WiseKG: Balanced Access to Web Knowledge Graphs

Amr Azzam*
Vienna Univ. of Economics & Business
amr.azzam@wu.ac.at

Christian Aebeloe*
Aalborg University
caebel@cs.aau.dk

Gabriela Montoya
Aalborg University
gmontoya@cs.aau.dk

Ilkcan Keles
Aalborg University
Turkcell
ilkcan.keles@turkcell.com.tr

Axel Polleres
Vienna Univ. of Economics & Business
Complexity Science Hub Vienna
axel.polleres@wu.ac.at

Katja Hose
Aalborg University
khose@cs.aau.dk

ABSTRACT

SPARQL query services that balance processing between clients and servers become more and more essential to handle the increasing load for open and decentralized knowledge graphs on the Web. To this end, Linked Data Fragments (LDF) have introduced a foundational framework that has sparked research exploring a spectrum of potential Web querying interfaces in between server-side query processing via SPARQL endpoints and client-side query processing of data dumps. Current proposals in between typically suffer from imbalanced load on either the client or the server. In this paper, to the best of our knowledge, we present the first work that combines both client-side and server-side query optimization techniques in a truly dynamic fashion: we introduce WiseKG, a system that employs a cost model that dynamically delegates the load between servers and clients by combining client-side processing of shipped partitions with efficient server-side processing of star-shaped subqueries, based on current server workload and client capabilities. Our experiments show that WiseKG significantly outperforms state-of-the-art solutions in terms of average total query execution time per client, while at the same time decreasing network traffic and increasing server-side availability.

ACM Reference Format:
Amr Azzam, Christian Aebeloe, Gabriela Montoya, Ilkcan Keles, Axel Polleres, and Katja Hose. 2021. WiseKG: Balanced Access to Web Knowledge Graphs. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021 ,Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449911

1 INTRODUCTION

The Semantic Web has over the past two decades seen a steady increase in the amount of data published as Linked Open Data (LOD), forming a Web of interconnected Knowledge Graphs (KG) [8]. Such KGs are accessible either via public SPARQL endpoints, downloadable data dumps, KG partitions, or dereferenceable URIs. The continued work on Linked Data is fueled by the prospects of such interconnected KGs finally making the vision of an interlinked Web of Data a reality, and at the same time providing scalable query processing to its users. However, as provisioning and maintaining access to KGs is still a huge burden for data publishers [2, 26, 30], low availability of public SPARQL endpoints [5, 29] remains one of the greatest obstacles to this vision.

In order to tackle this bottleneck, various recent proposals emphasize decentralization as a means to lift this burden off the data providers. On one hand, several approaches have focused on the decentralization of the data [2, 3, 9]. While these approaches increase the availability of the data, their query processing capabilities are significantly less efficient than approaches with a powerful centralized server or approaches that ship full data dumps to powerful clients for local processing. On the other hand, several recent studies [1, 6, 12, 19, 20] have focused on the decentralization of the query processing tasks. These approaches divide the processing burden between servers and clients. Even though the servers might be powerful, they will struggle with highly concurrent query loads. Therefore, the clients, which might have free resources, will take some of the query processing tasks for themselves rather than waiting for an overloaded server.

To this end, Triple Pattern Fragments (TPF) [30] reduces the server load significantly by processing joins on the client-side while only processing individual triple patterns on the server. To avoid processing non-selective triple patterns on the server, the client locally processes joins using previously obtained bindings in the request (one binding at a time), potentially leading to smaller intermediate results. Yet, this kind of processing potentially leads to a large number of server requests during query processing, creating a significant overhead on the network traffic. As opposed to just providing triple pattern execution on the server and full join capabilities on the client, two approaches have recently been proposed to optimize SPARQL query processing. Star Pattern Fragments (SPF) [1] exploits server-side evaluation of star-shaped subqueries, while smart-KG [6] exploits client-side evaluation of star-shaped subqueries by retrieving compressed KG partitions from the server. However, the potential benefit of being able to dynamically switch between strategies based on the current server load, caching, etc., remains mostly unexploited by the current state of the art.

*Both authors contributed equally to this research.

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3449911

Hence, in this paper, we present WiseKG, a novel approach that combines the strengths of the state of the art and further advances them by finding a novel balance between server and client load. Based on the current load on the server, WiseKG decides whether subqueries should be processed on the client or on the server. The underlying cost model considers parameters such as CPU load, estimated network transfer time, and currently available resources at the client to determine where to process a particular part of the query. By applying this cost model, servers can dynamically share the query processing tasks with the clients, making better use of server resources and retaining high performance even during high load. At the same time, they achieve significantly lower query processing times and, by processing subqueries locally on the server, avoid unnecessary data shipping during periods with an overall low query processing load.

In summary, we make the following contributions:

- We present WiseKG, a novel system that dynamically shifts the query processing load between client and server.
- WiseKG employs a cost model to minimize the total time consumed by client-side and server-side components while considering the current load on the server and the client.
- Our extensive evaluation using demanding query workloads on real-world KGs as well as synthetic KGs up to 1 billion triples shows that WiseKG significantly outperforms the state of the art.

The remaining sections are organized as follows. We cover background in Section 2. Section 3 provides a motivating example. Section 4 gives an overview of WiseKG, followed by a presentation of the server-side cost model. Section 5 details SPARQL query processing on the client- and server-side. Section 6 then presents an empirical evaluation of WiseKG. Last, we conclude the paper and provide an outlook on future work in Section 7.

2 BACKGROUND

RDF and SPARQL. We assume the readers' familiarity with base technologies such as RDF and SPARQL, from which we borrow standard notation such as RDF Turtle[1] or algebra operators [25]; by $subj(t)$, $pred(t)$, $obj(t)$ we to refer to the components of a single *RDF triple* $t \in G$, such that these components are RDF terms (i.e. URIs/IRIs, blank nodes, and literals). An RDF knowledge graph (KG) G is a set of such triples, where $subj(G)$, $pred(G)$, $obj(G)$ denote subjects, predicates, and objects in G.

RDF KGs can be queried using the query language SPARQL, which relies on matching graph patterns for easy access to RDF stores. The fundamental graph pattern is a *triple pattern* tp, which is an RDF triple that permits variables from an infinite set V of variables, disjoint with the previously mentioned RDF terms.

A *Basic Graph Pattern (BGP)* is a set of triple patterns $\{tp_1 \dots tp_n\}$ that can be viewed as a conjunctive query; note that while semantically, order is not relevant, we use sequences $(\dots)$ instead of sets $\{\dots\}$ in this paper to indicate execution (left-linear) order in a query plan, i.e., for instance $(tp_1, \dots, tp_n)$ stands for a left-linear query execution plan $(\dots (tp_1 \bowtie tp_2) \bowtie \dots) \bowtie tp_n)$, whereas non-left-linear plans will be denoted by respective explicit parentheses.

[1] http://www.w3.org/TR/2014/REC-turtle-20140225/

For any pattern P, we denote by $var(P)$ its variables. The solutions (or, answers, resp.) of a (query) pattern P over a graph G, denoted $[[P]]_G$, are given as sets Ω of *bindings*, i.e., mappings of the form, $\mu : var(P) \rightarrow R$ to the set R of RDF terms, such that $G \models \mu(P)$, i.e. $\mu(P)$ forms a (sub)graph entailed by G. Two mappings $\mu_1, \mu 2$ are called *compatible*, denoted as $\mu_1 || \mu_2$ if for any $v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$, cf. [25] for details.

A *star pattern* $sp = \{tp_1 ... tp_k\}$ is a BGP such that $subj(tp_i) = subj(tp_j)$ for all $i, j \in \{1, \dots, k\}$, i.e., the subjects of all triple patterns are the same. We refer to k as the *star-size* of sp.

Note that each complex BGP P can be decomposed into a set of star patterns $\mathcal{S}(P)$, called the *(star-)decomposition* of P as follows:

$$\mathcal{S}(P) = \{\{t \in P \mid subj(t) = s\} \mid s \in subj(P)\}$$

Along the above-mentioned notation for query plans as sequences of patterns being interpreted as left-linear query plans, we will analogously write query plans that evaluate patterns per stars as permutations of $\mathcal{S}(P)$, e.g., the query plan shown in Fig. 1b could be written as (sp_1, sp_2, tp_6), indicating an execution plan at the level of joining star patterns as follows: $((sp1 \bowtie sp2) \bowtie tp)$.

The primary focus of this paper is on evaluating BGPs as the fundamental retrieval functionality of SPARQL. However, more complex patterns, such as *Union*, *Optional*, and *Filters*, are covered by our proposed system, which implements the full SPARQL specification – for a more complete formalization we refer to [25].

