SMART-KG: Hybrid Shipping for SPARQL Querying on the Web

Amr Azzam
Vienna University of Economics and Business
amr.azzam@wu.ac.at

Javier D. Fernández
Vienna University of Economics and Business
jfernand@wu.ac.at

Maribel Acosta
Karlsruhe Institute of Technology
maribel.acosta@kit.edu

Martin Beno
Vienna University of Economics and Business
martin.beno@wu.ac.at

Axel Polleres
Vienna University of Economics and Business, Complexity Science Hub Vienna
axel.polleres@wu.ac.at

ABSTRACT

While Linked Data (LD) provides standards for publishing (RDF) and (SPARQL) querying Knowledge Graphs (KGs) on the Web, serving, accessing and processing such open, decentralized KGs is often practically impossible, as query timeouts on publicly available SPARQL endpoints show. Alternative solutions such as Triple Pattern Fragments (TPF) attempt to tackle the problem of availability by pushing query processing workload to the client side, but suffer from unnecessary transfer of irrelevant data on complex queries with large intermediate results. In this paper we present `smart-KG`, a novel approach to share the load between servers and clients, while significantly reducing data transfer volume, by combining TPF with shipping compressed KG partitions. Our evaluations show that `smart-KG` outperforms state-of-the-art client-side solutions and increases server-side availability towards more cost-effective and balanced hosting of open and decentralized KGs.

ACM Reference Format:
Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. 2020. SMART-KG: Hybrid Shipping for SPARQL Querying on the Web. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3366423.3380177

1 INTRODUCTION

Knowledge Graphs (KGs) have emerged as a promising data management foundation to provide scalable knowledge models that represent facts about entities as well as relations among these [13]. The adoption of the KG concept offers the potential for building innovative products and services that create new value in terms of commercial applications by the likes of Google, Microsoft and Bloomberg, to name but a few. Also, specific domains and initiatives, for instance, in biomedicine already make extensive use of KGs for the integration of diverse datasets in fields such as neurosciences, cancer research and drug discovery [29].

Openly available examples of interlinked KGs include DBpedia, Yago, and Wikidata, and indeed many openly available KGs are published now following the Linked Data [12] principles, using the semi-structured RDF data model and supporting query access through the SPARQL query language. However, there are still serious barriers to consume and use open RDF KGs published on the web. Indeed, concurrently querying highly demanded RDF graphs [22] such as DBpedia with multiple clients still imposes a significant bottleneck: each RDF graph is, at best, exposing its own SPARQL endpoint, but while RDF stores offer server-side solutions that have good performance in single queries, they are expensive to host and hard to maintain when large KGs are served or concurrent execution of complex queries is allowed to multiple users [15, 40]. This leads to well-known problems of Linked Data availability and resource limits [38]. As an example, SPARQLES [42], a service monitoring 565 SPARQL endpoints, shows that 64% of them are unavailable (as of October 2019). To mitigate the shortcoming of SPARQL endpoints, solutions to shift the server workload to the clients have been proposed [25, 43]. These solutions allow for cost-effective hosting of large KGs by performing most of the query processing at the client, however, at the cost of significant performance degradation for SPARQL queries that require the evaluation of several operators, and potentially shipping a large number of intermediate results that do not contribute to the final query answer.

In order to address these current limitations, we propose an approach relying on – rather than querying monolithic graphs – serving compressed partitions for KGs in a modular fashion, reducing drastically the need for redundant data transfers in shared server- and client-side query processing. In particular, we propose a novel client-server query paradigm, `smart-KG`, that aims at increasing server availability while achieving competitive performance. In our approach, smart servers maintain compressed and queryable graph partitions, that is, KG "slices" that can be shipped, cached and be locally queried by smart clients. We propose a graph partitioning technique based on Characteristic Sets [21, 37], that exploits the structure of RDF graphs to group entities described with the same sets of predicates. Furthermore, the smart clients implement query optimization and execution techniques to handle combinations of KG partitions and intermediate results of triple queries issued directly to the server, to evaluate SPARQL queries.

Our main contributions are:

- A novel paradigm, `smart-KG`, to distribute the evaluation of SPARQL queries among clients and servers by leveraging the transfer of compressed KG partitions.

WWW '20, April 20–24, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7023-3/20/04.
https://doi.org/10.1145/3366423.3380177

- A KG partition technique designed for graphs with skewed predicate distributions to trade-off the number of partitions to be maintained and transferred.
- Client-side query optimization and execution techniques that combine KG partition retrieval and intermediate results that ensure correct query evaluation.
- An empirical evaluation of `smart-KG` on synthetic and real-world KGs and queries, significantly outperforming state-of-the art on server- and client-side SPARQL query processing.

The remainder of this paper is organized as follows. In Section 2, we analyse the related work. Section 3 introduces preliminaries concepts used in this work. We present our `smart-KG` solution in Section 4. An empirical evaluation and results are discussed in Section 5. In Section 6, we conclude and outline future work.

2 RELATED WORK

The execution of SPARQL queries over remote KGs typically rely on architectures of clients (consuming SPARQL queries) and servers (exposing RDF graphs via SPARQL). In client-server environments, query workload distribution has been classified into three main types of shipping strategies [20]. *Query shipping* consists in processing query execution completely at the server and shipping only results back to the client. *Data shipping* exploits the processing capacity of clients (in the extreme case, meaning to simply serve dataset dumps for download) and thereby reduce the workload of servers. Finally, in *hybrid shipping* strategies, the execution of sub-queries and operators is distributed among clients and servers according to, e.g., the complexity of the queries and the server workload. In the following, we analyse existing solutions for processing SPARQL queries based on the client-server strategy implemented.

Query Shipping: SPARQL Endpoints. RDF KGs are traditionally exposed via SPARQL endpoints, i.e., APIs that serve SPARQL queries over the HTTP protocol [17]. Note that SPARQL endpoints often run on top of RDF triples stores (e.g. Virtuoso [16] or Stardog [5]).

SPARQL query processing over endpoints provides relatively high performance under low loads. However, with concurrent clients and query complexity, endpoints face overloads and large delays that lead to well-known problems of low availability and poor performance. Thus, most SPARQL endpoints turn into resource-hungry services, too costly to host and maintain for many potential data providers. Latest studies on public SPARQL endpoints [15, 38] confirm these issues and show that at least half of the endpoints do not answer at all, while others impose significant restrictions such as refusing complex queries or limiting result sizes [11].

Data Shipping: Client-side SPARQL. To overcome the low availability problem of SPARQL endpoints, different client-side solutions have been proposed [25, 26]. These approaches perform query execution by traversing the KG structure, dereferencing URI-identified entities appearing within the query and assuming that these – following the Linked Data Principles – allow to retrieve RDF data that can be processed locally. Unfortunately, for most non-trivial queries (e.g. patterns only including textual literals) an evaluation on the web is unfeasible [28]. Likewise, there are no established best practices on which parts of an enclosing dataset should be downloadable by dereferencing URIs, ranging from non-dereferenceable URIs in many datasets to having to download full KG dumps.

Hybrid Shipping. To overcome the deficiencies of query and data shipping, different hybrid shipping approaches for SPARQL query evaluation have been proposed.

Triple Pattern Fragment (TPF) [43] servers only support the evaluation of triple patterns. Then, TPF clients [8, 43] retrieve the intermediate results (typically paginated) of each triple pattern in the query and join them to compute the final query results. While experimental results [30, 43] show that TPF achieves higher server availability than traditional server-centric SPARQL endpoints, this typically comes at the expense of a significant increase in the network traffic due to considerable overheads from HTTP requests and data transfers. In particular, non-selective queries can be penalized by a high number of irrelevant intermediate results transferred, and costly client-side join operations. In contrast to TPF, our proposed approach is able to reduce the amount of data transferred from the server to the client which, in turn, speeds up query execution.

