Eurostars Project

3DFed – Dynamic Data Distribution and Query Federation

Project Number: E!114681 **Start Date of Project:** 2021/04/01 **Duration:** 36 months

Deliverable 1.2
Component Architecture and Interfaces

Dissemination Level	Public
Due Date of Deliverable	September 30, 2021
Actual Submission Date	September 30, 2021
Work Package	WP1, Requirements Elicitation & Conceptual Architecture
Deliverable	D1.2
Type	Report
Approval Status	Final
Version	1.2
Number of Pages	17

Abstract:
This report first presents a detailed survey of state-of-the-art techniques in RDF data partitioning and federated SPARQL query processing. Based on the requirements specified in Task 1.1 and the survey results, we then present a detailed technical architecture of 3DFed. The architecture define the generic interfaces for the front-end based on W3C standards (SPARQL, RDF, OWL). In addition, we identify the reusable components and the strategies for their integration into other components. We finally discuss the 3DFed integration into the partners' products.

3DFed Project by Eurostars.

History

Version	Date	Reason	Revised by
0.1	12/09/2021	Initial Template & Deliverable Structure	Muhammad Saleem
0.2	16/09/2021	Architecture and State of the Art	Muhammad Saleem
0.3	20/09/2021	Reusable Components and Components Integration	Milos Jovanovik
0.4	24/09/2021	UPB Reusable Components and Components Integration	Muhammad Saleem
0.5	29/09/2021	Reusable Components	Mirko Spasić
0.6	30/09/2021	Reusable Components, Components Integration and Product Integration related to elevait	Martin Voigtć
1.0	30/09/2021	Finalizing	Muhammad Saleem

Author List

Organization	Name	Contact Information
University of Paderborn	Muhammad Saleem	saleem@informatik.uni-leipzig.de
OpenLink Software	Milos Jovanovik	mjovanovik@openlinksw.com
OpenLink Software	Mirko Spasić	mspasic@openlinksw.com
elevait GmbH & Co. KG	Martin Voigt	martin.voigt@elevait.de

Contents

1 State Of The Art

State of the art pertaining to 3DFed project can be divided into three main categories namely RDF data partitioning, federated SPARQL query processing, and RDF datasets profiling.

1.1 RDF Data Partitioning

In distributed RDF stores and SPARQL engines, the data are partitioned over a cluster of machines in order to enable horizontal scale, where additional machines can be allocated to the cluster to handle larger volumes of data. However, horizontal scaling comes at the cost of network communication costs. Thus a key optimization is to choose a partitioning scheme that reduces communication costs by enforcing various forms of locality, principally allowing certain types of (intermediate) joins to be processed on each individual machine [3]. Formally, given an RDF graph G and n machines, an n-partition of G is a tuple of subgraphs $(G_1, \ldots, G_n)$ such that $G = \bigcup_{i=1}^{n} G_i$, with the idea that each subgraph G_i will be stored on machine i.[1] We now discuss different high-level alternatives for partitioning.

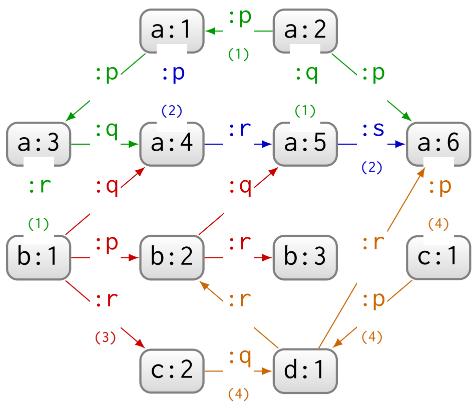
Triple/Quad-based Partitioning

A first option is to partition based on individual triples or quads without considering the rest of the graph. For simplicity we will speak about triples as the discussion generalizes straightforwardly to quads. The simplest option is to use *round robin* or *random partitioning*, which effectively places triples on an arbitrary machine. This ensures even load balancing, but does not support any locality of processing, and does not allow for finding the particular machine storing triples that match a given pattern.

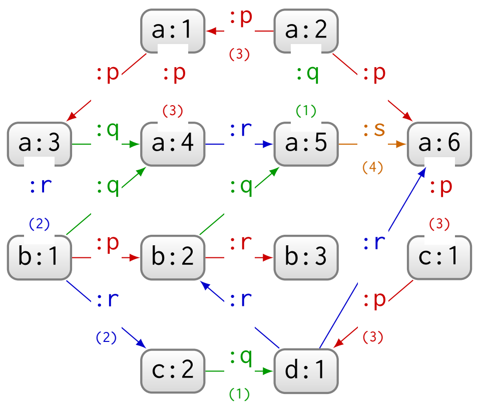
An alternative is to partition according to a deterministic function over a given key; for example, a partition key of S considers only the subject, while a partition key of PO considers both the predicate and object. Later given a triple pattern that covers the partition key (e.g., with a constant subject if the key is S), we can find the machine(s) storing all triples that match that pattern. We show some examples using different functions and partition keys in Figure 1 considering four machines. *Range-based partitioning* assigns a range over the partition key to each function, where the example of Figure 1 splits S into [a:1,a:3], [a:4,a:6], [b:1,b:3], [c:1,d:1]. This approach allows for range-based queries to be pushed to one machine, but requires maintaining a mapping of ranges to machines, and can be complicated to keep balanced. An alternative is *hash-based partitioning* where we compute the hash of the partition key modulo the number of machines, where the second example of Figure 1 splits P by hash. This does not require storing any mapping, and techniques such as consistent hashing can be used to rebalance load when a machine enters or leaves; however, if partition keys are skewed (e.g., one predicate is very common), it may lead to an unbalanced partition. A third option is to apply a *hierarchical-based partition* based on prefixes, where the third example of Figure 1 partitions O by their namespace. This may lead to increased locality of data with the same prefix [29], where different levels of prefix can be chosen to enable balancing, but choosing prefixes that offer balanced partitions is non-trivial.

