Optimizing SPARQL Queries over Decentralized Knowledge Graphs

Christian Aebeloe [a,*], Gabriela Montoya [a] and Katja Hose [a]

[a] *Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, DK-9220 Aalborg Ø, Denmark*
E-mails: caebel@cs.aau.dk, gmontoya@cs.aau.dk, khose@cs.aau.dk

Abstract. While the Web of Data in principle offers access to a wide range of interlinked data, the architecture of the Semantic Web today relies mostly on the data providers to maintain access to their data through SPARQL endpoints. Several studies, however, have shown that such endpoints often experience downtime, meaning that the data they maintain becomes inaccessible. While decentralized systems based on Peer-to-Peer (P2P) technology have previously shown to increase the availability of knowledge graphs, even when a large proportion of the nodes fail, processing queries in such a setup can be an expensive task since data necessary to answer a single query might be distributed over multiple nodes. In this paper, we therefore propose an approach to optimizing SPARQL queries over decentralized knowledge graphs, called LOTHBROK. While there are potentially many aspects to consider when optimizing such queries, we focus on three aspects: cardinality estimation, locality awareness, and data fragmentation. We empirically show that LOTHBROK is able to achieve significantly faster query processing performance compared to the state of the art when processing challenging queries as well as when the network is under high load.

Keywords: LOTHBROK, Peer-to-Peer, characteristic sets, query optimization, cardinality estimation, data locality, SPARQL, RDF, knowledge graphs

1. Introduction

Due to the popularity of decentralized knowledge graphs on the Web, more and increasingly large knowledge graphs encoded in RDF are becoming available [1]. Furthermore, RDF knowledge graphs made available today are becoming exceedingly large. For instance, Wikidata [2] and Bio2RDF [3] contain more than 14 billion triples each. As a result, data providers experience an increasing burden of maintaining access to the datasets; and without any monetary incentives to do so, datasets often end up becoming unavailable [4–6] and outdated [7].

In recent years, several decentralized systems [6–11] have been proposed to alleviate the aforementioned burden from the data providers by reducing the computational load required to keep the data available, albeit using different methods to do so. For instance, Linked Data Fragments (LDF)-based approaches [9–13] reduce the computational load on the server by distributing some of the query processing effort to the client, ensuring that the server only processes requests with low time complexity. On the other hand, Peer-to-Peer (P2P) systems [6–8] remove the centralized point of failure that a server represents and replicate the data across several nodes in a decentralized fashion, ensuring that even if the uploading node fails, the data is still accessible. For instance, RDFPeers [14] uses a structured overlay over a P2P network that relies on Dynamic Hash Tables (DHTs) to determine where to replicate certain data. However, in situations where nodes frequently leave or join the network (i.e., churn), and data is often uploaded to the network, nodes have to go through a costly adjustment process to update the overlay and redistribute

*Corresponding author. E-mail: caebel@cs.aau.dk.

the data. Instead, systems like PIQNIC [6] and COLCHAIN [7] use unstructured P2P systems as foundation, where there is no global control over where data is replicated, making the network more stable under churn.

COLCHAIN builds upon PIQNIC and divides the entire network into communities of nodes that not only replicate the same data, but also collaborate on keeping certain data (fragments) up-to-date. This is done by using blockchain technology [15–18] where *chains* of updates maintain the history of changes to the data fragments. By linking such update chains to the data fragments in a community, COLCHAIN allows community participants to collaborate on keeping the data up-to-date while using consensus to make malicious updates less likely and allowing users to roll-back updates to an earlier version on request. Furthermore, the decentralized nature of COLCHAIN also increases the availability of the uploaded data by replicating the data on nodes within the community.

Nevertheless, while PIQNIC and COLCHAIN already use decentralized indexes [19] to determine where data is located during query time, subgraphs needed to answer a query are usually scattered across multiple nodes. Furthermore, the indexes provide limited information that prevents the nodes from considering locality and accurately estimating join cardinalities when optimizing queries. As a result, such systems often experience an unnecessarily large amount of intermediate results when processing a query. This problem is exacerbated by the decentralized nature of the systems, since the intermediate results have to be transferred between nodes, causing a significant communication overhead.

While there are potentially many aspects to consider when optimizing queries in a decentralized setup, we will focus on three such aspects: cardinality estimation, locality awareness, and data fragmentation. Suboptimal solutions to any of these three aspects can lead to an increased communication overhead and lower performance. For instance, while fragmenting large knowledge graphs into smaller fragments ensures that nodes do not have to replicate entire knowledge graphs, using a fragmentation technique that spreads out the data relevant to a single (sub)query across several fragments can increase the communication overhead since nodes might have to send an excessive number of requests to obtain all relevant data to answer a particular query [20–23]. On the other hand, inaccurate cardinality estimations can lead to a suboptimal join strategy that increases the amount of intermediate results and therefore runtime [24, 25]. And while several approaches have proposed reasonably accurate cardinality estimation techniques [24–26] over knowledge graphs, and for federated engines in particular [25, 27–29], such approaches cannot easily be transferred to a decentralized setup since nodes in a decentralized setup lack a global overview of the network and the data is scattered across multiple nodes. Finally, considering locality of the data when processing queries can help ensure that larger subqueries are delegated to nodes that can process them without communicating with other nodes, lowering the data transfer overall.

Nevertheless, while an optimization approach that maximizes the degree to which entire queries can be processed by a single node could decrease the communication overhead, a study [20] found that processing entire queries on one node can actually decrease the overall performance when the network is under heavy load, and that it is equally important to balance out the query load between nodes. As such, there is a need for a more holistic approach to query optimization that is able to delegate the processing of subqueries to other nodes in the network, thus reducing the communication overhead to the extent possible. For instance, query optimization techniques that are based on star-shaped subqueries have previously been shown to increase performance by at least an order of magnitude [10–12, 30]. This, and the fact that conjunctive subqueries are relatively efficient to process [31], means that decomposing and processing queries based on star-shaped subqueries can significantly reduce the communication overhead in decentralized systems.

In this paper, we therefore extend our work on PIQNIC [6] and COLCHAIN [7] in three aspects that work together to reduce the communication overhead when processing SPARQL queries, and in doing so, improve query processing performance in an approach that we call LOTHBROK. LOTHBROK adapts Characteristic Sets [10–12, 24] to fragment data in decentralized P2P systems. Furthermore, LOTHBROK builds upon Prefix-Partitioned Bloom Filters (PPBFs) [19] and proposes a new indexing scheme called Semantically Partitioned Bloom Filters (SPBFs) to obtain more accurate cardinality estimations. Lastly, LOTHBROK also introduces a locality-aware query optimization strategy that takes advantage of the SPBF indexes and is able to delegate the processing of (sub)queries to neighboring nodes in the network holding relevant data.

Figure 1 shows a high-level overview of the contributions of LOTHBROK, following the approach described above. First, knowledge graphs are fragmented using the Characteristic Set fragmentation, and indexed using the SPBF indexes. The query optimizer uses the information available in the SPBF indexes to build a query execution plan

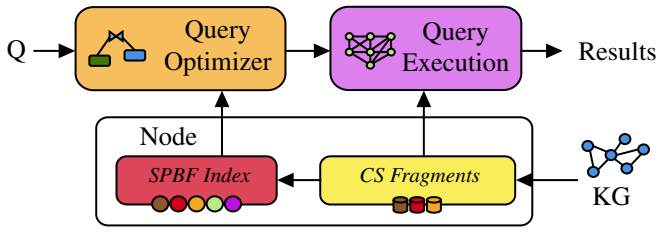


Fig. 1. Overview flow diagram of the contributions of LOTHBROK.

in consideration of data locality. To obtain the final results for a given query, the execution plan is finally executed over the network.

We evaluate LOTHBROK thoroughly using LargeRDFBench [32], a benchmark suite for federated RDF systems that comprises 13 datasets with over a billion triples and includes 40 queries of varying complexity and sizes of intermediate results. Furthermore, we evaluate LOTHBROK using synthetic data and queries from WatDiv [33] to test the scalability of LOTHBROK under load. Thus, in this paper, we focus on the query optimization problem for distributed knowledge graphs. Generalizing the approaches presented in the paper to other types of distributed graphs is an interesting topic for future work. Futhermore, since updates are managed by the underlying P2P layer, such as COLCHAIN [7], this paper focuses on static knowledge graphs. In summary, we make the following contributions:

- A data fragmentation technique that builds on Characteristic Sets [24]
- SPBF indexes adapted to the characteristic set fragmentation technique
- A cardinality estimation approach over decentralized RDF fragments using the SPBF indexes to provide more accurate cardinality estimations
- A locality-aware query optimization algorithm that uses SPBF indexes to delegate subqueries to neighboring nodes and reduce the communication overhead
- A thorough experimental evaluation of the impact of the presented techniques on query processing performance using real-world data from a well-known benchmark suite, and large-scale synthetic datasets

The paper is structured as follows: Section 2 discusses related work while Section 3 describes background information. Then, Section 4 presents LOTHBROK, Section 5 details how LOTHBROK optimizes queries, and Section 6 describes the query execution approach, while Section 7 presents our experimental evaluation. Lastly, Section 8 concludes the paper with an outlook to future work.

2. Related Work

The availability problem has prompted significant amount of research in the areas of decentralized query processing and decentralized architectures for knowledge graphs. In this section, we therefore discuss existing approaches related to LOTHBROK; client-server architectures, federated systems, and P2P systems.

2.1. Client-Server Architectures

SPARQL endpoints are Web services providing an HTTP interface that accepts SPARQL queries and remain some of the most popular interfaces for querying RDF data on the Web. However, several studies [4, 5] have found that such endpoints are often unavailable and experience downtime.

Linked Data Fragment (LDF) interfaces, such as Triple Pattern Fragments (TPF) [9], attempt to increase the availability of the server by shifting some of the query processing load towards the client while the server only processes requests with low time complexity. For instance, TPF servers only process individual triple patterns while the TPF clients process joins and other expensive operations. Today, several TPF clients exist that rely on either a greedy algorithm [9], a metadata based strategy [34], or star-shaped query decomposition combined with adaptive query processing techniques [35] to determine the join order of the triple patterns in a query. However, while in

all these approaches the server can handle more concurrent requests in comparison to SPARQL endpoints without becoming unresponsive, TPF naturally incurs a large network overhead when processing queries since intermediate bindings from previously evaluated triple patterns are transferred along with subsequently evaluated triple patterns to limit the amount of intermediate results, one by one. Furthermore, studies found that the performance of TPF is heavily affected by the type of triple pattern (i.e., the position of variables in the triple pattern) [13] and the shape of the query [36, 37].

Several different systems have since been proposed to lower the network overhead. For instance, Bindings-Restricted TPF (brTPF) [38] bulks bindings from previously evaluated triple patterns such that multiple bindings can be attached to a single request. While this reduces the number of requests made for a triple pattern, it still incurs a somewhat large data transfer overhead, since each request still evaluates a single triple pattern. hybridSE [39] combines a brTPF server with a SPARQL endpoint and takes advantage of the strengths of each approach; subqueries with large numbers of intermediate results are sent to the SPARQL endpoint to overcome the limitations posed by LDF systems. However, hybridSE often answers complex queries using the SPARQL endpoint and is thus vulnerable to server failure.

To further limit the network overhead, Star Pattern Fragments (SPF) [11] clients send conjunctive subqueries in the shape of stars (star patterns) to the server and process more complex patterns locally on the client. Such conjunctive subqueries can be processed relatively efficiently by the server [31], which results in the transfer of significantly fewer intermediate results than in systems like TPF and brTPF. On the other hand, Smart-KG [12] ships predicate-family partitions (i.e., characteristic sets) to the client and processes the entire query locally; however, triple patterns with infrequent predicate values (according to a certain threshold) are sent to and evaluated by the server. While this takes advantage of the distributed resources that the clients possess, Smart-KG often ends up transferring excessive amounts of data unnecessarily since entire partitions of a dataset are transferred regardless of any bindings from previously evaluated star patterns. WiseKG [10] combines SPF and Smart-KG and uses a cost model to determine which strategy (SPF or Smart-KG) is the most cost-effective to process a given star-shaped subquery. Like SPF and Smart-KG, WiseKG processes more complex patterns on the client. Nevertheless, all the aforementioned LDF approaches rely on a centralized server or a fixed set of servers that are subject to failure.

Lastly, different from LDF approaches, SaGe [40] decreases the load on the server by suspending queries after a fixed time quantum to prevent long-running queries from exhausting server resources; the queries can then be restarted by making a new request to the server. However, SaGe processes entire, and possibly complex, queries on the server, and as stated above, such servers are subject to failure.

2.2. Federated Systems

Federated systems enable answering queries over data spread out across multiple independent SPARQL endpoints [41–45] or LDF servers [46] offering access to different datasets. While such approaches spread out query processing over several servers, lowering the load on each individual server, they sometimes generate suboptimal query execution plans that increase the number of intermediate results and the load on individual servers [47]. As such, several approaches [25, 27–29, 48, 49] have attempted to optimize federated queries in different ways. For instance, [44] builds an index over time by remembering which endpoints in the federation can provide answers to which triple patterns. Furthermore, [48] decomposes queries into subqueries that can be evaluated by a single endpoint. While [48] uses a similar query decomposition strategy as LOTHBROK, they target federations over SPARQL endpoints, and as previously mentioned, such endpoints suffer from availability issues. On the other hand, [25, 49] estimate the selectivity of joins to produce more efficient join plans. For instance, [25] uses characteristic sets [24] and pairs [50] to index the data in the federation and combines this with Dynamic Programming (DP) to optimize query execution plans. Furthermore, [46] proposes an interface for processing federated queries over heterogeneous LDF interfaces. To achieve this, the query optimizer is adapted to the characteristics of the different interfaces as well as the locality of the data, i.e., knowledge of which nodes hold which data. Inspired by these approaches, LOTHBROK fragments knowledge graphs based on characteristic sets and uses a similar cardinality estimation technique to optimize join plans in consideration of data locality in the network.

2.3. Peer-to-Peer Systems

Peer-to-Peer (P2P) systems [6–8, 14, 23, 51, 52] tackle the availability issue from a different perspective: by removing the central point of failure completely and replicating the data across multiple nodes in a P2P network, they can ensure the data remains available even if the original node that uploaded the data fails. As such, they consist of a set of nodes (often resource limited) that act both as servers and clients, maintaining a limited local datastore. The structure of the network, i.e., connections between the nodes, as well as data placement (data allocation), varies from system to system. For instance, some systems [8, 14, 51] enforce data placement by applying a structured overlay over the network, such as Dynamic Hash Tables (DHTs) [53]. On the other hand, PIQNIC [6] imposes no structure on top the network; nodes are connected randomly to a set of neighbors that are shuffled periodically with another node's neighbors to increase the degree of joinability between the fragments of neighboring nodes. Lastly, COLCHAIN [7] extends PIQNIC and divides the entire network into smaller communities of nodes that collaborate on keeping certain data available and up-to-date. By applying community-based ledgers of updates and relying on a consensus protocol within a community, COLCHAIN lets users actively participate in keeping the data up-to-date.

Each P2P system has different ways of processing queries. For instance, due to the lack of global knowledge over the network, basic P2P systems have to flood the network with requests for a given horizon to increase the likelyhood of receiving complete query results. To counteract this, distributed indexes [19, 29, 54] like Prefix-Partitioned Bloom Filter (PPBF) indexes [19] determine which nodes may include relevant data for a given query and thus allow the system to prune nodes from consideration during query optimization. Yet, the aforementioned systems still experience a significant overhead partly caused by inaccurate cardinality estimations, query optimization that does not consider the locality of data, as well as data fragmentation that splits up closely related data. For instance, PIQNIC and COLCHAIN both use a predicate-based fragmentation strategy that creates a fragment for each predicate. This, together with the replication and allocation strategy used, means that data relevant to a single query is distributed over a significant number of fragments and nodes.

However, while an approach that maximizes the degree to which entire queries can be processed by one node can lower the communication overhead, distributing some of the query processing load across multiple nodes is equally important when optimizing queries in a decentralized context [20] to avoid overloading individual nodes. As such, LOTHBROK introduces a query optimization technique that distributes the processing of subqueries to nodes in the network based on data locality and fragment compatibility, while the characteristic set fragmentation technique allows entire star-shaped subqueries to be processed on the same node.

3. Background

A commonly used format for storing semantic data is the Resource Description Framework (RDF) [55]. RDF structures data as triples, defined as follows.

Definition 1 (RDF Triple)**.** *Let I, B, and L be the disjoint sets of IRIs, blank nodes, and literals. An RDF triple is a triple t of the form* $t = (s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$*, where s, p, and o are called subject, predicate, and object.*

Given the definition of an RDF triple, a *knowledge graph* $\mathcal{G}$ is a finite set of RDF triples. The most popular language to query knowledge graphs is SPARQL [56]. A SPARQL query consists of one or more *triple patterns*. A triple pattern t is a triple of the form $t = (s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ where V is the set of all variables. A Basic Graph Pattern (BGP) is a set of triple patterns. Without loss of generality, we focus our discussion in the main part of this paper on BGPs and describe in Section 5 how our approach can support other operators, such as `UNION` and `OPTIONAL`; our experimental evaluation in Section 7 includes queries with a variety of SPARQL operators including `UNION` and `OPTIONAL`.

A complex BGP P can be decomposed into a set of *star patterns*. A star pattern P' is a set of triple patterns that share the same subject, i.e., $\forall t_1 = (s_1, p_1, o_1), t_2 = (s_2, p_2, o_2)$ such that $t_1, t_2 \in P'$, it is the case that $s_1 = s_2$. Note that while star patterns can be defined as both subject-based and object-based star patterns, for ease of presentation, we focus on subject-based star patterns only since subject-subject joins are much more

common in real query loads [57]; LOTHBROK can trivially be adapted to object-based star patterns by using the same principles presented in this paper for object-object joins rather than subject-subject joins. Given a BGP $P = \{(s_1, p_1, o_1), \ldots, (s_n, p_n, o_n)\}$, the set of *subjects* in P, denoted S_P, is the set of distinct subject values across the triples in P, i.e., $S_P = \{s \mid (s, p, o) \in P\}$.

Definition 2 (Star Decomposition [11]). *Given a BGP $P = \{t_1, \ldots, t_n\}$ with subjects $S_P = \{s_1, \ldots, s_m\}$, the* star decomposition *of P, $\mathcal{S}(P) = \{P_s(P) \mid s \in S_P\}$, is a set of star patterns $P_s(P)$ for each $s \in S_P$, such that $P = \cup_{s \in S_P} P_s(P)$ where $P_s(P) = \{(s', p', o') \mid (s', p', o') \in P \wedge s' = s\}$.*

The answer to a BGP P over a knowledge graph $\mathcal{G}$ is a set of *solution mappings*, defined as follows.

Definition 3 (Solution mapping [7, 9]). *Given the sets I, B, L, V from above, a* solution mapping *μ is a partial mapping $\mu : V \mapsto (U \cup B \cup L)$.*

Given a BGP P and a solution mapping μ, the notation $\mu[P]$ denotes the triple (patterns) obtained by replacing variables in P according to the bindings in μ. Furthermore, $dom(\mu)$ returns the *domain* of μ, i.e., the set of variables that are bound in μ, and $vars(P)$ returns the variables in P. Given a knowledge graph $\mathcal{G}$ and BGP P, $[[P]]_\mathcal{G}$ denotes the set of solution mappings that constitutes the answer to P over $\mathcal{G}$, i.e., $\forall \mu \in [[P]]_\mathcal{G}, \mu[P] \subseteq \mathcal{G}$ and $dom(\mu) = vars(P)$. $[[P]]_\mathcal{G}$ contains all possible solution mappings that satisfy the previous conditions. A set of triples T is said to be *matching* a BGP P over a knowledge graph $\mathcal{G}$, denoted $T_\mathcal{G}[P]$, iff $\exists \mu \in [[P]]_\mathcal{G}$ where $T = \mu[P]$.