More recently, different approaches inspect a more balanced client-server load distribution. Bindings-Restricted Triple Pattern Fragments [27] (brTPF) gives a slight boost to the performance of TPF by attaching intermediate results to triple pattern requests along with distributing the join between the client and the server using the bind join strategy. Thus, brTPF reduces the number of HTTP requests and data received with respect to the original TPF solution [27]. However, the number of requests is still relatively high in addition to the attached intermediate results, combined with the need to transfer these intermediate results verbatim.

Finally, SaGe [36] is a SPARQL query engine tailored to address the undesirable starvation of simple queries waiting for long running queries to release the server resources. To this aim, SaGe proposes a preemptive execution model. SaGe introduces a scheduling mechanism that allocates a fixed time executing quantum. Once a query is executed for that given time period, the query is suspended and another query in processed. To resume queries, SaGe ships to the client the state of the query execution. Experiments show that the preemption mechanism enhanced average workload completion time per client. Yet, SaGe faces excessive number of requests, query context switching and potential client-side overheads.

Our approach, `smart-KG`, proposes a novel hybrid strategy to provide a more balanced client/server distribution. As opposed to prior solutions such as TPF, `smart-KG` does not rely solely on shipping intermediate results, but rather on shipping modular, query-relevant partitions of a KG that can be directly queried locally by a client.

3 PRELIMINARIES

RDF and SPARQL. The Resource Description Framework (RDF) [41] is a graph-based data model to represent information about resources and their relationships in the form of triples (*subject*, *predicate*, *object*) $\in (I \cup B) \times I \times (I \cup B \cup L)$, where I, B, L are infinite, mutually disjoint sets of IRIs, blank nodes, and literals, respectively [23]. A finite set of such triples G is called an *RDF graph*. RDF graphs can be queried using the SPARQL [24] query language, which is based on graph pattern matching. The core query atom of SPARQL is a *triple pattern tp* from $(I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ where apart from RDF terms also variables from a set V of variables, disjoint from the aforementioned I, B and L, are allowed. Basic Graph Patterns (BGPs), i.e. sets of triple patterns, can

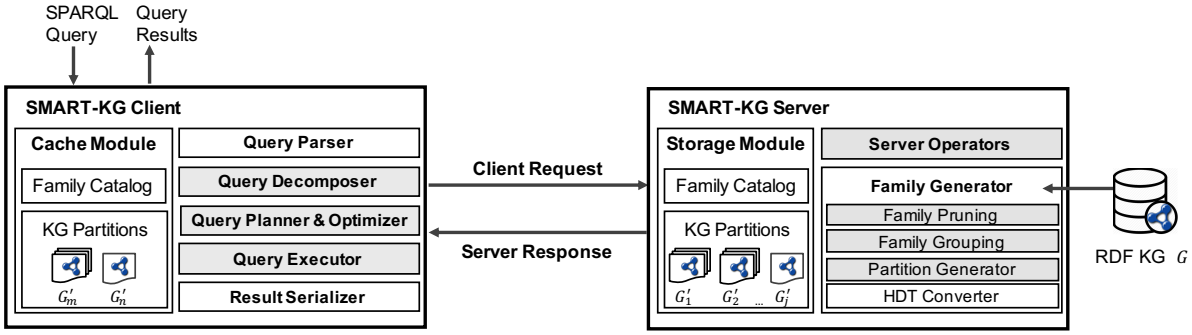


Figure 1: Overall architecture for the `smart-KG` client and server.

also be understood as conjunctive queries over G. The evaluation of a BGP Q over an RDF graph G, denoted $[\![Q]\!]_G = \Omega$, is defined as a set of solution mappings, s.t. each $\omega \in \Omega$ denotes a substitution from the variables in Q ($var(Q)$) to $I \cup B \cup L$ such that $\omega(Q) \subseteq G$. For the purposes of this paper, we focus on BGPs and leave out other patterns, which all build on top of BGPs as SPARQL's core retrieval functionality [14].

Family-Based Partitioning of RDF Graphs. RDF is a semi-structured data model which typically does not prescribe a fixed schema. In theory, this can lead to RDF graphs, where the set of predicates used to describe subjects and their relationships may vary greatly. However, in real-world RDF graphs, there typically is an inherent structure as there exist repetitions whenever subjects of the same kind are described in the same way. For instance, predicates for *songs* (e.g. duration, album, etc.) are different than those used to describe *persons* (e.g. birthday, nationality, etc.) in DBpedia, and the same combinations of predicates are shared across many subjects of the same type. Neumann and Moerkotte [21, 37] capture these structures with the notion of *characteristic sets*, also called *predicate families* [18] (or just families hereinafter). Let G be an RDF graph, and S_G, P_G, O_G be the sets of subjects, predicates and objects in G respectively. We define the *(predicate) family*, of a subject s, $F(s)$ as the set of predicates related to the subject s, that is:

$$F(s) = \{p \mid \exists o \in O_G : (s, p, o) \in G\} \quad (1)$$

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

$$F(G) = \{F(x) \mid x \in S_G\} \quad (2)$$

For simplicity, we name the different families in G as $F_1, F_2, ..., F_m$, where $m = |F(G)|$. In this paper, we use families as a means to define a graph partition, i.e., we consider – as the basis for our approach – a disjoint set of partitions that is a cover[1] $\mathcal{G} = \{G_1, G_2, \cdots, G_m\}$ of G based on its families, where each partition G_i is defined as:

$$G_i = \{(s, p, o) \in G \mid F(s) = F_i\} \quad (3)$$

Families provide structure-based means of partitioning an RDF graph used for (i) join and cardinality estimation [21, 37] for SPARQL optimization, (ii) RDF compression [31], and (iii) building indexes[2] to speed up SPARQL queries [35]. To the best of our knowledge, this paper is the first work using families in shipping strategies.

[1] i.e, $G = G_1 \cup G_2 \cup \cdots \cup G_m$, where $\forall i, j\ G_i \cap G_j = \emptyset$

[2] Meimaris et al. [35] extended the notion of characteristic sets also to object nodes

RDF HDT Compression. HDT [19] is a well-known compressed format for RDF graphs, which permits efficient triple pattern retrieval over the compressed data. HDT has three main components: (i) the *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. Both the HDT dictionary and triples are self-indexed to support efficient retrieval operations. The dictionary implements prefix-based Front-Coding compression [34], 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 [19].

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 [33]. 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.

4 SMART-KG: DESIGN AND OVERVIEW

`smart-KG` (cf. Fig. 1) defines client and server operations to combine the shipping of partitions based on RDF families with the shipping of the results of evaluating triple patterns to reduce query runtime.

Smart-KG servers generate families and corresponding partitions of a given knowledge graph (KG). The resulting KG partitions are materialized (in HDT) in the storage module along with a family catalog that contains metadata about the structure of the partitions. In addition, `smart-KG` servers also support operators to execute triple pattern queries and to transfer partitions to `smart-KG` clients.

Smart-KG clients are able to execute SPARQL queries by devising query plans that combine the shipping of triple pattern results and partitions. The query decomposition, planning, and optimization techniques implemented by the `smart-KG` client exploit the structure of KG partitions to reduce query execution time.

4.1 SMART-KG Server

The `smart-KG` server, upon loading an RDF graph, supports access to graph partitions and the evaluation of triple patterns using TPF.

To this end, the server implements a partition generator taking into consideration the families from the graph plus retrieval operations.

4.1.1 Partition Generator. The `smart-KG` server, upon loading a graph G, could decompose it into partitions $G_1, \dots G_m$ per family, as described in Eq. (3) and convert those partitions to HDT. In practice, however, the number of partitions can be relatively large for highly semi-structured RDF graphs. Thus, we introduce the concept of *predicate-restricted families*, where some particular predicates are not considered for the creation of families.

