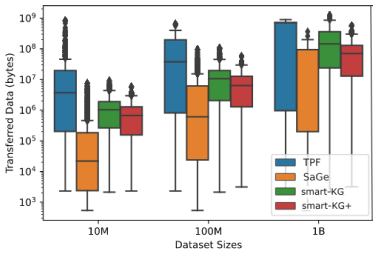
an overview of the trend of the average execution time of each category in Fig. 9, while the rest of the queries are analyzed in detail in Appendix C. In general, SaGe has an outstanding performance for all query shapes. This behavior can be explained by the size of the workload; the `watdiv-btt` workload includes only 20 queries per client inducing a low query arrival rate to the SaGe server. In contrast, TPF is significantly worse than most of the compared systems except in simple queries due to shipping large intermediate results and a high number of requests. In turn, `smart-KG` provides a relatively slow performance in L, S, and F queries since it ships partitions with unnecessary intermediate results for such selective queries. Yet, `smart-KG+` provides an efficient query performance in F and C queries. Interestingly, although `smart-KG+` still has to ship the same partitions as `smart-KG`, `smart-KG+` provides better performance in most query shapes thanks to the more accurate query planner. As expected, the performance of `smart-KG+` enhances gradually from simple L queries reaching its best performance in complex C queries.

Table 8 summarises the average execution time for all different query shapes over `watdiv100M` with 128 clients. For the L-workload, `smart-KG+` and SaGe offer comparable performance in L, with a better geometric mean in the simplest queries and highly selective queries with a small diameter. For the F queries, SaGe provides the best performance in F-workload compared to all systems. In turn, `smart-KG+` outperforms all the compared systems in C queries, except in C2, where both `smart-KG` and TPF timeouts are due to the large intermediate results. Finally, `smart-KG+` achieves the second smallest total geometric mean after SaGe that behaves `watdiv-btt` workload as a SPARQL endpoint since `watdiv-btt` has low number of queries (cf. Table 8, GM-T column).

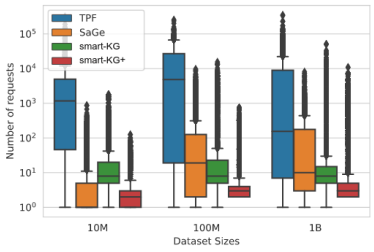
### *5.6. Resource Consumption*

**Network Load.** We report two main metrics to describe the network load: the total number of requests received by the server and the number of bytes transferred on the network between clients and the server. The results reported in the following do not account for queries that timed out.

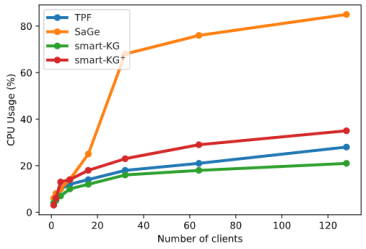
Fig. 10a shows the distribution of the number of transferred bytes on increasing KG sizes with 128 concurrent clients. SaGe transfers the least number of bytes over the network compared to all state-of-the-art systems since SaGe acts as a full SPARQL endpoint with a Web preemption as an additional feature to prevent query execution starvation with no intermediate results. SaGe only consumes a small extra data transfer overhead to send query plans of a long running-query in order to enable the clients to resume query execution afterward. In contrast, TPF




(a) Box plot summary for `transferred data` per query for the 128 clients over `watdiv10M`, `watdiv100M`, `watdiv1B` on `watdiv-sts` workload (log scale)




(b) Box plot summary for the `number of requests` per query for the 128 clients over `watdiv10M`, `watdiv100M`, `watdiv1B` on `watdiv-sts` workload (log scale)




(c) Avg. Server CPU Usage (in %) for increasing number of clients over `watdiv1B` dataset on `watdiv-sts` workload

Fig. 10. Server resource consumption with increasing number of clients and increasing dataset sizes on `watdiv-sts` workload

incurs the highest data transfer cost due to the enormous amount of shipped intermediate results leading to low query execution performance as already shown in Fig. 5 and Fig. 6.

`smart-KG+` requires less data transfer than `smart-KG`. This is expected for two main reasons. First, `smart-KG+` utilizes a star pattern reordering based on cardinality estimation which eventually reduces the intermediate results transferred on the network. Second, `smart-KG+` employs brTPF to handle single non-star triple patterns which reduces the data transfer compared to TPF. To be precise, `smart-KG+` requires to transfer on average 8.1MB and 86.8MB per query over `watdiv100M` and `watdiv1B`. As expected, `smart-KG+` transfers more data over the network than SaGe, but up to 87% and 40% less data than TPF and `smart-KG` per query over `watdiv100M` dataset. Yet, `smart-KG+` can leverage the transferred partitions by reusing them in future queries.

As shown in Fig. 10b, `smart-KG+` significantly reduces the number of requests in comparison to all of the compared systems. `smart-KG+` requires on average 8, 17 and 178 requests over `watdiv10M`, `watdiv100M`, and `watdiv1B`. In contrast, TPF incurs an enormous number of requests, reaching more than $10 - 30K$ requests per query (on average) over the different WatDiv datasets. For SaGe, the number of requests considerably increases as a consequence of the scheduling mechanism to allocate server resources among the workload queries.

In both versions of smart-KG, the implementation of a caching mechanism would potentially yield a substantial performance improvement. Two strategies can be employed: server-side caching of popular families in-memory and client-side caching, where families are stored locally upon shipment, enabling their reuse for subsequent queries involving the same families. Caching the partitions on the client-side will execute *streak queries* with minimal communication to the server. A streak [63] is defined as a sequence of queries that appear as subsequent modifications of a seed query.

**Server CPU Usage.** Fig. 10c shows that `smart-KG`, TPF, and `smart-KG+` only consume less than 30% of server CPU in order to process the `watdiv-sts` query workload on all number of clients setups. This is because the aforementioned interfaces limit the client to send certain query patterns (i.e less expressive queries) to the server (e.g. single triple patterns and star patterns). This allows for distributing the query execution computation cost between the client and the server. In contrast, SaGe offers a more expressive server interface with few operators executed on the clients. Thus, SaGe server is able to execute more complex queries which extensively use the server CPU leading to a rapid surge of CPU usage. In particular, SaGe uses less than 30% CPU usage for 1 up to 16 clients and then escalates up to $80 - 100\%$ for 32 to 128 clients.

**Server Disk Usage.** Table 9 presents a comparison of the required disk storage for all compared systems. We consider four KGs with diverse raw data sizes (in N-Triples). In practice, TPF and SaGe rely on the compressed HDT file format that offers a high space-efficient representation. In turn, `smart-KG` and `smart-KG+` rely on the family partitioning mechanism that demands additional disk space to store HDT partitions, specifically both systems mandate double the N-Triples format size of Watdiv KG. Note that DBpedia requires less storage space

Table 9
Comparison of storage requirements (in MB) for systems with HDT backend vs original graph size (raw)

| Dataset | Raw | family partitioning | *typed*-family partitioning | TPF/SaGe |
|---|---|---|---|---|
| WatDiv-10M | 1,471 | 2,783 | 5,632 | 112 |
| WatDiv-100M | 14,876 | 29,711 | 58,265 | 1,186 |
| WatDiv-1000M | 151,862 | 310,574 | 624,253 | 12,793 |
| DBpedia | 158,197 | 122,440 | 150,528 | 17,904 |

since we apply the pruning parameters to reduce the number of materialized HDT partitions. Considering that disk storage is the most economical server resource, smart-KG$^+$ supports an admissible trade-off to obtain better query performance alongside less server CPU consumption.

**Client CPU and RAM usage.** The SaGe client locally performs two main tasks: first, resuming the suspended query execution based on the saved plan received earlier from the SaGe server; second, executing the non-preemptable SPARQL operators including aggregation functions as well as OPTIONAL, ORDER BY, GROUP BY, DISTINCT, etc. Given the aforementioned tasks, SaGe clients, nevertheless, demand a feasible (on average 15%) CPU usage and reasonable RAM size ($\sim$ 2GB) for all workloads. In turn, TPF requires a higher computation cost (on average $45\%$) on the client-side than SaGe since TPF clients locally execute the expensive join operator. Similar to TPF, smart-KG performs the join processing of single triple patterns and star patterns queries over the shipped partitions, leading to a higher client CPU consumption (on average $70\%$) than TPF and SaGe which could be expensive for light client systems. In turn, smart-KG$^+$ demands (on average $55\%$) less client-side processing than smart-KG, as it processes fewer intermediate results on the client due to the bind join strategy supported by brTPF as well as the more efficient query plans devised by the server-side optimizer and planner.