Since updates to the data are managed by the underlying P2P layer, solution mappings (Definition 3) are defined independently from the updating process. That is, in the general case, solution mappings are obtained on query time over the latest version of the knowledge graphs. To expand our work to support dynamic datasets, we could make use of the underlying P2P layer; if a dataset changes, the original node recomputes the index and broadcasts the update throughout the network. This is what systems like COLCHAIN [7] do, and in our experimental evaluation (Section 7) we have already implemented LOTHBROK on top of COLCHAIN. Furthermore, conflicting or inconsistent datasets in a LOTHBROK network could lead to unexpected or erroneous results [58] when querying. However, the focus of this paper is on query optimization techniques and considering the quality of the datasets in LOTHBROK is outside its scope. Nevertheless, in the future, we could expand LOTHBROK with existing knowledge graph quality management techniques (e.g., [58–60]) to mitigate this problem. Thus, we refer to related work for more details on handling dynamic [7] or inconsistent [58–60] datasets.

3.1. Peer-to-Peer

In its simplest form, an unstructured P2P system consists of a set of interconnected nodes that all maintain a local datastore managing a set of (partial) knowledge graphs, where each node maintains a local view over the network, i.e., a set of *neighboring* nodes (nodes within the local view over the network).

Formally, we define a P2P network N as a set of interconnected nodes $N = \{n_1, \ldots, n_n\}$ where each node maintains a local datastore and a local view over the network. The data uploaded to a node in N is replicated throughout the network. Furthermore, in line with previous work [7, 19], each node maintains a distributed index describing the knowledge graphs reachable within a certain number of steps (also known as hops), called the *horizon* of a node. A node n is defined as follows:

Definition 4 (Node [6, 19]). *A node n is a triple $n = (G, I, N)$ where:*

- *G is the set of knowledge graphs in n's local datastore*
- *I is n's distributed index*
- *N is a set of neighboring nodes*

While maintaining the structure of the network is important for P2P systems, it is not relevant for the data and query processing techniques that this paper is focusing on. As such, we do not go into detail on network topology, data replication and allocation, and periodic shuffles. Instead, we refer the interested reader to related work such as [6, 7] for more details. In the following, we define data fragmentation and introduce a running example.

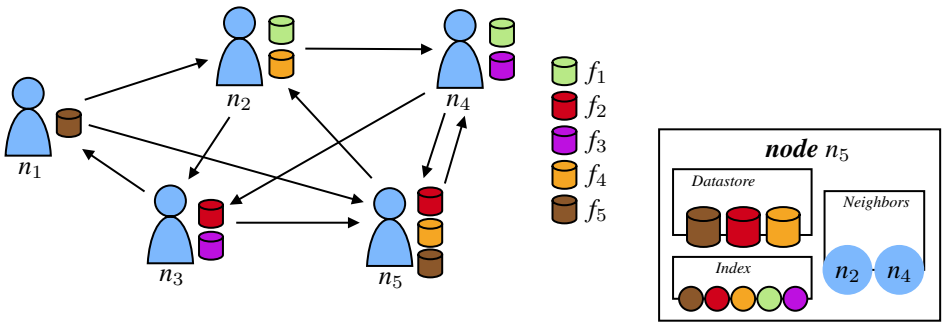
In line with previous work [6, 7], and to avoid having to replicate large knowledge graphs throughout the network, LOTHBROK divides knowledge graphs into smaller disjoint *fragments*, i.e., partial knowledge graphs, which can be

replicated more easily. Fragments can be obtained using a *fragmentation* function. A fragmentation function is a function that, given a knowledge graph, returns a set of disjoint fragments, and is formally defined as follows:

Definition 5 (Fragmentation Function [6, 7])**.** *A* fragmentation *function $\mathcal{F}$ is a function that maps a knowledge graph $\mathcal{G}$ to a set of disjoint* knowledge graph fragments*, i.e., $\mathcal{F} : \mathcal{G} \mapsto 2^{\mathcal{G}}$ such that $\bigcup_{f \in \mathcal{F}(\mathcal{G})} f = \mathcal{G}$, and $\forall f_1, f_2 \in \mathcal{F}(\mathcal{G})$, $f_1 \cap f_2 = \emptyset$.*

Different fragmentation functions can have different granularities. For instance, the most coarse-granular fragmentation function is $\mathcal{F}_C(\mathcal{G}) = \{\mathcal{G}\}$, i.e., the fragmentation function does not split up the original knowledge graph. ColChain [7] as well as Piqnic [6] use a *predicate-based* fragmentation function for $\mathcal{G}$, i.e., $\mathcal{F}_P(\mathcal{G}) = \{\{(s', p', o') \mid (s', p', o') \in \mathcal{G} \wedge p' = p\} \mid \exists s, o : (s, p, o) \in \mathcal{G}\}$, which creates a fragment for each unique predicate in $\mathcal{G}$. Lothbrok uses a fragmentation function based on characteristic sets [24] (i.e., predicate families) that is detailed in Section 4.2.

The fragments created by the fragmentation function are replicated and allocated at multiple nodes in the network to ensure availability in case the original provider of the knowledge graph becomes unavailable and to enable load balancing. The replication and allocation factor are parameters of the underlying network; for instance, in Piqnic [6], fragments are replicated and allocated across the node's neighbors, and nodes index all fragments available within a certain horizon. On the other hand, ColChain [7] replicates and allocates fragments at nodes that participate within the same communities. Since this paper focuses on data fragmentation and query optimization, we omit details on data replication and allocation and refer the interested reader to related work [6, 7] for details.



(a) Example network N **(b)** Node n_5

Fig. 2. (a) Example of an unstructured P2P network $N = \{n_1, \ldots, n_5\}$ and (b) architecture of a single node n_5 that indexes data within a horizon of 2 nodes.

Consider, as a running example, the unstructured P2P network in Figure 2a consisting of five nodes ($N = \{n_1, \ldots, n_5\}$) that replicate a total of five fragments ($f_1, \ldots, f_5$). In this example, each node maintains a set of two neighbors and each fragment is replicated across two nodes. For instance, node n_5 has $\{n_2, n_4\}$ as its set of neighbors, and replicates the fragments $\{f_2, f_4, f_5\}$ in its local datastore. While the running example is based on an unstructured network, such as the one presented in [6], Lothbrok could be adapted to more structured setups, such as the one presented in [7].

3.2. Distributed Indexes

To speed up query processing performance, systems like Piqnic [6] and ColChain [7] use distributed indexes [19, 54] to efficiently identify nodes holding relevant data for a given SPARQL query. The indexes capture information about the fragments stored locally at the node itself as well as information about fragments that can be accessed via its neighbors.

A distributed index, as defined in [7, 19], is a structure that maps the triple patterns in a query to nodes that hold relevant fragments to those triple patterns. In line with [7, 19], we thus define distributed indexes as follows.

Definition 6 (Distributed Index [7, 19])**.** *Let N be a P2P network, n be a node such that $n \in N$, $\mathcal{T}$ be the set of all possible triple patterns, and $\mathcal{F}$ be the set of fragments that n has access to within its local view over the network. A* distributed index *on n is a tuple $I_n = (\nu, \eta)$ with $\nu : \mathcal{T} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^N$. For a triple pattern t, $\nu(t)$ returns the set of fragments in $\mathcal{F}$ that t matches. For a fragment $f \in \mathcal{F}$, $\eta(f)$ returns the nodes on which f is located.*

Given a node n, n's distributed index is denoted I_n. Given the definition of a distributed index, we define a *node mapping* as a mapping from a triple pattern t in a BGP P to a set of nodes that contain relevant fragments to t, as follows:

Definition 7 (Node Mapping [7, 19])**.** *For any BPG P and distributed index I, there exists a function $match(P, I)$ that returns a* node mapping *$M : P \mapsto 2^N$, such that $\forall t \in P$, $M(t) = \bigcup_{f \in \nu(t)} \eta(f)$, i.e., $M(t)$ returns the indexed nodes that have fragments holding data matching the triple t.*

To build the index for a node's local view over the network, nodes share partial indexes, i.e., partial mappings, for the fragments that they have access to, called index *slices*. An index slice for a fragment is a partial mapping from triple patterns to the fragments that contain relevant triples to the triple patterns, as well as a mapping from the fragment to the nodes that replicate it, and is defined as follows:

Definition 8 (Index Slice [7, 19])**.** *Let f be a fragment. The index* slice *of f, s_f, is a tuple $s_f = (\nu', \eta')$, where $\nu'(t)$ returns $\{f\}$ if there exists a triple in f that matches t, or $\emptyset$ otherwise, and $\eta'(f)$ returns the set of all nodes that contain f in their local datastore. The function $s(f)$ returns the index slice describing f, i.e., $s(f) = s_f$.*

Index slices for the fragments that a node has access to are combined into a distributed index for that particular node using the $\oplus$ operator[1]. The distributed index is then used to check the relevancy and overlap of fragments during query time to optimize the query. Given a set of slices S, the index obtained by combining the slices in S, $I(S)$, can be computed using the formula in Equation 1 [7, 19].

$$I(S) = \left(\bigoplus_{s \in S} s.\nu', \bigoplus_{s \in S} s.\eta' \right) \tag{1}$$

While the definition of distributed indexes allows for several different types of indexes, the index slices used in PIQNIC [6] and COLCHAIN [7] correspond to Prefix-Partitioned Bloom Filters (PPBFs) [19], which extend regular Bloom filters [61]. Given a set $\mathcal{S}$ of IRIs, a Bloom filter $\mathcal{B}$ for $\mathcal{S}$ is a tuple $\mathcal{B} = (\hat{b}, H)$ where $\hat{b}$ is a bitvector of size m and H is a set of k hash functions [19]. Each hash function in H maps the elements from $\mathcal{S}$ (i.e., IRIs) to a position in $\hat{b}$; these positions are thus set to 1 whereas the positions not mapped to by a function in H are 0. In other words, index slices in [19] represent the set of entities in a fragment as bitvectors following the approach described above. Looking up whether an element e is in $\mathcal{S}$ using the Bloom filter for $\mathcal{S}$ is done by hashing e using the hash functions in H and checking the value of each position in $\hat{b}$. If at least one of those positions is set to 0, it is certain that $e \notin \mathcal{S}$. However, if all corresponding bits are set to 1, it is not certain that $e \in \mathcal{S}$, since it could be a false positive caused by hash collisions, i.e., different values are mapped to the same positions in the underlying bitvector. In this case, we say that e *may* be in $\mathcal{S}$, denoted $e \sqsubseteq \mathcal{S}$.

To check the compatibility of two fragments relevant for conjunctive triple patterns, we check whether or not they produce any join results. To do this, we could check whether or not the intersection of the bitvectors describing the subjects and objects of the fragments is empty (i.e., if they have some IRI in common). Given two Bloom filters $\mathcal{B}_1 = (\hat{b}_1, H)$ and $\mathcal{B}_2 = (\hat{b}_2, H)$, the intersection of $\mathcal{B}_1$ and $\mathcal{B}_2$ is approximated by the logic `AND` operation between $\hat{b}_1$ and $\hat{b}_2$, $\mathcal{B}_1 \cap \mathcal{B}_2 \approx \hat{b}_1 \& \hat{b}_2$.

To avoid exceedingly large bitvectors, PPBFs partition the bitvector based on the prefix of the IRIs. The prefix of an IRI u corresponds to the IRI of the namespace of u[2]. The name of an IRI is then the IRI minus the pre-

[1] $\oplus$ is defined in [7, 19] as $(f \oplus g)(x) = f(x) \cup g(x)$ if f and g are defined at x; $(f \oplus g)(x) = f(x)$ if f is defined at x and g is *not* defined at x; $(f \oplus g)(x) = g(x)$ if g is defined at x and f is *not* defined at x; $(f \oplus g)(x) = \emptyset$ if neither f nor g is defined at x.

[2] As defined in `https://www.w3.org/TR/rdf11-concepts/#vocabularies`.

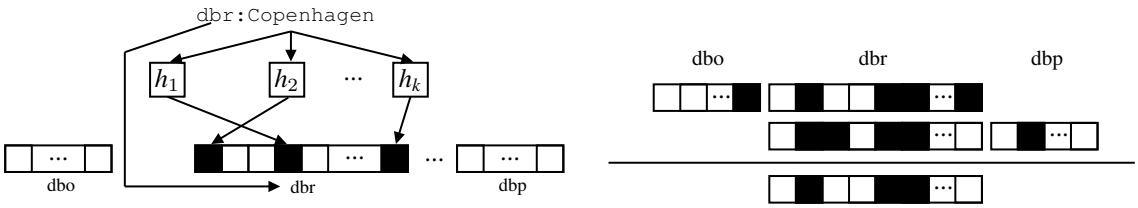
fix. For instance, the IRI `http://dbpedia.org/resource/Denmark` has the prefix (i.e., namespace IRI) `http://dbpedia.org/resource/` and the name `Denmark`. A PPBF is formally defined in [19] as follows.

Definition 9 (Prefix-Partitioned Bloom Filter [19])**.** *A PPBF $\mathcal{B}^P$ is a 4-tuple $\mathcal{B}^P = (P, \hat{B}, \theta, H)$ where*

- *P a set of prefixes*
- *$\hat{B}$ is a set of bitvectors such that $\forall \hat{b}_1, \hat{b}_2 \in \hat{B} : |\hat{b}_1| = |\hat{b}_2|$*
- *$\theta : P \rightarrow \hat{B}$ is a prefix-mapping function such that $\forall p_1, p_2 \in P$ where $p_1 \neq p_2$, $\theta(p_1) \neq \theta(p_2)$.*
- *H is a set of hash functions*

For each $p_i \in P$, $\mathcal{B}_i = (\theta(p_i), H)$ is the Bloom Filter that encodes the name of the IRIs with prefix p_i and is called a partition of $\mathcal{B}^P$.

Consider the example where the IRI `dbr:Copenhagen` is inserted into a PPBF, visualized in Figure 3a. In this case, the IRI is matched to the prefix `dbr`, and the IRI is hashed using each hash function in the PPBF; each corresponding bit in the bitvector for the `dbr` prefix is thus set to 1.



(a) PPBF $\mathcal{B}_1^P$ **(b)** $\mathcal{B}_1^P \cap \mathcal{B}_2^P$

Fig. 3. Example of (a) inserting an IRI into a PPBF $\mathcal{B}_1^P$ and (b) intersection between two PPBFs $\mathcal{B}_1^P \cap \mathcal{B}_2^P$ [19].

Like for regular Bloom filters, we say that an IRI i with prefix p *may* be in a PPBF $\mathcal{B}^P$, denoted $i \sqsubseteq \mathcal{B}^P$, if and only if all positions given by $h(i)$ such that $h \in H$ are set to 1 in the bitvector $\theta(p)$. PPBFs are used by PIQNIC and COLCHAIN to prune non-overlapping fragments of joining triple patterns from the query execution plan (i.e., the $match(P, I)$ function in Definition 7). This is done by finding the intersection of the two PPBFs to check whether or not they overlap; if the intersection of the two PPBFs is empty, the corresponding fragments do not produce any join results. The PPBF intersection is defined in [19] as follows.

Definition 10 (Prefix-Partitioned Bloom Filter Intersection [19])**.** *The intersection of two PPBFs with the same set of hash functions H and bitvectors of the same size, denoted $\mathcal{B}_1^P \cap \mathcal{B}_2^P$, is $\mathcal{B}_1^P \cap \mathcal{B}_2^P = (P_\cap, \hat{B}_\cap, \theta_\cap, H)$, where $P_\cap = \mathcal{B}_1^P.P \cap \mathcal{B}_2^P.P$, $\hat{B}_\cap = \{\mathcal{B}_1^P.\theta(p) \,\&\, \mathcal{B}_2^P.\theta(p) \mid p \in P_\cap\}$, and $\theta_\cap : P_\cap \rightarrow \hat{B}_\cap$.*

Consider the example intersection visualized in Figure 3b. As described above, the intersection of two PPBFs is the bitwise `AND` operation on the bitvectors for the prefixes that $\mathcal{B}_1^P$ and $\mathcal{B}_2^P$ have in common. In this example, $\mathcal{B}_2^P$ does not have a bitvector with the prefix `dbp`, thus this partition is omitted from the intersection. Similarly, the bitvector partition with the `dbo` prefix is omitted. Since both PPBFs have bitvectors for the `dbr` prefix, the resulting PPBF has one partition for the `dbr` prefix that is a result of the bitwise `AND` operation between the two corresponding partitions in $\mathcal{B}_1^P$ and $\mathcal{B}_2^P$.

Furthermore, given a partitioned bitvector $\mathcal{B}$ and $\hat{b} \in \mathcal{B}.\hat{B}$, let $t(\hat{b})$ be a function that returns the number of bits in $\hat{b}$ that are set. Then, the estimated cardinality of a partitioned bitvector $\mathcal{B}$, denoted $card^P(\mathcal{B})$, is the sum of the estimated cardinality for all bitvector partitions in $\mathcal{B}.\hat{B}$ [19, 62] and is formally defined as follows:

$$card^P(\mathcal{B}) = \sum_{\hat{b} \in \mathcal{B}.\hat{B}} \frac{ln(1 - t(\hat{b})/|\hat{b}|)}{|\mathcal{B}.H| \cdot ln(1 - 1/|\hat{b}|)} \tag{2}$$

Consider, for instance, a PPBF $\mathcal{B}^P$ such that $|\mathcal{B}^P.H| = 5$ and that $|\hat{b}| = 20000$ for all $\hat{b} \in \mathcal{B}^P.\hat{B}$ with two partitions, `dbr` and `dbp`. Since the partitioned bitvector has two partitions, obtaining the estimated cardinality for $\mathcal{B}^P$ is the

sum of estimating the cardinality of both prefix partitions. Let the number of set bits in the bitvector for the dbr prefix be 736 and the number of set bits in the bitvector for the dbp prefix be 249. Then, the estimated cardinality using Equation 2 is:

$$card^P(\mathcal{B}^P) = \frac{ln(1 - 736/20000)}{5 \cdot ln(1 - 1/20000)} + \frac{ln(1 - 249/20000)}{5 \cdot ln(1 - 1/20000)} \approx \frac{-0.0375}{-0.00025} + \frac{-0.0125}{-0.00025} \approx 150 + 50 \approx 200$$

4. The LOTHBROK Approach

Differently from PIQNIC and COLCHAIN, LOTHBROK uses a fragmentation strategy based on characteristic sets. To accommodate efficient query processing over such fragments, as well as to enable locality-awareness and more accurate cardinality estimation, LOTHBROK introduces an indexing scheme that maps star patterns to fragments rather than triple patterns. In the remainder of this section, we provide a brief overview of the LOTHBROK architecture and how LOTHBROK optimizes SPARQL queries over decentralized knowledge graphs, followed by a formal definition of the fragmentation and indexing approach. Query optimization with details on how to exploit locality-awareness and join ordering are explained in Section 5.

4.1. Design and Overview

LOTHBROK introduces three contributions, that altogether decrease the communication overhead and in doing so increases query processing performance. First, LOTHBROK creates fragments based on characteristic sets such that entire star patterns can be answered by a single fragment. This is beneficial since, as we discussed in Section 1, such star patterns are relatively efficiently processed by the nodes [31] and reduce the communication overhead. The characteristic set of a subject value (entity) is the set of predicates that occur in triples with that subject. As such, LOTHBROK creates one fragment per unique characteristic set and each fragment thus contains all the triples with the subjects that match the characteristic set of the fragment. Consider, for instance, the example network in Figure 2 and query Q shown in Figure 4a. Figure 4b shows the characteristic sets of each fragment in the network. Using this fragmentation method, each fragment can provide answers to entire star patterns; for instance, $P_3 \in \mathcal{S}(\mathcal{Q})$ can be processed over just f_5, since it is the only fragment containing triples with both predicates present in P_3. The formal definition of the fragmentation approach is presented in Section 4.2.

```
select * where {
  ?person dbo:nationality ?country . # tp1 (P1)
  ?person dbo:author ?publication . # tp2 (P1)
  ?country dbo:capital ?capital . # tp3 (P2)
  ?country dbo:currency ?currency . # tp4 (P2)
  ?publication dbo:publisher ?publisher . #tp5 (P3)
  ?publication dbp:language ?language . #tp6 (P3)
}
```

(a) Query Q

Fragment	CS
f_1	{dbo:nationality,dbo:author,dbo:deathDate}
f_2	{dbo:nationality,dbo:author}
f_3	{dbo:capital,dbo:currency,dbo:population}
f_4	{dbo:capital,dbo:currency}
f_5	{dbo:publisher,dbo:language}

(b) CSs of each fragment in the running example

Fig. 4. (a) Example SPARQL query Q and (b) corresponding characteristic sets in the example network.