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

$$F'(s) = \{p' \in P'_G \mid \exists o \in O_G : (s, p', o) \in G\} \tag{4}$$

Analogously, we denote as $F'(G) = \{F'_1, F'_2, \cdots, F'_m\}$, or just F', to the set of different predicate-restricted families for G w.r.t. P_G, where $m' = |F'(G)|$. These families correspond to a set $\mathcal{G}' = \{G'_1, G'_2, \dots, G'_m\}$ of disjoint partitions of a subgraph of G based on the P'_G-restricted families, with

$$G'_i = \{(s, p, o) \in G \mid F'(s) = F'_i\} \tag{5}$$

Note that, however $\mathcal{G}'$ is no longer a complete cover of G, but the graph $G' = \bigcup G'_i$ only contains the "projection" of G to P'_G. Serving predicate restricted families allows a `smart-KG` publisher to select P'_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 based on the cardinality of predicates in Section 4.1.3.

4.1.2 Family Grouping. Although the use of restricted families can control the number of generated families and avoid generating rarely used families to some extent, the number and volume of partitions are still determined further by other distribution features of the data. 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 attributes may vary across entities of the same type.

Thus, besides using predicate-restricted families, our partition shipping strategy further drastically benefits from merging (i.e. grouping) similar families into a single partition. For instance, all disjoint families containing a certain set of predicates e.g. F_1={`foaf:name`, `foaf:age`, `dct:title`} and F_2={`foaf:name`, `foaf:age`, `dbp:birthdate`} can be merged into a single family $F_{\{1,2\}}$={`foaf:name`, `foaf:age`}. The intuition behind merging such families covering overlapping predicates is that these overlapping predicate subsets may also occur as predicate families in query patterns more commonly. Therefore, instead of shipping the union of partitions contributing to a query, only the partition corresponding to the smallest merged families needs to be shipped.

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[3]:

$$F'_I = \bigcap\nolimits_{i \in I} F'_i \tag{6}$$

[3]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 $I \in 2^{\{1,\cdots,m\}}$

```
1  Initialize μ with the original families:
2  foreach f ∈ F'(G) do
3  |   μ(F'_i) ← {i}
4  repeat
5  |   μ'(·) ← μ(·)
6  |   foreach f ∈ dom(μ) do
7  |   |   foreach g ∈ dom(μ) do
8  |   |   |   if g ∩ f ≠ ∅ then
9  |   |   |   |   if g ∩ f ∈ dom(μ) then
10 |   |   |   |   |   μ(g ∩ f) ← μ(g ∩ f) ∪ μ(g) ∪ μ(f)
11 |   |   |   |   else
12 |   |   |   |   |   μ(g ∩ f) ← μ(g) ∪ μ(f)
13 until μ ≠ μ';
14 return μ
```

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

$$G'_I = \{(s, p, o) \in G \mid F(s) \subseteq F(I)\} \tag{7}$$

Following the example, if G'_1 and G'_2 are grouped into $G'_{\{1,2\}}$, then for evaluating a query that requires the predicates `foaf:name` and `foaf:age`, only $G'_{\{1,2\}}$ needs to be shipped, instead of $G'_1 \cup G'_2$.

Following similar premises, Gubichev and Neumann [21] 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:age`}, $F^2_{\{1,2\}}$={`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 was used to estimate join costs [21]) 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 this merged families map for all potentially non-empty intersections of sub-families, we start from $F'(G) = \{F'_1, \dots F'_m\}$, and iteratively construct a partial map μ 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 μ with $F'(G)$ (lines 2–4), and then, iteratively, until μ does not change any more (lines 5–13), 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 μ (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:

$$\mathcal{G}_{serv} = \{G'_{\mu(f)} \mid f \in dom(\mu)\} \tag{8}$$

Note that the partitions in $\mathcal{G}_{serv}$ are no longer non-overlapping. However, 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 μ.

4.1.3 Family Pruning. Note that, in practice, it might be too expensive to materialise partitions for *all* potential merges (intersections) of all families in G. 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 materialised partitions, where we (i) restrict considered predicates in P'_G based on their cardinality, (ii) avoid creation of small families that deviate only slightly from other overlapping, "core" families, and (iii) avoid materialisation of families over a certain size.

(i) Restrict predicates based on cardinality. The cardinality of predicates can play a fundamental role in the number and size of shipped partitions: firstly, infrequent predicates can be scattered through many different families but, at the same time, the overall size is limited and a TPF call for just triples with these infrequent predicates can easily bring all the information related to the predicate without the need to transfer large intermediate results; secondly, frequent predicates (e.g. `dbo:wikiPageExternalLink` in DBpedia) can potentially belong to most families, and unnecessary overload the size of each family, if not practically queried (e.g. `dbo:wikiPageLength`) or selectivity/distribution of frequent predicates is low/skewed (e.g. `rdf:type`) which could lead to the generation and transfer of large partitions. To address these issues we use thresholds τ_l, τ_h with $0 \leq \tau_l < \tau_h \leq 1$, to delimit the minimum and maximum percentage of triples per predicate, and define P'_G accordingly based on these thresholds as follows:

$$P'_G = \{p' \in P_G \mid \tau_l \leq \frac{|(s, p', o) \in G|}{|G|} \leq \tau_h\} \tag{9}$$

Note that publishers might still consider to include particular heavy hitters (e.g. `rdf:type`) which can be frequent in queries. While we allow for this possibility, we consider as future work the application of different techniques (e.g. clustering) to select P'_G (and thus families), e.g. by considering query logs.

(ii) Avoid the creation of small families. In order to address issue (ii), 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 α_s. That is, we define these core families as

$$F'_{core} = \{F'_i \in F' \mid \frac{|S_{G'_i}|}{|S_G|} \geq \alpha_s\} \tag{10}$$

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 subgraphs where entities are described with the same attributes.

(iii) Avoid the creation of large families. Finally, we avoid the materialisation of overly large (e.g. hundreds of millions of triples in DBpedia) merged partitions G_I with size G_I above a threshold α_t, which limits the size of the materialised merged partitions.

In order to only take core families into account for the creation of partitions, and limit merged families to sizes below α_t, it is sufficient to modify Equation (8) as follows:

$$\mathcal{G}_{serv} = \left\{ G'_{\mu(f)} \;\middle|\; \begin{array}{r} f \in dom(\mu) \wedge \\ \mu(f) \cap I_{core} \neq \emptyset \wedge \\ \sum_{i \in \mu(f)} |G'_i| \leq \alpha_t \end{array} \right\} \cup \{G_{\{i\}} | F'_i \in F'\} \tag{11}$$

In Equation (11), line 2 addresses issue (ii)[4] and line 3 addresses issue (iii)[5]. The last part ensures that, despite pruning, the non-merged partitions of families in F' remain being served.

As, due to these pruning steps, no longer all of the partitions corresponding to families in $dom(\mu)$ will be materialised in $\mathcal{G}_{serv}$. In practice, we define another mapping function, μ_G, that allows to directly map families from $dom(\mu)$ to "minimal" sets of matching partitions in $\mathcal{G}_{serv}$.[6] 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) materialised matching partitions, i.e. where $\mu_G(f) = \{I \mid G'_I \in \mathcal{G}^{\prec}_{serv}(f)\}$.[7] In this case, $\mathcal{G}^{\prec}_{serv}(f)\}$ is defined as follows: let $\mathcal{G}_{serv}(f) = \{G'_I \in \mathcal{G}_{serv} \mid f \subseteq \mu^{-1}(I)\}$ be all materialised 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, the intuition is to pick the partitions that are "subset-minimal with respect to their corresponding families".

4.1.4 Server Operators. The `smart-KG` server materialises all partitions in $\mathcal{G}_{serv}$ into HDT files, and provides operators to ship partitions and their metadata based on μ_G, or to respond to TPF requests. Overall, the following operations[8] are provided:

- *TPF(tp)* to retrieve the answer for a triple pattern *tp*, i.e., the `smart-KG` server returns the triples from G that match *tp*.
- *TPFcard(tp)* to retrieve the result cardinality of a triple pattern (this is a standard TPF API function).
- retrievePartition(*id*) to retrieve a partition by *id* (we use *id*s corresponding to partitions in $\mathcal{G}_{serv}$).
- *retrieveIDs(f)* to retrieve the IDs of $\prec$-*minimal* partitions matching a given family f (i.e., $\mu_G(f)$), plus metadata with descriptive statistics per ID (e.g, number of triples).
- *getPartitionMetadata()* to retrieve the pruning parameters used by the server (i.e., P'_{core}, τ_l, τ_h, α_s, and α_t).