Note that the aforementioned percentages de-escalate with an increasing number of clients due to the bottleneck on the server-side since the clients are almost idle awaiting to receive the server response. In other words, the network traffic dominates the query execution of TPF, smart-KG, and smart-KG$^+$ while context switching overhead and waiting queues dominate in the case of SaGe. Upon comparing the two versions of smart-KG, as depicted in Figure 10a, we observe that the smart-KG exhibits a higher data transfer and has the potential to consume a greater amount of client RAM compared to smart-KG$^+$. Compared to SaGe and TPF, smart-KG$^+$ needs a higher client RAM since it loads the HDT partitions in client memory, however it still affordable. For instance, smart-KG$^+$ requires up to 3 GB to execute the watdiv-sts workload over watdiv1B.

### *5.7. Typed-family Partitioning Evaluation*

In this part of the evaluation, we focus on evaluating typed-family partitioning using synthetic and real-world KGs on multiple KG sizes and on different query workloads. Therefore, we compare the smart-KG$^+$ implementation using only family partitioning and the extended version that additionally uses typed-family partitioning. smart-KG$^+$ adopts the server-side query planner and Client-side and Server-side Joining evaluated in Section 5.4 - Section 5.6.

#### *5.7.1. Typed-family evaluation on the WatDiv dataset*

Table 10 presents a comparison between typed-family partitioning and family partitioning on total transferred data and workload completion time on different sizes of the WatDiv dataset. Typed-family partitioning significantly decreases the number of transferred bytes (on average 27% and 32% over 10M and 100M datasets, respectively) shipped over the network compared to the original family partitioning on watdiv-sts workload. As expected, typed-family partitioning demands up to 41% and 46% less transferred data over watdiv10M and watdiv100M, respectively on watdiv-stf$_{bounded}$ and watdiv-stf$_{both}$ workloads since we only ship the family partitions that contains the exact solution bindings to the star-shaped subquery. We also show in Table 10 the impact of typed-family partitioning on the watdiv-sts workload completion time. Typed-family partitioning has substantially reduced (over 40%) the completion time for watdiv-stf$_{bounded}$ and watdiv-stf$_{both}$ on both watdiv10M and watdiv100M datasets.

Table 10

`Workload Transferred Data` and `Workload Completion Time` per client over `watdiv10M` and `watdiv100M` on `watdiv-stf`$_{bounded}$, `watdiv-stf`$_{unbounded}$ and `watdiv-stf`$_{both}$ workloads

| Query Workload | Workload Transferred Data (MB) | | | | | | Workload Completion Time (ms) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 10M | | | 100M | | | 10M | | | 100M | | |
| | Original | Typed | % | Original | Typed | % | Original | Typed | % | Original | Typed | % |
| watdiv-stf$_{bounded}$ | 42.14 | 24.8 | (-) 41% | 401.97 | 216,71 | (-) 46% | 22956 | 12621 | (-) 45% | 47167 | 27479 | (-) 42% |
| watdiv-stf$_{unbounded}$ | 28.82 | 28.85 | (+) 0.12 % | 236.21 | 236.24 | (+) 0.01% | 10538 | 11237 | (+) 7% | 12228 | 12691 | (+) 3% |
| watdiv-stf$_{both}$ | 6.02 | 2.29 | (-) 62% | 68.61 | 24.59 | (-) 64% | 7668 | 3702 | (-) 52% | 15022 | 5941 | (-) 60% |
| Summary | 76.99 | 55.95 | (-) 27% | 706.79 | 477.54 | (-) 32% | 41162 | 27560 | (-) 33% | 74417 | 46111 | (-) 38% |

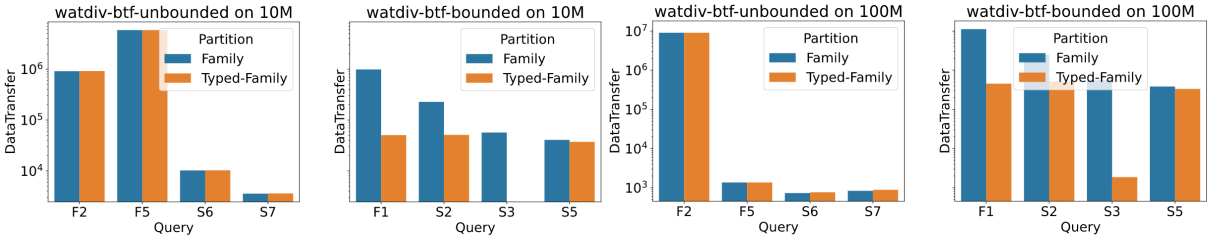



Fig. 11. Data transferred per query (in bytes) over `watdiv10M` and `watdiv100M` on `watdiv-btf`$_{unbounded}$ and `watdiv-btf`$_{bounded}$ workloads

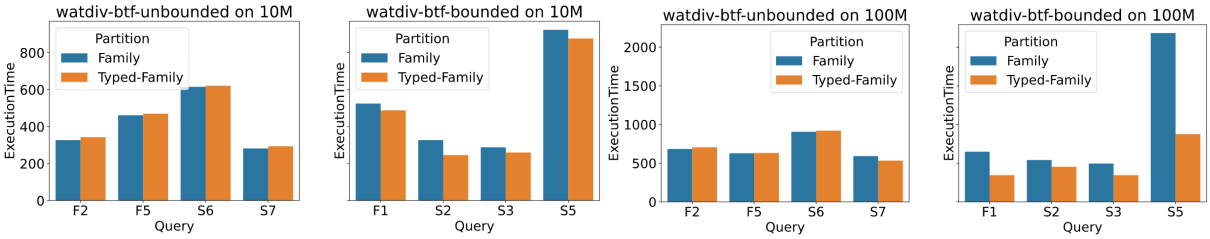


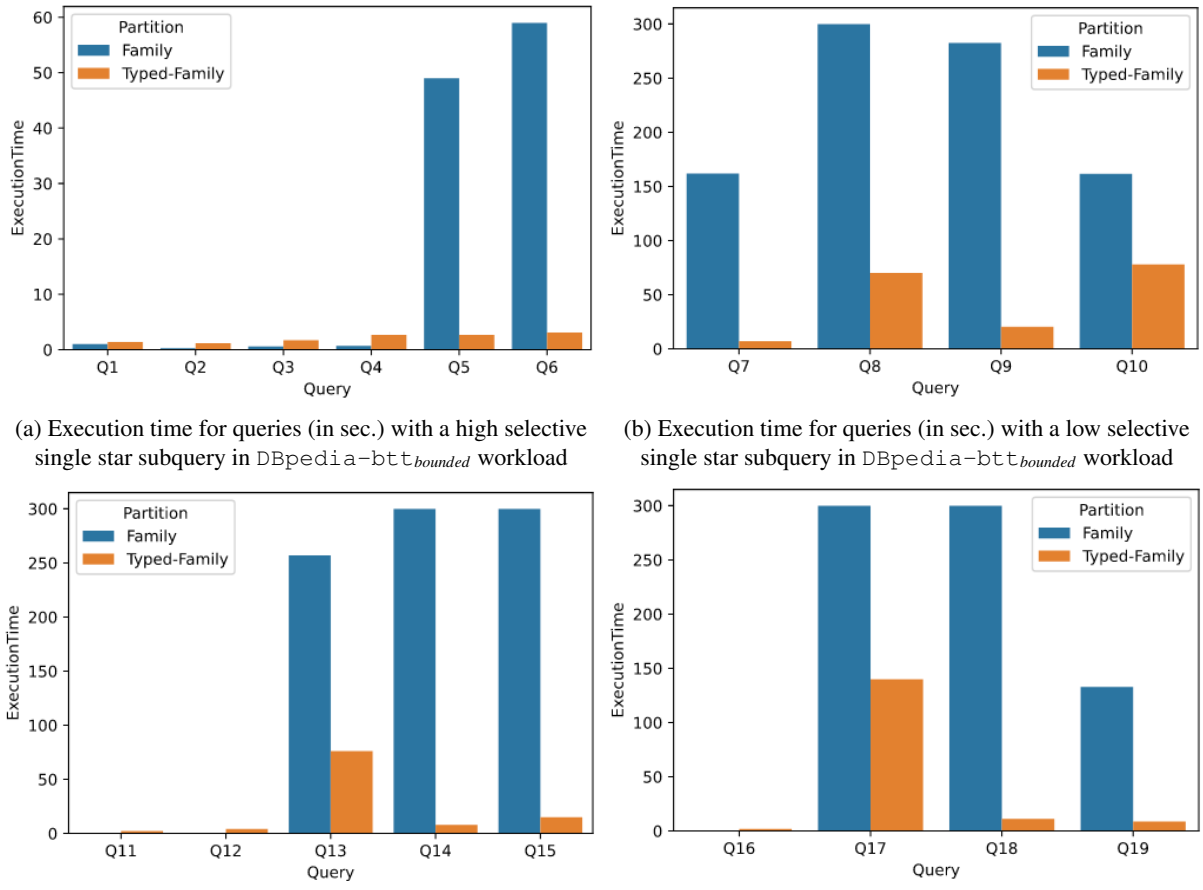

Fig. 12. Execution time per query (in ms) over `watdiv10M` and `watdiv100M` on `watdiv-btf`$_{unbounded}$ and `watdiv-btf`$_{bounded}$ workloads