Any such partitioning function will send any triple with the same partition key to the same machine, which ensures that (equi-)joins on partition keys can be pushed to individual machines. Hash-based partitioning is perhaps the most popular among distributed RDF stores (e.g., YARS2 [23], SHARD [37], etc.). Often triples will be hashed according to multiple partition keys in order to support different index permutations, triple patterns, and indeed joins across different types of partition keys (e.g, with S and O as two partition keys, we can push

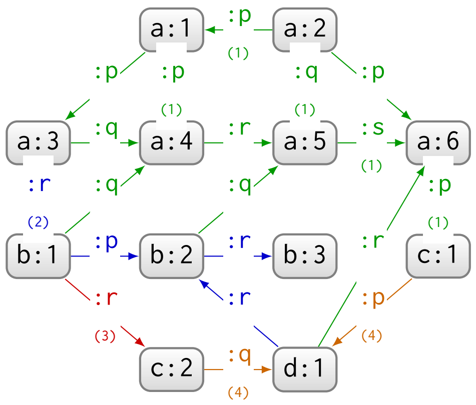
[1]We relax the typical requirement for a set partition that $G_i \cap G_j = \emptyset$ for all $1 \leq i < j \leq n$ to allow for the possibility of replication or other forms of redundancy.



Range-based (S)



Partition-based (P)



Hierarchy-based (O)

Figure 1: Examples of triple-based partitioning schemes

S–S, O–O and S–O (equi-)joins to each machine). When the partition key P is chosen, the scheme is analogous to distributed vertical partitioning. Care must be taken to avoid imbalances caused by frequent terms, such as the `rdf:type` predicate, or frequent objects such as classes, countries, etc. Omitting partitioning on highly-skewed partition keys may be advantageous for balancing purposes [23].

Graph-based Partitioning

Graph-based partitioning takes into consideration the entire graph when computing a partition. A common strategy is to apply a k-*way partition* of the RDF graph G [30]. Formally, letting $V = \mathrm{so}(G)$ denote the nodes of G, the goal is to compute a node partition $V_1, \ldots, V_n$ such that $V = \bigcup_{i=1}^{k} V_n$, $V_i \cap V_j = \emptyset$ for all $1 \leq i < j \leq k$, $\lfloor \frac{|V|}{k} \rfloor \leq |V_i| \leq \lceil \frac{|V|}{k} \rceil$ for all $1 \leq i \leq k$, and the number of triples $(s, p, o) \in G$ such that s and o are in different node partitions is minimised. In Figure 2, we show the optimal 4-way partitioning of the graph seen previously, where each partition has 3 nodes, there are 10 edges between partitions (shown dashed), and no other such partition leads to fewer edges (<10) between partitions. Edges between partitions may be replicated in the partitions they connect. Another alternative is to k-way partition the *line graph* of the RDF graph: an undirected graph where each triple is a node, and triples sharing a subject or object have an edge between them.

Finding an optimal k-way partition is intractable[2], where approximations are thus necessary for large-scale graphs, including *spectral methods*, which use the eigenvectors of the graph's Laplacian matrix to partition it; *recursive bisection*, which recursively partitions the graph in two; *multilevel partitioning*, which "coarsens" the graph by computes a hierarchical graph summary, then partitions the smaller graph summary (using, e.g., spectral methods), and finally "uncoarsens" by expanding back out to the original graph maintaining the partitions; etc. We refer for more details to Buluç et al. [9], who argue that multilevel partitioning is "*clearly the most successful heuristic for partitioning large graphs*". Such techniques have been used by H-RDF-3x [27], EAGRE [49], Koral [28], and more besides.

Query-based Partitioning

While the previous partitioning schemes only consider the data, other partitioning methods are (also) based on queries. Workload-based partitioning strategies have been broadly explored, where the idea is to identify common joins in query logs that can be used to partition or replicate parts of the graph in order to ensure that high-demand joins can be pushed to individual machines. Partitioning can then be *a priori*, for example, based on a query log; or *dynamic* (aka. adaptive), where the partitions change as queries are received. Such strategies are used by WARP [26], Partout [17], WORQ [33], chameleon-DB [5], AdPart [22], etc.

Dynamic Data Exchange

It is possible that the initial static partitioning of data is not optimized when deployed in real-world settings. This is because that the type data distribution may result in large intermediate results during query processing and consequently to unacceptably high network traffic, resource consumption, and overall query runtime. In that case, it is important to dynamically shuffle the data among data nodes or endpoints to fit according to the type of receiving workload. There is only little attention paid to this dynamic partitioning of data in RDF graphs and SPARQL querying. Bok et. al [7] proposes a dynamic partitioning method of RDF graphs to support load balancing in distributed environments where data insertion and change continue to occur. The proposed method generates clusters and subclusters by considering the usage frequency of the RDF graph that are used by queries as the criteria to perform graph partitioning. It creates a cluster by grouping RDF subgraphs with higher usage frequency while creating a subcluster with lower usage frequency. Workload-adaptive streaming partitioner named WASP, that aims to achieve low-latency and high-throughput online graph queries was proposed in [11]. They assume that workload typically contains frequent query patterns, WASP exploits the existing workload to capture active vertices and edges which are frequently visited and traversed, respectively. This information is used to heuristically improve the quality of partitions either by avoiding the concentration of active vertices in a few partitions proportional to their visit frequencies or by reducing the probability of the cut of active edges proportional to their traversal frequencies. Other systems such as WARP [26], PartOut [17], Loom [15], and Taper [16] have limited support for workload-adaptive partitioning [11].

Replication

Rather than partitioning data, data can also be replicated across partitions. This may vary from replicating the full graph on each machine, such that queries can be answered in full by any machine increasing query throughput (used, e.g., by DREAM [21]), to replicating partitions that are in high-demand (e.g., containing schema data, central nodes, etc.) so that more queries can be evaluated on individual machines and/or machines have equal workloads that avoid hot-spots (used, e.g., by Blazegraph [47] and Virtuoso [14]).

[2]Given a graph, deciding if there is a k-way partition with fewer than n edges between partitions is NP-complete.