Second, to accommodate processing entire star patterns over individual fragments, and to encode structural information that can be used for cardinality estimation and locality awareness, LOTHBROK introduces a novel indexing scheme, called Semantically Partitioned Blooms Filter (SPBF) Indexes, that builds upon the Prefix-Partitioned Bloom Filter (PPBF) indexes presented in [19]. In particular, SPBFs partition the bitvectors based on the IRI's position in the fragment, i.e., whether it is a subject, predicate, or object. For instance, in the running example, the SPBF for f_5 contains a partition encoding all the subjects with the characteristic set

{dbo:publisher,dbo:language}, as well as partitions encoding all the objects in f_5 that occur in a triple with each predicate. The formal definition of SPBF indexes is discussed in Section 4.3.

Third, LOTHBROK proposes a query optimization technique that takes advantage of the fragmentation based on characteristic sets and the SPBF indexes to estimate cardinalities and consider data locality while optimizing the query execution plan. First, LOTHBROK builds a *compatibility graph* using the SPBF indexes that describes, for a given query, which fragments are compatible with one another for each star join in the query (i.e., which fragments may produce results for the joins). In other words, the nodes in a compatibility graph denote the relevant fragments, and the edges denote which fragments may produce join results with one another for the given query. Then, LOTHBROK builds a query execution plan using a Dynamic Programming (DP) algorithm that considers the compatibility of fragments in the compatibility graph and the locality of the fragments in the index. The query execution plan built by the query optimizer follows a left-deep approach that uses the bindings obtained from previously evaluated subplans as input (filter) when processing joins. Furthermore, the plan obtained from this step includes all the relevant fragments (i.e., only non-relevant fragments are pruned).

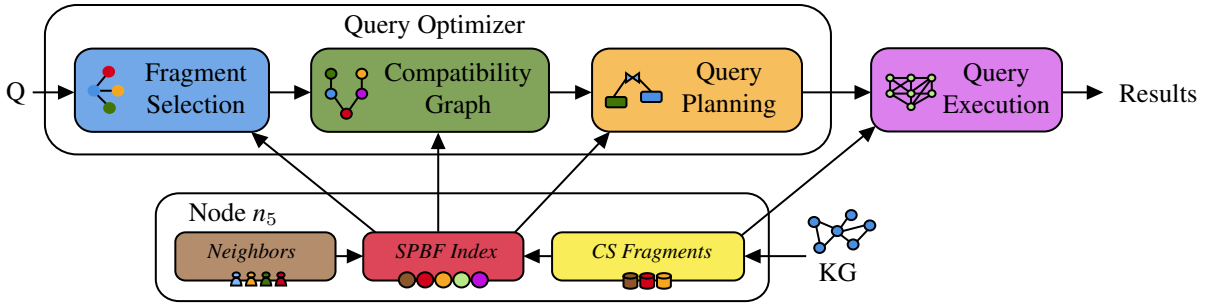


Fig. 5. Flow diagram of the contributions of LOTHBROK (extended from Figure 1).

Figure 5 shows an extended version of Figure 1 using the principles explained in Section 3.1 and the contributions explained above. The query optimizer thus contains three sequential steps; (1) fragment selection, (2) compatibility graph, and (3) query planning, that altogether computes the query execution plan for a given query.

Notice that for star patterns P with a large number of solution mappings, $[[P]]_{\mathcal{G}}$ could become computationally expensive to enumerate. However, as shown in our experimental evaluation in Section 7, LOTHBROK actually decreases the number of intermediate results to be enumerated by up to two orders of magnitude compared to triple pattern-based query executors using the optimization techniques explained above. Furthermore, we apply pagination of large result sets, which related studies [9, 38] have already shown can effectively limit the effects of a large number of solution mappings, even for star pattern-based query execution [11].

In the remainder of this section, we detail data fragmentation (Section 4.2) and indexing (Section 4.3) in LOTHBROK. Section 5 details the query optimization approach used by LOTHBROK.

4.2. Data Fragmentation

As discussed in Section 1, star-shaped subqueries can be processed relatively efficiently over a fragment [31], thus they can also help achieving a better balance between reducing the communication overhead and distributing the query processing load [10–12]. To facilitate processing such star patterns on single nodes, we propose to fragment the uploaded knowledge graphs based on *characteristic sets* [10, 12, 24]. This is the first contribution as explained in Section 4.1, and corresponds to the *CS Fragments* step in Figure 5. Formally, a characteristic set is defined as follows:

Definition 11 (Characteristic Set [10, 12, 24])**.** *The characteristic set for a subject s in a given knowledge graph $\mathcal{G}$, $C_{\mathcal{G}}(s)$, is the set of predicates associated with s, i.e., $C_{\mathcal{G}}(s) = \{p \mid (s,p,o) \in \mathcal{G}\}$. The set of characteristic sets of a knowledge graph $\mathcal{G}$ is $C(\mathcal{G}) = \{C_{\mathcal{G}}(s) \mid (s,p,o) \in \mathcal{G}\}$.*

In other words, the characteristic set of a subject is the set of predicates (i.e., predicate combination) used to describe the subject, i.e., that occur in the same triples as the subject. For instance, if the triples (`dbr:Denmark`,`dbo:capital`,`dbr:Copenhagen`) and (`dbr:Denmark`,`dbo:currency`,`dbr:Danish_Krone`) are the only ones with subject `dbr:Denmark`, then this subject is described by the characteristic set {`dbo:capital`,`dbo:currency`}.

Characteristic sets were first introduced in [24], used for cardinality estimation and, in extension of that, join ordering. WiseKG [10] and Smart-KG [12] used the notion of characteristic sets for fragmentation of knowledge graphs in LDF systems to balance the query load between clients and servers. In this paper, we use characteristic set based fragments as an alternative to the purely predicate-based fragmentation used by for example PIQNIC. We define the characteristic set based fragmentation function as follows:

Definition 12 (Characteristic Set Fragmentation Function)**.** *Let $\mathcal{G}$ be a knowledge graph, then the* characteristic set fragmentation function *of $\mathcal{G}$, $\mathcal{F}_C(\mathcal{G})$, is defined using the notation introduced in Definition 11, as:*

$$\mathcal{F}_C(\mathcal{G}) = \{\{(s,p,o) \mid (s,p,o) \in \mathcal{G} \wedge C_{\mathcal{G}}(s) = C_i\} \mid C_i \in C(\mathcal{G})\} \tag{3}$$

That is, the characteristic set fragmentation function creates a fragment for each characteristic set in the knowledge graph. In the characteristic sets shown in Figure 4b, f_4 thus contains all triples of all subjects that are described by the characteristic set {`dbo:capital`,`dbo:currency`}.

Depending on the complexity of the knowledge graph, however, using fragmentation purely based on characteristic sets can quickly lead to an unwieldy number of fragments. In our experimental evaluation in Section 7, fragmenting the data from LargeRDFBench [32] using Equation 3 led to 181,859 distinct fragments most of which contain very few subjects. Consider, for instance, in the running example, the situation where the following five characteristic sets are found in the uploaded knowledge graph; for illustration purposes we have extended the notation with the number of subjects covered by each characteristic set:

$$CS_1 = (\{\texttt{dbo:nationality}, \texttt{dbo:author}, \texttt{dbo:deathDate}\}, 500)$$
$$CS_2 = (\{\texttt{dbo:nationality}, \texttt{dbo:author}\}, 550)$$
$$CS_3 = (\{\texttt{dbo:publisher}, \texttt{dbo:language}\}, 1000)$$
$$CS_4 = (\{\texttt{dbo:nationality}, \texttt{dbo:author}, \texttt{dbo:language}\}, 2)$$
$$CS_5 = (\{\texttt{dbo:nationality}\}, 1)$$

The fragments in the above example are skewed with a significant difference between the largests and smallest fragments. For instance, a separate fragment is created for CS_4 even though it does not carry very much information because it describes only two subjects. As documented in previous studies [24, 25, 50], similar approaches using DP to optimize the join order often struggle when presented with a large number of sources since they generally have to compare the cost of each possible combination of the sources. Concretely, in our case, the DP algorithm presented in Section 5 has polynomial time complexity with the number of relevant fragments; in the worst case, the DP algorithm has to check compatibility between each pair of relevant fragments. This means that as the number of relevant fragments increases, the potential number of compatibility checks increases polynomially as well. Clearly, this could affect lookup time during query optimization.

To partially address the data skew issue, we merge fragments with infrequent characteristic sets into fragments with more frequent characteristic sets, similar to [24]. While such an approach potentially has the tradoff that some of the information in the merged fragments is lost, which could in rare cases lead to incomplete queries, we did not encounter such incomplete results in our experimental evaluation (Section 7). After fragmenting datasets using Equation 3, we iteratively merge the fragments with the lowest number of distinct subject values into other fragments until the total number of fragments is below a threshold, or all fragments with fewer subject values than a threshold have been merged, using the following strategy. In our experiments (Section 7), we computed two sets of fragments for each dataset; one where the total number of fragments matched the number of predicate-based

fragments, and one where all fragments with fewer than 50 subjects (determined empirically based on the data used in our experiments) were merged.

First, for infrequent fragments f_1 with characteristic set CS_1 where there exists another fragment f_2 with a more frequent characteristic set CS_2, such that $CS_1 \subseteq CS_2$, we merge f_1 into f_2 by adding the triples of f_1 to f_2; if there are multiple candidates for f_2, we select the one with the smallest set of predicates, since that fragment has the fewest additional predicates. In the example above, for instance, since $CS_5 \subseteq CS_2$, we merge CS_5 into CS_2 by adding the triples from f_5 to f_2.

Second, the remaining fragments f, i.e., ones that cannot directly be merged into any frequent fragments, are split into a set of disjoint fragments, $\{f_1, \ldots, f_n\}$, such that each $f_i \in \{f_1, \ldots, f_n\}$ can be merged into other fragments with more frequent characteristic sets using the first step. This is done by continuously selecting the largest possible set of predicates that can be merged into other fragments, until no predicates are left. In the (rare) case that some predicates do not occur in any frequent fragments, we store the (small) fragments containing those predicates separately; however, this never happened in our experimental evaluation in Section 7. For instance, in the example above, since CS_4 is not a subset of any frequent characteristic set, we have to split f_4 into smaller fragments. As such, we first create a fragment f_4' with the characteristic set $\{\texttt{dbo:nationality}, \texttt{dbo:author}\}$, since this is the largest subset of CS_4 that can be merged into other fragments (either f_1 or f_2). Then, we create a fragment f_4'' with the characteristic set $\{\texttt{dbo:language}\}$, since this is the only predicate left in the original fragment, and merge f_4' with f_2 and f_4'' with f_3 according to the first step above.

The steps above are sequential, i.e., all fragments that can be merged into other fragments without splitting are merged according to the first step, whereas the remaining infrequent fragments afterwards have to be split and merged according to the second step. In the example above, we end up with the following fragments:

$$CS_1 = (\{\texttt{dbo:nationality}, \texttt{dbo:author}, \texttt{dbo:deathDate}\}, 500)$$
$$CS_2 = (\{\texttt{dbo:nationality}, \texttt{dbo:author}\}, 553)$$
$$CS_3 = (\{\texttt{dbo:publisher}, \texttt{dbo:language}\}, 1002)$$

Clearly, the data skew caused by the fragmentation approach is affected by the structuredness and heterogeneity of the datasets. In LOTHBROK, we logically expect well-structured datasets to perform well, while unstructured datasets should lead to a large number of fragments. This is also what we see in our experimental evaluation in Section 7, where a few very unstructured datasets in LargeRDFBench [32] (e.g., the DBPedia subset) caused a large number of fragments some of which have very similar characteristic sets, whereas more structured datasets (e.g., LinkedTCGA-E) resulted in fewer fragments (Table 3) with less similar characteristic sets. Furthermore, we were able to decrease the number of fragments in our experiments by up to two orders of magnitude. For LargeRDFBench specifically, we decreased the number of fragments from 181,859 to 2,160, which significantly reduces the number of compatibility checks in the DP algorithm as well; as shown in our experimental evaluation (Section 7), using the merging procedure presented above, we were able to achieve significantly improved performance compared to triple-pattern-based query processors.

Since the merging procedure described above has already been reasoned and documented by previous studies [24, 25, 50], and our experimental results (Section 7) are in line with those studies, we will not provide an in-depth analysis of the benefits and tradeoffs of the merging procedure. It is, however, a topic for future work. Furthermore, a complete analysis of the effects of graph complexity measures on the different fragmentation approaches and on the data skew incurred by each fragmentation approach, is a topic for future work.

4.3. Semantically Partitioned Bloom Filter Indexes

The updated fragmentation function described in Section 4.2 often results in fragments that contain characteristic sets with several predicates. However, as described in Section 3.2, the PPBF indexes from [19] encode the set of entities in a fragment without any regard for the position of the entity in the fragment or the connection between the subjects, predicates, and objects.

Hence, we propose a novel indexing schema called *Semantically Partitioned Bloom Filters* (SPBFs), which builds upon PPBF as baseline. This is the second contribution of LOTHBROK described in Section 4.1. Specifically, SPBFs encode the subject values in a single prefix-partitioned bitvector, while there is one prefix-partitioned bitvector for each predicate in the fragment that encodes the objects occurring in triples with that predicate. This structural change in the index lets us do two things: (1) by checking the overlap of the partitioned bitvectors that correspond to the position of the join, we can more accurately determine whether or not fragments produce join results with one another, and (2) we can maintain the link between the subjects, predicates, and objects. As explained in Section 4.1, the SPBF indexes are the second contribution of LOTHBROK, and corresponds to *SPBF Index* in Figure 5. The change in indexing structure requires adjustments in the following query optimization procedure. This procedure is outlined in Figure 5 and detailed in Section 5, and involves source selection based on the compatibility of the fragments and Dynamic Programming (DP).

Note that since LOTHBROK fragments and indexes data based on characteristic sets, the query optimizer described in Section 5 decomposes queries into star patterns. The following description therefore does not mirror exactly the definitions from Section 3.2 [19], since SPBF indexes have to match entire star patterns to fragments rather than triple patterns.

Formally, an SPBF is defined as follows:

Definition 13 (Semantically Partitioned Bloom Filter)**.** *An SPBF $\mathcal{B}^S$ is a 5-tuple $\mathcal{B}^S = (P, \mathcal{B}_s, B_o, \Phi, H)$ where:*

- *P is a set of distinct predicate values*
- *$\mathcal{B}_s$ is the prefix-partitioned bitvector that summarizes the subjects*
- *B_o is the set of prefix-partitioned bitvectors that summarize the objects*
- *$\forall \mathcal{B}_i \in \{\mathcal{B}_s\} \cup B_o$, $\mathcal{B}_i = (P_i, \hat{B}_i, \theta_i)$ where:*
  - *P_i is a set of prefixes*
  - *$\hat{B}_i$ is a set of bitvectors such that $\forall \hat{b}_1, \hat{b}_2 \in \hat{B}_i : |\hat{b}_1| = |\hat{b}_2|$*
  - *$\theta_i : P_i \rightarrow \hat{B}_i$ is a prefix-mapping function such that $\forall p_1, p_2 \in P_i$ where $p_1 \neq p_2$, $\theta_i(p_1) \neq \theta_i(p_2)$*
- *$\Phi : P \rightarrow B_o$ is a predicate-mapping function such that $\forall p \in P : \Phi(p) \in B_o$*
- *H is a set of hash functions*

Given a fragment f, $\mathcal{B}^S(f)$ describes the SPBF for f. For instance, in the running example (Figure 4), the SPBF for f_2, $\mathcal{B}^S(f_2)$, contains a prefix-partitioned bitvector encoding all the subject values in f_2, $\mathcal{B}^S(f_2).\mathcal{B}_s$, as well as a separate prefix-partitioned bitvector for each predicate encoding the object values for those predicates, i.e., the partition $\mathcal{B}^S(f_2).\Phi(\texttt{dbo:author})$ that describes the objects that are connected with the `dbo:author` predicate, and so on. The SPBF for f_2 is visualized in Figure 6b; Figure 6 further visualizes the SPBFs for each fragment in the running example (Figure 4).

Similarly to PPBFs (Section 3.2), we say that an IRI i at position $\rho \in \{s, p, o\}$ *may* be in an SPBF $\mathcal{B}^S$, denoted $i \sqsubseteq^\rho \mathcal{B}^S$, if and only if $i \sqsubseteq \mathcal{B}^S.\mathcal{B}_s$ if $\rho = s$, $\exists p \in \mathcal{B}^S.P : i \sqsubseteq \mathcal{B}^S.\Phi(p)$ if $\rho = o$, or $i \in \mathcal{B}^S.P$ if $\rho = p$. For instance, $\texttt{dbo:nationality} \sqsubseteq^p \mathcal{B}^S(f_2)$ means that the IRI `dbo:nationality` *may* be in f_2 on the predicate position. Furthermore, $\mathcal{B}_p(\mathcal{B}^S)$ denotes a function that computes and returns the prefix-partitioned bitvector that contains all predicates in $\mathcal{B}^S.P$.

To adapt the general definition of distributed indexes from Section 3.2 [19] to the characteristic set fragmentation and star-shaped query decomposition of LOTHBROK, in the following, we change the definition of the $I_n.\nu(t)$ function from Definition 6 to map entire star patterns to the relevant fragments rather than triple patterns. To do this, we formally define the $relevantFragment(P, f)$ function as a binary function that, for a given star pattern P and fragment f, determines whether f is a relevant fragment to P, as follows.

Definition 14 (Fragment Relevance)**.** *Given a star pattern P and a fragment f, $relevantFragment(P, f)$ is a binary function such that:*

- *$relevantFragment(P, f) = \texttt{true}$ iff $\forall t = (s, p, o) \in P$, the following conditions hold:*
  - *$s \in V$ or $s \sqsubseteq^s \mathcal{B}^S(f)$*
  - *$p \in V$ or $p \sqsubseteq^p \mathcal{B}^S(f)$*
  - *$o \in V$ or $o \sqsubseteq \mathcal{B}^S(f).\Phi(p)$*

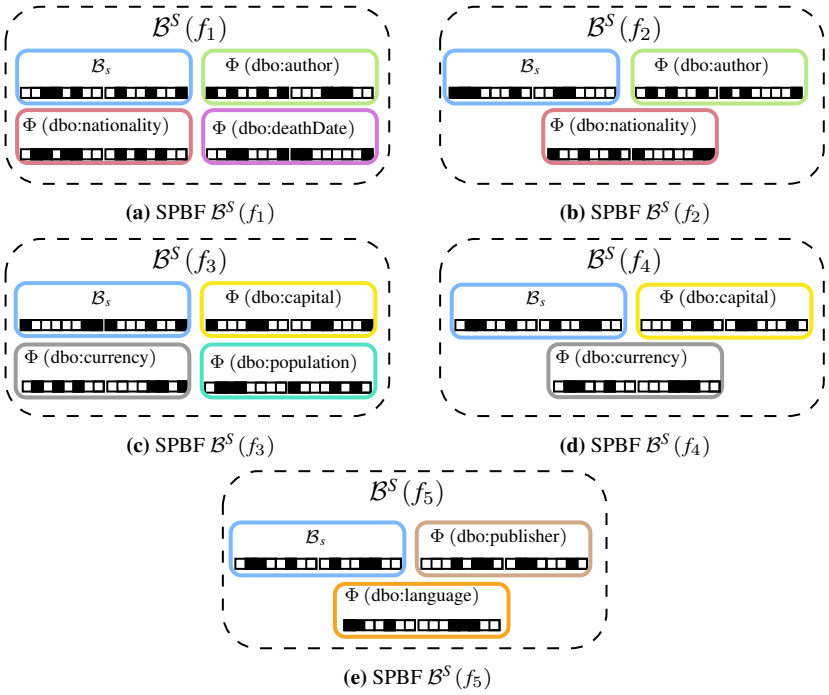


Fig. 6. SPBFs of the fragments in the running example from Figure 4.