In Fig. 11 and Fig. 12, we show at the query level the impact of typed-family partitioning on the execution of different query shapes extracted from the WatDiv Basic Testing query set. Fig. 11 shows that typed-family partitioning significantly decreases the data transferred of `watdiv-btf`$_{bounded}$ workload queries. For instance, using typed-family partitioning, query S3 requires only 3% and 1% of the transferred data required over `watdiv10M` and `watdiv100M`, respectively, in comparison to using original family partitioning. In addition, when using typed-family partitioning, queries F1 and S2 demand 97% - 99% less data transfer than when using family partitioning. Note that typed-family partitioning has no major influence on the queries of the `watdiv-btf`$_{unbounded}$.

Fig. 12 shows the execution time of the workloads. For the `watdiv-btf`$_{bounded}$ workload, the execution time of queries has been significantly reduced thanks to typed-partitioning for downsizing the shipped partitions, e.g., with a percent of decrease between -30% and -60 % in `watdiv-100M` dataset. Yet, typed-family partitioning has a positive bearing on the query performance of the `watdiv-btf`$_{bounded}$ workload for several queries. For the `watdiv-btf`$_{unbounded}$ workload, the runtime with typed-family partitioning has a slight increase of 5-8 ms for unbounded queries. This increase is attributed to the query planner needing to search through additional metadata associated with typed family partitioning. Yet, this delay is an implementation detail that can be optimized further.

### *5.7.2. Typed-family evaluation on the DBpedia dataset*

In this section, we analyze the impact of typed family-partitioning compared to family partitioning on the execution time of the `DBpedia-btt`$_{bounded}$ and workload. Note that `smart-KG` relies on family partitioning where we do not materialize any partition that contains the predicate `rdf:type` since `rdf:type` $\notin P'_G$.




(a) Execution time for queries (in sec.) with a high selective single star subquery in DBpedia-btt$_{bounded}$ workload

(b) Execution time for queries (in sec.) with a low selective single star subquery in DBpedia-btt$_{bounded}$ workload

(c) Execution time for queries (in sec.) with a single star and multiple triple patterns in DBpedia-btt$_{bounded}$ workload

(d) Execution time for queries (in sec.) with multiple stars in DBpedia-btt$_{bounded}$ workload

Fig. 13. Execution time per query (in sec) on DBpedia-btt$_{bounded}$ workload

In Fig. 13, we divide the queries in Dbpedia-btt$_{bounded}$ into four different categories based on the number of star-shaped subqueries, subquery selectivity, and a star-shaped query combined with single triple patterns. Fig. 13a shows the performance for highly selective star queries. smart-KG$^+$ with typed family-partitioning executes queries Q1 up Q4 slightly slower than smart-KG$^+$ with family partitioning. This is due to the fact that family partitioning does not materialize any partitions that contain rdf:type predicate, since the percentage of triples with this predicate is higher than the defined threshold $\tau_{p_{high}}$. In queries Q5 and Q6, smart-KG$^+$ with typed family-partitioning achieves a better performance since it ships a partition that resolves the query locally while brTPF has a poor performance since some of the triple patterns are non-selective (even though the entire star-query is highly selective).

Fig. 13b shows that relying on typed family-partitioning achieves better query processing performance compared to family partitioning. This is because smart-KG$^+$ ships a typed partition that contains the solution mappings of the entire star query, while on using family-based partitioning, smart-KG$^+$ will utilize brTPF to resolve the triple pattern with rdf:type, which requires an enormous number of requests to join with a non-selective star subquery.

Fig. 13c presents the execution time of queries that combine a star subquery with a couple of single triple patterns in a BGP. In queries Q11 and Q12, smart-KG$^+$ with family partitioning performs slightly better than with typed family-partitioning, since these queries include highly selective triple patterns and do not require shipping an entire partition to resolve the query. On the other hand, Q13, Q14, and Q15 show a significant improvement when using typed family-partition since it reduces the amount of data transferred.

Table 11
Impact of Typed-Family Partitioning on WiseKG's Performance on watdiv10M dataset (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in milliseconds, TO: Timeouts.)

| Workload | WiseKG$^{Family}$ | | | | WiseKG$^{Typed-Family}$ | | | |
|---|---|---|---|---|---|---|---|---|
| | Req | DT | ET | TO | Req | DT | ET | TO |
| watdiv-sts | 2452 | 101.81 | 39610 | 6 | **2301** | **85.72** | **39093** | 6 |
| watdiv-btf | 179 | 27.32 | 23666 | 1 | 179 | 27.32 | 23680 | 1 |
| watdiv-stf$_{bounded}$ | 528 | 22.54 | 17097 | 0 | **377** | **10.20** | **12285** | 0 |
| watdiv-stf$_{unbounded}$ | 36 | 9.0 | 3286 | 0 | 36 | 9.0 | 3235 | 0 |
| watdiv-stf$_{both}$ | 98 | 5.24 | 2787 | 0 | 98 | **1.49** | **1970** | 0 |

Fig. 13d shows that smart-KG$^+$ has a better performance in queries Q17, Q18, and Q19 when relying on typed family-partitioning and better performance in query Q16 when using a family partition. Note that the query performance highly depends on the selectivity of the star-typed subquery $Q_s$ (i.e the size of the shipped partition in case of typed family-partitions) compared to the selectivity of $Q'_s$ after decomposing the typed star to $Q'_s$ and $Q''_s$. Finally, in queries with no bound types, the observed performance of family partitions and typed-family partitions is almost identical. This is consistent with the results on WatDiv watdiv-btf$_{unbounded}$ presented in Section 5.7.1.

### *5.7.3. Assessing the impact of typed-family partitioning on* WiseKG

WiseKG [25] is an LDF interface that dynamically shifts the query processing load between client and server. WiseKG combines two LDF APIs (SPF and smart-KG) that enable server-side and client-side processing of star-shaped sub-patterns. WiseKG decides whether the star-subqueries should be processed on the client or on the server. For this, WiseKG relies on a cost model that picks the best-suited API per sub-query based on the current server load, client capabilities, estimation of necessary data transfer between client and server, and network bandwidth. By leveraging this cost model, WiseKG dynamically distributes query processing tasks between servers and clients, better-utilizing server resources and maintaining high-performance levels even under conditions of heavy load.

Earlier experiments have demonstrated that WiseKG outperforms state-of-the-art stand-alone LDF interfaces, especially under highly demanding workloads. Consequently, this section evaluates the impact of our typed-family partitioning on WiseKG's performance. To do so, the following two versions of WiseKG are developed:

- WiseKG$^{Family}$: The original version of WiseKG relies on the family generator from smart-KG.
- WiseKG$^{Typed-Family}$: An extension of the earlier version where we incorporate typed-family partitioning proposed in smart-KG$^+$.

The experimental results, as presented in Table 11, are based on watdiv10M dataset. We observe that WiseKG$^{Typed-Family}$ achieves significant reductions in data transfer by 16%, 54%, and 71% for watdiv-sts, watdiv-stf$_{bounded}$, and watdiv-stf$_{both}$, respectively, when compared to WiseKG$^{Family}$. Additionally, WiseKG$^{Typed-Family}$ requires 7% fewer requests than WiseKG$^{Family}$ for watdiv-sts and 28% fewer requests for watdiv-stf$_{bounded}$. This performance improvement is attributed to the adoption of typed-family partitioning, which effectively reduces data transfer and the number of requests for queries involving bounded star-typed patterns. It is worth noting that the number of requests and the data transferred by WiseKG remain unaffected by typed-family partitioning in the case of the watdiv-stf$_{unbounded}$ workload, consistent with the results presented in Section 5.7.1. Moreover, the impact of typed-partitioning on WiseKG's performance with the watdiv-btf workload is minimal due to its relatively smaller size query workload, and the cost model of WiseKG effectively executes most of the queries on the server-side using the SPF API.