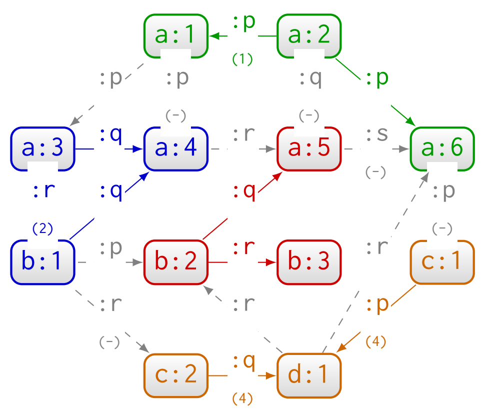


Figure 2: Example of optimal k-way partitioning ($k = 4$)

Discussion

Triple/quad-based partitioning is the simplest to compute and maintain, requiring only the data present in an individual tuple, allowing joins on the same partition key to be pushed to individual machines. Graph-based partitions allow for evaluating more complex graph patterns on individual machines, but are more costly to compute and maintain (considering, e.g., dynamic data). Information about queries, where available, can be used for the purposes of workload-based partitioning, which partitions or replicates data in order to enable locality for common sub-patterns. Replication can further improve load balancing, locality and fault-tolerance at the cost of redundant storage.

1.2 Federated SPARQL Query Processing

In this section, we give a brief overview of the federated SPARQL query processing engines over multiple endpoints. The focus is the cost-based federation engines, as planned in the 3DFed project. In particular, we describe how the cardinality estimations for triple patterns and joins between triple patterns are performed in state-of-the-art cost-based federated SPARQL query processing engines.

DARQ [36] is an index-assisted federated query engine based on index-only source selection and nested loop join for query optimization. However, DARQ can generate a large number of endpoint requests due to the nested loop join implementation. SPLENDID [19] makes use of a hybrid (index+SPARQL ASK) source selection approach, a cost-based optimization using datasets statistics. While SPLENDID makes use of bind as well as hash joins , it does not perform a join-aware source selection and thus often overestimates the number of relevant sources. FedX [44] is an index-free approach based on ASK queries for source selection. It performs bind joins in a block nested loop fashion. Since FedX does not store any statistics about the datasets, the join ordering is based solely on a heuristic, i.e., counting the free variables among triple patterns. Thus, it can generate query execution plans that are not efficient.

ANAPSID [2] is an adaptive query federation engine that adapts its query execution at runtime. It makes use of a hybrid source selection approach and implements adaptive group and adaptive dependent joins. SemaGrow [10] is an index-assisted approach based on stored statistics about datasets. SemaGrow implements bind, hash, and merge joins. However, like SPLENDID, it overestimates the set of relevant sources and can result in extra intermediate results. Lusail [1] is an index-free query federation engine that exploits the data locality for optimized query processing. Odyssey [34] is index-assisted query engines based on cardinality estimations to generate optimized query execution plans. Inspired by our CostFed [42] engine, we are proposing a cost-based

SPARQL query federation engine. The join ordering of the plans it generates will be based on an estimation of the cardinality of the triple patterns and the joins contained in the input query. The decision between the join implementation to use (bind or symmetric hash join) will be based on the join cost estimation function it implements. In addition, it will implement means to keep its index up-to-date based on regular intervals as well as at fixed times. Some others recent relevant work can be found in [**?**, 6, 32, 45]

Now we go into further details of the cost-based federation engines.

SPLENDID: SPLENDID [18] also uses VoID statistics to generate a query execution plan. It uses a dynamic programming approach to produce a query execution plan. SPLENDID makes use of both hash ($\bowtie_h$) and bind ($\bowtie_b$) joins.

Triple pattern cardinality is estimated as follows:

$$\begin{aligned}
&\text{card}_d(?,p,?) = \text{card}_d(p)\\
&\text{card}_d(s,?,?) = |d| \cdot \text{sel}\,.s_d\\
&\text{card}_d(s,p,?) = \text{card}_d(p) \cdot \text{sel}.\,\text{s}_d(p)\\
&\text{card}_d(?,?,o) = |d| \cdot \text{sel}\,.o_d\\
&\text{card}_d(?,p,o) = \text{card}_d(p) \cdot \text{sel}\,.o_d(p)\\
&\text{card}_d(s,?,o) = |d| \cdot \text{sel}\,.s_d \cdot \text{sel}\,.o_d
\end{aligned}$$

where the $card_d(p)$ is the number of triple patterns in the data source d having predicate p. The total number of triples in a data source d is defined as $|d|$. If we have a bound predicate then the average selectivity of subject and object is defined as $sel.s_d(p)$ and $sel.o_d(p)$ respectively; if the predicate is not bound then the average selectivity of subject and object is defined as $sel.s_d$ and $sel.o_d$ respectively. In star-shaped queries, SPLENDID estimates the cardinality of triple patterns having the same subject separately. All triples with same subjects are grouped and then the minimum cardinality of all triple patterns with bound objects is calculated. Lastly, the cardinality of remaining triples with unbound objects is multiplied with the average selectivity of subjects and the minimum value. Formally, the equation is defined as:

$$\begin{aligned}
&\text{card}_d(T) =\\
&\quad \min\left(\text{card}_d\left(T_{bound}\right)\right) \cdot \prod\left(sel.s_d \cdot \text{card}_d\left(T_{unbound}\right)\right)
\end{aligned}$$

Join cardinality is estimated as follows:

$$\text{card}(q1 \bowtie q2) = \text{card}\left(q_1\right) \cdot \text{card}\left(q_2\right) \cdot \text{sel}_{\bowtie}(q1, q2)$$

In these equations the $sel_{\bowtie}$ is the join selectivity of two input relations. It defines how many bindings are matched between two relations. SPLENDID uses the average selectivity of join variables as join selectivity.