– $relevantFragment(P, f) =$ `false` *otherwise*

Consider, as an example, the star pattern P_1 in the example query in Figure 4a and the SPBFs for the fragments in Figure 6. In this example, $relevantFragment(P_1, f_1) =$ `true`, since both `dbo:nationality` $\sqsubseteq^p \mathcal{B}^S(f_1)$ and `dbo:author` $\sqsubseteq^p \mathcal{B}^S(f_1)$, and so we say the f_1 is a *relevant* fragment to P_1.

Note, that fragment relevance in LOTHBROK is a binary decision, $relevantFragment(P, f)$ returns either `true` or `false`, and is not based on a relevance score. However, extending our work to consider relevance scores could be interesting future work.

Given the definition of the $relevantFragment(P, f)$ function above, we define an SPBF index as follows:

Definition 15 (Semantically Partitioned Bloom Filter Index [7, 19])**.** *Let n be a node and $\mathcal{N}$ be the set of nodes within n's local view of the network, $\mathcal{P}$ be the set of all possible star patterns, and $\mathcal{F}$ be the set of fragments stored by at least one node in $\mathcal{N}$. The* SPBF index *on n is a tuple $I_n^S = (\upsilon, \eta)$ with $\upsilon : \mathcal{P} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^{\mathcal{N}}$. $\upsilon(P)$ returns the set of fragments F such that $\forall f \in F$, $relevantFragment(P, f) =$ `true`. $\eta(f)$ returns the set of nodes N such that $f \in n_i.G$, $\forall n_i \in N$ and $n_i \in \mathcal{N}$.*

Consider again the running example in Figure 4a and the example network and fragment distribution in Figure 2a. In this case, given the SPBF index $I_{n_1}^S$ that comprises the SPBFs from all the fragments in Figure 6, then $I_{n_1}^S.\upsilon(P_1) = \{f_1, f_2\}$, since f_1 and f_2 both contain partitions with the `dbo:nationality` and `dbo:author` predicates. Furthermore, in this example, $I_{n_1}^S.\eta(f_1) = \{n_2, n_4\}$, since they are the nodes replicating f_1 in their local datastore.

Since LOTHBROK, like PIQNIC and COLCHAIN, builds partial indexes, i.e., *slices* (cf. Section 3.2), for each fragment that are combined to form the node's distributed index, we define an SPBF index slice similar to Definition 8 as follows:

Definition 16 (SPBF Slice)**.** *Let f be a fragment. The* SPBF slice *describing f is a tuple $s_f^S = (\upsilon', \eta')$ where $\upsilon'(P)$ returns $\{f\}$ if and only if $relevantFragment(P, f) =$ `true`, or $\emptyset$ otherwise, and $\eta'(f)$ returns the set of all nodes that contain f in its local datastore.*

In the running example, for instance, the SPBF slice for f_1, $s^S_{f_1}$, is the SPBF visualized in Figure 6a, and $s^S_{f_1}.\upsilon'(P_1) = \{f_1\}$, since $relevantFragment(P_1, f_1) = \texttt{true}$, as explained above. In other words, the SPBF slice describing a particular fragment is the SPBF obtained from the respective fragment. The function $s^S(f)$ finds the SPBF slice describing f.

Since the SPBF indexes presented in this section are more complex than the ones presented in [19], combined with the more complex query optimization technique outlined in Section 5, we expect a slightly increased query optimization overhead compared to existing approaches [19]. However, this overhead should be compensated with the increased query execution efficiency that is partially obtained from the usage of the SPBF indexes. In fact, this is also what our experimental evaluation in Section 7 shows. Nevertheless, a deeper analysis of this tradeoff using even more diverse real-world datasets and queries is part of our future work.

Like the fragmentation approach (Section 4.2), the indexes are structurally affected by graph complexity measures. Unstructured datasets can lead to skewed partitions where some partitioned bitvectors encode a large number of values. Nevertheless, a complete analysis of the effect of the graph complexity on the indexing approach is a topic for future work. In Section 5, we detail how SPBF indexes are used to optimize queries using cardinality estimations and the locality of the data.

5. Query Optimization

Besides the characteristic set fragmentation method (Section 4.2) and the SPBF indexes (Section 4.3), LOTHBROK introduces a query optimizer that uses the SPBF indexes to build query execution plans in consideration of data locality, that minimizes the number of intermediate results nodes have to transfer between one another. Doing so significantly reduces the network overhead, as we see in our experimental evaluation in Section 7.

As explained in Section 4.1, and visualized in Figure 5, the query optimizer consists of three sequential steps. The first step is *fragment selection*, which matches relevant fragments to each star pattern in the query. We use the $I^S_n.\upsilon(P)$ function from Definition 15 for this purpose. As in Section 4.3, we again emphasize that the relevance of fragments in LOTHBROK is a binary decision; defining the relevance of different fragments based on the rate of overlap could be interesting future work.

In the second step of the query optimizer, LOTHBROK uses the mapping of relevant fragments from the first step to build a *compatibility graph* that describes which fragments are compatible (i.e., joinable) for the star patterns in the query, i.e., which fragments produce join results with one another for the given query. As such, the nodes in a compatibility graph are the relevant fragments, and the edges connect the compatible ones. Compatibility graphs encapsulate two things; (1) fragments that do not contribute to the overall query result are pruned (based on joinability), and (2) different branches of a compatibility graph for the same subqueries can be processed in parallel.

Using the compatibility graph from the previous step, the third step from Figure 5 uses a Dynamic Programming (DP) algorithm similar to [24, 25] to build a query execution plan that specifies which parts of the query can be processed in parallel on which nodes. To decrease the network overhead, the cost function used by the DP algorithm to reduce the number of intermediate results that have to be transferred over the network.

The output of the query optimization is an annotated query execution *plan* specifying join order, join delegations, and which subqueries can be processed in parallel on which nodes. Formally, a query execution plan is defined as:

Definition 17 (Query Execution Plan)**.** *A* query execution plan Π *specifies the node that processes the plan, called a* delegation*, and can be one of four types:*

- Join $\Pi = \Pi_1 \bowtie^n \Pi_2$ *where* Π_1 *and* Π_2 *are two (sub)plans and n is the node the join is delegated to.*
- Cartesian product $\Pi = \Pi_1 \times^n \Pi_2$ *where* Π_1 *and* Π_2 *are two (sub)plans and n is the node the Cartesian product is delegated to.*
- Union $\Pi = \Pi_1 \cup \Pi_2$ *where* Π_1 *and* Π_2 *are two (sub)plans.*
- Selection $\Pi = [[P]]^n_f$ *where P is a star pattern, f is the fragment that P is processed over, and n is the node the selection is delegated to.*

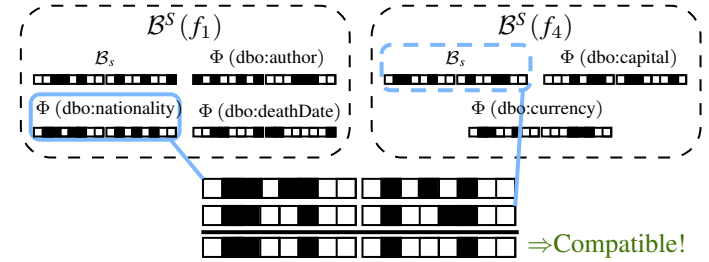
Since unions are not explicitly executed by any node, the partial results of each subplan in the union are transferred to the node that uses those intermediate results. Hence, we omit the specification of delegations for unions from Definition 17 and the description below. Furthermore, we assume that query execution plans are always left-deep, i.e., the right side of a join can only consist of a selection or a union of selections. As part of our future work, we will investigate whether execution plans that are not left-deep could further improve the potential for optimization in certain cases. As an example, the execution plan for query Q, $\Pi = ((([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$ (Figure 13g) specifies that the join $[[P_2]]_{f_4} \bowtie [[P_1]]_{f_1}$ is delegated to n_2 and processed in parallel with $[[P_2]]_{f_3} \bowtie [[P_1]]_{f_2}$ on n_3 (specified by the union), the result of which is transferred to n_1 and joined with $[[P_3]]_{f_5}$. We denote the empty plan $\Pi_\emptyset$.

In the remainder of this section, we detail the compatibility graph and query planning steps from Figure 5. Section 6 then details how the query execution plan is processed over a network.

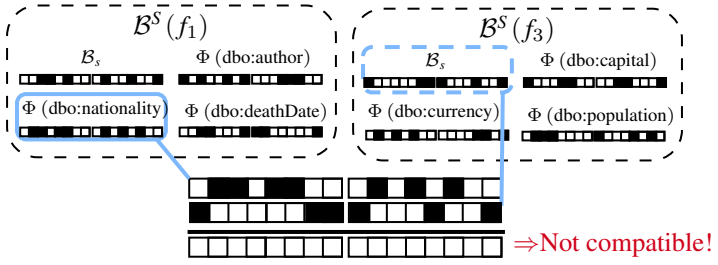
5.1. Fragment and Source Selection

In order to prune fragments that do not contribute to the query result, as well as to determine subqueries that can be processed in parallel, LOTHBROK builds a *compatibility graph* (Figure 5), describing which of the relevant fragments are compatible for the given query, i.e., which fragments produce join results with one another for each join in the query. Specifically, two fragments are said to be compatible for a given query if the intersection of the corresponding SPBF partitions is non-empty, i.e., if the sets of entities represented by these partitions could have some common elements.

As an example, consider again the query Q in Figure 4a and fragments from the running example in Figure 2. In this case, $\{f_1, f_2\}$ are the relevant fragments for P_1 and $\{f_3, f_4\}$ are the relevant fragments for P_2. Since P_1 and P_2 join on the `?country` variable, we can check the compatibility of each combination of fragments for those star patterns, by checking the overlap of the partitioned bitvector on the object position for the `dbo:nationality` predicate for P_1 and the partitioned bitvector on the subject position for P_2, since those are the positions of `?country` in P_1 and P_2. Figure 7 visualizes this computation for two of the fragment combinations. For instance, in Figure 7a, we see that $\{f_1, f_4\}$ are compatible, since the intersection of the corresponding partitions is non-empty. On the other hand, $\{f_1, f_3\}$ (Figure 7b) are not compatible, since the intersection of the corresponding SPBF partitions is empty.



(a) Checking compatibility of $\{f_1, f_4\}$ over $P_1 \bowtie P_2$



(b) Checking compatibility of $\{f_1, f_3\}$ over $P_1 \bowtie P_2$

Fig. 7. Checking compatibility of fragments for the join $P_1 \bowtie P_2$ in the example query Q (Figure 2).

In other words, a compatibility graph is an undirected graph where nodes are the relevant fragments and edges describe the compatible ones. Structurally, a compatibility graph is thus defined as follows:

Definition 18 (Compatibility Graph)**.** *A* compatibility graph G^C *tuple* $G^C = (F, C)$ *where:*

- F *is a set of fragments*
- $C : F \times F$ *is a set of compatible fragments, such that* $\forall (f_1, f_2) \in C$, $f_1 \in F$ *and* $f_2 \in F$

Consider again the running example in Figure 2, and the example compatibility graph in Figure 8g. As we saw in Figure 7, f_1 and f_3 are not compatible since they do not produce any join results for the example query, meaning the compatibility graph in Figure 8g has no edge between those two fragments.

In the following, we detail how Algorithm 1 does so by going through the algorithm showing a step-by-step example of how the compatibility graph is built in the running example in Figure 2 (visualized in Figure 8). Recall the function $\mathcal{B}^S(f)$ that returns the SPBF for a fragment f, and let $vars(P)$ be a function that returns all the variables in a star pattern P. Furthermore, given an SPBF $\mathcal{B}^S$, a star pattern P, and a variable v, let $\mathcal{B}(\mathcal{B}^S, P, v)$ denote a function that returns (assuming v can only occur once in P) $\mathcal{B}^S.\mathcal{B}_s$ if v is the subject in P, $\mathcal{B}^S.\Phi(p)$ if v is the object with predicate p, i.e., $(s, p, v) \in P$, or $\mathcal{B}_p(\mathcal{B}^S)$ if v is a predicate in P. Algorithm 1 defines the $G^C(P, I^S)$ function in lines 1-16 that computes a compatibility graph given a BGP P and SPBF index I^S.

Algorithm 1 Compute the Compatibility Graph of a BGP over an SPBF index

Input: A BGP $P = P_1 \cup \ldots \cup P_n$; an SPBF index $I^S = (\upsilon, \eta)$
Output: A compatibility graph G^C

1: **function** $\mathrm{G}^C(P, I^S)$
2: $\quad P' \leftarrow P_k$ where $P_k \in \mathcal{S}(P)$ and $card_B(P_k) \leq card_B(P_j) \forall P_j \in \mathcal{S}(P)$;
3: $\quad P_\epsilon \leftarrow P'$;
4: $\quad F, C \leftarrow \emptyset$;
5: $\quad$ **for all** $f \in I^S.\upsilon(P')$ **do**
6: $\quad\quad G^C_\epsilon \leftarrow \texttt{buildBranch}(P - P', I^S, f, P', P_\epsilon)$;
7: $\quad\quad F \leftarrow F \cup G^C_\epsilon.F$;
8: $\quad\quad C \leftarrow C \cup G^C_\epsilon.C$;
9: $\quad$ **if** $P - P_\epsilon \neq \emptyset$ **then**
10: $\quad\quad G^C_\epsilon \leftarrow \mathrm{G}^C(P - P_\epsilon, I^S)$;
11: $\quad\quad$ **if** $G^C_\epsilon = G^C_\emptyset$ **then return** $G^C_\emptyset$
12: $\quad\quad$ **for all** $f_1 \in F, f_2 \in G^C_\epsilon.F$ **do**
13: $\quad\quad\quad C \leftarrow C \cup \{(f_1, f_2)\}$;
14: $\quad\quad F \leftarrow F \cup G^C_\epsilon.F$;
15: $\quad\quad C \leftarrow C \cup G^C_\epsilon.C$;
16: $\quad$ **return** (F, C);
17: **function** BUILDBRANCH(P,I^S,f,P',P_ϵ)
18: $\quad$ **if** $P = \emptyset$ **or** $\forall P'' \in \mathcal{S}(P) : vars(P') \cap vars(P'') = \emptyset$ **then**
19: $\quad\quad$ **return** $(\{f\}, \emptyset)$;
20: $\quad F, C \leftarrow \emptyset$;
21: $\quad$ **for all** $P'' \in \mathcal{S}(P)$ s.t. $vars(P') \cap vars(P'') \neq \emptyset$ **do**
22: $\quad\quad P'_\epsilon \leftarrow P_\epsilon \cup P''$;
23: $\quad\quad V \leftarrow vars(P') \cap vars(P'')$;
24: $\quad\quad$ **for all** $f' \in I^S.\upsilon(P'')$ s.t. $\forall v \in V : \mathcal{B}(\mathcal{B}^S(f), P', v) \cap \mathcal{B}(\mathcal{B}^S(f'), P'', v) \neq \emptyset$ **do**
25: $\quad\quad\quad G^C_\epsilon \leftarrow \texttt{buildBranch}(P - P'', I^S, f', P'', P'_\epsilon)$;
26: $\quad\quad\quad$ **if** $G^C_\epsilon \neq G^C_\emptyset$ **then**
27: $\quad\quad\quad\quad F \leftarrow F \cup G^C_\epsilon.F \cup \{f\}$;
28: $\quad\quad\quad\quad C \leftarrow C \cup G^C_\epsilon.C \cup \{(f, f')\}$;
29: $\quad\quad P_\epsilon \leftarrow P_\epsilon \cup P'_\epsilon$;
30: $\quad$ **return** (F, C);

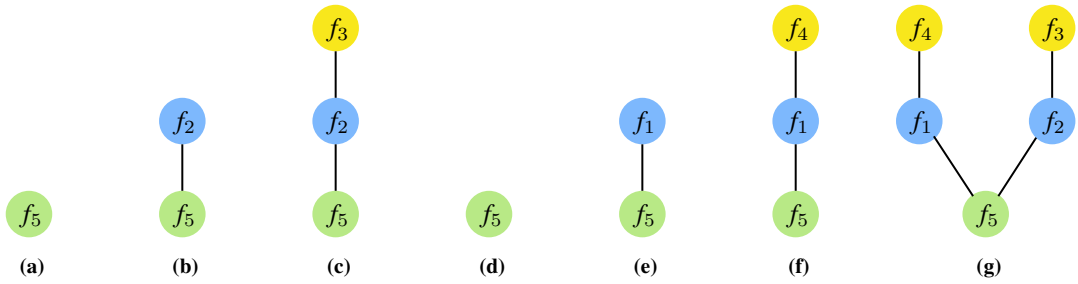


Fig. 8. Recursively building the compatibility graph for the query in Figure 4a by applying Algorithm 1 resulting in $G^C(Q, I^S_{n_1})$. Yellow nodes denote the fragments relevant for P_2, blue nodes the fragments relevant for P_1, and the green nodes the fragments relevant for P_3.

Figure 8 shows how Algorithm 1 builds the compatibility graph for query Q in Figure 4a. In the following, we go through each intermediate step of the algorithm, describing the intermediate compatibility graphs built in the process. First, the G^C function selects the star pattern in $\mathcal{S}(P)$ with the lowest estimated cardinality in line 2 (cardinality estimation is detailed in Section 5.2). Assume in the running example, that P_2 is the star pattern with the lowest estimated cardinality (Section 5.2), and that it is therefore selected in line 2 as the first star pattern. Furthermore, assume that f_1 is only compatible with f_4 and f_2 is compatible with f_3.

Then, the relevant fragments for the selected star pattern are found using the $I^S.\upsilon$ function from the SPBF index (Definition 15) and iterated over in the for loop in lines 5-8; for each of these fragments, the function calls the `buildBranch`$(P, I^S, f, P', P_\epsilon)$ function in lines 17-30 that builds the (sub)graph starting from the current fragment. In the example, the loop in lines 5-8 iterates over $\{f_3, f_4\}$, since these are the fragments relevant for P_2.

The `buildBranch`$(P, I^S, f, P', P_\epsilon)$ function defines a recursive function that builds a sub-graph starting from a specific fragment and star pattern. In the first iteration in the running example (i.e., for f_3), `buildBranch` is called with $P = P_1 \cup P_3$, $f = f_3$, and $P' = P_2$ as parameters. First, if P does not contain any star patterns that join with P', i.e., if P' is the outer-most star pattern in the join tree or for a Cartesian product, the function returns the compatibility graph just containing f without any edges (lines 18-19). In the example, since P_1 joins with P_2, the algorithm does not enter the if statement in line 19.

Instead, the for loop in lines 21-29 iterates through the star patterns $P'' \in P$ that join with P', i.e., star patterns that have at least one variable in common. For each fragment f' relevant for P'' (again found using the SPBF index), the function checks the compatibility of f and f' for each join variable v in line 24, i.e., whether or not f and f' may produce join results for each join variable, by intersecting the corresponding partitioned bitvectors in $\mathcal{B}^S(f)$ and $\mathcal{B}^S(f')$. If the fragments may produce join results, a recursive call is made in line 25 with the $P = P - P''$, $f = f'$, and $P' = P''$ as parameters. In the example, the for loop in line 21 has only one iteration for $P'' = P_1$, i.e., the only star pattern in $\mathcal{S}(P)$ that joins with P_2. Hence, the for loop in line 24 checks the compatibility of each fragment relevant for P_1 (f_1 and f_2) with f_3 (since $f = f_3$ in this call to the function). Since f_2 is compatible with f_3 (cf. the join cardinalities in Table 1), a recursive call is made in line 25 with $P = P_3$, $f = f_2$, and $P' = P_1$.