2.1 Existing KG Interfaces

In this section, we define query interfaces for KGs following the principles set by Linked Data Fragments (LDF) [30]. In essence, LDF characterizes APIs that allow access to fragments of a KG G through (specific to a particular instantiation of LDF) a limited range of allowed query patterns that a client can submit to the server; often with the goal to limit server-side computation cost and to enable effective HTTP caching, while leaving evaluations of more complex patterns to the client. Variations of LDF also offer additional controls to ship intermediate bindings alongside with queries or to control the "chunk size" of results through specifying page sizes into which the results should be batched. Note that in line with LDF [30] we also assume G to be blank node-free.

In this paper, we omit details on LDF, such as metadata that is sent along with query results and hypermedia controls. However, we borrow from the original specification [30] and align formal definitions and notations to uniformly present different APIs:

Definition 1 (adapted from [30]). *An LDF API of a KG G accessible at an endpoint URI u[2] is a tuple $f = \langle s, \Phi \rangle$ with*

- *a selector function $s(G, P, \Omega)$ that defines how a fragment $\Gamma \subseteq G$, or alternatively a set of fragments[3] $\Gamma^* \subseteq 2^G$ is constructed upon calls to the API.*
- *a paging mechanism $\Phi(n, l, o)$ parameterized by $n, l, o \in \mathbf{N}_0$ denoting maximum page size, limit, and offset.*

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

[3] 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.

The selector function s is parameterized by a graph G, pattern P, and a set of bindings Ω[4], where we define two variants, $s(\cdot)$ and $s^*(\cdot)$, which differ essentially in terms of returning either a single graph or one subgraph per solution $\mu \in [[P]]_G$:

$$s(G,P,\Omega) = \{t \in \mu(P) \mid \exists \mu \in [[P]]_G : G \models \mu(P) \wedge (\exists \mu' \in \Omega : \mu' || \mu)\}$$

$$s^*(G,P,\Omega) = \{\mu(P) \mid \exists \mu \in [[P]]_G : G \models \mu(P) \wedge (\exists \mu' \in \Omega : \mu' || \mu)\}$$

As we will see, all LDF APIs discussed in this section can indeed be expressed in terms of one of these two default selector functions.

The general paging mechanism Φ we use in this paper shall enable returning the result in batches, e.g., for LDF use cases where Γ (or, resp., Γ^*) would be very large or retrieving the whole result is not required or possible. Hence, we assume that $\Phi(n,l,o)$ simply defines a mechanism to divide Γ into partitions (or *pages*) $\{\Gamma_1, \ldots, \Gamma_k\}$, where for each page Γ_i it is guaranteed that $|\Gamma_i| < n$ (i.e., Γ_i does not contain more than n triples), and l and o, resp. would allow to request the pages from Γ_o to Γ_{o+l}[5]. We assume l to default to $l = \infty$, o to default to $o = 1$, and finally $n = \infty$ signifying that whole graph Γ should be returned.

In the following, we will explain different approaches on the LDF framework's spectrum, wrt. implications on server availability under high numbers of concurrent clients:

Data Dumps offer the clients simple access to the entire KG. In order to perform a SPARQL query, the clients have to download the whole KG and run a local SPARQL engine themselves. This can be a very beneficial solution for many clients with sufficient resources but puts high processing cost on the client, plus the need for high amounts of data transfers whenever the KG evolves/changes. Data dumps can be characterized in terms of LDF by

- the selector function $s(\cdot)$ as defined above,
- the only admissible form of P and Ω are $P = \{(?s, ?p, ?o)\}$ and $\Omega = \{\emptyset\}$, i.e., $s(G,P,\Omega)$ boils down to the identity function,
- Φ: the only admissible parameter for $\Phi(n,l,o)$ is $\Phi(\infty,1,1) = \{\Gamma_1\} = \{\Gamma\}$.

SPARQL endpoints provide efficient querying on the server side; the query shipped to the server is typically evaluated in an efficient triple store such as Virtuoso, Blazegraph, and Jena, etc., without work for the clients, who receive the ready end result. This can be characterized in terms of LDF as follows:

- while SPARQL endpoints usually directly return sets of bindings, they can also be viewed as a variant of $s^*(\cdot)$ by returning subgraphs of the form $\mu(P)$[6],
- any pattern P is admissible;
- $\Omega = \{\emptyset\}$, unless `VALUES` patterns are considered, which could be viewed as equivalent to binding restrictions a la LDF,
- Φ: while some SPARQL endpoints support other forms of paging, the standard LIMIT and OFFSET operators for BPGs could be considered as `LIMIT` l and `OFFSET` o such that $n = |P|$; however, note that subsequent calls of SPARQL queries with consecutive OFFSETs are in general not guaranteed to behave deterministically.

SaGe is – in essence – a SPARQL endpoint with the ability to interrupt queries under too much concurrent load on the server side [19]. That is, in principle, we can view SaGe as a variant of SPARQL endpoints that, given a query P whose execution exceeds a timeout τ, suspends it and only returns a partial result $\{\Gamma_1, \ldots, \Gamma_{o-1}\}$, along with additional state information to the client. The client can (with additional hypermedia controls using this state information) deterministically continue exactly at offset o in a subsequent call. Hence, in times of high query load, SaGe uses this strategy to suspend clients to avoid starvation of others. We note that, while the SaGe *server* itself is stateless, i.e., it does not store the intermediate states of the suspended queries, it handles the overall query execution load incl. join processing for BGPs.

Triple Pattern Fragment (TPF) [30] is an interface to enable live SPARQL querying with high availability and scalability by restricting server capabilities to only answer single triple pattern fragments and shifting processing of more complex patterns to the client-side (with the expenses of a substantial increase in the network traffic). In terms of the generic LDF framework, TPF is the most straightforward "incarnation", defined as:

- the selector function is $s(\cdot)$ as defined above,
- the only admissible form of P are triple patterns and $\Omega = \{\emptyset\}$,
- $\Phi(n,l,o)$: allows results to be "batched" into chunks of n triples, whereas limit l and offset o cannot be set explicitly in TPF.

Binding-Restricted Triple Pattern Fragments (brTPF) [12] is an extension of TPF that reduces the network load through additionally permitting arbitrary $\Omega \neq \emptyset$. This ensures fewer requests to the server plus faster query processing. However, brTPF still potentially struggles with high numbers of concurrent clients or queries with large intermediate results.

Star Pattern Fragments (SPF) [1] proposes to generalize brTPF from single triples to handling star-shaped subqueries 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 relatively efficiently by the server [25]. 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:

- $s_{SPF}(G,P,\Omega) = s^*(G,P,\Omega)$, i.e., $s^*(\cdot)$ is used to return results per pattern solution,
- the only admissible form of P are star-shaped BGPs,
- Ω can be any set of bindings,
- $\Phi(n,l,o)$: as solutions are returned per pattern solution, n is fixed to the star pattern of size k but SPF also allows to paginate over solutions, i.e., retrieving results in chunks of l (iterating over increasing offsets $o := o + l$).

[4] We note that this strict definition of allowed parameters for s is not made in [30], but we will rather use those here to describe the considered APIs uniformly.

[5] As such l,o should be viewed synonmous SPARQL's LIMIT and OFFSET modifiers.

[6] Deriving μ is straightforward since, given P, μ and $\mu(P)$ are in a trivial 1-to-1 correspondence. We prefer this interpretation of the LDF metaphor to SPARQL endpoints over – as suggested in a side note in [30] – relying on encoding result sets as RDF triples (such as using e.g. the informal RDF SPARQL result format from the SPARQL1.1 Test Case Structure, cf. https://www.w3.org/2009/sparql/docs/tests/README.html) since the latter would not return subgraph(s) of G.

Experiments [1] 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 comparably low network load.

smart-KG [6] (SKG) is another alternative paradigm that combines TPF with the idea to ship graph partitions per star-shaped patterns. To this end, the server holds (compressed and queryable) partitions per common predicate families of G, defined as follows:

Definition 2 (Predicate family). *We define a* predicate family $F(s)$ *wrt. to KG* G *as the set of predicates associated with subject* s*:*

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

We denote the set of families of a graph G *as* $F(G)$ *or* F *for simplicity whereby* $F(G) = \{F(x) | x \in subj(G)\}$.

Predicate families, also known as *characteristic sets*, were introduced in [23] as an RDF query cardinality estimation method while SKG uses families as a basis for inducing a graph partitioning of G, with one partition G_f per $f \in F(G)$ [6].