LHD: LHD [48] is a cardinality-based and index-assisted approach that aims to maximize parallel execution of sub-queries. It makes use of the VoID statistics for estimating the cardinality of triple patterns and joins between triple patterns. LHD only uses Bind joins for query execution. LHD implements a response-time-cost model by making an assumption that the response time of a query request is proportional to the total number of bindings transferred. LHD determines the total number of triples t_d, distinct subjects s_d and objects o_d from the VoID description of a dataset d. The VoID file also provides other information, such as the number of triples $t_{d.p}$, distinct subjects $s_{d.p}$ and distinct objects $o_{d.p}$ in the dataset d for a predicate p. The federation engine makes an assumption about uniform distribution of objects and subjects in datasets. Let's assume a triple pattern

$T: \{SPO\}$ [3], the function to get the set of relevant datasets of T is defined as $S(T)$, the selectivity of x with respect to $S(T)$ is defined as $selT(x)$, and the cardinality of x with respect to $S(T)$ is defined as $cardT(x)$.

For single triple pattern cardinality estimation, the selectivity of each part is estimated as follows:

$$\mathrm{sel}_T(S) = \begin{cases} \frac{\sum_{d\in S(T)} t_d/s_d}{\sum_{d\in S(T)} s_d} & \text{if } \mathrm{var}(P) \wedge \neg\, \mathrm{var}(S) \\ \frac{\sum_{d\in S(T)} t_{d\cdot p}/s_{d\cdot p}}{\sum_{d\in S(T)} s_{d\cdot p}} & \text{if } P = p \wedge \neg\, \mathrm{var}(S) \\ 1 & \text{if } \mathrm{var}(S) \end{cases}$$

$$\mathrm{sel}_T(P) = \begin{cases} \frac{\sum_{d\in S(T)} t_{d.p}}{\sum_{d\in S(T)} t_d} & \text{if } P = p \\ 1 & \text{if } \mathrm{var}(P) \end{cases}$$

$$\mathrm{sel}_T(O) = \begin{cases} \frac{\sum_{d\in S(T)} t_d/o_d}{\sum_{d\in S(T)} o_d} & \text{if } \mathrm{var}(P) \wedge \neg\, \mathrm{var}(O) \\ \frac{\sum_{d\in S(T)} t_{d.p}/o_{d.p}}{\sum_{d\in S(T)} o_{d.p}} & \text{if } P = p \wedge \neg\, \mathrm{var}(O) \\ 1 & \text{if } \mathrm{var}(O) \end{cases}$$

After calculating the selectivity of each part, LHD estimates the cardinality of the triple pattern as follows:

$$\mathrm{card}(T) = t \cdot \mathrm{sel}_T(S) \cdot \mathrm{sel}_T(P) \cdot \mathrm{sel}_T(O)$$

Given two triple patterns T1 and T2, LHD calculates the join selectivity by using the following equations:

$$\mathrm{sel}\,(T_1 \bowtie T_2) = \begin{cases} \frac{\sum_{d\in S(T_1)} s_{d.p1} \cdot \sum_{d\in S(T_2)} s_{d.p2}}{\sum_{d\in S(T_1)} s_d \cdot \sum_{d\in S(T_2)} s_d} & \text{if joined on } S_1 = S_2 \\ \frac{\sum_{d\in S(T_1)} o_{d.p1} \cdot \sum_{d\in S(T_2)} o_{d.p2}}{\sum_{d\in S(T_1)} o_d \cdot \sum_{d\in S(T_2)} o_d} & \text{if joined on } O_1 = O_2 \\ \frac{\sum_{d\in S(T_1)} s_{d.p1} \cdot \sum_{d\in S(T_2)} o_{d.p2}}{\sum_{d\in S(T_1)} s_d \cdot \sum_{d\in S(T_2)} o_d} & \text{if joined on } S_1 = O_2 \\ 1 & \text{if no shared variables.} \end{cases}$$

Using the join selectivity values, join cardinality is estimated by the following equation:

$$\mathrm{card}\,(T_1 \bowtie T_2 \bowtie \ldots \bowtie T_n) = \prod_{i=1}^{n} \mathrm{card}\,(T_i) \cdot \mathrm{sel}\,(T_1 \bowtie T_2 \bowtie \ldots \bowtie T_n)$$

[3] In this section, the letters with a question mark (e.g. ?x) denote a variable in an RDF triple, a literal value is represented by a lower-case letter (e.g. o) , and a variable or a literal value is defined by an upper-case letter (e.g. S)

SemaGrow: SemaGrow [10] query planning is based on VoID[4] statistics [4] about datasets. It makes use of the VoID index as well as SPARQL ASK queries to perform source selection. Three types of joins, i.e, bind, merge and hash, are used during the query planning. The selection to perform the required join operation is based on a cost function. It uses a reactive model for retrieving results of the joins as well as individual triple patterns. As with CostFed, SemaGrow recursively defines SPARQL expressions. Given a data source S, the cardinality estimations of triple patterns and joins are explained below.

SemaGrow contains a Resource discovery component, which returns the list of relevant sources to a triple pattern along with statistics. The statistics related to the data source include (1) the number of estimated distinct subjects, predicates and objects matching the triple pattern, and (2) the number of triple patterns in the data sources matching the triple pattern. The cardinality of a triple pattern is provided by the Resource Discovery component. On the other hand, for more complex expressions, SemaGrow needs to make an estimation based on available statistics. In order to estimate complex expressions based on the aforementioned basic statistics, SemaGrow adopts the formulas described by LHD [48]. The cardinality of each expression (E) in a data source S, is defined as $Card([E], S)$).