As for the retrieveIDs(f) operation, it essentially scans $dom(\mu_G)$ to determine the single (cardinality-wise) smallest $f' \supseteq f$ in $dom(\mu_G)$ and retrieves IDs corresponding to the index-sets $\mu_G(f')$. Note that f' is uniquely determined, which can be proven by contradiction: i.e. assume two cardinality-wise smallest $f'_1, f'_2 \in dom(\mu_G)$ with $f'_1 \neq f'_2$ and $f \subset f'_1, f \subset f'_2$; then, also $f \subset f'_1 \cap f'_2$, where (by assumption $f'_1 \neq f'_2$) it holds that $|f'_1 \cap f'_2| < |f'_1|$ or $|f'_1 \cap f'_2| < |f'_2|$. However, by construction of μ_G, this also implies that $f'_1 \cap f'_2 \in dom(\mu_G)$, which contradicts the assumption.

[4] since $S_{G'_i} \cap S_{G'_j} = \emptyset$ for all base families $F'_i, F'_j \in F'$, by construction it holds that $|S_{G'_I}| = \sum_{i \in I} |S_{G'_I}|$

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

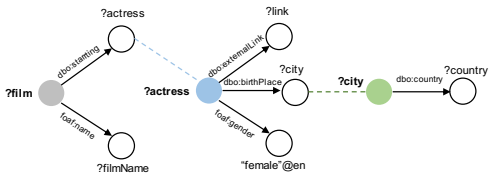
[6] In practice, Alg. 1 computes the partitions $\mathcal{G}_{serv}$ along with μ_G in one go.

[7] For $G'_{\mu(f)} \in \mathcal{G}_{serv}$, i.e., if the resp. partition is materialised, then $\mu_G(f) = \{\mu(f)\}$.

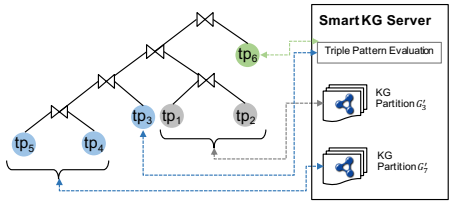
[8] We assume that a server handles a single graph. For multiple graphs, the *id* of the graph can be added as a parameter.

```
SELECT * WHERE {
?film  dbo:starring  ?actress  .          # tp1
?film  foaf:name ?filmName .              # tp2
?actress  dbo:wikiPageExternalLink  ?link  .  # tp3
?actress  dbo:birthPlace  ?city  .        # tp4
?actress  foaf:gender "female"@en .       # tp5
?city  dbo:country  ?country  . }         # tp6
```

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



(b) Star-shaped query decomposition



(c) Shipping plan based on the decomposition

Figure 2: Example of processing a SPARQL query with the `smart-KG` client.

4.2 SMART-KG Client

The `smart-KG` client (cf. Fig. 1) implements partition and triple pattern shipping to efficiently execute SPARQL queries over the `smart-KG` server. The `smart-KG` client maintains a *catalog* with metadata about the families available at a `smart-KG` server obtained with the server operation *getPartitionMetadata()*. The input of the client is a SPARQL query, which the *query parser* translates into the corresponding SPARQL algebra expressions. Then, the *query decomposer* splits the BGPs within the query into star-shaped subqueries around the same subject. Based on this decomposition, the *query optimizer* implements heuristics to determine the order of stars and triple patterns within the stars, and the shipping strategies to evaluate them. The *query executor* evaluates the plan and combines the results locally by joining the data retrieved from the server. The results produced by the engine are translated by the *results serializer* into the format specified by the user. The partitions downloaded from the `smart-KG` server during query evaluation can be stored in the *family cache* and reused for subsequent query evaluations. In the following, we will describe the main `smart-KG` client components: query decomposer, optimizer, and executor.

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

$$Q_s = \{tp \in Q \mid tp = (s, p, o)\} \tag{12}$$

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

$$F(Q_s) = \{p \mid \exists o : (s, p, o) \in Q_s, p \in I\} \tag{13}$$

Given the SPARQL query in Fig. 2a, the BGP is decomposed into $\mathcal{Q} = \{Q_{?\texttt{film}}, Q_{?\texttt{actress}}, Q_{?\texttt{city}}\}$ around the three subjects (cf. Fig. 2b). Each of the star families $F(Q_s)$ that can be mapped to existing predicate families in $dom(\mu_G)$ on the server has a non-empty answer. For example, $Q_{?\texttt{film}} = \{(?\texttt{film}, \texttt{dbo:starring}, ?\texttt{actress}), (?\texttt{film}, \texttt{foaf:name}, ?\texttt{filmName})\}$ has $F(Q_{?\texttt{film}}) = \{\texttt{dbo:starring}, \texttt{foaf:name}\}$; based on the decomposition $\mathcal{Q}$, the `smart-KG` client's shipping-based query optimizer next has to devise a query plan.

4.2.2 Shipping-based Query Planner & Optimizer. The `smart-KG` client query planner devises plans where both triple pattern results (using TPF) and partitions in $\mathcal{G}_{serv}$ are transferred from the server to resolve the sub-queries in $\mathcal{Q}$. To decide whether and for which sub-queries to use triple pattern or partition shipping, and in which order to execute them, the optimizer implements heuristics based on the sub-queries in $\mathcal{Q}$ and the server's partition metadata.

Partition Shipping (P-S). Shipping relevant partitions to evaluate a star $Q_s \in \mathcal{Q}$ needs to take into consideration the materialized partitions at the server. Since graph partitions are generated based on the core families (cf. Section 4.1), only stars with $F(Q_s) \subseteq P'_{core}$ can be fully evaluated by served partitions. Therefore, the optimizer first partitions each $Q_s \in \mathcal{Q}$ into the disjoint sets Q'_s and Q''_s, where $Q'_s = \{(s, p, o) \in Q_s \mid p \in P'_{core}\}$, i.e., the part of the star that can be evaluated over the served partitions, whereas the remaining triple patterns in $Q''_s = Q_s \setminus Q'_s$ are delegated to TPF requests.[9] Then, the optimizer implements the following additional heuristics: partition shipping is only followed if $|Q'_s| > 1$, as in practice, the transfer of graph partitions to resolve a single triple patterns usually takes longer than delegating to a TPF request directly.

Triple Pattern Shipping (TP-S). Triple patterns tp delegated to TPF will be evaluated using a *TPF(tp)* request to the server. This involves the triples patterns in Q''_s and Q'_s with $|Q'_s| = 1$.

The query optimizer, given $\mathcal{Q}$ and P'_G as input, devises a query plan Π_Q, based on the described sub-decomposition into P-S and TP-S patterns. It accordingly proceeds in two phases, first iterating over each star $Q_s \in \mathcal{Q}$ to perform the partitioning into Q'_s and Q''_s, additionally collecting the cardinality for each triple pattern $tp_i \in Q_s$ using *TPFcard()* server requests. Then, the optimizer devises sub-plans, for Q'_s and Q''_s that can be efficiently executed, by join ordering based on these cardinalities, using the construct *Plan*, that comprises a pair of a left-linearly ordered query plan, along with a shipping strategy. Join order is determined by the cardinality of triple patterns, where smaller triple patterns are evaluated first. For each sub-query Q_s, the optimizer creates a shipping-based subplan Π_s which is added to the set of current *subplans*.