We can interpret SKG as an LDF interface as follows:

- admissible patterns are defined by submitting a predicate family $f' = \{p_1, \dots p_k\}$, which may be interpreted as a pattern $\bigcup_{i=1}^{k} \{(?S, p_i, ?P_i)\}$, or resp., in SPARQL syntax, as {?S p_1 ?P$_1$; p_2 ?P$_2$; . . . ; p_k ?P$_k$.},
- $\Omega = \{\emptyset\}$ is the only admissible binding set, i.e., SKG does not consider binding restrictions,
- the selector function may be viewed as a variation of $s(\cdot)$ as follows: while the SKG server API returns a graph G_f per family $f \in F(G)$ matching P, the union of all these graphs is defined as

$$s_{\text{SKG}}(G, P, \Omega) = \{t \in G \mid \exists t' \in s(G, P, \Omega) : subj(t) = subj(t')\}$$

  That is, while strictly speaking, indeed rather *several* partitions G_f are returned, $s_{\text{SKG}}(G, P, \Omega) = \bigcup_{f \supseteq f'} G_f$ defines the union of *all* these partitions $G_f \subseteq G$ such that $f' \subseteq f$ which are sent to the client,
- Φ: only $n = \infty$ is admissible, i.e., no paging is supported since the union of all relevant partitions is returned – unlike SPF an over-estimation representing all *subgraphs relevant to a star-shaped subquery*

An SKG client hence decomposes BGPs into families f' of star-shaped subqueries – on an abstract level, discarding variables or concrete bindings – and fetches via this API the subgraphs G_f (that are available in compressed form on the server) matching f'; single non-star triples in the BGP are retrieved via TPF and joins between star-shaped subqueries, and single triple queries are then computed on the client-side. Evaluations [6] show that this approach is highly competitive for many concurrent clients due to its low server and (due to partition compression also) network footprint. As for Φ, note that it would not make sense to decompose family-based partitions into chunks since chunking up the HDT-compressed partitions would require decompression.

2.2 RDF HDT Compression

It is worthwhile to also explain the HDT [10] binary compression format for RDF datasets that is used "under the hood" in all of the previously mentioned interfaces, namely (br)TPF, SaGe, SKG, and SPF, as well as in our novel approach presented in this paper. HDT offers efficient search and retrieval over the compressed RDF graphs without the need for decompression and offering query relevant statistics directly in its metadata. The main compression idea relies on ordering triples by *SPO*, grouping repetitive RDF terms. An HDT file could be viewed as a compressed, directly queryable *SPO*-ordered index. In addition, HDT provides a compressed binary utility index built upon loading time covering *OPS PSO* to achieve a high performance for resolving any SPARQL triple pattern. TPF and SPF rely on an HDT of the whole graph G to evaluate triple and star patterns on the server with a low computation footprint, whereas SKG profits from the compression also lowering the network footprint when shipping family partitions G_f.

3 MOTIVATING EXAMPLE

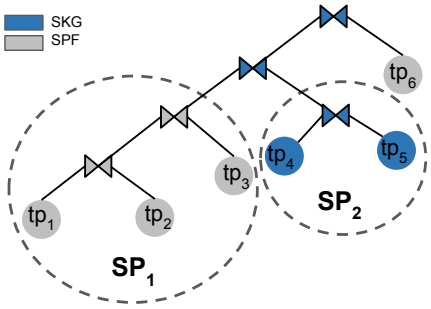
All thus far described KG APIs alone suffer from an imbalanced load on either client-side (dumps, TPF, SKG) or server-side (SPARQL endpoints, SaGe, SPF). In this paper, we therefore advocate that, based on decomposing BGPs into star-shaped subqueries and characteristics of these subqueries (e.g., selectivity and intermediate result cardinality estimation), we can optimally distribute the query processing load between client and server. Hence, given statistics as well as information about the current server workload and the client's capabilities, we can pick the best suited KG API.

In particular, the factors that our cost model considers are server load, client computing resources, and the number/size of intermediate results to be transferred over the network (in combination with available bandwidth), since several sources [1, 6, 13, 21, 22] identified these as important dimensions when accessing KGs.

To elaborate, let us consider query Q given in Figure 1a. All triple patterns of Q have quite large cardinalities, meaning that both single pattern interfaces (TPF, brTPF) would need to send enormous numbers of requests to the server and ship large intermediate results to the client when processing the query.

For both star-based interfaces (SPF and SKG), the query would be decomposed into two stars and a single triple pattern: $sp_1 = \{tp1, tp2, tp3\}$, $sp_2 = \{tp4, tp5\}$, and $tp6$. sp_1 has 89,366 solution mappings, and sp_2 has 600,349 solution mappings. Both, SKG and SPF would estimate the result sizes of star patterns and, in essence, order the query execution plan accordingly to (sp_1, sp_2, tp_6), i.e., starting with sp_1. SKG ships a partition containing 1,628,572 stars in total leading to excessive data transfer even though the partition is HDT-compressed. SPF, on the other hand, only ships the 86,366 stars that actually match sp_1, resulting in less of a network overhead and faster query processing. However, in order to process the join between sp_1 and sp_2, SPF's client join processor would batch the 89,366 bindings into groups of 30 bindings each, sending one request per batch, amounting to 2,979 requests. This overhead could be conveniently mitigated by instead shipping the compressed partition for sp_2 and joining on the client: this example illustrates how a combination of SPF's server-side star evaluation with SKGs partition shipping could outperform either approach alone. Moreover, note that in case of a high server workload, the additional network overhead for transferring the partition for sp_1 might still be affordable, compared to server-side SPF processing of sp_1 using the overloaded server.

```
select * where {
  ?album dbo:artist ?artist . # tp1: 146,716 matches (sp1)
  ?album rdf:type dbo:Album . # tp2: 147,917 matches (sp1)
  ?album dbo:releaseDate ?date . # tp3: 212,290 matches (sp1)
  ?artist dbo:genre ?genre . # tp4: 576,000 matches (sp2)
  ?artist foaf:name ?name . # tp5: 4,146,579 matches (sp2)
  ?song dbo:writer ?artist . # tp6: 200,969 matches
}
```



(a) Show artists' albums, genres, and the songs they have written **(b) Query execution plan for $(sp_1^{SPF}, sp_2^{SKG}, tp_6^{SPF})$**

Figure 1: Example of processing a SPARQL query with WiseKG

4 WISEKG

In the spirit of the example presented in Section 3, WiseKG enables to leverage (i) the characteristics of the star-shaped subqueries as well as (ii) information on the currently available client and server resources, to estimate the cost of processing each star-shaped subquery on the client (using SKG) or on the server (using SPF), – choosing the most efficient execution strategy dynamically.

4.1 Overview

WiseKG employs a dynamic cost model to determine an *annotated query plan*: in order to denote query execution plans (cf. Section 2) with particular interfaces to be used per subquery, we will use superscripts *SPF* and *SKG*, i.e., for our example the annotated plan $\Pi = (sp_1^{SPF}, sp_2^{SKG}, tp_6^{SPF})$. In case of the example in Figure 1b this would mean that sp_1 is evaluated via SPF on the server, sp_2 is executed using SKG on the client, and the resulting bindings from joining both are given as input Ω to a call of tp_6 executed again using SPF on the server[7].

Upon receiving a BGP P from the client, the WiseKG server will decompose it into star-shaped subqueries, and use its cost-model to create an annotated query plan Π that is returned to the client, along with a timestamp τ denoting plan expiry. The client then, in the order specified by the server, executes Π using the APIs specified in the plan annotations. In case the execution is not completed by τ, the client needs to request a new annotated plan, which may look different – as mentioned before and illustrated in the example, the choice of API per subquery taken by the server may depend on its current load, as discussed in the following.

Formally, the WiseKG server API offers the following interface calls access KG G:
an SPF LDF API control $SPF(P, \Omega)$ returning $s_{SPF}(G, P, \Omega)$,
an SKG LDF API control $SKG(P, \Omega)$ returning $s_{SKG}(G, P, \emptyset)$[8],
an execution plan interface $Plan(P)$ returning a $pair(\Pi_P, \tau)$. We will use the notation $c(P, \Omega)$ to denote that a (star-shaped) subpattern P is executed by a control $c \in \{SPF, SKG\}$ – in the spirit of LDF, we expect also other (hypermedia) controls to be callable in addition to *SPF* and *SKG* in the future. Further, in this paper we assume that the call to $c(P, \Omega)$ on the client side is converted to a set of bindings through a function $eval_c(P, \Omega) = \Omega \bowtie [[P]]_G$. Note that, depending on whether the underlying selector function of $c(P, \Omega)$ is already accepting bindings, directly returning $\Omega \bowtie [[P]]_G$ (such as for SPF) or only returning a graph of which $[[P]]_G$ can be computed and then joined with Ω on the client (such as for SKG), $eval_c$ incurs more or less work on the client side.

$Plan(P)$ maps a BPG P to an annotated plan Π_P along with the expiry timestamp $\tau = \tau_C + \iota$, where τ_C corresponds to the current time, and ι is a fixed time quantum per query[9]. Π_P is constructed from $\mathcal{S}(P)$ by (i) identifying the best join amongst stars based on cardinality estimations and (ii) determining, based on factors such as the current load on the server and the estimated network/processing cost, the best interface (SPF or SKG) per subquery. Before we explain (server and client) query processing in more detail (cf. Section 5), we first present the server cost model, which is used to make this latter choice.