## *5.8. Lesson Learned*

Concluding the evaluation of the experimental results for our proposed approach, we provide a summary of our lessons learned in the following:

*Ablation Study*

- **Server-side query planning (NP) enhances complex query performance**: The proposed solution (brTPF+NP) enhances the performance of complex queries (C) by up to 50% through effective star reordering. However, our solution encounters timeouts in some cases, mainly because of attaching large intermediate results back to the server and query planning mismatches.
- **Trade-off in query planning strategies**: The new server-side query planner in smart-KG$^+$ achieves better query plans, but this improvement comes at the cost of increased server requests. Additionally, the strategy to query brTPF achieves better performance in queries requiring a small number of intermediate results, while querying TPF is more efficient for queries involving many intermediate results.

*System Performance Evaluation*

- **Improved throughput, resource consumption and scalability**: smart-KG$^+$ scales better, achieving higher query throughput with an increasing number of clients over various graph sizes. Our solution requires fewer transferred data and sends a lower number of requests per query compared to other systems. Additionally, it performs efficiently on different graph sizes, outperforming TPF and SaGe in workload completion time, achieving up to 18x and 7x faster performance with increasing the number of concurrent clients. In summary, smart-KG$^+$ demonstrates efficient resource consumption and scalability.
- **Query shape matters**: smart-KG$^+$ excels in complex queries (C), it may not be as efficient in simple queries (L) and moderately selective queries (F).

*Typed-family Partitioning*

- **Typed-Family partitioning reduces data transfer**: The evaluation shows that typed-family partitioning significantly decreases the amount of data transferred over the network compared to using only family partitioning. On average, using typed-family partitioning results in a reduction of 27% and 32% in transferred bytes for 10M- and 100M-sized datasets, respectively, in the watdiv-sts workload. This reduction can be even higher (up to 46%) in certain cases, such as watdiv-stf$_{bounded}$ workload.
- **Query completion time improvement**: Typed-family partitioning substantially reduces the completion time for queries in the watdiv-stf$_{bounded}$ workload on both 10M- and 100M-sized datasets. The improvement in completion time ranges from 40% to 46% for these specific workloads.
- **Minimal impact/overhead on unbounded queries**: For queries without bound types (unbounded queries), the observed performance of family partitions and typed-family partitions is almost identical.
- **Extension to WiseKG**: The evaluation also extends the evaluation to WiseKG, an LDF interface that dynamically shifts query processing load between clients and servers. The results show that typed-family partitioning significantly reduces data transfer and the number of requests of WiseKG in star queries with bound *rdf:type* predicates.

## 6. Conclusion and Future Work

We introduced smart-KG$^+$, a hybrid shipping approach to efficiently query Knowledge Graphs (KGs) on the Web, while balancing the load between servers and clients. We combine the Bindings-Restricted Triple Pattern Fragment (brTPF) strategy with shipping compressed graph partitions that can be locally queried at the client. The served partitions are based on grouping entities described with the same sets of properties and classes benefiting from the special nature of the rdf:type predicate. In smart-KG$^+$, we implement a server-side query planner to provide accurate plans to optimize the trade-off between brTPF and partition shipping.

Our evaluation shows that smart-KG$^+$ performs on average 10 times faster, use 5 times less network traffic, and sends 20 times fewer requests, with 5 times less server CPU, outperforming the state-of-the-art approaches. We show an extensive experimental study on synthetic and real datasets that, at the cost of reasonable server disk storage, smart-KG$^+$ improves the execution time of the query workloads and reduces network cost.

There are several research directions that can be followed in future work. This includes the extension of our framework to define interfaces and their admissible queries for other types of partitioning strategies including workload-

aware and k-way partitioning. Furthermore, future work can investigate other partitioning strategies to reduce the network traffic since this is one of the main factors impacting the performance as shown in our experiments. For example, a study [77] shows that a tiny portion of the KG is actually accessed by a typical DBpedia query workload. Thus, future work can consider the query workload during the KG partitioning (including online partitioning [49]) to minimize the size of materialized partitions to transfer only the data that is required for the query load [46, 54, 55].

In the future, we also plan to investigate the automation of parameter settings for family partitioning in `smart-KG+`. We aim to create a system that autonomously analyzes the RDF graph and suggest optimal parameter settings, based on the trade-off of disk usage and graph coverage with the partitions.

Lastly, in terms of query processing, future work can focus on client-side optimization and devise novel techniques for selecting appropriate physical join operators (e.g., [26, 78]) but that can work on partitioned-based interfaces using, for example, sampled statistics from the partitions [79]. Additionally, the integration of partition-based LDF interfaces in the heterogeneous KG federations landscape can be investigated [80].

**Acknowledgments.** We would like to thank Prof. Katja Hose and Christian Aebeloe from University of Aalborg for allowing us to perform some of our experiments on their servers. The research was partially funded by the EU H2020 research and innovation programme under grant agreement No 957402 (TEAMING.AI).

## References

[1] A. Azzam, J.D. Fernández, M. Acosta, M. Beno and A. Polleres, SMART-KG: Hybrid Shipping for SPARQL Querying on the Web, in: *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Y. Huang, I. King, T. Liu and M. van Steen, eds, ACM / IW3C2, 2020, pp. 984–994. doi:10.1145/3366423.3380177.

[2] P.A. Bonatti, S. Decker, A. Polleres and V. Presutti, Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web (Dagstuhl Seminar 18371), *Dagstuhl Reports* **8**(9) (2018), 29–111. doi:10.4230/DagRep.8.9.29.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z.G. Ives, DBpedia: A Nucleus for a Web of Open Data, in: *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, K. Aberer, K. Choi, N.F. Noy, D. Allemang, K. Lee, L.J.B. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber and P. Cudré-Mauroux, eds, Lecture Notes in Computer Science, Vol. 4825, Springer, 2007, pp. 722–735. doi:10.1007/978-3-540-76298-0_52.

[4] K.D. Bollacker, C. Evans, P.K. Paritosh, T. Sturge and J. Taylor, Freebase: a collaboratively created graph database for structuring human knowledge, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, J.T. Wang, ed., ACM, 2008, pp. 1247–1250. doi:10.1145/1376616.1376746.

[5] F.M. Suchanek, G. Kasneci and G. Weikum, Yago: a core of semantic knowledge, in: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, C.L. Williamson, M.E. Zurko, P.F. Patel-Schneider and P.J. Shenoy, eds, ACM, 2007, pp. 697–706. doi:10.1145/1242572.1242667.

[6] D. Vrandecic and M. Krötzsch, Wikidata: a free collaborative knowledgebase, *Commun. ACM* **57**(10) (2014), 78–85. doi:10.1145/2629489.

[7] C. Guéret, P. Groth, F. van Harmelen and S. Schlobach, Finding the Achilles Heel of the Web of Data: Using Network Analysis for Link-Recommendation, in: *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, P.F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J.Z. Pan, I. Horrocks and B. Glimm, eds, Lecture Notes in Computer Science, Vol. 6496, Springer, 2010, pp. 289–304. doi:10.1007/978-3-642-17746-0_19.

[8] C.B. Aranda, A. Hogan, J. Umbrich and P. Vandenbussche, SPARQL Web-Querying Infrastructure: Ready for Action?, in: *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J.X. Parreira, L. Aroyo, N.F. Noy, C. Welty and K. Janowicz, eds, Lecture Notes in Computer Science, Vol. 8219, Springer, 2013, pp. 277–293. doi:10.1007/978-3-642-41338-4_18.

[9] M. Salvadores, M. Horridge, P.R. Alexander, R.W. Fergerson, M.A. Musen and N.F. Noy, Using SPARQL to Query BioPortal Ontologies and Metadata, in: *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J.X. Parreira, J. Hendler, G. Schreiber, A. Bernstein and E. Blomqvist, eds, Lecture Notes in Computer Science, Vol. 7650, Springer, 2012, pp. 180–195. doi:10.1007/978-3-642-35173-0_12.