Fig. 2c shows the shipping strategies for each sub-plan from our example. For the sub-query $Q_{?\texttt{actress}}$, the optimizer created $Plan((tp_5 \bowtie tp_4), P\text{-}S)$. Yet, the triple pattern tp_6 in $S_{?\texttt{actress}}$ is evaluated using triple pattern shipping as the optimizer determined that the predicate `dbo:wikiPageExternalLink` is not in P'_G.

In the second phase, the optimizer combines the sub-plans that share variables to build the final plan Π_Q. Again, the optimizer uses a heuristic to determine join order based on selecting the sub-plan Π_i containing the overall smallest (i.e., assumed most selective) triple pattern from *subplans* first, and so on, iteratively joining subplans to Π_Q. The resulting query plan Π_Q comprises sub-plans annotated with the corresponding shipping strategy, and join operators to be evaluated locally by the client.

[9] Note that Q''_s also includes triple patterns with predicate variables, i.e., $p \in V$.

Algorithm 2: Query Executor: *evalPlan*

Input: Query plan Π
Output: Ω the result set of executing Π
1 **if** $\Pi = Plan(\Pi_s$, *P-S*$)$ **then**
2 Q_s is the sub-query associated with Π_s
3 $G^* = \{getPartition(id) \mid id \in retrieveIds(F(Q_s))\}$
4 $\Omega \leftarrow \{\omega_\emptyset\}$
5 **for** $tp_i \in P_s$ **do**
6 $\Omega \leftarrow \Omega \bowtie \cup_{G_j \in G^*} [\![tp_i]\!]_{G_j}$
7 **else if** $\Pi = Plan(\Pi_s$, *TP-S*$)$ **then**
8 $\Omega \leftarrow TPF(\Pi_s)$
9 **else**
10 Π is $(\Pi_l \bowtie \Pi_r)$
11 $\Omega \leftarrow evalPlan(\Pi_l) \bowtie evalPlan(\Pi_r)$
12 **return** Ω

4.2.3 Query Executor. The function *evalPlan* evaluates the plan Π_Q by traversing the tree of sub-plans (cf. Alg. 2). The shipping strategies are implemented by calling the respective `smart-KG` server operators (cf. Sec. 4.1.4). Depending on the structure of the sub-plans, the query executor distinguishes the following cases.

Case: (P-S) Sub-plans. P-S sub-plans are evaluated (cf. Alg. 2, lines 1–6) by determining relevant served partition IDs for Q'_s, through calling $retrieveIds(F(Q'_s))$, and retrieving each ID from the server (line 3). The query executor evaluates the triple patterns tp_i against each such partition and merges the results using the SPARQL algebra union operator (line 4). The intermediate results of each triple pattern are joined in following the plan Π_s (line 5–6).
Case: (TP-S) Sub-plans. TP-S sub-plans are composed of single triple patterns, executable by calling the *TPF(tp)* `smart-KG` server operator (cf. Alg. 2, lines 7–8).
General Case. Joins the results of two recursively evaluated sub-plans Π_l and Π_r (cf. Alg. 2, lines 9–11).

The outcome of the query executor is the result set Ω of evaluating the query Q. In practice, the executor implements an iterator model to push intermediate results of evaluating one subplan to the next operator in the plan. This allows the `smart-KG` client for streaming results incrementally as the data arrives from the server.

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*$(Q, G) = [\![Q]\!]_G$.[10]

5 EXPERIMENTAL EVALUATION

We compare the performance of `smart-KG` with state-of-the-art SPARQL engines. All datasets, queries, and results of our experiments using different workloads are available online.[10]

Knowledge Graphs. We use four RDF graphs (cf. Table 1): three synthetic datasets from the Waterloo SPARQL Diversity Benchmark (WatDiv) [7] [10], with sizes of 10M, 100M and 1000M triples; plus, we use the real-world DBpedia [32] dataset (v.2015A).

Queries and Workloads. For WatDiv, we consider 80 workloads (one per client), each of them with 154 SPARQL queries that were selected uniformly at random from the WatDiv stress test queries [1].

The queries contain up to 10 joins with varying selectivity and shapes (star, path, and snowflake). All workloads follow nearly the same distribution of query selectivities and shapes. For DBpedia, we use queries from the real-world LSQ query log [39]; here, we are interested in highly-demanding queries, hence, we randomly selected 12 BGP queries (out of 259) with runtime higher than 1s. We report the average measures of three independent executions.

Compared Systems. We evaluate the following systems:
- `smart-KG`**:** We implement both client and server in Java[10], extending the TPF implementations [2]. HDT indexes and data are stored on the server's disk, with no client-side family caching.
- **Triple Pattern Fragments (TPF):** We use the node.js TPF client, recommended by the authors, plus the Java TPF server [2], as the `smart-KG` TPF handler is also implemented in Java.
- **Virtuoso:** We run Virtuoso [16] (v7.2.5), without quotas or limits.
- **SaGe:** We use the Python SaGe server and the Java SaGe client with recommended configurations [4].

We configured Virtuoso and SaGe to run with 4 workers [36]. We omit the comparison with brTPF [27], as existing evaluations report that the performance lie between TPF and SaGe [36], and our preliminary tests show scalability problems of the brTPF server implementation [1] for the WatDiv-100M and Watdiv-1000M graphs.

Hardware Setup. We use the following technical infrastructure.
- **Client specifications:** Clients ran on 1, 20 10, 40 and 80 physical machines, each with identical hardware specifications: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz, 32GB of DDR4 RAM, 512GB M.2 NVMe SSD, running Fedora 29 (Linux Kernel v 5.0.14).
- **Server specifications:** The servers run on a VM hosted on a machine running QEMU+ KVM hypervisor with Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz, 384 GB of DDR3 RAM, running Centos 7 (Linux Kernel v3.10.0). Compared server systems were running on VMs configured with 4 CPUs and 32 GB RAM.
- **Network configuration:** For emulating realistic internet connection bandwidth from consumer internet service providers, we limited network speed of each client to 20 MBit, using `tc` [9].

Metrics. Our evaluation considers the following metrics:
- **Number of Timeouts:** Number of queries that time out. We set a timeout of 5 min. for WatDiv and 30 min. for DBpedia queries.
- **Execution Time:** Elapsed time spent by a client executing a workload, measured with the `time` command of Linux.
- **Resource Consumption:** We report on CPU usage per core, RAM usage, and network traffic, all measured with `psutil` [3].

5.1 Creation of Family-based Partitions

For each graph G, Table 1 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). These numbers are obtained fixing $\tau_l = 0.01/100$ for all G, while we set $\tau_h = 0.1/100$ for DBpedia, as we empirically tested that the resultant predicate set filters out both infrequent and heavy hitters. Likewise, we empirically set $\alpha_t = 0.05$ for both datasets, $\alpha_s = 0$ for WatDiv, and $\alpha_s = 0.01/100$ for DBpedia. Fig. 3 shows an ablation study in DBpedia to determine the number of generated families with different values of such parameters[11].

Table 1 also shows that $|F'_{core}|$, $|\mathcal{G}_{serv}|$, and the computation time are sub-linearly increasing with the graph sizes. In WatDiv,

[10] Experiments and the proof are available online https://ai.wu.ac.at/smartkg

[11] We omit α_t as this study analyzes the size of families in the graph, while the extremely large families pruned by α_t tend to be generated when merging families.