4.2 Server-Side Cost Model

In this section, we present WiseKG's server cost model used to determine the choice between client-side evaluation using SKG or server-side evaluation using SPF. The cost model is inspired by the classic R^* optimizer [18] from the field of distributed databases [18, 31]. In the R^* model, the total time is the sum of four time components (CPU processing, messaging, data transfer, and I/O) that can be estimated for a query Q as:

$cost(Q) = processing + Messaging + data\ transfer + I/O$

Following the R* model, we consider, in our client-server architecture, the following components to approximate the total time consumed by client and server to process a star subquery: estimated number of CPU instructions (#*CPU*), estimated number of I/O operations (#*IO*), as well as two communication cost components – estimated number of requests (#*M*) and estimated number of transferred bytes (#*BYT*) over the network per query. WiseKG's cost model for a given star subquery is then defined as

$$cost(sp) = \underbrace{W_{CPU} \times (\#CPU)}_{\text{Processing}} + \underbrace{W_{MSG} \times (\#M)}_{\text{Messaging}} + \underbrace{W_{BYT} \times (\#BYT)}_{\text{Data transfer}} + \underbrace{W_{IO} \times (\#IO)}_{\text{I/O}} \quad (2)$$

[7] Note that for triple patterns, SPF is equivalent to brTPF so we can use the SPF interface also for single triple patterns.
[8] Note that SKG does not allow to ship bindings, cf. Section 2.
[9] Somewhat similar to/inspired by SAGE's[19] query suspension timeouts.

where the weights W_{CPU}, W_{MSG}, W_{BYT}, and W_{IO} help estimating the time required by the client and server hardware configuration to perform a CPU instruction, the time required to send an (HTTP) request message from a client to a server over the network, the time required to transfer one byte from a server to a client over the network, as well as the time required for a disk I/O operation. It is important to note that WiseKG's server optimizer is tailored to embed dynamic factors to reflect the current server load. These weights are estimated as follows:

W_{CPU}: We estimate time per CPU instruction as the inverse of the CPU's IPS (Instructions per second) rate, damped by the current CPU load in percent[10]:

$$W_{CPU} = \frac{1}{IPS \times (100\% - CPU_{usage})}$$

W_{MSG}: The average time to transmit an HTTP request from a client to the server. In our experiments and network setup, similar to SaGe's experiments [19], we assume as a constant value of $W_{MSG} = 50ms$ for all clients. In a real world scenario, we would measure this delay based on an initial HTTP request per client.

W_{BYT}: We estimate W_{BYT} by the conservative minimum between the available server bandwidth bw_{serv} (which we estimate as the difference between bandwidth of the server network card reduced by the average data transfer over the network in the last 1 minute, again checking every second) and the client bandwidth bw_{client}, which we estimate as $20Mb/sec$ in our setup, similar to [6]. This way, W_{BYT} takes into account the current network usage of concurrent clients. In our experiments, .

$$W_{BYT} = \frac{1}{Min(bw_{client}, bw_{serv})}$$

W_{IO}: We measure I/O in terms of loading chunks of 1MB from disk, i.e., we estimate W_{IO} as the time required to read 1MB to the memory. In WiseKG, the I/O times differ per chosen API: for SPF, a single HDT file of the entire graph G is used and mapped into memory while auxiliary bitmap indexes remain in memory to help localize potential mapping solutions (using approx. 3% of the entire HDT file altogether [10]). Thus, the I/O time accounts for transferring non-cached blocks that might contain the mapping solutions to memory. In SKG, the I/O time is due to the server reading HDT *partitions* from disk in order to ship those to the client; on the client side, we assume processing continues in memory, thus not involving further I/O operations.

We note that our experiments have shown that in fact I/O is a negligible factor in our setup; for both SPF and SKG (we perform a respective experiment with a stress-testing workload described in Section 6.1), we verified that the amount and difference in I/O times in both approaches was dwarfed by the communication costs. Therefore, we leave out this factor in our cost estimation model ($W_{IO} = 0$).

The final time cost estimates of client-side SKG evaluation based on shipped partitions vs. server-side SPF evaluation of star patterns are given in Definition 3 and Definition 4. For a query BGP P, these costs are estimated for each star pattern $sp \in \mathcal{S}(P)$.

Definition 3 (Cost of SKG Star Pattern Evaluation). *Given a star pattern $sp \in \mathcal{S}(P)$ and a plan Π_P, as well as the set of families $F_{sp} = \{f \in F(G) \mid f \supseteq pred(sp)\}$ relevant for sp in G, the cost in time of evaluating sp using SKG is estimated as follows:*

$$cost_{SKG}(sp, \Pi) = W_{CPU_{client}} \times \underbrace{card(sp, \Pi) \times i_t}_{\#CPU} + W_{MSG} \times \underbrace{|F_{sp}|}_{\#M} + W_{BYT} \times \underbrace{\left(\sum_{f \in F_{sp}} size(f) \right)}_{\#BYT} + W_{I/O} \times \underbrace{\left(\sum_{f \in F_{sp}} size(f) \right)}_{\#IO}$$

Definition 4 (Cost of SPF Star Pattern Evaluation). *Given sp, Π_P, and F_{sp}, the cost in time of evaluating sp using SPF is estimated as follows:*

$$cost_{SPF}(sp, \Pi) = W_{CPU_{serv}} \times \underbrace{card(sp, \Pi) \times i_t}_{\#CPU} + W_{MSG} \times \underbrace{\frac{card(sp, \Pi)}{\Phi(n)}}_{\#M} + W_{BYT} \times \underbrace{card(sp, \Pi) \times b_t}_{\#BYT} + W_{IO} \times \underbrace{size(G)}_{\#IO}$$

Definitions 3 and 4 use the following functions and variables:

- $card(sp, \Pi)$ returns an estimated result cardinality for evaluating star pattern sp using an estimation of the number of bindings for previously evaluated star patterns in Π. This estimate (based on statistics about the sizes of subgraphs per characteristic set) is described in [23].
- $size(\cdot)$ is either the size of an HDT file (plus index) for a partition corresponding to a family $f \in F(G)$ or, for $size(G)$ the size of the HDT file for the entire graph G[11].
- i_t is the number of CPU instructions needed to process each triple in the result set. In general, we rely on HDT algorithmic costs which are sub-linear and close to constant for most operations. [10]; we only measured one millisecond (or at most a few milliseconds) in our experiments. We therefore set this factor to $i_t = 1$. Different IPS rates in the server and client are considered in the different weights: $W_{CPU_{serv}}$ and $W_{CPU_{client}}$.
- b_t is the average number of bytes per triple in the result; we estimate this factor by averaging the size of the triples in each family partition.

5 QUERY PROCESSING

In this section, we detail how the WiseKG 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.

[10] We estimate this current CPU load as the average percentage of CPU_{usage} in the previous minute (checking every 1sec). Note that for our experiments we only compute this CPU usage on the server side, i.e. for $W_{CPU_{serv}}$, whereas for $W_{CPU_{client}}$ we assume $CPU_{usage} = 0$, i.e., full availability of client resources.

[11] Note that SPF relies on a single HDT for G whereas SKG only transfers the HDT files corresponding to F_{sp}.

5.1 Server-Side Query Processing

Since the server-side processing of star-shaped subqueries in SPF and SKG APIs running on the server are explained in detail in [6] and [1], we mainly focus on the creation of the annotated execution plan in this section: when the WiseKG server receives a $Plan(P)$ request for a BGP P, it creates a query execution plan specific to P, which it returns along with the expiry timestamp τ to the client for execution; the resp. algorithm to compute $Plan(P)$ is shown in Alg. 1.

Algorithm 1: Create an annotated query execution plan

Input: $P = \{tp_1, tp_2, \ldots, tp_n\}$ // a BGP
Output: (Π_p, τ) // an annotated plan and its expiry time

1 **function** $Plan(P)$
2 $S \leftarrow \mathcal{S}(P)$
3 $\Pi_p \leftarrow ()$
4 **while** $S \neq \emptyset$ **do**
5 **for** $sp \in S$ **do**
6 $cnt_{sp} \leftarrow card(sp, \Pi_p)$
7 **if** $cnt_{sp} = 0$ **then**
8 **return** ()
9 $sp_i \leftarrow sp$ where $sp \in S$ and $cnt_{sp} \leq cnt_{sp'}$ for all $sp' \in S$
10 **if** $cost_{SPF}(sp_i, \Pi_P) \leq cost_{SKG}(sp_i, \Pi_P)$ **then**
11 $\Pi_P \leftarrow append(\Pi_P, (sp_i^{SPF}))$
12 **else**
13 $\Pi_P \leftarrow append(\Pi_P, (sp_i^{SKG}))$
14 $S \leftarrow S \setminus \{sp_i\}$
15 $\tau \leftarrow \tau_C + \iota$
16 **return** (Π_P, τ)

The first step is to decompose the query into star-shaped subqueries (line 2). To create the execution plan, we find the star-subquery with the lowest cardinality estimation (line 5-8) and add it to the plan; when we find a query with an empty result (e.g. in case no matching family partition exists [6]), we can stop since the final result will then also be empty. The star pattern with the lowest cardinality estimation is selected first (line 9), thus overall in the final plan, patterns are ordered by estimated cardinality.