[10] A. Polleres, M.R. Kamdar, J.D. Fernández, T. Tudorache and M.A. Musen, A more decentralized vision for Linked Data, *Semantic Web* **11**(1) (2020), 101–113. doi:10.3233/SW-190380.

[11] P. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan and C.B. Aranda, SPARQLES: Monitoring public SPARQL endpoints, *Semantic Web* **8**(6) (2017), 1049–1065. doi:10.3233/SW-170254.

[12] O. Hartig, I. Letter and J. Pérez, A Formal Framework for Comparing Linked Data Fragments, in: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, C. d'Amato, M. Fernández, V.A.M. Tamma, F. Lécué, P. Cudré-Mauroux, J.F. Sequeda, C. Lange and J. Heflin, eds, Lecture Notes in Computer Science, Vol. 10587, Springer, 2017, pp. 364–382. doi:10.1007/978-3-319-68288-4_22.

[13] R. Verborgh, M.V. Sande, O. Hartig, J.V. Herwegen, L.D. Vocht, B.D. Meester, G. Haesendonck and P. Colpaert, Triple Pattern Fragments: A low-cost knowledge graph interface for the Web, *J. Web Semant.* **37-38** (2016), 184–206. doi:10.1016/j.websem.2016.03.003.

[14] O. Hartig and C.B. Aranda, Bindings-Restricted Triple Pattern Fragments, in: *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*, C. Debruyne, H. Panetto, R. Meersman, T.S. Dillon, E. Kühn, D. O'Sullivan and C.A. Ardagna, eds, Lecture Notes in Computer Science, Vol. 10033, 2016, pp. 762–779. doi:10.1007/978-3-319-48472-3_48.

[15] A. Gubichev and T. Neumann, Exploiting the query structure for efficient join ordering in SPARQL queries, in: *EDBT*, 2014.

[16] T. Neumann and G. Moerkotte, Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins, in: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, S. Abiteboul, K. Böhm, C. Koch and K. Tan, eds, IEEE Computer Society, 2011, pp. 984–994. doi:10.1109/ICDE.2011.5767868.

[17] M. Saleem, M.I. Ali, A. Hogan, Q. Mehmood and A.N. Ngomo, LSQ: The Linked SPARQL Queries Dataset, in: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, M. Arenas, Ó. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan and S. Staab, eds, Lecture Notes in Computer Science, Vol. 9367, Springer, 2015, pp. 261–269. doi:10.1007/978-3-319-25010-6_15.

[18] G. Schreiber and Y. Raimond, *RDF 1.1 Primer*, W3C Working Group Note, 2014, https://www.w3.org/TR/rdf11-primer/.

[19] C. Gutierrez, C.A. Hurtado, A.O. Mendelzon and J. Pérez, Foundations of Semantic Web databases, *J. Comput. Syst. Sci.* **77**(3) (2011), 520–541. doi:10.1016/j.jcss.2010.04.009.

[20] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C, 2013, https://www.w3.org/TR/sparql11-query/.

[21] J. Pérez, M. Arenas and C. Gutierrez, Semantics and complexity of SPARQL, *ACM Trans. Database Syst.* **34**(3) (2009), 16:1–16:45. doi:10.1145/1567274.1567278.

[22] J.D. Fernández, M.A. Martínez-P., C. Gutiérrez, A. Polleres and M. Arias, Binary RDF representation for publication and exchange (HDT), *J. Web Semant.* **19** (2013), 22–41. doi:10.1016/j.websem.2013.01.002.

[23] M.A. Martínez-Prieto, N.R. Brisaboa, R. Cánovas, F. Claude and G. Navarro, Practical compressed string dictionaries, *Inf. Syst.* **56** (2016), 73–108. doi:10.1016/j.is.2015.08.008.

[24] M.A. Martínez-Prieto, M.A. Gallego and J.D. Fernández, Exchange and Consumption of Huge RDF Data, in: *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, E. Simperl, P. Cimiano, A. Polleres, Ó. Corcho and V. Presutti, eds, Lecture Notes in Computer Science, Vol. 7295, Springer, 2012, pp. 437–452. doi:10.1007/978-3-642-30284-8_36.

[25] A. Azzam, C. Aebeloe, G. Montoya, I. Keles, A. Polleres and K. Hose, WiseKG: Balanced Access to Web Knowledge Graphs, in: *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, J. Leskovec, M. Grobelnik, M. Najork, J. Tang and L. Zia, eds, ACM / IW3C2, 2021, pp. 1422–1434. doi:10.1145/3442381.3449911.

[26] M. Acosta and M. Vidal, Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data, in: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, M. Arenas, Ó. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan and S. Staab, eds, Lecture Notes in Computer Science, Vol. 9366, Springer, 2015, pp. 111–127. doi:10.1007/978-3-319-25007-6_7.

[27] L. Heling and M. Acosta, Cost- and Robustness-Based Query Optimization for Linked Data Fragments, in: *The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, Athens, Greece, November 2-6, 2020, Proceedings, Part I*, J.Z. Pan, V.A.M. Tamma, C. d'Amato, K. Janowicz, B. Fu, A. Polleres, O. Seneviratne and L. Kagal, eds, Lecture Notes in Computer Science, Vol. 12506, Springer, 2020, pp. 238–257. doi:10.1007/978-3-030-62419-4_14.

[28] R. Taelman, J.V. Herwegen, M.V. Sande and R. Verborgh, Comunica: A Modular SPARQL Query Engine for the Web, in: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, D. Vrandecic, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee and E. Simperl, eds, Lecture Notes in Computer Science, Vol. 11137, Springer, 2018, pp. 239–255. doi:10.1007/978-3-030-00668-6_15.

[29] C. Aebeloe, I. Keles, G. Montoya and K. Hose, Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns, *CoRR* **abs/2002.09172** (2020). https://arxiv.org/abs/2002.09172.

[30] C.B. Aranda, A. Polleres and J. Umbrich, Strategies for Executing Federated Queries in SPARQL1.1, in: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C.A. Knoblock, D. Vrandecic, P. Groth, N.F. Noy, K. Janowicz and C.A. Goble, eds, Lecture Notes in Computer Science, Vol. 8797, Springer, 2014, pp. 390–405. doi:10.1007/978-3-319-11915-1_25.

[31] T. Minier, H. Skaf-Molli and P. Molli, SaGe: Web Preemption for Public SPARQL Query Services, in: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, L. Liu, R.W. White, A. Mantrach, F. Silvestri, J.J. McAuley, R. Baeza-Yates and L. Zia, eds, ACM, 2019, pp. 1268–1278. doi:10.1145/3308558.3313652.

[32] D.J. Abadi, A. Marcus, S. Madden and K. Hollenbach, SW-Store: a vertically partitioned DBMS for Semantic Web data management, *VLDB J.* **18**(2) (2009), 385–406. doi:10.1007/s00778-008-0125-y.

[33] M. Cossu, M. Färber and G. Lausen, PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies, in: *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, M.H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu and K. Hose, eds, OpenProceedings.org, 2018, pp. 469–472. doi:10.5441/002/edbt.2018.49.

[34] D. Graux, L. Jachiet, P. Genevès and N. Layaïda, SPARQLGX in Action: Efficient Distributed Evaluation of SPARQL with Apache Spark, in: *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016*, T. Kawamura and H. Paulheim, eds, CEUR Workshop Proceedings, Vol. 1690, CEUR-WS.org, 2016. https://ceur-ws.org/Vol-1690/paper68.pdf.

[35] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu and G. Lausen, Sempala: Interactive SPARQL Query Processing on Hadoop, in: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C.A. Knoblock, D. Vrandecic, P. Groth, N.F. Noy, K. Janowicz and C.A. Goble, eds, Lecture Notes in Computer Science, Vol. 8796, Springer, 2014, pp. 164–179. doi:10.1007/978-3-319-11964-9_11.

[36] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic and G. Lausen, S2RDF: RDF Querying with SPARQL on Spark, *Proc. VLDB Endow.* **9**(10) (2016), 804–815. doi:10.14778/2977797.2977806. http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf.

[37] X. Chen, H. Chen, N. Zhang and S. Zhang, SparkRDF: Elastic Discreted RDF Graph Processing Engine With Distributed Memory, in: *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, WI-IAT 2015, Singapore, December 6-9, 2015 - Volume I*, IEEE Computer Society, 2015, pp. 292–300. doi:10.1109/WI-IAT.2015.186.