Table 1: 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)
WatDiv-10M	10,916,457	521,585	86	1,005,832	59	59	10,106	21,210	1
WatDiv-100M	108,997,714	5,212,385	86	9,753,266	59	59	22,855	37,392	7
WatDiv-1000M	1,092,155,948	52,120,385	86	92,220,397	59	59	39,046	52,885	12
DBpedia	837,257,959	113,986,155	60,264	221,623,898	218	84	35	29,965	23

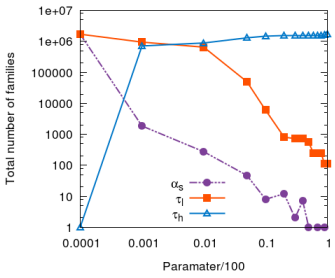
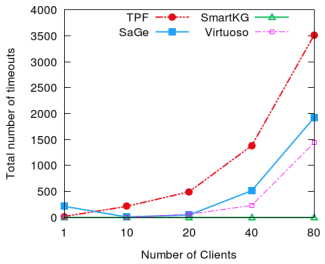
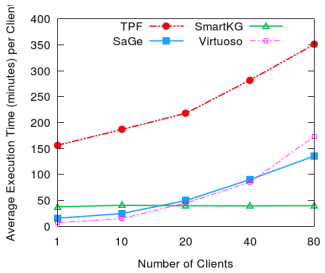


Figure 3: Ablation study in DBpedia to select the parameters in partitioning algorithm



(a) Number of timeouts



(b) Average execution time

Figure 4: Performance on the WatDiv-100M workload

$F'_{core} = F'(G)$, whereas in DBpedia, the initial number of P'_G-restricted[12] families $|F'(G)|$ is >600K: the family pruning strategy allows `smart-KG` to identify $|F'_{core}| = 35$ core families, which are merged into ~30K materialised partitions.

5.2 Overall Query Performance

We report on performance for the WatDiv query workload, at increasing number of clients and dataset size. The performance of `smart-KG` always considers the family grouping/pruning strategies mentioned in Sec.4.1.3; we also tested `smart-KG` without grouping/merging, which however did not scale, due to requiring shipping large numbers of small partitions with many redundant triples.

Increasing Number of Clients. In this part of the study, we focus on the graph WatDiv-100M as this is in line with the size of open KGs published in the LOD Cloud [6], with an average of 183M RDF triples. Fig. 4 shows the results of executing the WatDiv-100M workload on the query performance at different number of concurrent clients (1, 10, 20, 40 and 80) in terms of (a) number of timeouts, and (b) average workload execution time per client.

Fig. 4a shows that `smart-KG` produces no timeouts at such relative modest but state-of-the-art graph sizes. That is, even with 80 concurrent clients, our approach is able to successfully finish all queries in the workload for all concurrent clients. In contrast, TPF was not able to answer all queries within a 5 minutes timeout, even in the single client configuration. The percentage of timeouts escalates with increasing number of clients, from 10% in 1-client workload to an average of 28% with 80 concurrent clients. These results confirm the scalability limitations of the system.

On WatDiv-100M, SaGe times out in less queries than TPF, but timeouts increase significantly with the number of clients, reaching a non-negligible 15% of the queries for 80 concurrent clients.

The average workload execution time per client, in Fig. 4b, shows superior performance and scalability of our approach, where performance remains constant irrespective of the number of clients, as `smart-KG` limits the server load and joins are mostly performed on clients over shipped KG partitions. For less than 20 concurrent clients, SaGe starts slightly ahead of `smart-KG`. From this point and on, SaGe suffers from excessive delays, and overall performance is degrading, e.g., `smart-KG` is up to 3.5 times faster with 80 clients. This is because SaGe executes SPARQL queries using a *round robin* policy to avoid the convoy effect but, with an increasing number of clients, the increased waiting time and server usage lead to degrading average completion time for queries.

In turn, TPF is significantly worse – up to three times slower – than the other systems due to the enormous number of requests and the excessive data transfer. As we will show in the next section, traffic is substantially higher with larger datasets because clients need to ask for several server responses to evaluate a single query.

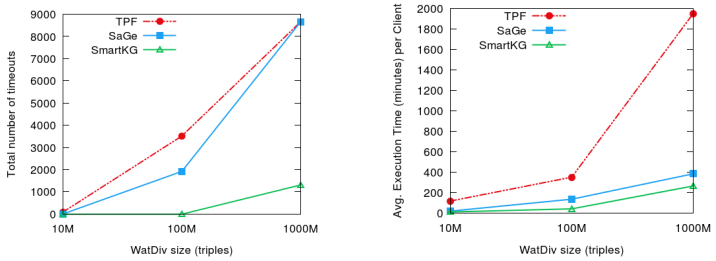
To complete the comparison, we also evaluate the performance of query shipping strategies using a Virtuoso SPARQL endpoint. As shown in Fig. 4, Virtuoso behaves very similar to SaGe with the difference that i) it shows no timeout and similar performance for 10 clients, but ii) its execution time is slightly degrading with 80 clients. Given these results and in line with previous studies [36], in the following sections, we focus on comparing the performance of `smart-KG` with the shipping strategies of TPF and SaGe only.

Increasing KG Size. Fig. 5 shows the performance of the evaluated systems at increasing KG sizes, fixing the scenario to 80 concurrent clients. We execute the WatDiv workloads over 10M to 1000M triples, which constitutes, to the best of our knowledge, the largest experiment on client-side SPARQL query approaches to date.

Fig. 5a shows again the number of query timeouts. As expected, timeouts of TPF and SaGe significantly increase with the size of the graph. TPF is not able to scale for the WatDiv-1000M dataset, failing to answer 75% of the queries. Although SaGe is slightly better than TPF, it fails to answer 68% of the queries with 80 concurrent clients. In contrast, `smart-KG` reports the best results at scale, timing out only in the largest graph for 10% of the queries.

Fig. 5b presents the average workload execution time for all systems and different sizes, with 80 concurrent clients. As expected, average workload completion time increases with the larger KGs, while `smart-KG` remains the fastest in all scenarios. TPF has the

[12]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 [39].



(a) Number of Timeouts (b) Average Workload Execution Time

Figure 5: Performance on the workloads (80 clients) at increasing KG sizes

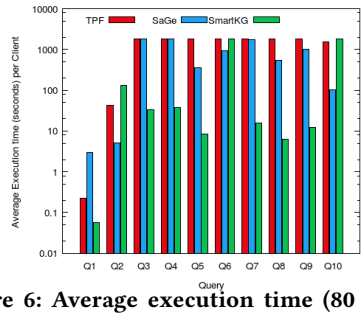
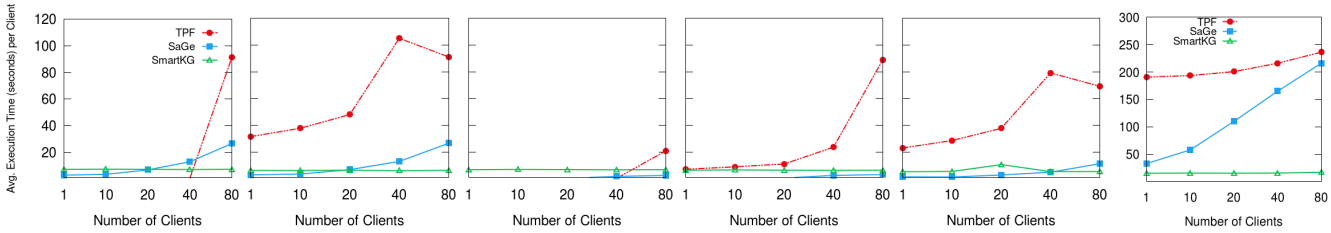


Figure 6: Average execution time (80 clients) with DBpedia high-demanding queries



(a) L1 (b) L2 (c) L3 (d) L4 (e) L5 (f) Complex

Figure 7: Avg. execution time per client on the standard WatDiv-100M, for simplest L queries and complex queries

longest execution time (and significantly longer in Watdiv-1000M) while SaGe is on average 1.5 times slower than `smart-KG` in Watdiv-1000M. Note also that average execution times include timeouts: since we have shown already that we do better in numbers of timeouts, we may assume that full execution times with unlimited runtimes would be even more significantly in our favor.

DBpedia Queries. We evaluate the performance on DBpedia to consider real-wold data and high-demanding queries. Fig. 6 shows the performance results on 80 clients for 12 representative queries of the LSQ log, omitting Q11 and Q12 which time out (>30 minutes) in all systems. The results are in line with our previous analysis: TPF is the slowest (except Q1,Q2) and `smart-KG` is 2-3 levels of magnitude faster than SaGe in all queries except for Q2, Q6, and Q10. These are the cases where SmartKG depends heavily on TPF shipping, while SaGE can delegate to the SPARQL server.