For estimating the join cardinality we need to calculate the join selectivity (JoinSel ($[E1] \bowtie [E2]$)), which is given as follows:

$$\begin{aligned}
&\text{JoinSel } ([E1] \bowtie [E2]) = \\
&\qquad \min(\text{ JoinSel } [E1], \text{ JoinSel } [E2]) \\
&\text{JoinSel } ([T]) = \min(1/d_i)
\end{aligned}$$

In these equations, E1 and E2 reside any join expressions or triple patterns. The T is a single triple pattern. d_i represents the number of distinct values for the i-st join attribute in a T. Hence, the join cardinality is given as follows:

$$\begin{aligned}
&\text{Card}([E1] \bowtie [E2], S) = \\
&\text{Card}([E1], S) \cdot \text{Card}([E2], S) \cdot \text{ JoinSel } ([E1] \bowtie [E2])
\end{aligned}$$

Odyssey: Odyssey [34] makes use of the distributed characteristic sets (CS) [35] and characteristic pair (CP) [20] statistics to estimate cardinalities. Odyssey estimates the cardinality of each type of query differently using these statistics. For star-shaped queries, where the subject (or object) is the same for all joining triple patterns, estimated cardinality for a given set of properties P (predicates of joining triple patterns) is computed using CSs C_j containing all these properties. The common subject (or object) is defined as an entity. CSs can be computed by scanning once a dataset's triples are sorted by subject; after all the entity properties have been scanned, the entity's CS is identified. For each CS C, Odyssey computes statistics, i.e., ($count(C)$) represents the number of entities sharing C and ($occurrences(p, C)$) represents the number of triples with predicate p occurring with these entities.

Odyssey represents estimatedCardinality$_{Distinct}(P)$ as the estimated cardinality of queries that contain distinct keywords, and estimatedCardinality(P) as the estimated cardinality of those queries that do not contain the distinct keyword. Formally, estimated cardinality for star-shaped queries is defined as follows:

$$\begin{aligned}
&\text{estimatedCardinality }_{Distinct}(P) = \sum_{P \subseteq C_j} (\text{count } (C_j)) \\
&\text{estimatedCardinality } (P) = \\
&\qquad \sum_{P \subseteq C_j} \left(\text{count } (C_j) \cdot \prod_{p_i \in P} \frac{\text{ocurrences } (p_i, C_j)}{\text{count } (C_j)} \right)
\end{aligned}$$

[4]VoID vocabulary: `https://www.w3.org/TR/void/`

For arbitrary-shaped queries, Odyssey also considers the connections (links) between different CSs. Characteristic pairs (CPs) help in describing the links between Characteristic sets (CSs) using properties. For entities $e1$ and $e2$ the link is defined as $(CSs(e1), CSs(e2), p)$, given that $(e1, p, e2) \in s$, where s is data source. The number of links between two CSs: C_i and C_j, through a property p is represented in statistics, which is defined as: – $count(Ci, Cj, p)$. The equation for estimating the cardinality (pairs of entities with a set of properties P_k and P_l) for complex-shaped queries is defined as:

$$\text{estimatedCardinality}\left(P_k, P_l, p\right) =$$

$$\sum_{P_k \subseteq C_i \wedge P_l \subseteq C_j} (\text{count}\left(C_i, C_j, p\right) \cdot \prod_{p_k \in P_k - \{p\}}$$

$$\left(\frac{\text{ocurrences}\left(p_k, C_i\right)}{\text{count}(C_i)}\right) \cdot \prod_{p_l \in P_l} \left(\frac{\text{ocurrences}\left(p_l, C_j\right)}{\text{count}(C_j)}\right))$$

In order to reduce the complexity, Odyssey treats each star-shaped query as a single meta-node; assuming that the order of joins has already optimized within the star-shaped sub-queries. It uses Characteristics Pairs (CPs) to estimate the cardinality of joins between star-shaped queries (meta-nodes) and uses dynamic programming (DP) to optimize the join order and find the optimal plan.

1.3 RDF Datasets Profiling

SPORTAL [24] aims in providing basic information (e.g., distinct classes, properties, objects, etc.) about the public SPARQL endpoints. SPARQELS [8] monitors the public SPARQL endpoints and monitors their availability. The Linked SPARQL Queries (LSQ) dataset [38] provides meta-data information about real-world SPARQL queries collected from public SPARQL endpoints querying logs. Beside SPORTAL information, we are aiming to provide more detailed information (e.g., coherence of dataset, selectivities of subjects and objects for a given predicate, some data quality related statistics etc.). In addition, we will also monitor the SPARQL endpoints for availability and response times. We aim to present an LSQ like dataset in this task.

2 3DFed Architecture

Datasets suited for modern needs are commonly distributed and increasingly getting too large to fit in a single server. Current distributed solutions are designed for central storage or at best static data distribution, which can result in high network traffic and query runtimes. Modern end-user applications require results within milliseconds. Thus, there is an increasing need for intelligent and efficient data distribution and federated query engines to deal with these large amounts of data. The idea behind 3DFed is to develop generic approaches for the automatic redistribution and federated querying of large distributed datasets so as to facilitate the development of high-performance distributed data storage solutions. The final output will be W3C-standard-conform solutions/tools that implement automatic data distribution, federated query planning and execution, dynamic data exchange mechanisms, data storage profiling (containing useful information/statistics about the underlying data) and data monitoring.

A preliminary architecture for 3DFed is shown in Figure 3. Starting from the bottom, we first convert the different types of data into Resource Description Framework format (RDF), a W3C standard for the representation of knowledge on the Web. We then distribute the resulting RDF datasets among different data storage solutions, i.e., triple stores with public SPARQL endpoints, thus allowing users to query them by means of SPARQL, the W3C standard for querying RDF data. The triple stores dynamically exchange the data to exploit data locality for maximising/balancing the amount of computation in a single storage solution. The user can query the distributed triple stores using the 3DFed federation engine, which makes use of the data storage profiles collected beforehand.