Since P_3 joins with P_1, the for loop in line 24 checks the compatibility of f_5 and f_2 and makes another recursive call to the function in line 25 with $P = \emptyset$, $f = f_5$, and $P' = P_3$. In this iteration of the function, P is empty, thus the graph $(\{f_5\}, \emptyset)$ is returned in line 19. This graph is visualized in Figure 8a and contains only f_5 with no edges. Since this compatibility graph is non-empty, it is added to the output graph in lines 26-28 together with f_2 (since $f = f_2$ in this iteration of `buildBranch`) and the edge between f_5 and f_2. This graph is visualized in Figure 8b and returned by the current iteration of the `buildBranch` function. Upon receiving the graph in Figure 8b, the function adds f_3 (since $f = f_3$ in the current iteration) and an edge between f_2 and f_3 in lines 26-28, resulting in the compatibility graph shown on Figure 8c that is returned in line 30.

In the next iteration of the for loop in line 5, the `buildBranch` is called with $P = P_1 \cup P_3$, $f = f_4$, and $P' = P_2$. Following the same procedure as described above for f_3, we first build the subgraph containing only f_5 shown in Figure 8d. Then, f_1 is added to the graph along with an edge between f_1 and f_5 (since they produce join results), resulting in the subgraph shown in Figure 8e. Next, f_4 is added along with an edge between f_4 and f_1, resulting in

the compatibility graph for f_4 shown in Figure 8f. After merging this in lines 7-8 with the compatibility graph in Figure 8c, the resulting compatibility graph can be seen in Figure 8g.

The if statement in lines 2-5 ensures that subqueries with star patterns that do not join (i.e., in the case of Cartesian products) are included in the compatibility graph. This is done by keeping track of the considered star patterns in P using the accumulator P_ϵ defined in line 3 and updated in line 29. The example query contains no Cartesian products and so the compatibility graph on Figure 8g is returned by the algorithm.

The output of Algorithm 1 in the example is the compatibility graph shown in Figure 8g, specifying that f_1 is compatible with $\{f_4, f_5\}$ and f_2 is compatible with $\{f_3, f_5\}$.

Algorithm 1 and the definition of SPBF indexes (Definition 15) ensure that pruned fragments do not contribute to the query result, i.e., that our pruning method does not miss any potential results. This follows from the theorem:

Theorem 1. *For any BGP P, SPBF index I^S, and compatibility graph $G^C = G^C(P, I^S)$ (Algorithm 1), it is the case that $\forall P' \in \mathcal{S}(P)$, if $\exists f \in I^S.\nu(P')$ where $f \notin G^C.F$, then $[[P']]_f$ is incompatible with all the results of P over the fragments in I^S.*

A high-level sketch of the proof of Theorem 1 follows. Algorithm 1 only prunes fragments when the condition in line 24 is false. Furthermore, this condition is only false when the bitvectors for the two fragments do not overlap; if there is any kind of overlap, even on a single bit, the condition is true. If the two fragments contain a common value, then by definition this value is mapped to the same positions in the corresponding bitvectors, and they will overlap at least on those bits. Hence, Theorem 1 holds, and we do not miss any potential results.

5.2. Cardinality Estimation

In Section 4.2 we have described how LOTHBROK fragments knowledge graphs based on characteristic sets. Furthermore, in Section 4.3 we described how SPBF indexes connect the objects in a fragment to the predicates they occur in triples with. Since the SPBF of a fragment includes partitioned bitvectors describing the subjects and objects (Definition 15), we can estimate the number of values within these partitioned bitvectors and use those estimations to obtain cardinality estimations in a similar way as [24, 25].

Table 1
Estimated cardinalities for the SPBFs $\mathcal{B}^S(f_1)$, $\mathcal{B}^S(f_2)$, $\mathcal{B}^S(f_3)$, and $\mathcal{B}^S(f_4)$ for the running example in Figure 2

Partitioned Bitvector	card^P	Partitioned Bitvector	card^P
$\mathcal{B}^S(f_1).\mathcal{B}_s$	1000	$\mathcal{B}^S(f_3).\mathcal{B}_s$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:author})$	5000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:capital})$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality})$	1000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:currency})$	150
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:deathDate})$	1000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:population})$	100
$\mathcal{B}^S(f_2).\mathcal{B}_s$	2000	$\mathcal{B}^S(f_4).\mathcal{B}_s$	200
$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:author})$	3000	$\mathcal{B}^S(f_4).\Phi(\texttt{dbo:capital})$	200
$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality})$	2000	$\mathcal{B}^S(f_4).\Phi(\texttt{dbo:currency})$	500
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_3).\mathcal{B}_s$	0	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_3).\mathcal{B}_s$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s$	50	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s$	0
$\mathcal{B}^S(f_5).\mathcal{B}_s$	8000	$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:author}) \cap \mathcal{B}^S(f_5).\mathcal{B}_s$	500
$\mathcal{B}^S(f_5).\Phi(\texttt{dbo:publisher})$	8000	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:author}) \cap \mathcal{B}^S(f_5).\mathcal{B}_s$	1000
$\mathcal{B}^S(f_5).\Phi(\texttt{dbo:language})$	9000		

Note that the cardinality estimation technique presented in this section is used by the Dynamic Programming (DP) algorithm (Section 5.3) to find the cheapest costing query execution plan including all the relevant fragments, and is not used to rank relevant fragments according to the cardinalities. Recall the $card^P(\mathcal{B})$ function from Equation 2. Table 1 then shows the estimated cardinalities of each partitioned bitvector in the running example.

To estimate the cardinality of star-shaped subqueries, we utilize the fact that the subjects are described by a single partitioned bitvector. For a star-shaped subquery asking for the set of unique subject values described by a given set of predicates (i.e., queries with the `DISTINCT` keyword), the cardinality can be estimated as the sum of the number

of subjects in each fragment that includes all the predicates in the query. For instance, the cardinality of P_1 in the query in Figure 4a is the number of distinct subject values in f_1 and f_2.

Given a star pattern P and a fragment f, the cardinality of P over f, assuming that f is a relevant fragment for P, is the number of values in the partitioned bitvector on the subject position in $\mathcal{B}^S(f)$, and is formally defined as:

$$card_D(P, f) = card^P(\mathcal{B}^S(f).\mathcal{B}_s) \tag{4}$$

For queries not including the `DISTINCT` keyword, we need to account for duplicates by considering, on average, the number of triples for each non-variable predicate value in P that each subject value is associated with. Given a star pattern P and fragment f, let $preds(P)$ denote the non-variable predicate values in P (in the case of a variable on the predicate position in P, we consider the average number of predicate occurences in the characteristic set). The cardinality of P is thus estimated as follows [24, 25]:

$$card_S(P, f) = card_D(P, f) \cdot \prod_{p_i \in preds(P)} \frac{card^P(\mathcal{B}^S(f).\Phi(p_i))}{card^P(\mathcal{B}^S(f).\mathcal{B}_s)} \tag{5}$$

Henceforth, we will refer to the more generalized function $card$ rather than $card_D$ and $card_S$ to be equivalent to $card_D$ for queries with the `DISTINCT` modifier and $card_S$ for queries without. Using Equations 4 or 5, the cardinality of a star pattern P over a node n's SPBF index is, for all queries (both with and without the `DISTINCT` keyword), the aggregated cardinality over each relevant fragment to P, and is formally defined as follows:

$$card_n(P) = \sum_{f \in I_n^S.\eta(P)} card(P, f) \tag{6}$$

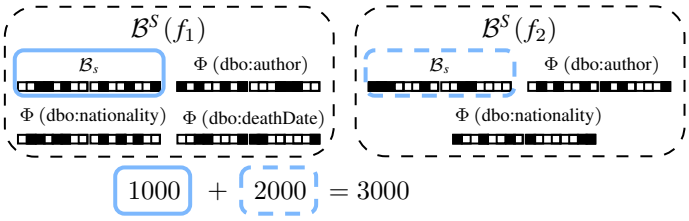


Fig. 9. Estimating the cardinality of P_1 with the `DISTINCT` modifier as the number of subjects in f_1 and f_2 found using Equation 4.

Consider, for instance, in the running example, the star-shaped BGP P_1 in Figure 4a and the estimated cardinalities of the partitioned bitvectors for each fragment in Table 1. Assume in this case that the `DISTINCT` keyword is given in the query. Then, $card_{n_1}(P_1)$ is computed as the aggregated estimation of subject values in f_1 and f_2, $card_{n_1}(P_1) = 1000 + 2000 = 3000$. This is visualized in Figure 9.

If, instead, the `DISTINCT` keyword was not included in the query, the cardinality $card_{n_1}(P_1)$ is, for each relevant fragment (f_1 and f_2), the number of subject values within the fragment multiplied with the average number of triples with each predicate $p_i \in preds(P_1)$ that each subject value is associated with, $card_{n_1}(P_1) = 1000 \cdot (5000/1000) \cdot (1000/1000) + 2000 \cdot (3000/2000) \cdot (2000/2000) = 5000 + 3000 = 8000$. Figure 10 visualizes the above computations and shows which bitvector each value is computed from.

Until now, the cardinality estimations presented in this section are useful for estimating the cardinality of individual star patterns in a query [24, 25]. However, to estimate the cardinality of arbitrary BGPs, [50] introduced characteristic pairs that describe the connections between IRIs described by different characteristic sets. In our case, however, we rely on the SPBFs of the relevant fragments to compute characteristic pairs without storing additional

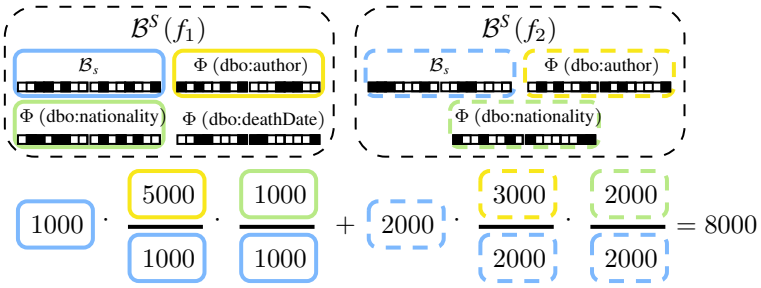


Fig. 10. Estimating the cardinality of P_1 without the `DISTINCT` modifier. Outlines show which bitvector each value is computed from.

information; by intersecting the partitioned bitvectors on the positions corresponding to the join variable, we can estimate the selectivity of a given join and use that to estimate the cardinality of the join.

To achieve this, we first extend the framework for cardinality estimation described above to enable cardinality estimation of an entire query execution plan. This is straightforward for Cartesian products, unions, and selections; for Cartesian products it is the multiplication of the cardinality of the operands, for unions it is the sum of the cardinality of the operands, and for selections it is the cardinality of the star pattern over a specific fragment defined in Equations 4 and 5. Given the reasoning above, we define the cardinality of a query execution plan Π, $card(\Pi)$, covering all types of Π, as follows:

$$card(\Pi) = \begin{cases} card(\Pi_1) \cdot card(\Pi_2), & \text{if } \Pi = \Pi_1 \times^n \Pi_2 \\ card(\Pi_1) + card(\Pi_2), & \text{if } \Pi = \Pi_1 \cup \Pi_2 \\ card(P, f), & \text{if } \Pi = [[P]]^n_f \\ card(\Pi_1 \bowtie^n \Pi_2), & \text{if } \Pi = \Pi_1 \bowtie^n \Pi_2 \end{cases} \tag{7}$$

To compute the cardinality of any join $\Pi = \Pi_1 \bowtie^n \Pi_2$ (e.g., including joins between a BGP with multiple star patterns and a star pattern), we consider two cases: (1) where Π_2 is a union $\Pi_2 = \Pi'_2 \cup \Pi''_2$, and (2) where Π_2 is a selection $\Pi_2 = [[P]]^{n_1}_f$. The cardinality of the join can thus be estimated using the following formula:

$$card(\Pi_1 \bowtie^n \Pi_2) = \begin{cases} card(\Pi_1 \bowtie^n \Pi'_2) + card(\Pi_1 \bowtie^n \Pi''_2), & \text{if } \Pi_2 = \Pi'_2 \cup \Pi''_2 \\ card^{\bowtie}(\Pi_1, P, f), & \text{if } \Pi_2 = [[P]]^{n_1}_f \end{cases} \tag{8}$$

The function $card^{\bowtie}(\Pi, P, f)$ in the second case of Equation 8 computes the cardinality of the join for a particular selection on the right side of the join, $[[P]]_f$. To achieve this estimation, we consider the estimated cardinality of Π and the selectivity of the join similar, i.e., the chance on average that each value on the left side of the join corresponds to a value in the join. Furthermore, to avoid a significant overestimation due to the possible correlation between multiple join variables in the same join, we only consider the most selective join variable for any join.

Recall the $\mathcal{B}(\mathcal{B}^S, P, v)$ function that returns the partitioned bitvector in $\mathcal{B}^S$ that corresponds to v's position in P, and let $S(\Pi, P)$ denote the set of star patterns in Π that join with P and $F(\Pi, f)$ denote the set of fragments in Π that join with f. For instance, for the execution plan in Figure 13d and the compatibility graph in Figure 8g, $S(\Pi, P_3) = \{P_1\}$ and $F(\Pi, f_5) = \{f_1, f_2\}$. Furthermore, given two star patterns P_1 and P_2, let $v(P_1, P_2) = \{v \mid v \in vars(P_1) \cap vars(P_2)\}$, i.e., the set of join variables. The cardinality of the join between a plan Π and a selection $[[P]]_f$ is, given the `DISTINCT` keyword is defined as follows:

$$card^{\bowtie}_D(\Pi, P, f) = card(\Pi) \cdot \min_{P' \in S(\Pi,P) \wedge v \in v(P,P')} \left(\frac{\sum_{f' \in F(\Pi,f)} card^P(\mathcal{B}(\mathcal{B}^S(f), P, v) \cap \mathcal{B}(\mathcal{B}^S(f'), P', v))}{\sum_{f' \in F(\Pi,f)} card^P(\mathcal{B}(\mathcal{B}^S(f'), P', v))} \right) \tag{9}$$

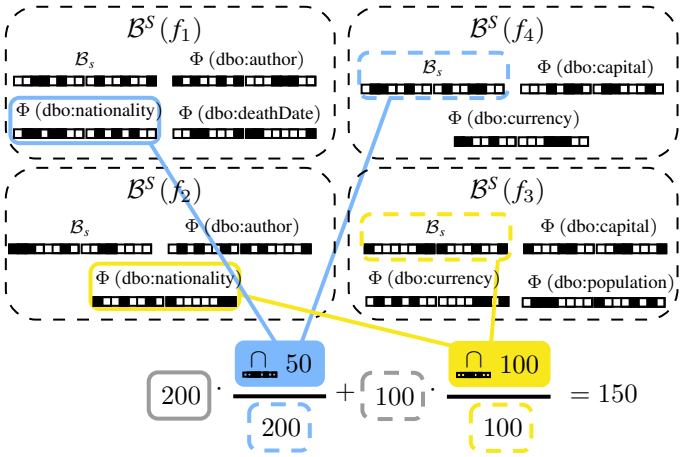
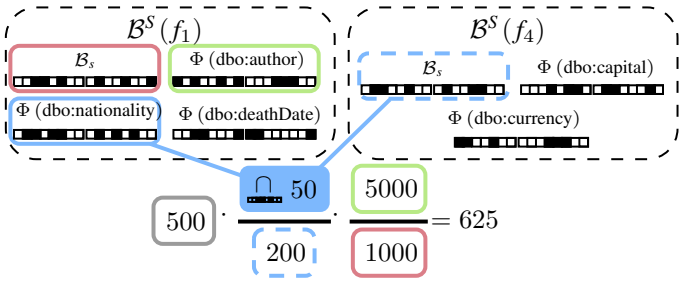


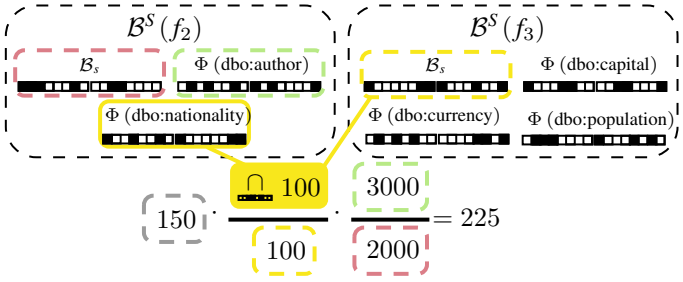
Fig. 11. Estimating the cardinality of $\Pi = ([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$ with the `DISTINCT` keyword using the cardinalities from Table 1 and Equation 9.

As an example, consider computing the cardinality $card(\Pi)$ of the plan Π visualized in Figure 13d using the `DISTINCT` keyword. Since Π is a union, we compute the cardinality of $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$ and $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$ and let $card(\Pi) = card(\Pi_1) + card(\Pi_2)$. Using Equation 9 on Π_1 and Π_2, we get the formula $card(\Pi) = 200 \cdot (50/200) + 100 \cdot (100/100) = 150$ as visualized in Figure 11 (the gray values are the cardinalities of the left selections in each join obtained using Equation 4).

For queries without the `DISTINCT` keyword, we once again consider the average predicate occurences. However, since the predicate occurrences in Π are already considered in $card(\Pi)$ in Equation 9, we only consider the average



(a) Cardinality of $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$



(b) Cardinality of $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$

Fig. 12. Estimating the cardinality of Π in Figure 13d without the `DISTINCT` modifier for (a) $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$ and (b) $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$. The output of Equation 7 is thus the sum of the two formulas ($625 + 225 = 850$).

number of occurrences in f for each triple pattern in P that does not join with Π on the object. The cardinality of the join between a plan Π and selection $[[P]]_f$, without the `DISTINCT` keyword, is computed as:

$$card^{\bowtie}_S(\Pi, P, f) = card^{\bowtie}_D(\Pi, P, f) \cdot \prod_{p \in preds(P): (s,p,o) \in P \wedge o \notin v(P,P') \forall P' \in S(\Pi,P)} \left(\frac{card^P(\mathcal{B}^S(f).\Phi(p))}{card^P(\mathcal{B}^S(f).\mathcal{B}_s)} \right) \quad (10)$$

Once again, computing the cardinality of Π in Figure 13d not including the `DISTINCT` keyword is $card(\Pi) = card(\Pi_1) + card(\Pi_2)$. Using Equation 10 on each of these yields the equation $card(\Pi) = 500 \cdot (50/200) \cdot (5000/1000) + 150 \cdot (100/100) \cdot (3000/2000) = 625 + 225 = 850$. Figure 12 visualizes this computation.

5.3. Optimizing Query Execution Plans

In the last step of the query optimizer (Figure 5), LOTHBROK builds an annotated query execution plan using a Dynamic Programming (DP) algorithm, that determines which parts of the query that can be processed in parallel based on the compatibility graph (as explained in Section 5.1). Furthermore, the DP algorithm is locality-aware, meaning it finds the join delegations that minimize the number of intermediate results that have to be transferred between nodes when executing the plan, called the *transfer cost*.

To do this, the DP algorithm incrementally builds the plan for each subquery, by checking the transfer cost of the possible join combinations and delegations, and selects the cheapest one. In the remainder of this section, we will first define the cost function used by the DP algorithm, after which we will detail the DP algorithm itself.