5.3 Evaluation of Simple & Complex Queries

While in the previous analysis, a workload consisted of queries with mixed characteristics, we have performed a separate performance analysis on two specific query categories predefined by the *WatDiv Basic Testing* [10]: *linear (L)*, which represents simple path queries, and *Complex (C)*, with more challenging queries including a combinations of low-selective star and path queries. WatDiv provides L-query templates and we randomly generate 3 queries for each subtype (L1-L5) per client. The benchmark has only three C queries (not templates), hence, we extend it by selecting 50 complex queries (based on low selectivity patterns and high execution time) from the initial intensive workload, for each client.

Fig. 7 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 5-7 seconds. SaGe has the best performance in the L3 and L4 workloads, with average execution times of less than 2 seconds per query. However, the SaGe execution time is affected by the number of clients for L1, L2, and L5. SaGe provides better execution time than `smart-KG` with up to 20 concurrent clients, while `smart-KG` is more competitive for larger number of clients. Finally, TPF is the slowest approach in L2, L4, L5 queries, while it excels in L1 and L3 up to 40 clients. TPF could not scale to 80 clients.

Fig. 7f shows the overall execution times for the C-queries workload (WatDiv-100M, 80 clients, 5min timeout). TPF is again the slowest solution, while `smart-KG` significantly outperforms SaGe as our partition shipping allows clients to work locally in parallel on their queries, without long waiting times for the server response. In contrast, SaGe preemption model introduces additional delays at heavy workloads, due to query swapping, long waiting queues and limited processing time per query on the server.

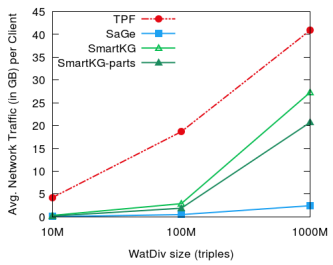
5.4 Resource Consumption

Server Network Load. Fig. 8a shows the average network traffic per client (in GB) on the intensive workload. We report the average per client, with 80 clients running in parallel on WatDiv-100M.

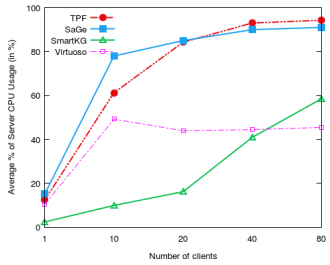
TPF incurs the highest communication costs due to the number of requests and the shipping of extensive intermediate results, which leads to poor query execution performance, as shown before.

In contrast, SaGe produces the least data transfer among all the systems, as it works as a SPARQL endpoint with a preemption model and no intermediate results are transferred. The only additional data transfer overheads in SaGe are the saved plans when queries are resumed, a relatively small cost depending on the number of calls required to finish each query. This factor depends on the complexity of the queries and the number of concurrent clients.

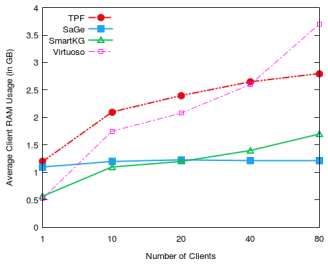
As expected, `smart-KG` requires more data transfer than SaGe, but up to 10× less data than TPF. Most of the data transferred is due to partition shipping (cf. Fig. 8a `smart-KG-parts`). Yet, the retrieved partitions can be reused for queries that require the same partitions.



(a) Network traffic per client (in GB) on the intensive workload at increasing KG sizes



(b) Avg. Server CPU Usage (in %) at increasing number of clients (WatDiv-100M)



(c) Avg. Server RAM Usage (in GB) at increasing number of clients (WatDiv-100M)

Figure 8: Resource consumption on the intensive workload.

Caching the partitions will execute streak queries with minimal communication to the server. A streak is a concept defined in [14] as sequence of queries that appear as subsequent modifications of a seed query.

Server CPU, RAM, and Disk Usage. Fig. 8b shows that TPF and SaGe extensively use the server CPU. In particular, SaGe and TPF respectively consume 80% and 60% of the CPU to execute 10 clients in parallel, and both rapidly increase to 100% for 40 and 80 clients. In practice, this reduces query throughput as most CPU time is allocated to query processing while new requests are queued. In contrast, `smart-KG` only uses 60% of the CPU to handle the workload on 80 parallel clients, which still gives room for serving additional clients in the current hardware configuration. `smart-KG` server consumes limited CPU thanks to its mixed triple pattern and partition shipping, which hardly requires server CPU usage.

Fig. 8c shows that TPF has the overall highest server memory usage, while SaGe's consumption remains constant and low, thanks to its preemption model. `smart-KG` uses least memory consumption up to 20 clients and then slightly increases at 40 and 80 clients to 0.5 GB more than SaGe (due to TPF triple evaluation on the server). We additionally compare to the resource consumption of Virtuoso, which shows relatively constant CPU (~40%) and increasing RAM consumption, exceeding TPF for 80 clients. These values are in line with the significant timeouts reported previously (see Fig. 4).

Table 2 shows the raw data sizes (in N-Triples) of the graphs and storage requirements for the evaluated systems. As expected, TPF and SaGe require a single HDT file, which highly compacts storage needs. In contrast, the high number of HDT partitions managed by `smart-KG` results in additional costs in disk space, doubling the raw size of the WatDiv graphs (DBpedia uses less space given its more restricted pruning). Given that disk space is relatively affordable for servers, `smart-KG` provides a reasonable tradeoff for faster and more balanced, SPARQL query execution.

Client CPU and RAM usage. As for client-side resources, as expected, Virtuoso excels with 80 clients in the WatDiv-100M workload (fully in the server, hence the clients run with 8% RAM size). SaGe also shows reasonable (average 15%) usage of the client CPUs, as it performs only two main operations on the client side: first, resuming query execution through received saved plans from the server; second, a subset of SPARQL operators such as filter, order by, aggregations, are offered. TPF performs joins of triple pattern results all locally on the client-side which is costly, leading to higher

Table 2: Comparison of storage requirements (in MB) for systems with HDT backend vs original graph size (raw)

Dataset	Raw	SmartKG	TPF/SaGe
WatDiv-10M	1,471	2,783	112
WatDiv-100M	14,876	29,711	1,186
WatDiv-1000M	151,862	310,574	12,793
DBpedia	158,197	122,440	17,904

Virtuoso takes ~ 3 times the space of TPF/SaGe.

(55%) client average CPU usage. `smart-KG` finally also depends on the client to execute query stars as well as TPF Join processing, so that client average CPU usage is higher, with 70% yet visible for most of the current client systems.

These percentages decrease with higher numbers of clients, because network waiting time dominates in the case of TPF and `smart-KG` and the long waiting queues in the case of SaGe. Client memory consumption remains fairly constant and low for both SaGe and TPF. `smart-KG` consumes more client memory, however still reasonable. For instance, `smart-KG` utilizes up to 3 GB RAM.

6 CONCLUSION AND FUTURE WORK

We introduced `smart-KG`, a hybrid approach to efficiently query Knowledge Graphs (KGs) on the Web, balancing the load between servers and clients. We combine the Triple Pattern Fragment (TPF) strategy with shipping compressed graph partitions that can be locally queried. The served partitions are based on predicate families and different pruning parameters control the sizes and numbers of the partitions. The `smart-KG` client implements a query decomposer, planner, and executor tailored to trade off TPF and partition shipping. Our evaluation shows that `smart-KG` significantly outperforms the state of the art, especially with increasing number of concurrent clients, and on challenging BGP queries. We also show that, at the cost of reasonable client resources, `smart-KG` improves server availability, consuming significantly less CPU and RAM than most of the evaluated systems, and reducing TPF's network traffic.