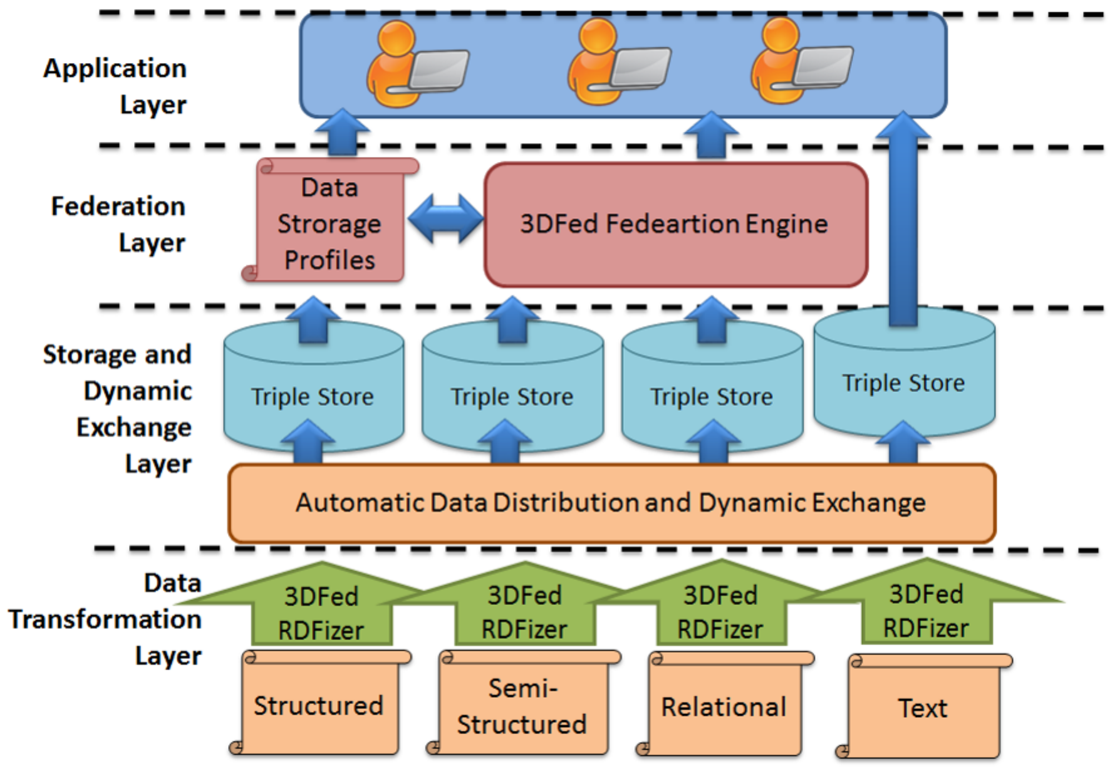


Figure 3: The 3DFed Generic Architecture

The architecture consists of 4 main layers:

2.1 Data Transformation Layer

In this layer, we convert the different types that are relevant for the business use cases into the Resource Description Framework format (RDF), a W3C standard for the representation of knowledge on the Web. The large number of datasets required by the use cases will each be hosted in a SPARQL endpoint, thus allowing the users to query them by means of SPARQL, the W3C standard for querying RDF data. Semi-structured and structured data can already be converted to RDF data efficiently by using systems such as D2R, Triplify and Sparqlify. Given that these systems all rely on manual specifications, we focus on devising automatic approaches for the conversion of structured and semi-structured data into RDF and link them to existing dataset already available on the LOD cloud.

The time-efficient and accurate extraction of RDF from unstructured data sources remains an important challenge that needs to be addressed. In this WP, we will hence also develop methods for the automatic transformation of unstructured data into Linked Data. Special attention will be paid to spatial data in this context.

2.2 Storage and Dynamic Layer

In 3DFed, we develop generic algorithms for the intelligent and automatic distribution and dynamic exchange of large linked datasets among storage solutions. We will first start the automatic distribution of the linked data among the available data storage solutions or data nodes (in a clustered version of the data storage solution). We then make use of the query logs to dynamically exchange the data between the data storage solutions. The overall goal is to improve the query runtime performance and distribute the load among data storage solutions to improve their availability. Consequently, this will facilitate the development of high-performance federation engines. To this end, we aim to exchange the data between storage solutions dynamically and exploit data locality to maximise/balance the amount of computation in a single storage solution. A data security model will ensure that the data sharing abides by the restrictions pertaining to data sharing across two storage solutions (e.g., that storage solutions that should not share data never share data). The dynamic exchange of data can be a deletion, an insertion or the insertion of a replicated chunk of data from another storage solution. The decision of dynamic exchange will be mostly based on the monitoring of the storage solutions, in particular on the federated queries that were issued to the distributed storage solution.

2.3 Federation Layer

In this layer, we are proposing a cost-based SPARQL query federation engine. The join ordering of the plans it generates will be based on an estimation of the cardinality of the triple patterns and the joins contained in the input query. The decision between the join implementation to use (bind or symmetric hash join) will be based on the join cost estimation function it implements. In addition, it will implement means to keep its index up-to-date based on regular intervals as well as at fixed times.

2.4 Application Layer

In this layer, we provide user friendly interfaces for querying the distributed data hosted by different data storage solutions. Additionally, we use interfaces for gathering the statistics during the data profiling.

3 Reusable Components

The 3DFed architecture will include some already existing tools, storage solutions, algorithms and models, which represent reusable components. They will be integrated with other components as part of the full 3DFed architecture.

3.1 Virtuoso

The architecture will use the Virtuoso Universal Server as an RDF storage solution. Virtuoso is a high-performance virtual database engine which provides transparent access to existing data sources, which are typically databases from different database vendors, or RDF knowledge graphs [13, 14, 12]. It supports data access via the standard SPARQL protocol, but also offers drivers for the Jena, Sesame, and Redland frameworks.

It also provides support for partitioning data across multiple servers, via Virtuoso Cluster. Virtuoso Cluster partitions each index of all tables containing RDF data separately. The partitioning is by hash. The result is that the data is evenly distributed over the selected number of servers. Immediately consecutive triples are generally in the same partition, since the low bits of IDs do not enter in into the partition hash. This means that key compression works well.

Since RDF tables are in the end just SQL tables, SQL can be used for specifying a non-standard partitioning scheme. For example, one could dedicate one set of servers for one index, and another set for another index.

In the GeoKnow project[5], it is shown how geospatial clustering and distributed geospatial querying can be done within Virtuoso. Characteristic set, as a structure-aware RDF store feature, can be used to optimize storage of RDF data in cases where the data exhibits regular, i.e. when there is a relational-like structure in the data. Also, some benefits from reorganizing physical data placement according to geospatial properties have been demonstrated using quad-tree procedure [43]. Squares of different sizes containing the same number of geometries are constructed and distributed among the cluster nodes, acting like a load balancer.