Algorithm 2 Compute the transfer cost of a query execution plan

Input: A query execution plan Π; a node n
Output: The estimated transfer cost *cost*

```
1:  function TRANSFERCOST(Π,n)
2:      cost ← 0;
3:      if Π = [[P]]^{n_i}_f then
4:          if n ≠ n_i then cost ← card(P, f);
5:      else if Π = Π_1 ∪ Π_2 then
6:          cost ← transferCost(Π_1, n) + transferCost(Π_2, n);
7:      else if Π = Π_1 ×^{n_i} Π_2 then
8:          cost ← transferCost(Π_1, n_i) + transferCost(Π_2, n_i);
9:          if n_i ≠ n then cost ← cost + card(Π);
10:     else if Π = Π_1 ⋈^{n_i} Π_2 then
11:         if Π_2 = Π'_2 ∪ Π''_2 then
12:             cost ← transferCost(Π_1 ⋈^{n_i} Π'_2, n) + transferCost(Π_1 ⋈^{n_i} Π''_2, n);
13:         else if Π_2 = [[P]]^{n_j}_f then
14:             cost ← transferCost(Π_1, n_i);
15:             if n_i ≠ n_j then cost ← cost + card^⋈_S(Π_1, P, f);
16:         if n ≠ n_i then cost ← cost + card(Π);
17:     return cost;
```

Using the cardinality estimation technique in Section 5.2, Algorithm 2 shows how the transfer cost of a query execution plan Π on a node n is computed taking into account the locality of the fragments. First, if $\Pi = [[P]]^{n_i}_f$, i.e., Π is a selection, the algorithm checks whether $n = n_i$ (line 4); if they are equal it is 0 (since it incurs no transfer cost), otherwise the transfer cost of Π is equal to the cardinality of the selection (Equation 5). For instance, the transfer cost of the execution plan shown in Figure 13c ($[[P_3]]^{n_1}_{f_5}$) on n_1 is 0 since f_5 is available on n_1.

If, instead, $\Pi = \Pi_1 \cup \Pi_2$, i.e., Π is a union, the transfer cost is the sum of the transfer costs for Π_1 and Π_2 (line 6). For instance, the transfer cost of the execution plan shown in Figure 13a ($[[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}$) on n_1 is $5000 + 3000 = 8000$, since neither f_1 or f_2 is available on n_1.

Otherwise, if $\Pi = \Pi_1 \times^{n_i} \Pi_2$, i.e., Π is a Cartesian product, the transfer cost is the sum of the transfer costs for Π_1 and Π_2 (line 6), plus the cardinality of the Cartesian product if it is delegated to a different node than the one processing the (sub)plan, i.e., if $n \neq n_i$ (since they have to be transferred from n_i to n).

Finally, if $\Pi = \Pi_1 \bowtie^{n_i} \Pi_2$, i.e., Π is a join, we once again take advantage of the fact that the right side of a join is always either a selection or a union of selections; in the latter case, we aggregate the transfer cost over each subplan in the union (line 12). However, if the right side of the join is a selection $\Pi_2 = [[P]]^{n_j}_f$, we start by estimating the transfer cost of the left side of the join (line 14); if $n_j \neq n_i$, we further add in line 15 the cardinality of the join (since these results should have to be sent back to n_i). Furthermore, if $n_i \neq n$, we add in line 16 the cardinality of the execution plan to the cost, since the results have to be transferred from n_i to n.

In addition to the transfer cost in Algorithm 2, we add the cardinality of the execution plan to the cost function used by the DP algorithm, since these results also have to be transferred to the user. The cost of processing a query execution plan Π over a node n is formally defined as follows.

$$cost_n(\Pi) = transferCost(\Pi, n) + card(\Pi) \tag{11}$$

Using the cost function in Equation 11, the DP algorithm finds the lowest costing execution plan by incrementally merging the cheapest (sub)plans for the smaller subqueries, and computing the cost of each possible join combination and delegation. Furthermore, to merge the subplans, the DP algorithm uses the compatibility graph computed in the second step (Figure 5), to determine which parts of those plans can be joined in parallel.

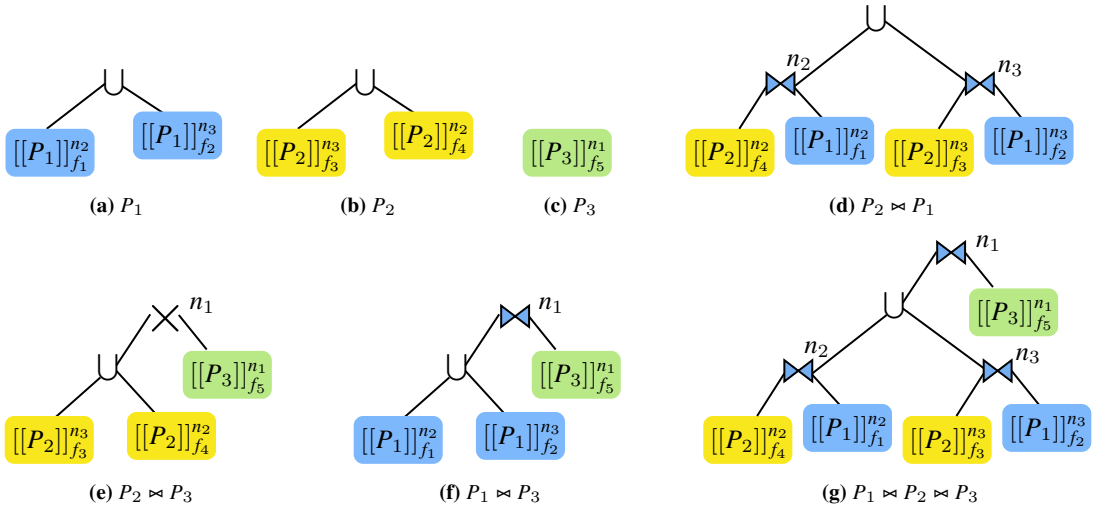
Algorithm 3 shows how the DP table is appended with the lowest costing execution plan for a given (sub)BGP P, node n, and compatibility graph G^C by defining the APPENDDPTABLE$(P, n, G^C, DPTable)$ function in lines 1-13. Table 2 shows the output of the DP table after applying the algorithm over each subquery in Q (Figure 4a), and Figure 13 visualizes each execution plan in Table 2. Notice, that each plan includes all the relevant fragments found in the source selection step (Section 5.1); Table 2 then shows the cheapest *costing* plan for each subquery.

Consider, in the running example, the situation where the APPENDDPTABLE function is called with the parameters $P = P_1 \cup P_2$, $n = n_1$, and where G^C is the compatibility graph in Figure 8g. In this case, the DPTable already contains entries for P_1, P_2, and P_3, i.e., Figures 13a-13c and the first three rows in Table 2, which are used to compute the execution plan for $P_1 \bowtie P_2$.

Since $|\mathcal{S}(P)| \neq 1$, the for loop in lines 7-12 iterates through each join combination in P to find the lowest costing join. In the first iteration of the for loop, where $P' = P_1$, the MERGEPLANS(P, Π, n, G^C) function used in line 9 (defined in lines 14-31) merges the subplans for P' (i.e., P_1) and $P \setminus \{P'\}$ (i.e., P_2) to assess the cost of this particular join combination, and is called with the parameters $P = P_1$, Π is the execution plan in Figure 13b, $n = n_1$, and G^C is the compatibility graph in Figure 8g. The GETDELEGATIONS(Π, I^S_n) function used in line 10 then appends the resulting execution plan Π with the delegations that result in the cheapest cost by trying each combination of possible delegations according to the locality of the relevant fragments in the index I^S_n. The if statement in lines 11-12 then checks whether the resulting execution plan has a lower cost than the currently cheapest plan for P; if the cost is lower, the DP table is updated in line 12.

In the call to MERGEPLANS in the first iteration of the for loop in lines 7-12, the function finds in line 15 the relevant fragments to P_1, again using the $I^S_n.\upsilon(P)$ function from Definition 15 and the compatibility graph G^C. In the example outlined above, this results in $F = \{f_1, f_2\}$. Since there is at least one relevant fragment that is compatible with any of the fragments in the subplan Π, it is not a cartesian product, so we enter the while loop in lines 19-30, which iteratively builds each subplan that, according to the compatibility graph, can be processed in parallel. The function GETONE(F) in line 20 returns one of the relevant fragments in F; in the example, we end up with $f = f_1$, $F = \{f_2\}$, and $F' = \{f_1\}$ after running line 20.

In the for loop in lines 21-22, the function determines, based on the edges in the compatibility graphs, which fragments in F that should be processed together, i.e., which fragments depend on some common fragments for the



(a) P_1 **(b)** P_2 **(c)** P_3 **(d)** $P_2 \bowtie P_1$

(e) $P_2 \bowtie P_3$ **(f)** $P_1 \bowtie P_3$ **(g)** $P_1 \bowtie P_2 \bowtie P_3$

Fig. 13. Best query execution plan for each subquery in the DP table (Table 2).

joining star patterns. In the above example, since f_1 and f_2 do not overlap on compatible fragments for P_2, applying lines 20-22, results in the $F' = \{f_1\}$.

In line 23, we then determine the set of subplans in Π that the fragments in F' are dependent on, according to the compatibility graph. That is, the SUBPLAN(f, Π) function returns, if $\Pi = \Pi_1 \cup (\ldots \cup (\Pi_n))$, the subplan Π_i in Π that f occurs in, called the *corresponding branch* of f, or Π otherwise. For instance, SUBPLAN(f_4, Π_2), where Π_2 is the execution plan in Figure 13b, results in the the plan $[[P_2]]_{f_4}$. As a result, applying line 23 to $F' = \{f_1\}$ in the example results in $Plans = \{[[P_2]]_{f_4}\}$.

Finally, the fragments in F' and the plans in *Plans* are combined in lines 24-28 and added to the accumulator in lines 29-30, leading to the intermediate execution plan $[[P_2]]_{f_4} \bowtie [[P_1]]_{f_1}$. The function GETUNION$(P, F')$ used in line 28 (and lines 4 and 17) finds the query execution plan describing the union of selections of the star pattern P over each relevant fragment $f \in F'$. Since $F = \{f_2\}$, we once again run the while loop in lines 19-30, this time with $f = f_2$, leading to the execution plan $([[P_2]]_{f_4} \bowtie [[P_1]]_{f_1}) \cup [[P_2]]_{f_3} \bowtie [[P_1]]_{f_2}$ as visualized in Figure 13d, which is returned in line 31.

Notice, that the function in lines 14-31 is called for each star pattern in the subquery, i.e., both P_1 and P_2 in this example, in order to check the cost of each possible join combination; Table 2 and Figure 13 only visualize the least costing join combination.

Table 2

Entries in the DP table for query Q (Figure 4a)

Subquery	Execution Plan	Cardinality	Cost
P_1	$[[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}$	8,000	8,000
P_2	$[[P_2]]^{n_3}_{f_3} \cup [[P_2]]^{n_2}_{f_4}$	650	650
P_3	$[[P_3]]^{n_1}_{f_5}$	9,000	9,000
$P_2 \bowtie P_1$	$([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$	**850**	1,700
$P_2 \bowtie P_3$	$([[P_2]]^{n_3}_{f_3} \cup [[P_2]]^{n_2}_{f_4}) \times^{n_1} [[P_3]]^{n_1}_{f_5}$	5,850,000	5,850,650
$P_1 \bowtie P_3$	$([[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$	1,688	9,688
$P_2 \bowtie P_1 \bowtie P_3$	$(([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$	154	1,004

Algorithm 3 Append DP Table for a specific subquery

Input: A BGP P; a node n; a compatibility graph G^C; Current $DPTable$
Output: Updated $DPTable$

function APPENDDPTABLE(P,n,G^C,$DPTable$)
  if $|\mathcal{S}(P)|= 1$ **then**
    $F \leftarrow G^C.F \cap I^S_n.\upsilon(P)$;
    $\Pi \leftarrow$ GETUNION(P, F);
    $\Pi \leftarrow$ GETDELEGATIONS(Π, I^S_n);
    return $DPTable.append(P, \Pi, card(\Pi), cost_n(\Pi))$;
  for all $P' \in \mathcal{S}(P)$ **do**
    $P'' \leftarrow P \setminus P'$;
    $\Pi \leftarrow$MERGEPLANS$(P', DPTable.get(P'').plan(), n, G^C)$;
    $\Pi \leftarrow$GETDELEGATIONS(Π, I^S_n);
    if $cost_n(\Pi) \leq DPTable.get(P).cost()$ **then**
      $DPTable.append(P, \Pi, card(\Pi), cost_n(\Pi))$;
  return $DPTable$;
function MERGEPLANS(P,Π,n,G^C)
  $F \leftarrow G^C.F \cap I^S_n.\upsilon(P)$;
  if $\nexists f_2 \in$ FRAGMENTS$(\Pi) : (f_1, f_2) \in G^C.C$ for any $f_1 \in F$ **then**
    return $\Pi \times$ GETUNION(P, F);
  $\Pi_\epsilon \leftarrow \Pi_\emptyset$;
  while $F \neq \emptyset$ **do**
    $f \leftarrow$ GETONE$(F), F \leftarrow F \setminus \{f\}, F' \leftarrow \{f\}$;
    for all $f' \in F$ s.t. $\bigcap_{f_\epsilon \in \{f,f'\}} \{f'' \mid (f_\epsilon, f'') \in G^C.C \wedge f'' \in$ FRAGMENTS$(\Pi)\} \neq \emptyset$ **do**
      $F \leftarrow F \setminus \{f'\}, F' \leftarrow F' \cup \{f'\}$;
    $Plans \leftarrow \bigcup_{f' \in F'} \{$SUBPLAN$(f'', \Pi) \mid (f', f'') \in G^C.C \wedge f'' \in$ FRAGMENTS$(\Pi)\}$;
    $\Pi' \leftarrow \Pi_\emptyset$;
    for all $\Pi'' \in Plans$ **do**
      if $\Pi' = \Pi_\emptyset$ **then** $\Pi' \leftarrow \Pi''$;
      else $\Pi' \leftarrow \Pi' \cup \Pi''$;
    $\Pi' \leftarrow \Pi' \bowtie$ GETUNION(P, F');
    if $\Pi_\epsilon = \Pi_\emptyset$ **then** $\Pi_\epsilon \leftarrow \Pi'$;
    else $\Pi_\epsilon \leftarrow \Pi_\epsilon \cup \Pi'$;
  return Π_ϵ;

6. Query Execution

Until now, we have described in Section 5 how LOTHBROK obtains a query execution plan using compatibility graphs and locality information provided by the SPBF indexes. In this section, we detail the last step from Figure 5, i.e., the *Query Execution* step, and thus how LOTHBROK evaluates a query given a query execution plan.

Given a BGP P, a compatibility graph $G^C = G^C(P, I^S)$, and a query execution plan Π over P and G^C, LOTHBROK processes P by processing the operations specified in Π and, in doing so, delegating joins and Cartesian products to the nodes specified in Π. The intermediate results from previous steps are used as input to subqueries at a later stage in the query execution plan. In case of a distributed join, the intermediate results are transferred along with the partial query to use local bind joins similar to [11, 38]. To formalize how star patterns in the query execution plan are processed over the fragments, we define a so-called *selector* function in line with related work [7, 11, 38]. The selector function returns the results of processing the star pattern over a fragment given a set of solution mappings, i.e., the set of stars in the fragment that constitute the answer to the star pattern, as follows:

Definition 19 (Selector Function [7, 11, 38])**.** *Given a node n, a star pattern P, and a finite set of distinct solution mappings Ω, the star pattern-based selector function for P and Ω, denoted $s_{(P,\Omega)}$ is for every fragment f in n's local datastore defined as follows.*

$$s_{(P,\Omega)}(f) = \begin{cases} \{t \in T \mid T \subseteq f \wedge T_f[P]\} & \text{if } \Omega = \emptyset \\ \{t \in T \mid T \subseteq f \wedge \exists \mu \in [[P]]_f, \mu' \in \Omega : \mu[P] = T \wedge \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

In line with [7, 11, 38], and to avoid long-running requests on each node, we apply pagination to the results of star pattern requests, i.e., we group the results into reasonably sized pages to avoid excessive data transfer. The page size used in our experimental evaluation (Section 7) is the page size recommended by related work [7, 11, 38], i.e., 100. However, for ease of presentation, we assume that all results can fit into one page when presenting the approach to query processing. Furthermore, to avoid underestimating costs caused by the selector function returning some duplicate values (e.g., when the same subject has multiple object values for a specific predicate), our implementation always uses $card_S$ (Equation 5) and $card_S^{\bowtie}$ (Equation 10) for cardinality estimations, regardless of whether or not the `DISTINCT` keyword is given. Last, given a star pattern P, a node n, a fragment f_i, and a finite set of solution mappings Ω, $sel_n(f_i, P, \Omega)$ denotes the result of invoking $s_{(P,\Omega)}(f_i)$ on n.

Algorithm 4 Evaluate a join plan

Input: A join plan Π; a node n; a set of solution mappings Ω
Output: A set of solution mappings Ω

1: **function** EVALUATEPLAN(Π,n,$\Omega = \{\emptyset\}$)
2: **if** $\Pi = \Pi_1 \times^{n_i} \Pi_2$ **then**
3: $\Omega_1 \leftarrow$ `evaluatePlan`(Π_1, n_i, Ω);
4: $\Omega_2 \leftarrow$ `evaluatePlan`(Π_2, n_i, Ω);
5: $\Omega \leftarrow \Omega_1 \times \Omega_2$;
6: **else if** $\Pi = \Pi_1 \bowtie^{n_i} \Pi_2$ **then**
7: $\Omega \leftarrow$ `evaluatePlan`(Π_1, n_i, Ω);
8: $\Omega \leftarrow$ `evaluatePlan`(Π_2, n_i, Ω);
9: **else if** $\Pi = \Pi_1 \cup \Pi_2$ **then**
10: $\Omega_1 \leftarrow$ `evaluatePlan`(Π_1, n, Ω);
11: $\Omega_2 \leftarrow$ `evaluatePlan`(Π_2, n, Ω);
12: $\Omega \leftarrow \Omega_1 \cup \Omega_2$;
13: **else if** $\Pi = [[P]]_f$ **then**
14: $N \leftarrow I_n^S.\eta(f)$;
15: **if** $n \in N$ **then** $n_i \leftarrow n$;
16: **else** $n_i \leftarrow$ `takeOne`(N);
17: $\phi \leftarrow sel_{n_i}(f, P, \Omega)$;
18: $\Omega \leftarrow \Omega \bowtie \{\mu \mid dom(\mu) = vars(P)$ and $\mu[P] \in \phi\}$;
19: **return** Ω;

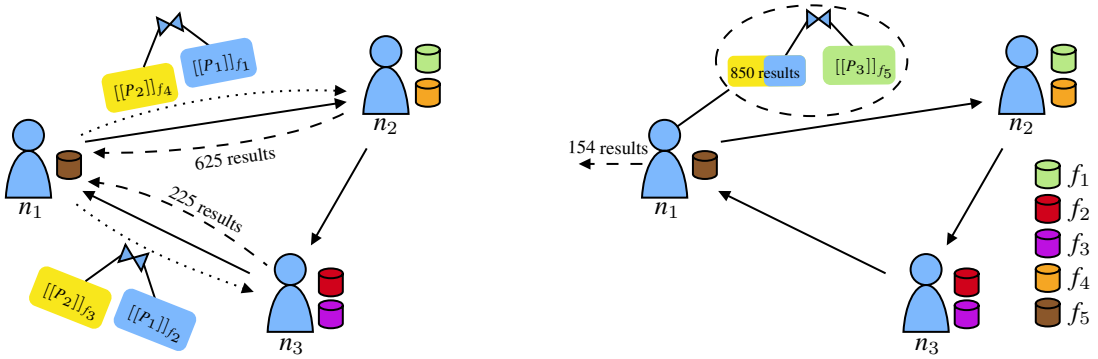
Let I_n^S denote a node n's SPBF index. The `evaluatePlan` function in Algorithm 4 defines a recursive function that processes a query execution plan on a node n by using the selector function defined in Definition 19 for selections in the plan and making recursive calls to the nodes specified in the plan.

Consider, for instance, the query execution plan Π shown in Figure 13g for query Q in Figure 4a processed by node n_1 in the running example. Figure 14 shows an overview of which parts of the query are sent to which node during query processing. Since Π is of type join, the function enters the if statement in line 6. Here, the function first makes a recursive call (since the join was delegated to node n_1) with the left-most subplan, i.e., $\Pi_1 = ([[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}) \cup ([[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2})$ (visualized in Figure 13d), in line 7. Notice that n_1 does not

need to wait for completion of the left-most subplan before processing the join. In fact, the current implementation starts processing joins as soon as partial results from the left side of the join have been returned.