Future work includes the following directions. First, we argue (and have experimentally demonstrated) that our approach using families provides a reasonable trade-off of shipping sizes; still comparing to other partitioning strategies (e.g. predicate-wise or hash partitioning) is on our agenda. Next, we plan to *exploit query logs* of KGs served on the web to generate query-driven partitions. This includes strategies to adapt to new workloads or updated KGs. Finally, we will investigate *shipping-based cost models* to enhance query performance and further reduce network traffic.

ACKNOWLEDGMENTS

This work has been supported by the European Union Horizon 2020 research and innovation programme under grant 731601 (SPECIAL) and by the Austrian Research Promotion Agency (FFG) grant no. 861213 (CitySPIN) and by the German Research Foundation (DFG) under grant 316669855 (SoRa).

REFERENCES

[1] [n. d.]. brTPF. http://olafhartig.de/brTPF-ODBASE2016/. http://olafhartig.de/brTPF-ODBASE2016/
[2] [n. d.]. Linked Data Fragments.http://linkeddatafragments.org/software/. https://github.com/sage-org/
[3] [n. d.]. psutil. https://psutil.readthedocs.io/. psutil.https://psutil.readthedocs.io/.
[4] [n. d.]. SaGe.https://github.com/sage-org/. https://github.com/sage-org/
[5] [n. d.]. Stardog. https://www.stardog.com/. https://www.stardog.com/.
[6] [n. d.]. The Linked Open Data Cloud.https://lod-cloud.net/. https://lod-cloud.net/.
[7] [n. d.]. Waterloo SPARQL Diversity Benchmark. https://dsg.uwaterloo.ca/watdiv/. https://dsg.uwaterloo.ca/watdiv/
[8] Maribel Acosta and Maria-Esther Vidal. 2015. Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data. In *International Semantic Web Conference*. 111–127.
[9] Werner Almesberger et al. 1999. Linux network traffic control—implementation overview.
[10] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *Proc. of ISWC*. Springer, 197–212.
[11] Carlos Buil Aranda, Axel Polleres, and Jürgen Umbrich. 2014. Strategies for Executing Federated Queries in SPARQL1.1. In *Proc. of ISWC*. 390–405. https://doi.org/10.1007/978-3-319-11915-1_25
[12] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked Data - The Story So Far. *Int. J. Semantic Web Inf. Syst.* 5, 3 (2009), 1–22.
[13] Piero Andrea Bonatti, Michael Cochez, Stefan Decker, Axel Polleres, and Valentina Presutti (Eds.). 2018. *Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web (Dagstuhl Seminar 18371)*. Schloss Dagstuhl, Germany. http://polleres.net/bona-etal-DagstuhlReport18371.pdf to appear.
[14] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An analytical study of large SPARQL query logs. *Proceedings of the VLDB Endowment* 11, 2 (2017), 149–161.
[15] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL web-querying infrastructure: Ready for action?. In *Proc. of ISWC*. Springer, 277–293.
[16] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*. Springer, 7–24.
[17] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. 2013. SPARQL 1.1 Protocol. *Recommendation, W3C, March* (2013).
[18] J. D. Fernández, M. A. Martínez-Prieto, P. de la Fuente Redondo, and C. Gutiérrez. 2018. Characterizing RDF Datasets. *Journal of Information Science* 44, 2 (2018), 203–229. https://doi.org/10.1177/0165551516677945
[19] Javier D. Fernández, Miguel A. Martınez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF Representation for Publication and Exchange (HDT). *J. Web Sem.* 19, 2 (2013). http://www.websemanticsjournal.org/index.php/ps/article/view/328
[20] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. 1996. Performance Tradeoffs for Client-Server Query Processing. In *Proc. of SIGMOD*. 149–160. https://doi.org/10.1145/233269.233328
[21] Andrey Gubichev and Thomas Neumann. 2014. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, Vol. 14. 439–450.
[22] Christophe Guéret, Paul T. Groth, Frank van Harmelen, and Stefan Schlobach. 2010. Finding the Achilles Heel of the Web of Data: Using Network Analysis for Link-Recommendation. In *Proc. of ISWC*. 289–304. https://doi.org/10.1007/978-3-642-17746-0_19
[23] C. Gutiérrez, C. Hurtado, A. O. Mendelzon, and J. Perez. 2011. Foundations of Semantic Web Databases. *JCSS* 77 (2011), 520–541.
[24] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language.
[25] Olaf Hartig. 2013. SQUIN: a traversal based query execution system for the web of linked data. In *Proc. of SIGMOD*. ACM, 1081–1084.
[26] Olaf Hartig, Christian Bizer, and Johann Christoph Freytag. 2009. Executing SPARQL Queries over the Web of Linked Data. In *Proc. of ISWC*. 293–309. https://doi.org/10.1007/978-3-642-04930-9_19
[27] Olaf Hartig and Carlos Buil-Aranda. 2016. Bindings-Restricted Triple Pattern Fragments. In *Proc. of ODBASE (LNCS)*, Vol. 10033. 762–779. https://doi.org/10.1007/978-3-319-48472-3_48
[28] Olaf Hartig and Giuseppe Pirrò. 2015. A context-based semantics for SPARQL property paths over the web. In *Proc. of ESWC*. Springer, 71–87.
[29] Ali Hasnain, Maulik R Kamdar, Panagiotis Hasapis, Dimitris Zeginis, Claude N Warren, Helena F Deus, Dimitrios Ntalaperas, Konstantinos Tarabanis, Muntazir Mehdi, and Stefan Decker. 2014. Linked biomedical dataspace: lessons learned integrating data for drug discovery. In *Proc. of ISWC*. Springer, 114–130.
[30] Lars Heling, Maribel Acosta, Maria Maleshkova, and York Sure-Vetter. 2018. Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study. In *Proc. of ISWC*. Springer, 86–102.
[31] A. Hernández-Illera, M.A. Martínez-Prieto, and J.D. Fernández. 2015. Serializing RDF in Compressed Space. In *Proc. of DCC*. 363–372.
[32] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195. https://doi.org/10.3233/SW-140134
[33] M.A. Martínez-Prieto, M. Arias, and J.D. Fernández. 2012. Exchange and Consumption of Huge RDF Data. In *Proc. of ESWC*. 437–452.
[34] M.A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. 2016. Practical Compressed String Dictionaries. *Information Systems* 56 (2016), 73–108.
[35] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos. 2017. Extended Characteristic Sets: Graph Indexing for SPARQL Query Optimization. In *Proc. of ICDE*. 497–508. https://doi.org/10.1109/ICDE.2017.106
[36] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. 2019. SaGe: Web Preemption for Public SPARQL Query Services. (2019), 1268–1278.
[37] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proc. of ICDE*. IEEE, 984–994.
[38] Axel Polleres, Maulik R. Kamdar, Javier D. Fernández, Tania Tudorache, and Mark A. Musen. 2018. A More Decentralized Vision for Linked Data. In *Proc. of DeSemWeb@ISWC*. http://ceur-ws.org/Vol-2165/paper1.pdf
[39] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: the linked SPARQL queries dataset. In *Proc. of ISWC*. Springer, 261–269.
[40] Manuel Salvadores, Matthew Horridge, Paul Alexander, Ray Fergerson, Mark Musen, and Natasha Noy. 2012. Using SPARQL to query bioportal ontologies and metadata, Vol. 7650. 180–195. https://doi.org/10.1007/978-3-642-35173-0_12
[41] G. Schreiber and Y. Raimond. 2014. *RDF 1.1 Primer*. W3C Working Group Note. https://www.w3.org/TR/rdf11-primer/.
[42] Pierre-Yves Vandenbussche, Jürgen Umbrich, Luca Matteis, Aidan Hogan, and Carlos Buil-Aranda. 2017. SPARQLES: Monitoring public SPARQL endpoints. *Semantic Web* 8, 6 (2017), 1049–1065.
[43] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. *Journal of Web Semantics* 37–38 (March 2016), 184–206. https://doi.org/doi:10.1016/j.websem.2016.03.003