[38] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.N. Ngomo and H. Jabeen, Distributed Semantic Analytics Using the SANSA Stack, in: *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II*, C. d'Amato, M. Fernández, V.A.M. Tamma, F. Lécué, P. Cudré-Mauroux, J.F. Sequeda, C. Lange and J. Heflin, eds, Lecture Notes in Computer Science, Vol. 10588, Springer, 2017, pp. 147–155. doi:10.1007/978-3-319-68204-4_15.

[39] B. Djahandideh, F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz and S. Zampetakis, CliqueSquare in action: Flat plans for massively parallel RDF queries, in: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, J. Gehrke, W. Lehner, K. Shim, S.K. Cha and G.M. Lohman, eds, IEEE Computer Society, 2015, pp. 1432–1435. doi:10.1109/ICDE.2015.7113394.

[40] A. Schätzle, M. Przyjaciel-Zablocki and G. Lausen, PigSPARQL: mapping SPARQL to Pig Latin, in: *Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, Athens, Greece, June 12, 2011*, R.D. Virgilio, F. Giunchiglia and L. Tanca, eds, ACM, 2011, p. 4. doi:10.1145/1999299.1999303.

[41] V. Khadilkar, M. Kantarcioglu, B. Thuraisingham and P. Castagna, Jena-HBase: A Distributed, Scalable and Effcient RDF Triple Store, in: *Proceedings of the ISWC 2012 Posters & Demonstrations Track, Boston, USA, November 11-15, 2012*, B. Glimm and D. Huynh, eds, CEUR Workshop Proceedings, Vol. 914, CEUR-WS.org, 2012. https://ceur-ws.org/Vol-914/paper_14.pdf.

[42] J. Du, H. Wang, Y. Ni and Y. Yu, HadoopRDF: A Scalable Semantic Data Analytical Engine, in: *Intelligent Computing Theories and Applications - 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings*, D. Huang, J. Ma, K. Jo and M.M. Gromiha, eds, Lecture Notes in Computer Science, Vol. 7390, Springer, 2012, pp. 633–641. doi:10.1007/978-3-642-31576-3_80.

[43] A. Akhter, A.N. Ngomo and M. Saleem, An Empirical Evaluation of RDF Graph Partitioning Techniques, in: *Knowledge Engineering and Knowledge Management - 21st International Conference, EKAW 2018, Nancy, France, November 12-16, 2018, Proceedings*, C. Faron-Zucker, C. Ghidini, A. Napoli and Y. Toussaint, eds, Lecture Notes in Computer Science, Vol. 11313, Springer, 2018, pp. 3–18. doi:10.1007/978-3-030-03667-6_1.

[44] K. Rohloff and R.E. Schantz, High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store, in: *SPLASH Workshop on Programming Support Innovations for Emerging Distributed Applications*, ACM, 2010, p. 4. doi:10.1145/1940747.1940751.

[45] R.T. Whitman, B.G. Marsh, M.B. Park and E.G. Hoel, Distributed Spatial and Spatio-Temporal Join on Apache Spark, *ACM Trans. Spatial Algorithms Syst.* **5**(1) (2019), 6:1–6:28. doi:10.1145/3325135.

[46] L. Galárraga, K. Hose and R. Schenkel, Partout: a distributed engine for efficient RDF processing, in: *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, C. Chung, A.Z. Broder, K. Shim and T. Suel, eds, ACM, 2014, pp. 267–268. doi:10.1145/2567948.2577302.

[47] A. Harth, J. Umbrich, A. Hogan and S. Decker, YARS2: A Federated Repository for Querying Graph Structured Data from the Web, in: *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, K. Aberer, K. Choi, N.F. Noy, D. Allemang, K. Lee, L.J.B. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber and P. Cudré-Mauroux, eds, Lecture Notes in Computer Science, Vol. 4825, Springer, 2007, pp. 211–224. doi:10.1007/978-3-540-76298-0_16.

[48] S. Gurajada, S. Seufert, I. Miliaraki and M. Theobald, TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing, in: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C.E. Dyreson, F. Li and M.T. Özsu, eds, ACM, 2014, pp. 289–300. doi:10.1145/2588555.2610511.

[49] R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim and M. Sahli, Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning, *VLDB J.* **25**(3) (2016), 355–380. doi:10.1007/s00778-016-0420-y.

[50] D. Janke, S. Staab and M. Thimm, Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores, in: *Joint Proceedings of BLINK2017: 2nd International Workshop on Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 21st - to - 22nd, 2017*, R. Usbeck, A.N. Ngomo, J. Kim, K. Choi, P. Cimiano, I. Fundulaki and A. Krithara, eds, CEUR Workshop Proceedings, Vol. 1932, CEUR-WS.org, 2017. https://ceur-ws.org/Vol-1932/paper-05.pdf.

[51] A. Harth, CumulusRDF: Linked Data Management on Nested Key-Value Stores, 2011.

[52] K. Lee and L. Liu, Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning, *Proc. VLDB Endow.* **6**(14) (2013), 1894–1905. doi:10.14778/2556549.2556571. http://www.vldb.org/pvldb/vol6/p1894-lee.pdf.

[53] G. Aluç, M.T. Özsu, K.S. Daudjee and O. Hartig, chameleon-db: a Workload-Aware Robust RDF Data Management System, 2013.
[54] K. Hose and R. Schenkel, WARP: Workload-aware replication and partitioning for RDF, in: *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C.Y. Chan, J. Lu, K. Nørvåg and E. Tanin, eds, IEEE Computer Society, 2013, pp. 1–6. doi:10.1109/ICDEW.2013.6547414.
[55] A. Madkour, A.M. Aly and W.G. Aref, WORQ: Workload-Driven RDF Query Processing, in: *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I*, D. Vrandecic, K. Bontcheva, M.C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L. Kaffee and E. Simperl, eds, Lecture Notes in Computer Science, Vol. 11136, Springer, 2018, pp. 583–599. doi:10.1007/978-3-030-00671-6_34.
[56] X. Zhang, L. Chen, Y. Tong and M. Wang, EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud, in: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C.S. Jensen, C.M. Jermaine and X. Zhou, eds, IEEE Computer Society, 2013, pp. 565–576. doi:10.1109/ICDE.2013.6544856.
[57] J. Huang, D.J. Abadi and K. Ren, Scalable SPARQL Querying of Large RDF Graphs, *Proc. VLDB Endow.* **4**(11) (2011), 1123–1134. http://www.vldb.org/pvldb/vol4/p1123-huang.pdf.
[58] Z. Kaoudi and I. Manolescu, RDF in the clouds: a survey, *VLDB J.* **24**(1) (2015), 67–91. doi:10.1007/s00778-014-0364-z.
[59] I. Abdelaziz, R. Harbi, Z. Khayyat and P. Kalnis, A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data, *Proc. VLDB Endow.* **10**(13) (2017), 2049–2060. doi:10.14778/3151106.3151109. http://www.vldb.org/pvldb/vol10/p2049-abdelaziz.pdf.
[60] W. Ali, M. Saleem, B. Yao, A. Hogan and A.N. Ngomo, A survey of RDF stores & SPARQL engines for querying knowledge graphs, *VLDB J.* **31**(3) (2022), 1–26. doi:10.1007/s00778-021-00711-3.
[61] D.J. Abadi, A. Marcus, S. Madden and K.J. Hollenbach, Scalable Semantic Web Data Management Using Vertical Partitioning, in: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, C. Koch, J. Gehrke, M.N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C.Y. Chan, V. Ganti, C. Kanne, W. Klas and E.J. Neuhold, eds, ACM, 2007, pp. 411–422. http://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf.
[62] A. Hogan, Canonical Forms for Isomorphic and Equivalent RDF Graphs: Algorithms for Leaning and Labelling Blank Nodes, *ACM Trans. Web* **11**(4) (2017). doi:10.1145/3068333.
[63] A. Bonifati, W. Martens and T. Timm, An analytical study of large SPARQL query logs, *VLDB J.* **29**(2–3) (2020), 655–679. doi:10.1007/s00778-019-00558-9.
[64] M. Martin, J. Unbehauen and S. Auer, Improving the Performance of Semantic Web Applications with SPARQL Query Caching, in: *The Semantic Web: Research and Applications*, L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral and T. Tudorache, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 304–318. ISBN 978-3-642-13489-0.
[65] A. Bonifati, W. Martens and T. Timm, Navigating the Maze of Wikidata Query Logs, in: *The World Wide Web Conference*, WWW '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 127–138–. ISBN 9781450366748. doi:10.1145/3308558.3313472.
[66] G. Karypis and V. Kumar, A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, *SIAM J. Sci. Comput.* **20**(1) (1998), 359–392. doi:10.1137/S1064827595287997.
[67] J.D. Fernández, M.A. Martínez-Prieto, P. de la Fuente Redondo and C. Gutiérrez, Characterising RDF Data Sets, *J. Inf. Sci.* **44**(2) (2018), 203–229–. doi:10.1177/0165551516677945.
[68] A. Hernández-Illera, M.A. Martínez-Prieto and J.D. Fernández, Serializing RDF in Compressed Space, in: *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, A. Bilgin, M.W. Marcellin, J. Serra-Sagristà and J.A. Storer, eds, IEEE, 2015, pp. 363–372. doi:10.1109/DCC.2015.16.
[69] M. Meimaris, G. Papastefanatos, N. Mamoulis and I. Anagnostopoulos, Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization, in: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, IEEE Computer Society, 2017, pp. 497–508. doi:10.1109/ICDE.2017.106.
[70] L.M. Haas, D. Kossmann, E.L. Wimmers and J. Yang, Optimizing Queries Across Diverse Data Sources, in: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos and M.A. Jeusfeld, eds, Morgan Kaufmann, 1997, pp. 276–285. http://www.vldb.org/conf/1997/P276.PDF.
[71] G. Aluç, O. Hartig, M.T. Özsu and K. Daudjee, Diversified Stress Testing of RDF Data Management Systems, in: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C.A. Knoblock, D. Vrandecic, P. Groth, N.F. Noy, K. Janowicz and C.A. Goble, eds, Lecture Notes in Computer Science, Vol. 8796, Springer, 2014, pp. 197–212. doi:10.1007/978-3-319-11964-9_13.
[72] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P.N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer and C. Bizer, DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia, *Semantic Web* **6**(2) (2015), 167–195. doi:10.3233/SW-140134.
[73] C. Fellbaum, *WordNet: An Electronic Lexical Database*, Bradford Books, 1998.
[74] J. Hoffart, F.M. Suchanek, K. Berberich and G. Weikum, YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia, *Artif. Intell.* **194** (2013), 28–61. doi:10.1016/j.artint.2012.06.001.
[75] M. Ley, The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives, in: *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*, A.H.F. Laender and A.L. Oliveira, eds, Lecture Notes in Computer Science, Vol. 2476, Springer, 2002, pp. 1–10. doi:10.1007/3-540-45735-6_1.