Since Π_1 is of type union, Algorithm 4 in lines 10-11 makes two recursive calls for the two subplans $\Pi_1 = [[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}$ and $\Pi_2 = [[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2}$. Note that these two recursive calls can be processed concurrently and indeed is done so in the implementation of LOTHBROK. This step is shown in Figure 14a where Π_1 is sent to node n_2 and Π_2 is sent to node n_3. Since both subplans follow the same structure, and thus the same evaluation process, we will only explain what happens when processing Π_1.

When processing the plan Π_1 from above, Algorithm 4 first calls the `evaluatePlan` on node n_2 for the subplan $[[P_2]]_{f_4}$, i.e., the selection for P_2 over f_4, in line 7. The `takeOne` function in line 16 selects a random node with the fragment in its local datastore if the node that processes the subquery does not store the fragment locally. In this case, since n_2 stores f_4, it calls the selector function for P_2 over f_4 locally in line 17. The 500 results of processing P_2 over f_4 (cf. Table 1) are then joined with the singleton set of bindings Ω that includes the empty mapping (i.e., a mapping compatible with any mapping) in line 18 and returned in line 19.



(a) Delegating subqueries to n_2 and n_3

(b) Processing $[[P_3]]_{f_5}$ locally on node n_1

Fig. 14. Processing Π in Figure 13g on n_1 by (a) delegating $[[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}$ to n_2 and $[[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2}$ to n_3 concurrently and (b) processing the join between these 850 results and $[[P_3]]_{f_5}$ locally on n_1 to achieve the 154 results (solid arrows denote neighbors, dotted arrows subquery delegation, and dashed arrows transferring of intermediate results). n_1 can send intermediate results to n_3 since it is within its horizon.

Upon receiving the 500 results in line 7, Algorithm 4 makes another recursive call in line 8 to `evaluatePlan` on node n_2 for the subplan $[[P_1]]_{f_1}$, i.e., the selection for P_1 over f_1 with the 500 intermediate results in Ω. Again, n_2 calls the local selector for P_1 over f_1 using the intermediate results in Ω as bindings. This results in 625 intermediate results in Ω that are the result of processing $P_1 \bowtie P_2$ over f_1 and f_4, which are returned by the function in line 19.

While n_2 found the 625 results from processing $[[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}$ in the recursive call in line 10, n_3 found the additional 225 results of processing $[[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2}$ in the recursive call in line 11 following the same steps as described above for n_2. In line 12, these results are combined and 850 bindings are returned in line 19, which is visualized on Figure 14a as n_2 returning 625 results to n_1 and n_3 returning 225 results to n_1.

The 850 intermediate results in Ω found by processing $([[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}) \cup ([[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2})$ in line 7 are used as bindings for the recursive call made in line 8 for the subplan $[[P_3]]_{f_5}$. This is visualized in Figure 14b. Since n_1 stores f_5 locally, it calls the local selector for P_3 over f_5 and Ω in line 17. The 154 results of processing P_3 over f_5 are joined with Ω in line 18 and returned as the final results in line 19.

As mentioned above, our implementation uses pagination of the results meaning, for instance, when processing the subplan $[[P_2]]_{f_4}$ in line 7, the 500 results would be split into multiple pages. In the implementation of LOTHBROK, nodes at subsequent steps in the pipeline start processing joins as soon as they receive some intermediate bindings. For instance, in the running example, n_1 starts processing the join between $P_2 \bowtie P_1 \bowtie P_3$ locally as soon as it receives results for $P_2 \bowtie P_1$ from either n_2 or n_3.

7. Experimental Evaluation

The experimental evaluation compares LOTHBROK with two state-of-the-art approaches building on P2P systems: PIQNIC [6] and COLCHAIN [7] with the query optimization approach outlined in [19]. To do this, we implemented[3] the fragmentation, indexing, and cardinality estimation approach as a separate package in Java 8 and modified PIQNIC's and COLCHAIN's query processors to use it. Like COLCHAIN and PIQNIC, LOTHBROK's query processor is implemented as an extension to Apache Jena[4]. Fragments in our implementation are stored as HDT files [63], allowing for efficient processing of the star patterns. We provide all source code, experimental setup (queries, datasets, etc.), and the full experimental results on our website[5].

7.1. Experimental Setup

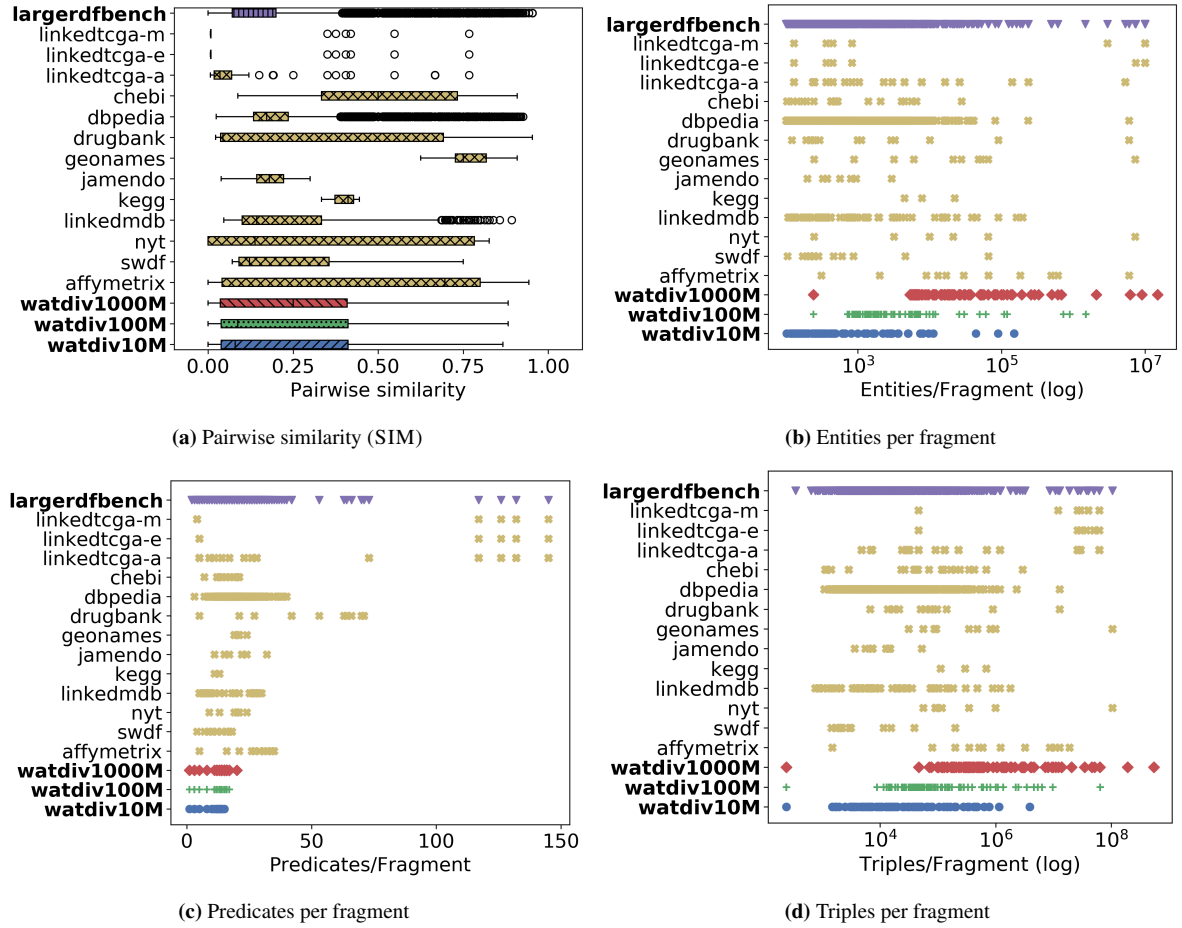
In this section, we detail the experimental setup, including a characterization of the used datasets and queries, the hardware and software setup, experimental configuration, as well as the evaluation metrics.

Datasets. We used two different benchmark suites for data in our experiments, LargeRDFBench [32] and WatDiv [33], with a total of four datasets, detailed in Table 3 along with some characteristics and statistics. LargeRDFBench is a well-known benchmark suite for federated RDF engines that comprises 13 different, interlinked datasets with over a billion triples in total, used to test LOTHBROK in a realistic setting where users would upload several interlinked datasets to a network and ask queries with varying complexity. Notice that the total number of fragments over the datasets in LargeRDFBench exceed the number of fragments for LargeRDFBench overall. This is due to some fragments spanning multiple datasets, e.g., LinkedTCGA-M and LinkedTCGA-E span the exact same fragments. WatDiv is a synthetic benchmark used to test the scalability of the approaches when the network is under heavy load, and to assess the impact of the query pattern on performance and network usage. We generated three differently sized WatDiv datasets, from 10 million triples to 1 billion triples.

Table 3
Characteristics of the used datasets

Dataset	#triples	#subjects	#predicates	#objects	#fragments	Struct. [64]
LargeRDFBench	1,003,960,176	165,785,212	2,160	326,209,517	2,160	0.926
LinkedTCGA-M	415,030,327	83,006,609	6	166,106,744	8	1
LinkedTCGA-E	344,576,146	57,429,904	7	84,403,402	8	1
LinkedTCGA-A	35,329,868	5,782,962	383	8,329,393	209	0.98
ChEBI	4,772,706	50,477	28	772,138	21	0.34
DBPedia-Subset	42,849,609	9,495,865	1,063	13,620,028	1,774	0.196
DrugBank	517,023	19,693	119	276,142	11	0.726
GeoNames	107,950,085	7,479,714	26	35,799,392	23	0.518
Jamendo	1,049,647	335,925	26	440,686	7	0.961
KEGG	1,090,830	34,260	21	939,258	3	0.919
LinkedMDB	6,147,996	694,400	222	2,052,959	135	0.729
NYT	335,198	21,666	36	191,538	6	0.731
SWDF	103,595	11,974	118	37,547	12	0.426
Affymetrix	44,207,146	1,421,763	105	13,240,270	12	0.506
watdiv10M	10,916,457	521,585	86	1,005,832	86	0.42
watdiv100M	108,997,714	5,212,385	86	9,753,266	86	0.42
watdiv1000M	1,092,155,948	52,120,385	86	92,220,397	86	0.42

[3]Code is available on the following GitHub repository: https://github.com/dkw-aau/Lothbrok-Java
[4]https://jena.apache.org
[5]https://relweb.cs.aau.dk/lothbrok



(a) Pairwise similarity (SIM)

(b) Entities per fragment

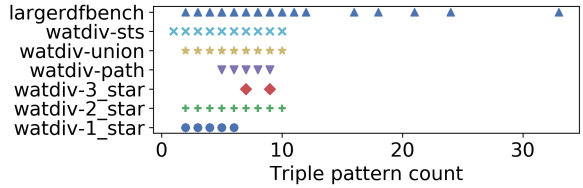
(c) Predicates per fragment

(d) Triples per fragment

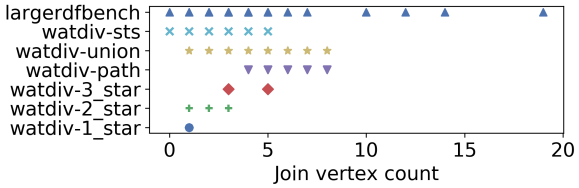
Fig. 15. Characteristics of the computed fragments over all the included datasets.

Fragments. To provide a fair comparison between the systems with and without LOTHBROK, we created an equal number of fragments for both fragmentation methods, characteristic sets (Section 4.2) and predicate-based, following the approach outlined in Section 4.2. Figure 15 shows an overview of the following characteristics of each fragment over each dataset: pairwise similarity SIM (Figure 15a): given two fragments f_1 and f_2 with characteristic sets CS_1 and CS_2, $\text{SIM}(f_1, f_2) = |CS_1 \cap CS_2|/|CS_1 \cup CS_2|$, i.e., Jaccard similarity, number of entities per fragment (Figure 15b), number of predicates per fragment (Figure 15c), and number of triples per fragment (Figure 15d). Additionally, to assess the impact of reducing the number of characteristic sets on query completeness, we ran similar experiments where we did not create an equal number of fragments for LOTHBROK, i.e., where we created one fragment for each characteristic set that describes at least 50 subjects and provide the results on our website[3]; since these results are quite similar to the ones presented in this section, we will not report on them further.

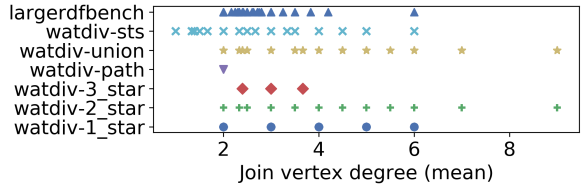
Queries. LargeRDFBench includes 40 queries [32] in five different categories of varying complexity: Simple (S), Complex (C), Large Data (L), and Complex and High Data Sources (CH). For WatDiv, we used WatDiv *star query loads* from [11] consisting of 1-3 star patterns, called the `watdiv-1_star`, `watdiv-2_star`, and `watdiv-3_star` query loads, as well as a query load consisting of path queries, i.e., queries where each star pattern only has one triple pattern, called the `watdiv_path` query load. Each of these query loads consists of 6,400 different queries. Furthermore, we combine the aforementioned query loads into a single query load called `watdiv-union`. Last, we created a query load with 19,968 queries from the WatDiv stress testing query templates (156 per node) called `watdiv-sts`. The complete set of queries is available on our website[3].



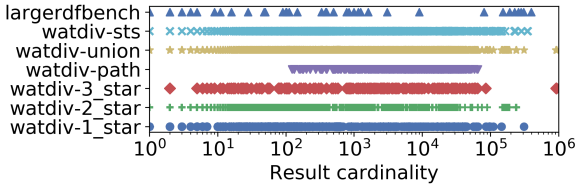
(a) Triple pattern count (#TP)



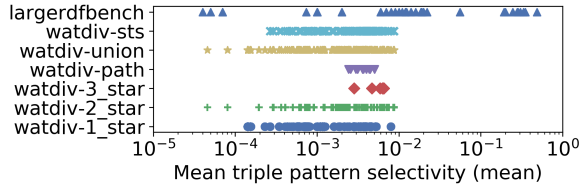
(b) Join vertex count (#JV)



(c) Join vertex degree (DEG)



(d) Result cardinality (#Results)



(e) Mean TP selectivity ($\text{SEL}_{\mathcal{G}}(tp)$)

Query load	SS	SO	OO
watdiv-1_star	100%	0%	0%
watdiv-2_stars	55.38%	34.31%	10.31%
watdiv-3_stars	57.67%	30.77%	11.53%
watdiv-paths	0%	100%	0%
watdiv-union	53.27%	41.27%	5.46%
watdiv-sts	55.57%	35.83%	8.6%
largerdfbench	41.41%	45.81%	12.78%

(f) Join vertex type over each query load

Fig. 16. Characteristics of all query loads (WatDiv query loads over `watdiv100M`; statistics over the `watdiv10M` and `watdiv1000M` datasets can be found on our website[3]).

Figure 16 shows an overview of the following characteristics of each load [11, 65]: Triple pattern count #TP (Figure 16a), join vertex count #JV (Figure 16b), join vertex degree DEG (Figure 16c), result cardinality #Results (Figure 16d), mean triple pattern selectivity $\text{SEL}_{\mathcal{G}}(tp)$ (Figure 16e), and join vertex type (Figure 16f).

Experimental Configuration. We compare the following systems: (1) PIQNIC [6] using PPBF indexes [19] (PIQNIC), (2) LOTHBROK on top of PIQNIC (LOTHBROK$_{\text{PIQNIC}}$), (3) COLCHAIN [7] using PPBF indexes (COLCHAIN), and (4) LOTHBROK on top of COLCHAIN (LOTHBROK$_{\text{COLCHAIN}}$). All configurations were run on networks with 128 nodes. To assess the scalability of LOTHBROK under load, we ran 156 `watdiv-sts` queries concurrently on each node over 8 different configurations where 2^i nodes issue queries concurrently such that $0 \leq i \leq 7$ (i.e., up to all 128 nodes). Furthermore, to analyze the impact of the query pattern on performance, we ran the WatDiv star query loads over each WatDiv dataset size such that for each star query load, each node issued 50 queries. Lastly, we tested the performance of LOTHBROK over each individual query in LargeRDFBench by running the queries sequentially in random order on three randomly selected nodes and report the average result.

Hardware Configuration. For all configurations and P2P systems, we ran 128 nodes concurrently on a virtual machine (VM) with 128 vCPU cores with a clock speed of 2.5GHz, 64KB L1 cache, 512KB L2 cache, 8192KB L3, and a total of 2TB main memory. To spread out resources evenly across nodes, all nodes were restricted to use 1 vCPU core and 15GB memory, enforced using the `-Xmx` and `-XX:ActiveProcessorCount` options for the JVM. Furthermore, to simulate a more realistic scenario, where nodes are not run on the same machine, we simulated a connection speed of 20 MB/s.

Evaluation Metrics. We used measured the following metrics:

- *Workload Time (WT):* The amount of time (in milliseconds) it takes to complete an entire workload including queries that time out.
- *Throughput (TP):* The number of completed queries in the workload divided by the total workload time (i.e., number of queries per minute).

- *Number of Timeouts (NTO):* The number of queries that timed out (timeout being 1200 seconds).
- *Query Execution Time (QET):* The amount of time (in milliseconds) elapsed between when a query is issued and when its processing has finished.
- *Query Response Time (QRT):* The amount of time (in milliseconds) elapsed between when a query is issued and when the first result is computed.
- *Query Optimization Time (QOT):* The amount of time (in miliseconds) elapsed between when a query is issued and when the optimizer has finished (i.e., when query execution starts).
- *Number of Requests (REQ):* The number of requests made between nodes when processing a query (including requests made from nodes that have been delegated subqueries).
- *Number of Transferred Bytes (NTB):* The amount of data (in bytes) transferred between nodes when processing a query (including data transferred to and from nodes that have been delegated subqueries).
- *Number of Relevant Nodes (NRN):* The number of distinct nodes that replicate fragments containing relevant data to a query.
- *Number of Relevant Fragments (NRF):* The number of distinct fragments containing relevant data to a query.