Then, the estimated costs for SPF and SKG are compared in Line 10; depending on the cost models from Section 4.2, each subquery is annotated with the resp. control for evaluating the star pattern on the server, i.e., *SPF* (line 11) or the client *SKG* (line 13). Here, the *append* function just appends the annotated star pattern to the end of the plan. When there are no more subqueries left in the star decomposition, the algorithm returns the plan (line 16) after computing the expiry timestamp (line 15).

For the query Q shown in Figure 1a, this algorithm could compute the execution plan in the join order visualized in Figure 1b (unless the server load is too high, in which case SP_1 could also potentially be suggested to be executed using SKG).

Finally, as a side note, we note that, based on the fact that not all family partitions in SKG are necessarily materialized on the server – SKG does not materialize HDT files over a certain partition cardinality threshold (for details, cf. [6, Section 4.1]); in such cases the concrete implementation of Alg. 1 defaults to SPF, i.e., server-side evaluation of the resp. star pattern, independent of the cost.

5.2 Client-Side Query Processing

Processing queries on a WiseKG client relies on an approach similar to the one presented in [1], which we adapt herein to accommodate for client-side processing of HDT shipped family partitions. In the following, we describe the basic ingredients that the client needs to process full SPARQL queries: WiseKG is able to process full SPARQL queries including operators such as `UNION` and `OPTIONAL`, `FILTER`, etc.,[12] which are all evaluated on the client-side. Herein, we only focus on the BGP evaluation part.

The general approach for processing BGPs P is as follows:

(1) Retrieve the query execution plan and time quantum for P from the server by calling $Plan(P) = (\Pi_P, \tau)$.
(2) For each star pattern $sp^c \in \Pi_P$ with control $c \in \{SPF, SKG\}$ in Π_P and solution mappings from previously evaluated operators Ω, iteratively do the following:
   (a) If $\tau < \tau_C$, i.e., the plan has expired, the client requests a new execution plan/expiry based on the remainder of P that has not yet been processed.
   (b) Otherwise we call the interface $c(sp, \Omega)$ and convert it to a set of bindings using $eval_c(sp, \Omega)$, which as mentioned above, in the case of $c = SKG$ involves client-side evaluation of the star-shaped pattern on the shipped HDT, whereas SPF directly returns the result bindings.

The exact algorithm implementing these steps in a recursive manner is shown in Alg. 2.

Algorithm 2: Processing a Query Execution Plan

Input: $\Pi = (sp_1^{c_1}, \ldots, sp_n^{c_n})$ // an execution plan;
τ // expiry timestamp;
Ω' // a set of bindings
Output: Ω // set of solution bindings

1 **function** $evalPlan(\Pi, \tau, \Omega)$
2 **if** $\tau < \tau_C$ **then**
3 $(\Pi, \tau) \leftarrow Plan(BGP(\Pi))$
4 **if** $\Pi = sp^c$ **then**
5 $\Omega \leftarrow eval_c(sp, \Omega')$
6 **else**
7 $\Omega \leftarrow evalPlan((sp_1^{c_1}, \ldots, sp_{n-1}^{c_{n-1}}), \tau, \Omega')$
8 $\Omega \leftarrow evalPlan(sp_n^{c_n}, \tau, \Omega)$
9 **return** Ω

Line 2 checks whether the plan has not yet expired; in that case, the algorithm calls $Plan(\Pi)$ to reevaluate the plan on the server (line 3)[13]. The way this is currently done can be understood as follows: assuming the originally requested plan is $(sp_1^{c_1} \ldots sp_i^{c_i} \ldots sp_n^{c_n})$ and the client reaches τ at step i. Then the client will restart calling $Plan(\{sp_i, \ldots sp_n\})$ receiving a new plan $\Pi_{\{sp_i, \ldots sp_n\}}$ upon which it continues; obviously this could change the interface choices per star for the remaining plan, based on the current server load situation.

[12] with the exception of GRAPH query patterns, since HDT does not support named graphs.
[13] Here, $BGP(\Pi)$ denotes the corresponding (non-annotated) BGP for plan Π.

Continuing on Alg. 2, in case the plan is associated with a single star pattern sp (line 4), we call the control $c \in \{SPF, SKG\}$ to retrieve the output plan and obtain the output solution mappings (line 5). Otherwise, the algorithm will make a recursive call for the left subtree (line 8) the resulting bindings of which are handed over to the call of the right subtree (line 9).

6 EXPERIMENTAL EVALUATION

In this section, we compare the performance of WiseKG with the state of the art SPARQL query processing interfaces.

6.1 Experimental Setup

In this section, we describe the experimental setup, including the systems we compare against, datasets, queries, and hardware and software configurations.

Implementation details. We implemented both WiseKG client and server in Java[14] extending the TPF implementations[15] so that we ensure comparability and compatibility with the spectrum of Linked Data Fragment (LDF) approaches including TPF, SPF, and smart-KG. The WiseKG server relies on SPF star pattern fragments for server-side processing of star-subqueries. Furthermore, the WiseKG server adopts the family generator component from smart-KG [6] to generate, manage, and store the HDT files of the family-based partitions. In our server-side cost model, we depend on a cross-platform operating system and hardware information library for Java[16] to retrieve system information about clients and the server resources usage including network and CPU usage. The WiseKG client implements a pipeline of nested iterators similar to brTPF and SPF client implementations.

Configuration. To assess the performance of our system under different loads, we perform experiments over eight configurations with 2^i clients ($0 \leq i \leq 7$) issuing queries concurrently for each configuration (up to 128 concurrent clients). Each concurrent client executes one query at a time, i.e., at most 128 queries are executed at the same time.

Datasets. We use three different sizes of the Waterloo SPARQL Diversity Benchmark (WatDiv) [4] to test the scalability of our approach: 10M, 100M, and 1B triples. In addition to these, we also use the real-world dataset DBpedia [16] (v.2015A). The characteristics of the evaluated RDF graphs are described in Table 1.

Table 1: Characteristics of the used datasets

Dataset	#triples	#subjects	#predicates	#objects	#families
watdiv10M	10,916,457	521,585	86	1,005,832	21,210
watdiv100M	108,997,714	5,212,385	86	9,753,266	37,392
watdiv1B	1,092,155,948	52,120,385	86	92,220,397	52,885
DBpedia	837,257,959	113,986,155	60,264	221,623,898	29,965

Queries. We consider three different query workloads for the WatDiv datasets: (i) a *basic testing* workload named `watdiv-btt` that consists of queries obtained from WatDiv basic testing templates[17]. Each client has a set of 20 queries including star queries (S), linear queries (L), snowflake queries (F), and complex queries (C); and (ii) a diverse *stress testing* workload named `watdiv-sts` that consists of queries obtained from the WatDiv stress-testing suite[4]. Each client has a set of 154 non-overlapping queries. In addition to these workloads, we randomly selected 16 queries from a real-world LSQ query log [27]; plus, we included 12 queries used to evaluate smart-KG [6].

Compared Systems. To test the effectiveness of dynamically shifting star-subquery processing between client-side and server-side based on the status of server-side resources disregarding the cost model defined in Section 4.2, we implemented a version of WiseKG named WiseKG$_{heuristic}$ that relies on more straightforward heuristics. Initially, WiseKG$_{heuristic}$ executes all star subqueries on the server side up to a predefined CPU usage threshold σ. When the threshold is reached, WiseKG$_{heuristic}$ produces an execution plan exclusively based on shipping family partitions. In addition, we evaluate WiseKG$_{cost}$, our main contribution, which is a version of WiseKG that relies on the cost model described in Section 4.2. Note that we use the recommended versions of both server and client for all the evaluated systems including Star Pattern Fragment (SPF) [1], smart-KG [6], SaGe [19], and Triple Pattern Fragments (TPF) [30].

Hardware configuration. We ran all 128 clients concurrently on a virtual machine with 128 2.5GHz vCPU cores, 64KB L1 cache, 512KB L2 cache, 8192KB L3 cache, and 2TB main memory. To ensure an even distribution of the resources between the clients, we limited each client (for all approaches) to run with a single vCPU core and 15GB main memory. WiseKG and all the compared system servers were run on the same server with 32 3GHz vCPU cores, 64KB L1 cache, 4096KB L2 cache, 16384KB L3 cache, and 128GB main memory. Clients and servers are located on the same 1 GBit network. In order to emulate a more realistic bandwidth scenario, we limited the network speed of each client to 20 MBit/sec.