All of these flexibilities allow us to use Virtuoso in 3DFed and extend the data storage model for cluster with the additional types of partitioning outlined in the previous sections. The existing interfaces of Virtuoso will also allow a seamless implementation of the dynamic data exchange algorithms planned in the 3DFed architecture, to maximize the architecture's performance.

3.2 Federation and Data Profiles

The DICE group at University of Paderborn already developed techniques for join-aware source selection algorithms for federated SPARQL query processing over multiple endpoints [40, 41, 25, 39, 31]. We will reuse the core idea of the join-aware source selection from these techniques. In addition, LSQ [38, 46], SPORTAL [24] data models will be reused to create profiles of the RDF datasets.

3.3 Data Transformation

In this project, elevait will provide its software platform for extracting and transforming information from arbitrary business documents, like handwritten orders or invoices. The software comprises of a bunch of web services for different machine learning tasks as well as AI Studio, based on Apache Nifi, for orchestrating these services on application level in a sophisticated extraction pipeline. In addition, the platform contains web applications that are mainly managing and reading the extracted information to present the insights to the users. The extraction pipeline already creates RDF data ins JSON-LD format. However, no SPARQL endpoint is used for writing or reading the data in the whole platform due to the lack of proper federation of the big datasets.

4 Components Integration

4.1 Integration with other Components

Virtuoso will be used as an RDF storage system for the 3DFed architecture. Aside from the general RDF and SPARQL storage and retrieval functionalities, the architecture will make use of Virtuoso's triple-hash-based partitioning on a Virtuoso cluster deployment.

As part of the project, the algorithms which will be developed both for partitioning the data on the cluster and for the dynamic data exchange, will seamlessly be integrated into or on top of Virtuoso. This will be achieved thanks to the flexibility of Virtuoso and its cluster deployment scheme, as well as its multiple standardized interfaces for reading and writing RDF data.

Source selection is one of the most important step in federated SPARQL query query processing. Our proposed 3DFed federated SPARQL query processing engine will first parse the given SPARQL queries to get the individual triple patterns. Then, it will perform the join-aware triple pattern-wise source selection. For this step, we will reuse (and extend) the HiBISCUS [40] source selection algorithm. It is implemented in java and

[5]`http://geoknow.eu/`

hence easily can be integrated to the proposed federation engine. The LSQ and SPORTAL models the data in RDF and can directly be integrated in to the proposed RDF-based data profiles.

As mentioned, the data transformation layer of elevaits software platform produces RDF data in JSON-LD format, currently stored document-based NoSQL databases. A main goal of elevait in 3DFed is, to leverage the real semantic power of the RDF data model in its platform. Regarding the integration, it means that all interfaces between the extraction pipeline and the web applications at on side and the database at the other side have to be analyzed in detail regarding the query complexity, especially aggregations, latency, throughput etc. After having the 1st prototypes ready, it is the goals to integrate 3DFed components with the underlying Virtuoso as SPARQL endpoint to write and read the data.

4.2 Integration into Partners' Products

Based on the foundational integration of elevaits platform with the federated SPARQL endpoint, the integration with the software products is mainly given due to the generic software architecture of the platform. However, to move on for a real product integration, we see 4 major steps. First, we must check the usage of 3DFed components with Virtuoso in the existing and constantly evolving continuous integration pipeline. That means unit, integration and performance tests must be updated and improved. Second, in close relation the deployment of the distributed database in a Kubernetes cluster must be enabled and topics like auto-scaling, monitoring, and altering need to be implemented and tested deeply. Third, if this works well, a dedicated migration path must be defined and implemented for the existing, productive installations. A main requirement is to ensure the zero-downtime on migration including no data-loss. Again, several tests must be developed and executed before its execution. Finally, and less uncritical than the steps before, the web applications enabling the users to work with the RDF data must be overhauled to leverage the real semantic power of the graph data. This includes check all existing visualization and interaction concepts.

5 Conclusion

We presented a detailed survey of state-of-the-art techniques in federated SPARQL query processing engines over multiple SPARQL endpoints and the RDF data partitioning. It is followed by the detailed technical layered-based architecture of the 3DFed. The reusable components and the strategies for their integration into other components as well as partners' products are discussed. We are aiming to develop a cost-based SPARQL query processing engine, a workload-based static data partitioner, and a component for dynamic RDF data exchange, when deployed in real-world environment.

References

[1] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. Lusail: A System for Querying Linked Data at Scale. *PVLDB*, 11(4):485–498, 2017.

[2] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *International Semantic Web Conference (ISWC)*, pages 18–34. Springer, 2011.

[3] Adnan Akhter, Axel-Cyrille Ngomo Ngonga, and Muhammad Saleem. An empirical evaluation of RDF graph partitioning techniques. In *European Knowledge Acquisition Workshop*, pages 3–18. Springer, 2018.

[4] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing Linked Datasets-On the Design and Usage of voiD, the'Vocabulary of Interlinked Datasets'. *Linked Data on the Web Workshop (LDOW)*, 2010.

[5] Günes Aluç, M Tamer Ozsu, Khuzaima Daudjee, and Olaf Hartig. chameleon-db: a workload-aware robust RDF data management system. *University of Waterloo, Tech. Rep. CS-2013-10*, 2013.

[6] Samita Bai and Shakeel A Khoja. Hybrid query execution on linked data with complete results. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 17(1):25–49, 2021.

[7] Kyoungsoo Bok, Junwon Kim, and Jaesoo Yoo. Dynamic partitioning supporting load balancing for distributed rdf graph stores. *Symmetry*, 11(7), 2019.

[8] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. Sparql web-querying infrastructure: Ready for action? In Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web – ISWC 2013*, pages 277–293, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[9] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[10] Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. SemaGrow: Optimizing Federated SPARQL Queries. In *International Conference on Semantic Systems (I-SEMANTICS)*, pages 121–128. ACM, 2015.

[11] Ali Davoudian, Liu Chen, Hongwei Tu, and Mengchi Liu. A workload-adaptive streaming partitioner for distributed graph stores. *Data Science and Engineering*, 6(2):163–179, 2021.