[76] M. Saleem, Q. Mehmood and A.N. Ngomo, FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework, in: *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, M. Arenas, Ó. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan and S. Staab, eds, Lecture Notes in Computer Science, Vol. 9366, Springer, 2015, pp. 52–69. doi:10.1007/978-3-319-25007-6_4.

[77] L. Rietveld, R. Hoekstra, S. Schlobach and C. Guéret, Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling, in: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C.A. Knoblock, D. Vrandecic, P. Groth, N.F. Noy, K. Janowicz and C.A. Goble, eds, Lecture Notes in Computer Science, Vol. 8797, Springer, 2014, pp. 81–96. doi:10.1007/978-3-319-11915-1_6.

[78] L. Heling and M. Acosta, Robust query processing for linked data fragments, *Semantic Web* **13**(4) (2022), 623–657. doi:10.3233/SW-212888.

[79] L. Heling and M. Acosta, Characteristic sets profile features: Estimation and application to SPARQL query planning, *Semantic Web* **14**(3) (2023), 491–526. doi:10.3233/SW-222903.

[80] L. Heling and M. Acosta, Federated SPARQL Query Processing over Heterogeneous Linked Data Fragments, in: *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, F. Laforest, R. Troncy, E. Simperl, D. Agarwal, A. Gionis, I. Herman and L. Médini, eds, ACM, 2022, pp. 1047–1057. doi:10.1145/3485447.3511947.

[81] M. Schmidt, M. Meier and G. Lausen, Foundations of SPARQL query optimization, in: *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, L. Segoufin, ed., ACM International Conference Proceeding Series, ACM, 2010, pp. 4–33. doi:10.1145/1804669.1804675.

## Appendix A. Proof of smart-KG$^+$ Correctness

PROPOSITION 1. The result of evaluating a BGP $Q$ over an RDF graph $G$ with `smart-KG+`, denoted *smart-eval*$(Q,G)$, is correct with respect to the SPARQL query language semantics, i.e., *smart-eval*$(Q,G) = [\![Q]\!]_G$.

*Proof.* For this proof, we first show that the `smart-KG+` query decomposer and planner are correct. By construction, the query decomposer is correct, as the combination of the star-shaped queries $Q_s$ corresponds to the original $Q$ (cf. Eq. 25). Furthermore, the tasks of the query planner are two-fold. First, the optimizer devises an ordering of the star-shaped queries $Q_i$ and the triple patterns within $Q_i$ (Alg. 2, lines 3–7). This first task ensures that the plans are correct since the join operator is commutative and associative in the SPARQL algebra [81]. The second task is to partition $Q_i$ into subsets $Q_i'$ and $Q_i''$ to be evaluated using the TPF or the SKG APIs (Alg. 2, lines 8–13). In the second task, it is easy to see that $Q_i = Q_i' \cup Q_i''$ and that $Q_i' \cap Q_i'' = \emptyset$, i.e., all triple patterns of the star-shaped query $Q_i$ are evaluated once either using the TPF or SKG APIs. Lastly, since all stars in the input decomposition $\mathcal{Q}$ are processed in Alg. 2, the produced plans are correct.

Now we proceed to show that the execution of plans (cf. Alg. 4) is also correct. For this proof, we assume that the server operators (i.e., the SKG API and the LDF API) are implemented correctly. By contradiction, let us assume that *smart-eval*$(Q,G) \neq [\![Q]\!]_G$. We distinguish three cases based on the shipping strategy used for evaluating $Q$.

**(i) $Q$ is evaluated with Partition Shipping.** For this case, we assume the correct implementation of the join operator. Therefore, by induction on the structure of the query, it is sufficient to prove this case when $Q$ is composed of a pair of triple patterns each with variable-free-predicates $p_1, p_2 \in P'_G$. With partition shipping, the evaluation of $Q$ is carried out against the set of corresponding partitions obtained with $SKG(Q,\emptyset)$. After applying the server operators, the query executor obtains the set of relevant partitions $G^*$ from $\mathcal{G}_{serv}$ (Eq. 24), i.e, $G^* \subseteq \mathcal{G}_{serv}$ for $Q$ (Alg. 4, line 2). Next, we consider two sub-cases. In the first sub-case, we have that *smart-eval*$(Q,G) \subset [\![Q]\!]_G$, i.e., there exists an RDF triple $t \in G$ with predicate $p_1$ (resp. $p_2$) such that $t \notin \bigcup_{G_j \in G^*} G_j$. Therefore, the partitions in $\mathcal{G}_{serv}$ are created incorrectly, which contradicts Equation 24. The sub-case $[\![Q]\!]_G \subset$ *smart-eval*$(Q,G)$ does not occur even in the case that $F(Q)$ is a subset of the predicates covered by $G_j$, as the executor performs triple pattern matching over each partition (Alg. 4, line 5) to get exact matches.

**(ii) $Q$ is evaluated with triple pattern shipping.** For this case, the evaluation of $Q$ is carried out as $TPF(Q,\Omega)$ and $Q$ corresponds to a single triple pattern (which is ensured by the query optimizer). Note that $\Omega$ can also be $\emptyset$ when there are no other intermediate results). By hypothesis, $TPF(Q,\Omega)$ does not produce $[\![Q]\!]_G$, which contradicts the definition of the *TPF* server operator.