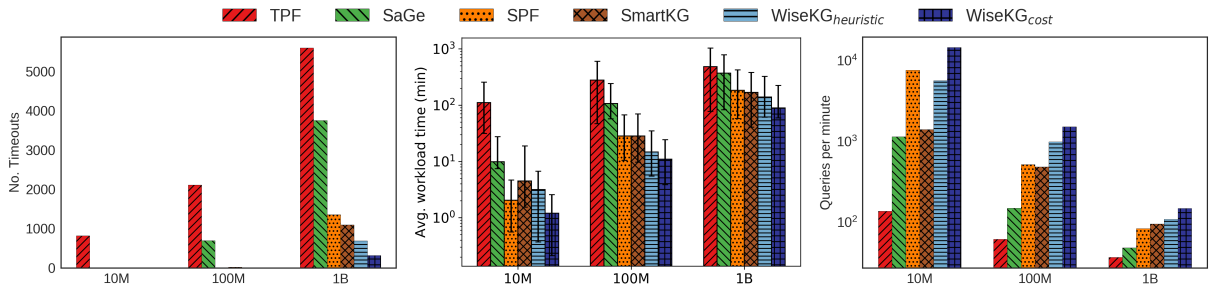
Evaluation metrics.

- **Timeouts**: number of queries that exceed the timeout.
- **Workload Completion Time**: the total time required by a client to complete a workload.
- **Query Execution Time**: the average time it takes to complete a query.
- **Server CPU load**: the average percentage of server CPU usage during the execution of a query workload.
- **Number of Requests made to the Server**: the number of requests a client sends to the server.
- **Number of Transferred Bytes**: the number of bytes transferred between server and client, i.e., the sum of both directions.

Software configuration. Following the experiments performed in [1, 6, 19], we used a timeout of 300 seconds, i.e., 5 minutes, for all approaches. That is, after 5 minutes we suspend the query execution. The page size $\Phi(n)$ for TPF, SPF, and WiseKG was set to $n = 100$ (as in [1, 30]) and the maximum number of bindings attached to a request for SPF and WiseKG was set to $|\Omega| = 30$ as it was in [1]. In order to assess our approach against the others using as similar as possible configurations, we set the time quantum ι to the same value as the overall timeout for all systems, i.e., 5 minutes.[18]

[14] https://github.com/WiseKG/WiseKG-Java
[15] https://github.com/LinkedDataFragments/Server.java
[16] https://github.com/oshi/oshi
[17] https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml

[18] In our current setup and evaluation covering widely used benchmarks in the area, the expiry timestamp was hardly reached. While we already significantly outperform all state-of-existing approaches, we still deem the addition of a plan expiry needed both conceptually (as the system resources change dynamically over time and our



(a) Number of Timeouts **(b) Avg. Workload Time** **(c) Queries per minute**

Figure 2: Number of timeouts, average workload time, and throughput for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` on `watdiv-sts`

6.2 Experimental Results

Due to space restrictions, this section focuses on the most important results. All results, incl. additional experiments, details on the implementation and configurations used in the experiments (datasets and queries) are available online[19].

System Performance Evaluation. In this part of the evaluation, we focus on analyzing the behavior of the compared systems in the scenario of increasing KG size with the highest number of concurrent clients (128 clients) using the `watdiv-sts` workload. As shown in Figure 2, WiseKG$_{heuristic}$, the vanilla version of WiseKG, performs significantly better than the state-of-the-art systems in terms of performance and scalability, not to mention WiseKG$_{cost}$ (just WiseKG hereafter) has even surpassed WiseKG$_{heuristic}$.

Figure 2a shows that WiseKG produces no timeouts over the `watdiv10M` and `watdiv100M` datasets for 128 concurrent clients. Moreover, even in the case of `watdiv1B`, WiseKG only incurs 2% timeouts of the total workload queries. In contrast, none of the compared systems was able to process all queries with a 5-minute timeout, except SPF and SaGe on the `watdiv10M` dataset. When queries are executed over the `watdiv1B` dataset, the percentages of timeouts reach 13% and 21% for smart-KG and SPF, respectively. For SaGe and TPF, the percentages of timeouts increase up to 55%. These results confirm the superior scalability of WiseKG compared to state-of-the-art systems. These experiments show that even for a high number of clients, WiseKG is able to handle large scale KGs.

Figure 2b shows that the average workload completion time including queries that timed out. WiseKG is up to 4 times faster than SPF and smart-KG, and up to an order of magnitude faster than SaGe and TPF over `watdiv1B` with a load of 128 concurrent clients. In addition, Figure 2b also shows that SPF and smart-KG have comparable average workload time. smart-KG performs slightly better for `watdiv100M` and `watdiv1B` datasets. This is not surprising since they similarly rely on star decomposition; SPF executes the star subqueries on the server side while smart-KG ships the relevant partitions for the subqueries and executes them on the client. Compared to SPF and smart-KG, WiseKG provides a significant performance improvement as a result of the proposed cost model that optimizes query processing by leveraging the subqueries' cardinality estimation as well as available client and server resources to determine an efficient execution plan. To provide a comprehensive evaluation, we also include TPF and SaGe in our experiments. As shown in Figure 2, our experiments confirm a previous study [6] that SaGe performs far better than TPF for small datasets. However, when dataset size increases and the number of concurrent clients is high, the difference between TPF and SaGe becomes less visible. Note that we did not include a SPARQL endpoint (e.g Virtuoso) in our experiments, since several previous studies [1, 6, 19, 30] have already shown that SPARQL endpoints are not able to scale well with an increasing number of clients.

We compare the performance of WiseKG to state-of-the-art interfaces considering real-world queries on DBpedia. Figure 3 presents the execution times of these 28 queries for all systems. The results confirm that WiseKG significantly outperforms the compared systems for the real-world queries. Figure 3 shows that TPF is the slowest or the second to slowest in all queries. On the one hand, smart-KG suffers from excessive delays in queries that require non-materialized partitions such as Q2, Q4, Q8, Q12, Q15, Q19, Q21 and Q25 since, in this case, smart-KG depends on TPF in addition to queries with high selectivity such as Q6, Q16, Q20, and Q26 as it is more resource-efficient to process on the server-side. On the other hand, SPF has a robust performance in most of the queries due to its efficient server-side star pattern execution, except the queries with low selectivity such as Q24 and Q28 due to the excessive transfer of intermediate results. Moreover, SaGe has worse performance than WiseKG for the less selective queries with large intermediate results, such as Q7 and Q28, due to these queries putting more load on the server and incurring more requests to the server. The queries where SaGe has slightly better performance than WiseKG, such as Q2 and Q4, are generally queries where the overhead of computing the execution plan for WiseKG is a considerable part of the overall execution time (i.e, very simple queries).

Finally, WiseKG$_{heuristic}$ is faster than WiseKG for the queries with execution time less than 0.1 seconds. This is because WiseKG has the overhead of computing the best query plan.

Performance evaluation on different query shapes. In this part of the evaluation, we analyze the effect of the query shapes on the performance of the systems. We use queries of 4 different shapes including linear (L), star (S), snowflake (F), and complex

model needs to consider the current "promises" it made to clients) and useful for future workloads on larger knowledge graph.

[19] https://github.com/WiseKG/WiseKG-Experiments

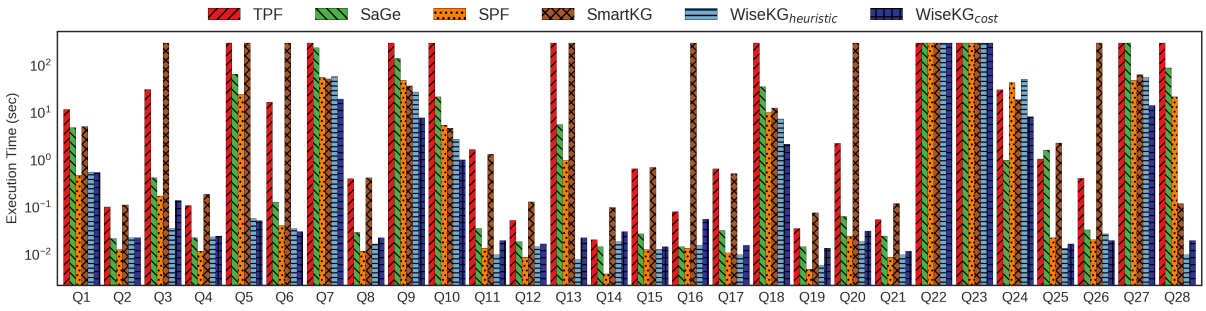
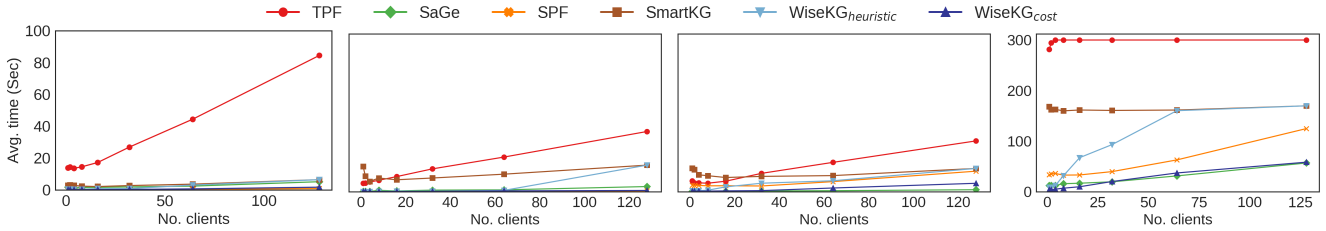


Figure 3: Execution time (in seconds) for 28 diverse queries over the `dbpedia` dataset.