[12] Orri Erling. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[13] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[14] Orri Erling and Ivan Mikhailov. *Virtuoso: RDF Support in a Native RDBMS*, pages 501–519. Springer, 2010.

[15] Hugo Firth and Paolo Missier. Workload-aware streaming graph partitioning. In *EDBT/ICDT Workshops*. Citeseer, 2016.

[16] Hugo Firth and Paolo Missier. Taper: query-aware, partition-enhancement for large, heterogenous graphs. *Distributed and Parallel Databases*, 35(2):85–115, 2017.

[17] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: A distributed engine for efficient RDF processing. In *WWW Companion*, pages 267–268. ACM, 2014.

[18] Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Workshop on Consuming Linked Data (COLD)*, pages 13–24. CEUR, 2010.

[19] Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VoID Descriptions. In *O. Hartig, A. Harth, and J. F. Sequeda, editors, 2nd International Workshop on Consuming Linked Data (COLD 2011) in CEUR Workshop Proceedings*, volume 782, October 2011.

[20] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *Extending Database Technology (EDBT)*, pages 439–450, 2014.

[21] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed Mehdi Reza Beheshti, and Sherif Sakr. DREAM: Distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015.

[22] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDBJ*, 25(3):355–380, 2016.

[23] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *International Semantic Web Conference (ISWC)*, pages 211–224. Springer, 2007.

[24] Ali Hasnain, Syeda Sana e Zainab, Qaiser Mehmood, and Aidan Hogan. Reflections on profiling and cataloguing the content of SPARQL endpoints using SPORTAL. In Claudia d'Amato and Lalana Kagal, editors, *Proceedings of the Journal Track co-located with the 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26, 2019*, volume 2576 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[25] Ali Hasnain, Qaiser Mehmood, Syeda Sana E Zainab, Muhammad Saleem, Claude Warren, Jr, Durre Zehra, Stefan Decker, and Dietrich Rebholz-Schuhman. BioFed: Federated query processing over life sciences linked open data. *J. of Bio. Sem.*, 8(1):13, 2017.

[26] Katja Hose and Ralf Schenkel. WARP: Workload-aware replication and partitioning for RDF. In *ICDE Workshops*, pages 1–6, 2013.

[27] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[28] Daniel Janke, Steffen Staab, and Matthias Thimm. Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In *Workshop on Benchmarking Linked Data (BLINK)*. CEUR, 2017.

[29] Daniel Janke, Steffen Staab, and Matthias Thimm. On Data Placement Strategies in Distributed RDF Stores. In *International Workshop on Semantic Big Data (SBD)*, pages 1–6. ACM, 2017.

[30] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[31] Yasar Khan, Muhammad Saleem, Aftab Iqbal, Muntazir Mehdi, Aidan Hogan, Axel-Cyrille Ngonga Ngomo, Stefan Decker, and Ratnesh Sahay. Safe: policy aware sparql query federation over rdf data cubes. In *Proceedings of the 7th International Workshop on Semantic Web Applications and Tools for Life Sciences, Berlin, Germany, December 9-11, 2014.*, 2014.

[32] Muideen Lawal. *On Cost Estimation for the Recursive Relational Algebra*. PhD thesis, Université Grenoble Alpes [2020-....], 2021.

[33] Amgad Madkour, Ahmed M. Aly, and Walid G. Aref. WORQ: Workload-Driven RDF Query Processing. In *International Semantic Web Conference (ISWC)*, pages 583–599. Springer, 2018.

[34] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. The Odyssey Approach for Optimizing Federated SPARQL Queries. *International Semantic Web Conference (ISWC)*, page 471–489, 2017.

[35] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *International Conference on Data Engineering (ICDE)*, pages 984–994. IEEE, IEEE, 2011.

[36] Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. In *European Semantic Web Conference (ESWC)*, pages 524–538. Springer, 2008.

[37] Kurt Rohloff and Richard E. Schantz. High-Performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-Store. In *Programming Support Innovations for Emerging Distributed Applications (PSI EtA)*. ACM, 2010.

[38] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: The Linked SPARQL Queries Dataset. In *International Semantic Web Conference (ISWC)*, pages 261–269. Springer, 2015.

[39] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A fine-grained evaluation of SPARQL endpoint federation systems. *Sem. Web J.*, 7(5):493–518, 2016.

[40] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *Extended Semantic Web Conference (ESWC)*, pages 176–191. Springer, 2014.

[41] Muhammad Saleem, Shanmukha S. Padmanabhuni, Axel-Cyrille Ngonga Ngomo, Aftab Iqbal, Jonas S. Almeida, Stefan Decker, and Helena F. Deus. TopFed: TCGA Tailored Federated Query Processing and Linking to LOD. *J. Bio. Sem.*, 5:47, 2014.

[42] Muhammad Saleem, Alexander Potocki, Tommaso Soru, Olaf Hartig, and Axel-Cyrille Ngonga Ngomo. CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation. In *SEMANTICS*, pages 163–174. Elsevier, 2018.

[43] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 6 1984.

[44] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *The Semantic Web – ISWC 2011*, pages 601–616. Springer, 2011.

[45] Adrián Andrés Soto Suárez. Efficient processing of recursive and federated queries in sparql. 2021.

[46] Claus Stadlera, Muhammad Saleema, Qaiser Mehmoodb, Carlos Buil-Arandac, Michel Dumontierd, Aidan Hogane, and Axel-Cyrille Ngonga Ngomoa. Lsq 2.0: A linked dataset of sparql query logs. 2021.

[47] Bryan B Thompson, Mike Personick, and Martyn Cutcher. The bigdata® rdf graph database. In *Linked Data Management*, pages 193–237. CRC Press, 2014.

[48] Xin Wang, Thanassis Tiropanis, and Hugh Davis. LHD Optimising Linked Data Query Processing Using Parallelisation. In *Workshop on Linked Data on the Web (LDOW)*. CEUR, 2013.

[49] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *International Conference on Data Engineering (ICDE)*, pages 565–576, 2013.