**(iii) $Q$ is evaluated following a hybrid shipping**. This proof follows from the correctness of the query decomposer and optimizer, the cases (i) and (ii). Without loss of generality, assume that $Q$ is composed of two subqueries $Q_1$ and $Q_2$ evaluated using the APIs, and $Q_2$ is evaluated using the TPF API. From cases (i) or (ii), it follows that *smart-eval*$(Q_1, G)$ is correct and produces the intermediate results $\Omega$. Then, the executor proceeds with the evaluation of $Q_2$ with intermediate results $\Omega$ as $TPF(Q_2, \Omega)$; from case (ii), it follows that *smart-eval*$(Q_2, G)$ is correct. Therefore, we conclude that *smart-eval*$(Q, G)$ is also correct. □

## Appendix B. Family partitioning on real-world KGs

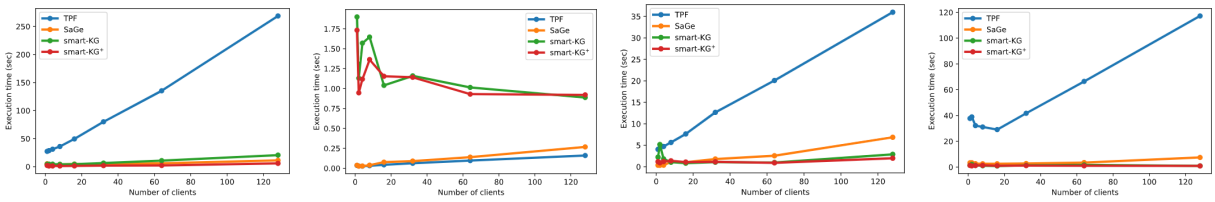
In Table 12, we present additional real-world Knowledge Graphs (KGs) partitioned using family-based techniques. Freebase and Yago2 follow the parametrization of DBpedia due to their similar characteristics (see Table 5). DBLP and WordNet use the same setup as WatDiv due to their comparable characteristics.

Table 12
Characteristics of the evaluated knowledge graphs

| RDF Graph $G$ | # triples $\|G\|$ | # subjects $\|S_G\|$ | # predicates $\|P_G\|$ | # objects $\|O_G\|$ | $\|P'_G\|$ | $\|P'_{core}\|$ | $\|F'_{core}\|$ | $\|\mathcal{G}_{serv}\|$ | C.Time (h) |
|---|---|---|---|---|---|---|---|---|---|
| Freebase | 2,067,068,155 | 102,001,451 | 770,415 | 438,832,462 | 530 | 171 | 479 | 11979 | 18 |
| Yago2 | 158,991,568 | 67,813,972 | 104 | 22,354,760 | 35 | 19 | 65 | 638 | 5 |
| DBLP | 88,150,324 | 5,125,936 | 27 | 36,413,780 | 27 | 27 | 270 | 990 | 3 |
| WordNet | 5,558,748 | 647,215 | 64 | 2,483,030 | 64 | 64 | 777 | 1156 | 0.5 |

## Appendix C. Detailed Performance evaluation on different query shapes

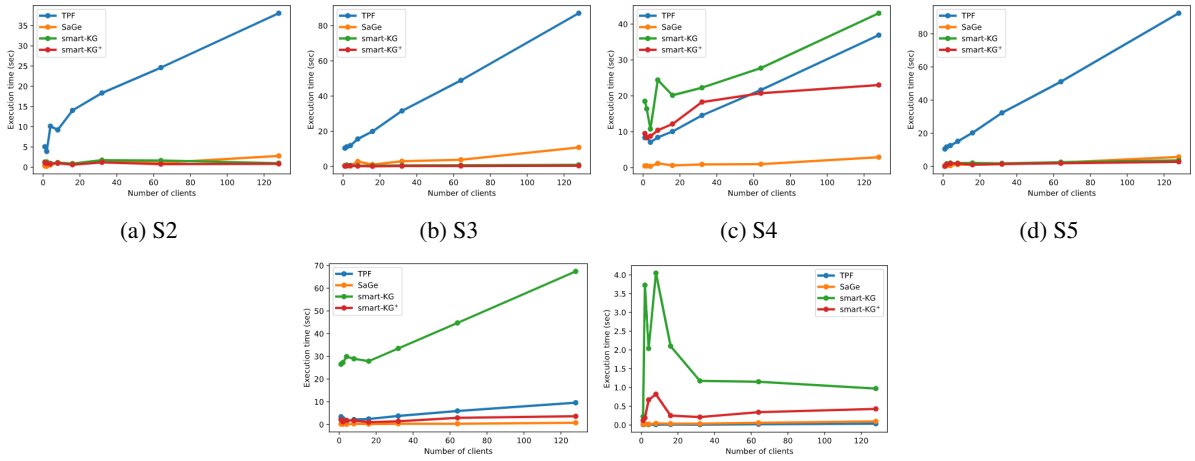
This appendix contains detailed experimental results of the compared systems on different query shapes extracted from the WatDiv Basic Testing [71]. Fig. 14 shows the performance in the simplest L-queries of the different systems on WatDiv-100M. Similar to our previous results, `smart-KG+` reports a stable query execution time, which ranges between 1-5 seconds. `smart-KG+` performs better than the original `smart-KG` due to the asynchronous pipeline of iterators executing first the most selective iterator. As expected, SaGe provides excellent performance in L queries (i.e. simple queries), with the best performance in the L3 query with an average execution time of less than 1 second. The main reason is that SaGe server in the case of L queries acts as a SPARQL endpoint since it requires a single request to process L query. Finally, TPF is the slowest approach in L2, L4, and L5 queries, while it excels in L1 and L3 with up to 40 clients since the queries are very selective and do not require pagination.




(a) L2 (b) L3 (c) L4 (d) L5

Fig. 14. Avg. execution time per client on the standard WatDiv-100M, for simplest L queries

Fig. 15 shows the query execution time of S-queries. `smart-KG+` provides a more efficient performance than `smart-KG`, since in this case, `smart-KG+` server query planner generates far more accurate triple pattern ordering than `smart-KG`, relying on pre-computed cardinality estimations (i.e. characteristics sets) stored on the server-side. SaGe maintains a solid performance in S-queries (very selective) requiring less time on the server. As shown in Fig. 16, SaGe provides the best execution time for F queries (i.e snowflake queries). `smart-KG` has on average a slow query execution time in F queries (i.e snowflake queries) since snowflake queries require a join operation between the shipped stars which are typically connected with a non-selective single triple pattern evaluated by a

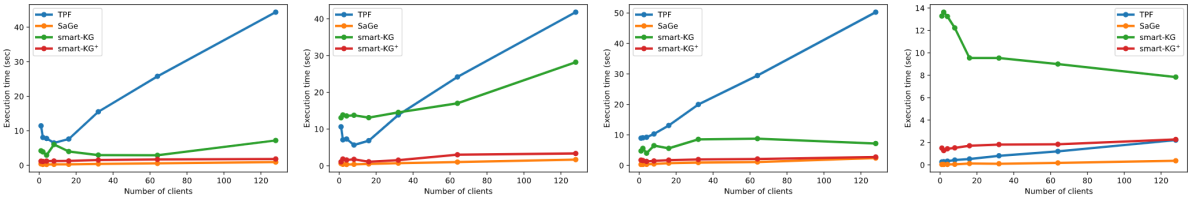



(a) S2 (b) S3 (c) S4 (d) S5

(e) S6 (f) S7

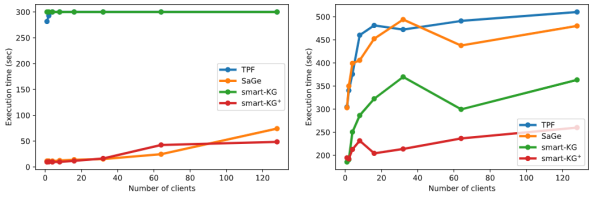
Fig. 15. Avg. execution time per client on the standard WatDiv-100M, for Star S queries

high number of TPF requests. However, `smart-KG+` significantly outperforms TPF and `smart-KG` thanks to the accurate server-side query planning. Finally, Fig. 17 shows the overall execution times for the C-queries workload on (WatDiv-100M, 80 clients, 5min timeout). TPF is again the slowest solution, while `smart-KG+` significantly outperforms all the compared systems. For instance, `smart-KG` timeouts at C2 since the query includes 3 stars and 3 single triple patterns with high cardinalities causing a tremendous number of TPF requests. `smart-KG+` avoids the large intermediate results by better subqueries reordering. For C3 (unbounded star query), `smart-KG` and `smart-KG+` provide the best performance since they are optimized for star queries. In contrast, SaGe suffers from additional delays in case of complex queries to maintain the fair resources allocation policy.




(a) F2 (b) F3 (c) F4 (d) F5

Fig. 16. Avg. execution time per client on the standard WatDiv-100M, for Snowflake F queries




(a) C2 (b) C3

Fig. 17. Avg. execution time per client on the standard WatDiv-100M, for Complex C queries