(a) L1-L5 **(b) S1-S7** **(c) F1-F5** **(d) C1-C3**

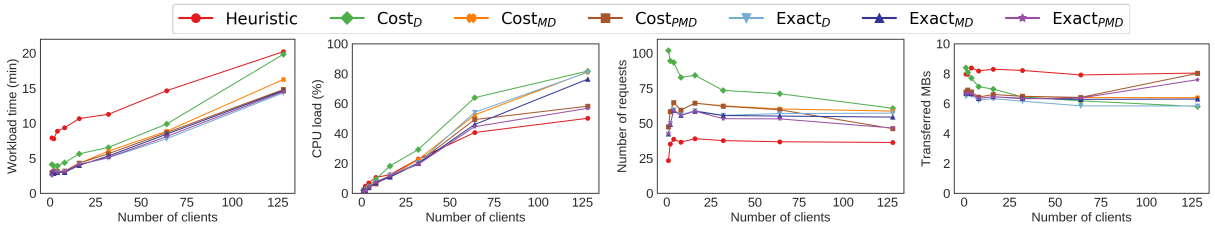
Figure 4: Avg. execution time per client over `watdiv100M` for the `watdiv-btt` workload.

(C) queries. These queries are part of the `watdiv-btt` workload and executed against `watdiv100M`. The `watdiv-btt` queries were executed in a different (random) order on each client and the results were recorded as the overall execution time per query shape averaged across all clients. Figure 4 shows the average query execution time for each shape.

In compliance with the system performance analysis, WiseKG outperforms all state-of-the-art systems for all different query shapes. For the L-workload, all systems have a similarly efficient performance since this workload includes the simplest queries with a small diameter. As shown in Fig. 4b, SPF provides an excellent performance for S-workload – as expected since it is optimized for star queries with high selectivity. On the other hand, smart-KG performs worse than SPF since it sends an entire partition with unnecessary intermediate results for such queries. In general, SaGe has an outstanding performance for all query shapes, especially for the F-workload as shown in Fig. 4c. This is due to the fact that the `watdiv-btt` workload includes only 20 queries per client (i.e low query arrival rate) and we use a medium-size `watdiv-100M` dataset for this experiment.

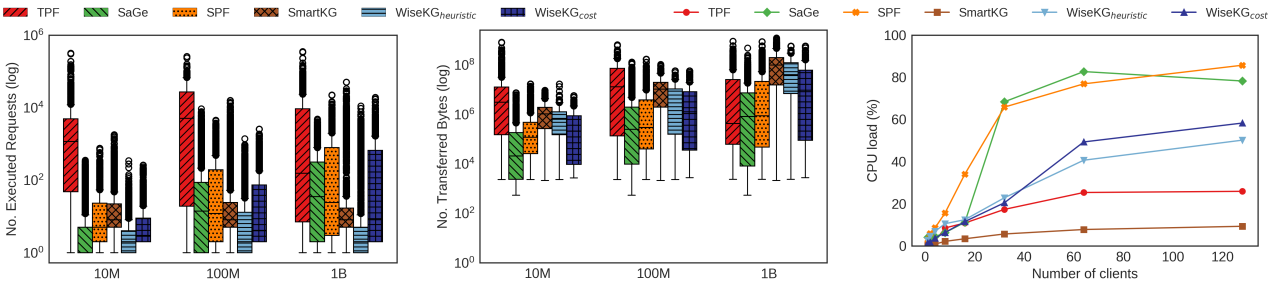
Fig. 4d shows that the behavior of the compared systems dramatically changes for the C-workload. For instance, WiseKG significantly outperforms state-of-the-art interfaces, even SaGe in the single client configuration. SaGe starts ahead of smart-KG up to 16 clients, then smart-KG performs better with higher numbers of concurrent clients. SPF suffers excessive delays in C1 since the query includes 3 stars that have intermediate results with high cardinalities. For query C2, SaGe outperforms all the compared systems. In contrast, smart-KG and TPF are significantly worse (both time out) than SPF due to SPF's better handling of triple patterns with large cardinalities by shipping bindings along with star-shaped subquery requests. Interestingly, although WiseKG$_{heuristic}$ times out in C2, WiseKG was able to efficiently perform the query with a slightly higher average time compared to SaGe. This is due to the accurate estimations of the cost model. Finally, for C3, though SPF and smart-KG are optimized for star queries, e.g., C3 is a single unbounded star, WiseKG is up to three times faster with 128 clients.

Impact of cost model components. We performed an experiment with several different configurations of the cost model over `watdiv100M` on the `watdiv-sts` workload in order to evaluate the impact of the cost model components on WiseKG query performance and resource consumption. To measure the impact of the cost model components, we configured three different versions of WiseKG including data transfer component only ($Cost_D$), data transfer and messaging components ($Cost_{MD}$), and finally, a version with processing, messaging, and data transfer components ($Cost_{PMD}$). For this experiment, we used WiseKG$_{heuristic}$ as baseline. Figure 5a shows that for the configuration with 128 clients $Cost_{PMD}$ improves the average workload completion time (14 min) compared to $Cost_D$ and $Cost_{MD}$ (19*min* and 16*min*, respectively). In addition, Figures 5b and 5c show that $Cost_{PMD}$ requires on average less CPU usage and number of requests than $Cost_D$ and $Cost_{MD}$. This is due to the fact that the $Cost_{PMD}$ configuration includes the processing component which significantly contributes to lowering the CPU load on the server. Although $Cost_D$ has the lowest transferred data compared to the rest of the configurations, $Cost_D$ is the slowest configuration. The reason for this behavior is that it does not take into account the HTTP request latency, which is an important factor to determine the incurred latency especially, in subqueries that require high numbers of result pages. It is important to note that all the configurations remain faster than WiseKG$_{heuristic}$, and



(a) Avg. Workload Time **(b) CPU Usage** **(c) Avg. requests per query** **(d) Avg. data transfer per query**

Figure 5: Impact of the cost model components on the performance and resources consumption over `watdiv100M`



(a) Number of requests to the server for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` (log).

(b) Number of transferred bytes for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` (log).

(c) Avg. Server CPU Usage (in percentage) for increasing numbers of clients over `watdiv1B`.

Figure 6: Number of requests to the server and number of transferred bytes for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B`, and CPU load for increasing numbers of clients over `watdiv1B` on the `watdiv-sts` workload

since WiseKG$_{heuristic}$ is faster than all the state-of-the-art systems (Figure 2), so are all the configurations.

Moreover, to evaluate the impact of using *characteristic set* [23] as a cardinality estimation method on the cost model components, we replaced the cardinality estimation function in the WiseKG configurations described earlier with the true cardinality, creating the configurations $Exact_D$, $Exact_{DC}$, and $Exact_{PMD}$, respectively. Figures 5b, 5c, and 5d show that $Exact_D$ and $Exact_{DC}$ provide faster performance and better resource utilization compared to their peers with cardinality estimation $Cost_D$ and $Cost_{PMD}$. Figure 5a shows that the configurations with the true cardinality have a comparable workload execution time ($\approx 14min$). This performance is similar to the performance of $Cost_{PMD}$ even though $Exact_{PMD}$ has a lower resource consumption.

Finally, our experimental results show that relying on characteristic sets as a cardinality estimation method provides a comparable performance to the configurations with the true cardinality – demonstrating a very subtle impact of the cardinality miss-estimates on the overall performance of WiseKG. We plan to investigate diverse cardinality estimators as future work in order to explore the impact of different cardinality estimation techniques on WiseKG query execution time [17, 24].

Resource consumption. In this part of the evaluation, we focus on the server resource usage including network and CPU consumption.

We report two main metrics to demonstrate the network traffic: the number of requests sent to the server (NRS) and the number of transferred bytes between client and server (NTB). Figures 6a and 6b show the distribution of the number of requests to the server per query as well as the distribution of the number of transferred bytes per query, with 128 concurrent clients on increasing KG sizes (`watdiv10M`, `watdiv100M`, and `watdiv1B`) for the `watdiv-sts` workload. As expected, TPF incurs the highest number of requests and the data transfer leading to a substantial increase in network load. Even though smart-KG relies on TPF to execute singular triple patterns and star patterns with no materialized partition, smart-KG significantly reduces the number of requests compared to TPF since it only sends a single request per star pattern. Figure 6a also demonstrates that WiseKG requires the lowest average number of requests among all systems due to three main reasons: first, WiseKG potentially reduces the number of requests required based on the communication component in the cost model which can be observed in the difference between the number of requests $Cost_D$ and $Cost_{DC}$ as shown in Figure 5c; second, WiseKG, in contrast to smart-KG, ships bindings along with the triple pattern requests (as presented in brTPF [12] that requires fewer requests than TPF); third, WiseKG has an advantage over SPF to require less requests in case of star patterns with low selectivity. Figure 6b shows that SaGe incurs the least data transfer among all compared systems since SaGe is essentially a SPARQL endpoint with a preemption model that only transfers the final results. As expected, WiseKG incurs less data transfer than TPF, smart-KG, and SPF. To be precise, WiseKG transfers on average $5.5MB$ per query while SPF and smart-KG transfer $7MB$ and $13MB$ over `watdiv100M` dataset. WiseKG demands on average less intermediate results than SPF and smart-KG thanks to the cardinality estimation aware cost model.

Figure 6c presents the average server CPU usage per system when the `watdiv-sts` workload is executed over the `watdiv100M` dataset. SPF and SaGe consume more CPU on the server side. This is expected since SPF processes star pattern queries on the server side and SaGe utilizes a SPARQL endpoint that does all the work on the server side. As one can see from Figure 6c, the CPU usage of these two interfaces approach the CPU processing capabilities when the concurrent number of clients is set to 128. In contrast, CPU consumption of smart-KG and TPF remain almost constant and quite low; under 20% and 30%, respectively. This low consumption is inline with restricted capabilities of these servers: partition shipping in case of smart-KG and triple pattern lookup in case of TPF. Figure 6c shows that WiseKG's CPU usage is almost in the middle between SPF and smart-KG, where it gradually increases up to 60% in the case of 128 concurrent clients, which enables WiseKG to serve more queries given the current server capabilities (Figure 2a).

7 CONCLUSIONS AND FUTURE WORK

We introduced WiseKG, a querying interface to efficiently access Web Knowledge Graphs. We propose an efficient query processing approach under high query loads by balancing SPARQL query execution load between servers and clients. To this end, we have combined two Linked Data Fragments APIs (SPF and smart-KG) that enable server-side and client-side processing of star-shaped sub-patterns. Our dynamic cost model picks the best suited API per sub-query based on the current server load, client capabilities, and estimation of necessary data transfer between client and server (for intermediate query results), and network bandwidth. Our experiments show that WiseKG significantly outperforms state-of-the-art stand-alone LDF interfaces on high demanding workloads, with increasing numbers of concurrent clients, with increasing KG sizes, and on different query shapes. We show that WiseKG's cost model improves average workload completion (reducing the number of timeouts) while also reducing resource consumption (including less CPU usage and network traffic) compared to existing interfaces.

In our future work, we plan to evaluate in more detail the influence of different hardware setups and mixes of clients with differing computational resources. We also, respectively, plan to expand our query optimizer to consider further aspects, such as additional hardware parameters, parallelism, network delays, etc. as well as to provide optimization support for additional types of queries incl. for instance aggregation [14, 15]. Moreover, we plan to extend our implementation, which is currently implemented as a standalone setup, into a framework that flexibly allows to integrate different LDF APIs [20] and also other cost models. The recently introduced Comunica [7, 28] platform could serve as a starting point for integration. While our implementation covers also full SPARQL patterns (incl. UNION, OPTIONAL, FILTER, etc.) computed on the client side, the current approach is not dealing with multiple (named) graphs and GRAPH queries. Looking into extensions of HDT towards handling quads [11] could address this current limitation.

Acknowledgments. This research was funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, Aalborg University's Talent Programme, and the Poul Due Jensen Foundation. The research was further funded by the EU H2020 research and innovation programme under grant agreement No 957402 (TEAMING.AI) and Marie Skłodowska-Curie grant agreement No. 860801 (KnowGraphs).

REFERENCES

[1] C. Aebeloe, I. Keles, G. Montoya, and K. Hose. 2020. Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns. *CoRR* abs/2002.09172 (2020).
[2] C. Aebeloe, G. Montoya, and K. Hose. 2019. A Decentralized Architecture for Sharing and Querying Semantic Data. In *ESWC 2019*. 3–18.
[3] C. Aebeloe, G. Montoya, and K. Hose. 2019. Decentralized Indexing over a Network of RDF Peers. In *ISWC 2019*. 3–20.
[4] G. Aluç, O. Hartig, M. Tamer Özsu, and K. Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *ISWC 2014*. 197–212.
[5] C. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. 2013. SPARQL Web-Querying Infrastructure: Ready for Action?. In *ISWC 2013*. 277–293.
[6] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, and A. Polleres. 2020. SMART-KG: Hybrid Shipping for SPARQL Querying on the Web. In *WWW 2020*. 984–994.
[7] A. Azzam, R. Taelman, and A. Polleres. 2020. Towards Cost-Model-Based Query Execution over Hybrid Linked Data Fragments Interfaces. In *ESWC 2020*. 9–12.
[8] P. A. Bonatti, S. Decker, A. Polleres, and V. Presutti. 2019. Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web (Dagstuhl Seminar 18371). *Dagstuhl* 8, 9 (2019), 29–111.
[9] M. Cai and M. R. Frank. 2004. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW 2004*. 650–657.
[10] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. 2013. Binary RDF representation for publication and exchange (HDT). *J. Web Semant.* 19 (2013), 22–41.
[11] J. D. Fernández, M. A. Martínez-Prieto, A. Polleres, and J. Reindorf. 2018. HDTQ: Managing RDF Datasets in Compressed Space. In *ESWC 2018*. 191–208.
[12] O. Hartig and C. B. Aranda. 2016. Bindings-Restricted Triple Pattern Fragments. In *ODBASE 2016*. 762–779.
[13] L. Heling and M. Acosta. 2020. Cost- and Robustness-Based Query Optimization for Linked Data Fragments. In *ISWC 2020*. 238–257.
[14] D. Ibragimov, K. Hose, T. Pedersen, and E. Zimányi. 2016. Optimizing Aggregate SPARQL Queries Using Materialized RDF Views. In *ISWC 2016*. 341–359.
[15] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. 2015. Processing Aggregate Queries in a Federation of SPARQL Endpoints. In *ESWC 2015*. 269–285.
[16] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. Mendes, S. Hellmann, M. Morsey, P. Kleef, S. Auer, and C. Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.
[17] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.
[18] L. F. Mackert and G. M. Lohman. 1986. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *VLDB 1986*. 149–159.
[19] T. Minier, H. Skaf-Molli, and P. Molli. 2019. SaGe: Web Preemption for Public SPARQL Query Services. In *WWW 2019*. 1268–1278.
[20] G. Montoya, C. Aebeloe, and K. Hose. 2018. Towards Efficient Query Processing over Heterogeneous RDF Interfaces. In *DeSemWeb@ISWC 2018*.
[21] G. Montoya, I. Keles, and K. Hose. 2019. Analysis of the Effect of Query Shapes on Performance over LDF Interfaces. In *QuWeDa@ISWC 2019*. 51–66.
[22] G. Montoya, M. Vidal, Ó. Corcho, E. Ruckhaus, and C. Buil Aranda. 2012. Benchmarking Federated SPARQL Query Engines: Are Existing Testbeds Enough?. In *ISWC 2012*. 313–324.
[23] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE 2011*. 984–994.
[24] Y. Park, S. Ko, Sourav S. Bhowmick, K. Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *SIGMOD 2020*. 1099–1114.
[25] J. Pérez, M. Arenas, and C. Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.
[26] A. Polleres, M. R. Kamdar, J. D. Fernández, T. Tudorache, and M. A. Musen. 2018. A More Decentralized Vision for Linked Data. In *DeSemWeb@ISWC 2018*.
[27] M. Saleem, M. Intizar Ali, A. Hogan, Q. Mehmood, and A. Ngonga Ngomo. 2015. LSQ: The Linked SPARQL Queries Dataset. In *ISWC 2015*. 261–269.
[28] R. Taelman, J. Van Herwegen, M. V. Sande, and R. Verborgh. 2018. Comunica: A Modular SPARQL Query Engine for the Web. In *ISWC 2018*. 239–255.
[29] P. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan, and C. Aranda. 2017. SPARQLES: Monitoring public SPARQL endpoints. *Semantic Web* 8, 6 (2017), 1049–1065.
[30] R. Verborgh, M. V. Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. 2016. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *J. Web Semant.* 37-38 (2016), 184–206.
[31] I. Zouaghi, A. Mesmoudi, J. Galicia, L. Bellatreche, and T. Aguili. 2020. Query Optimization for Large Scale Clustered RDF Data. In *DOLAP@EDBT/ICDT 2020*. 56–65.