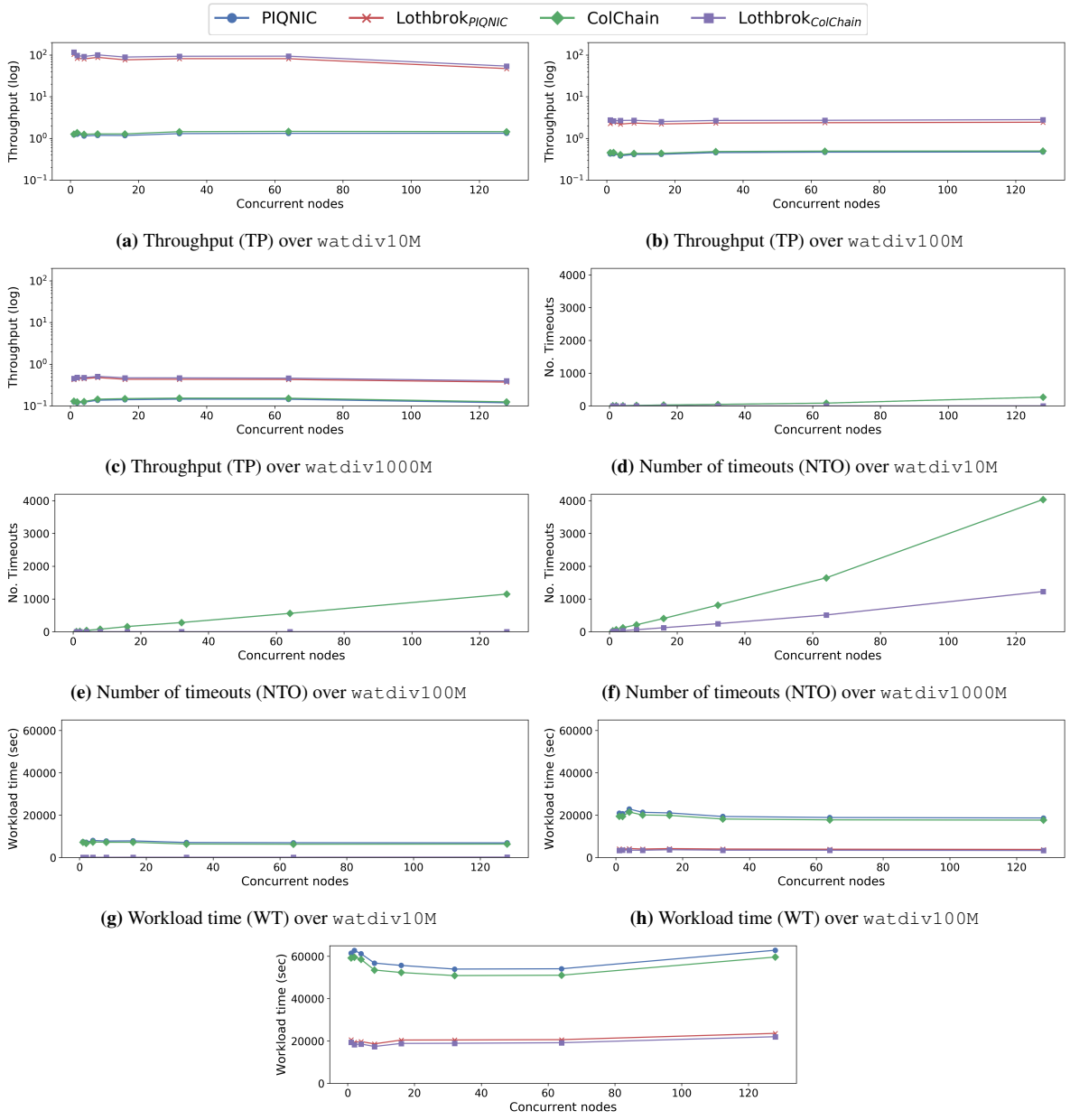
Software Configuration. Unless otherwise specified, we used the following parameters when running the systems. For ColChain, we used the following parameters recommended in [7]: Community Size: 20, Number of Communities: 200. For Piqnic, we use the following parameters recommended in [6]: Time-to-Live (number of hops): 5, Number of Neighbors: 5. The replication factor for Piqnic (i.e., the percentage of nodes replicating each fragment) was matched with the size of the communities in ColChain to provide a better comparison. Nodes were randomly assigned neighbors throughout the network. The page size (i.e., how many results can be returned with each request, was set to 100. Furthermore, to limit the size of HTTP requests, the number of results that each system was allowed to attach to each request (i.e., $|\Omega|$ in Section 6) was set to $|\Omega| = 30$ The timeout for all systems and queries was set to 20 minutes (1,200 seconds).

7.2. Scalability under Load

In these experiments, we ran the `watdiv-sts` queries over each WatDiv dataset in configurations where 2^i nodes issued 156 queries from the `watdiv-sts` query load concurrently such that $0 \leq i \leq 7$. Figures 17a-17c show the throughput (TP) of the `watdiv-sts` query load over each configuration in the scalability tests for the `watdiv10M` (Figure 17a), `watdiv100M` (Figure 17b), and `watdiv1000M` (Figure 17c) datasets in logarithmic scale. Clearly, Lothbrok has a significantly higher throughput across all datasets and configurations compared to the approaches that do not include Lothbrok (i.e., Piqnic and ColChain). In fact, for `watdiv10M`, this increase in throughput is close to two orders of magnitude. While the increase in throughput that Lothbrok provides is smaller for both `watdiv100M` and `watdiv1000M`, Lothbrok still increases the throughput by close to an order of magnitude for these datasets. Furthermore, while some results show that ColChain has a slightly higher throughput than Piqnic, both with and without Lothbrok on top, this difference is relatively negligible. Last, the results show that the throughput of Lothbrok is relatively stable when increasing numbers of nodes issue queries concurrently. In fact, even when every node in the network issue queries concurrently, the throughput is relatively close to the highest throughput throughout the configurations.

Figures 17d-17f show the number of queries that timed out (TO) of the `watdiv-sts` query load over each configuration for each WatDiv dataset. As expected, the number of timeouts increases relatively linearly with the number of nodes issuing queries concurrently. This is due to the fact that when more nodes issue queries, more queries in total are executed, meaning the total number of the queries that time out increases. Generally, the queries that time out correspond to query templates that result in a large number of intermediate results, e.g., by using the `owl:sameAs` predicate. Furthermore, Piqnic and ColChain incur significantly more timeouts without Lothbrok compared to with Lothbrok. In fact, for both `watdiv10M` and `watdiv100M`, Lothbrok experiences no timeouts while Piqnic and ColChain experience 267 timeouts for `watdiv10M` and 1,148 timeouts for `watdiv1000M`. Even for `watdiv1000M`, the number of timeouts experienced by Lothbrok is just 1,151 while Piqnic and ColChain both experience 4,036 timeouts. Furthermore, Piqnic and ColChain incur the exact same number of timeouts.

While queries can time out for several different reason, the queries that timed out in our experiments have some common characteristics. Specifically, they are typically the queries that result in a large number of intermediate



(a) Throughput (TP) over `watdiv10M`

(b) Throughput (TP) over `watdiv100M`

(c) Throughput (TP) over `watdiv1000M`

(d) Number of timeouts (NTO) over `watdiv10M`

(e) Number of timeouts (NTO) over `watdiv100M`

(f) Number of timeouts (NTO) over `watdiv1000M`

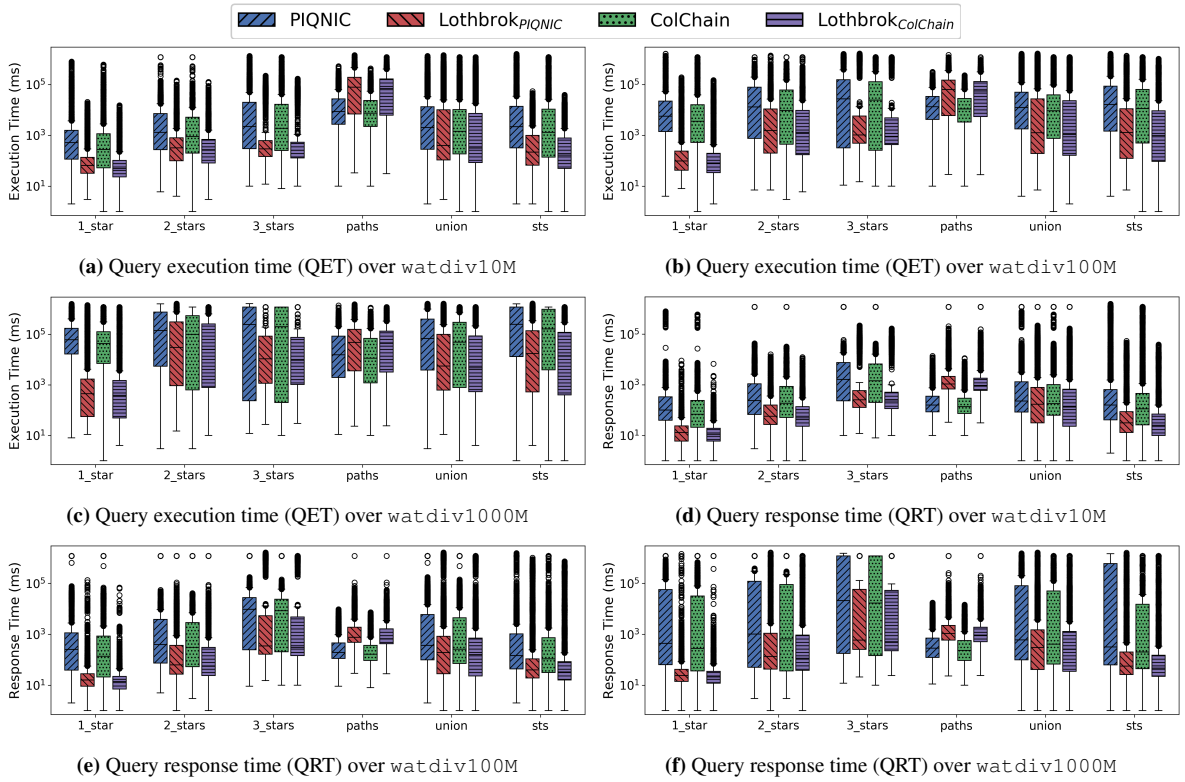
(g) Workload time (WT) over `watdiv10M`

(h) Workload time (WT) over `watdiv100M`

(i) Workload time (WT) over `watdiv1000M`

Fig. 17. Throughput (TP), number of timeouts (NTO), and workload time (WT) for `watdiv-sts` over the `watdiv10M`, `watdiv100M`, and `watdiv1000M` datasets.

results. This is particularly the case for PIQNIC and COLCHAIN, since general predicates such as `rdf:type` result in querying large fragments and many intermediate results. LOTHBROK is able to mitigate this effect by processing those triple patterns as part of a large star pattern, lowering the total number of intermediate results. On the other hand, the few queries that timeout for LOTHBROK over `watdiv1000M` are the queries specifically with a large number of star patterns with very common characteristic sets as we see in Section 7.3. A deeper analysis of what causes systems like PIQNIC and COLCHAIN to timeout requires further research that is out of scope for this paper; nevertheless, it is an interesting aspect to look into in the future.



(a) Query execution time (QET) over `watdiv10M`

(b) Query execution time (QET) over `watdiv100M`

(c) Query execution time (QET) over `watdiv1000M`

(d) Query response time (QRT) over `watdiv10M`

(e) Query response time (QRT) over `watdiv100M`

(f) Query response time (QRT) over `watdiv1000M`

Fig. 18. Query execution time (QET) and query response time (QRT) for the WatDiv datasets and star queries.

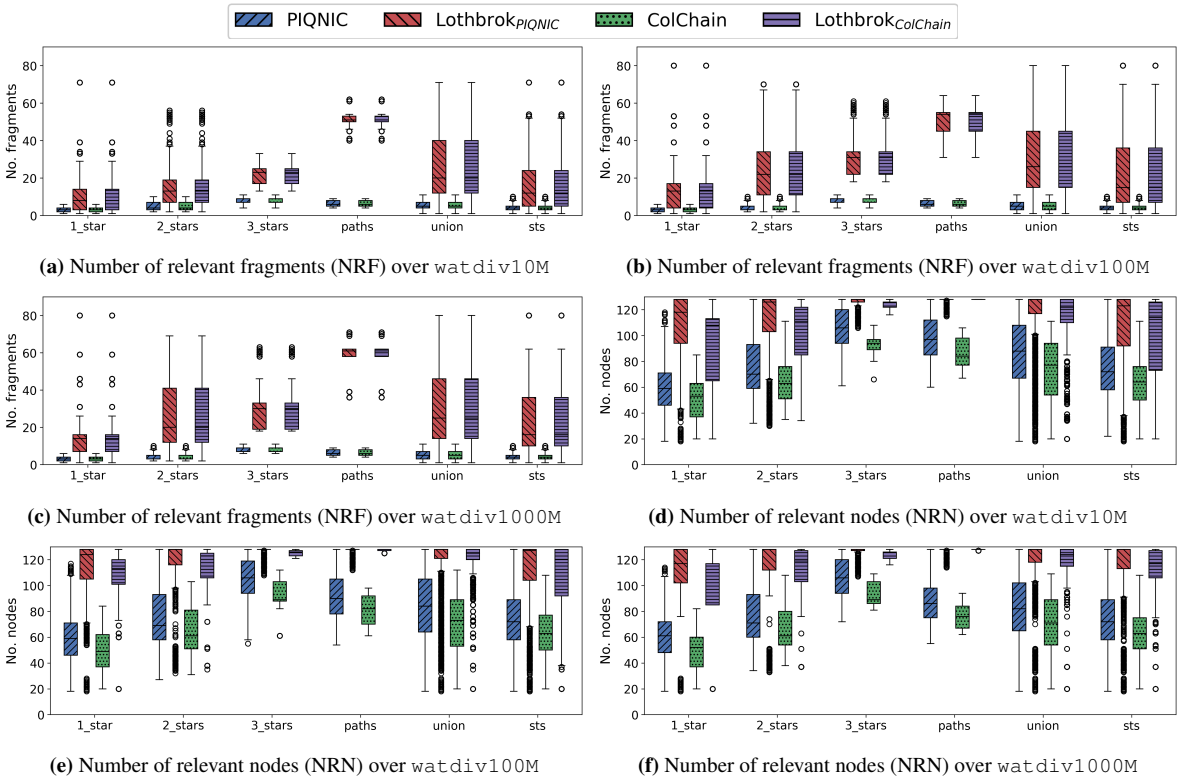
Figures 17g-17i show the workload time (WT) for each configuration. In line with the throughput and number of timeouts, LOTHBROK incurs a significantly lower average workload time than PIQNIC and COLCHAIN across all experiments and datasets. The slight decrease in the workload time for fewer nodes can be attributed to the network being able to process more queries concurrently when the overall load is relatively low. Nevertheless, the average workload time only increases slightly even when all nodes issue queries concurrently.

Overall, our experimental results show that, even when the network is under heavy query processing load, LOTHBROK increases the query throughput and decreases the average workload time significantly compared to state-of-the-art decentralized systems. In fact, the increase in performance is up to two orders of magnitude. As a result, LOTHBROK is also able to finish more queries without timing out.

7.3. Impact of Query Pattern

To test the impact of the query pattern on the performance of LOTHBROK, we ran the `watdiv-1_star`, `watdiv-2_star`, `watdiv-3_star`, `watdiv-path`, `watdiv-union`, and `watdiv-sts` query loads on each system; the `watdiv-sts` queries consist of, on average, more selective star patterns compared to the other WatDiv query loads (Figure 16).

Figures 18a-18c show the execution time (QET) for each WatDiv query load over each WatDiv dataset, and Figures 18d-18f show the response time (QRT) for each WatDiv query load in logarithmic scale. Our results show that LOTHBROK has significantly better performance across all datasets for almost every query load. As expected, the improvement in performance is more significant for the query loads with a lower number of star patterns. This is due to the fact that since the star patterns within these queries represent a large part of the query, LOTHBROK has to issue fewer requests overall, lowering the network overhead. For instance, the queries in the `watdiv-1_star` query



(a) Number of relevant fragments (NRF) over `watdiv10M`

(b) Number of relevant fragments (NRF) over `watdiv100M`

(c) Number of relevant fragments (NRF) over `watdiv1000M`

(d) Number of relevant nodes (NRN) over `watdiv10M`

(e) Number of relevant nodes (NRN) over `watdiv100M`

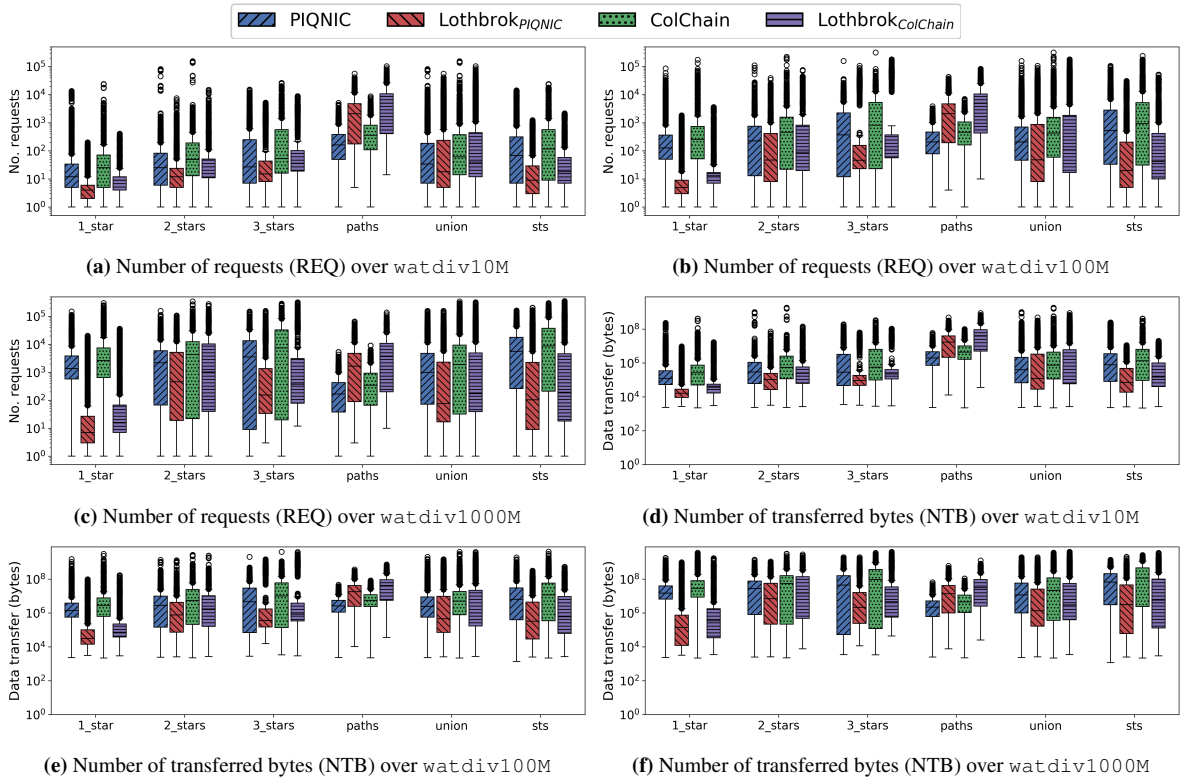
(f) Number of relevant nodes (NRN) over `watdiv1000M`

Fig. 19. Number of relevant fragments (NRF) and number of relevant nodes (NRN) for the WatDiv datasets and star queries.

load can by LOTHBROK be answered by issuing 0.89 requests per 90 results[6], whereas PIQNIC and COLCHAIN have to issue 9.27 requests per 90 results on average, for `watdiv1000M` in our experiments. In the `watdiv-3_star` query load, the improvement in performance is more modest across the datasets since each star pattern is a relatively small part of the query resulting in a higher number of requests; however, on average, we still see a performance increase of up to an order of magnitude.

We notice that for the `watdiv-path` query load, LOTHBROK actually has a slightly worse performance both in terms of QET and QRT compared to PIQNIC and COLCHAIN due to higher network usage. Figure 19 shows the number of relevant fragments (NRF) and the number of relevant nodes (NRN) for each query load over each dataset after optimization (similar figures are provided for NRF and NRN before optimization on our website[3]). Analyzing these results, we see that the decreased performance for `watdiv-path` is caused by LOTHBROK having a significantly larger number of relevant fragments and by extension a larger number of relevant nodes compared to PIQNIC and COLCHAIN. In fact, this is the case for all the WatDiv query loads (9 times larger for `watdiv-path` while up to 5 times larger for the other query loads); however, for the other query loads, this is compensated by the increased performance that the query optimization approach provides. This analysis is corroborated by the number of fragments pruned during optimization for each query load (figures provided on our website[3]); the `watdiv-path` query load has significantly less pruned fragments compared to the other query loads except `watdiv-1_star`. For PIQNIC and COLCHAIN, the number of relevant fragments will always equal the number of unique predicates in the query since one fragment is created per predicate; however, due to fragmenting the data based on characteristic sets, LOTHBROK can encounter multiple fragments for each unique predicate in the query. Furthermore, the number of relevant fragments is, on average, more than twice as high for LOTHBROK over the `watdiv-path` query load than over the other query loads. This is because most of the path queries use common predicates like `owl:sameAs`.

[6]Even though one request can fetch up to 90 results, the average number of requests is lower than 1 since the nodes store some data locally.



(a) Number of requests (REQ) over `watdiv10M`

(b) Number of requests (REQ) over `watdiv100M`

(c) Number of requests (REQ) over `watdiv1000M`

(d) Number of transferred bytes (NTB) over `watdiv10M`

(e) Number of transferred bytes (NTB) over `watdiv100M`

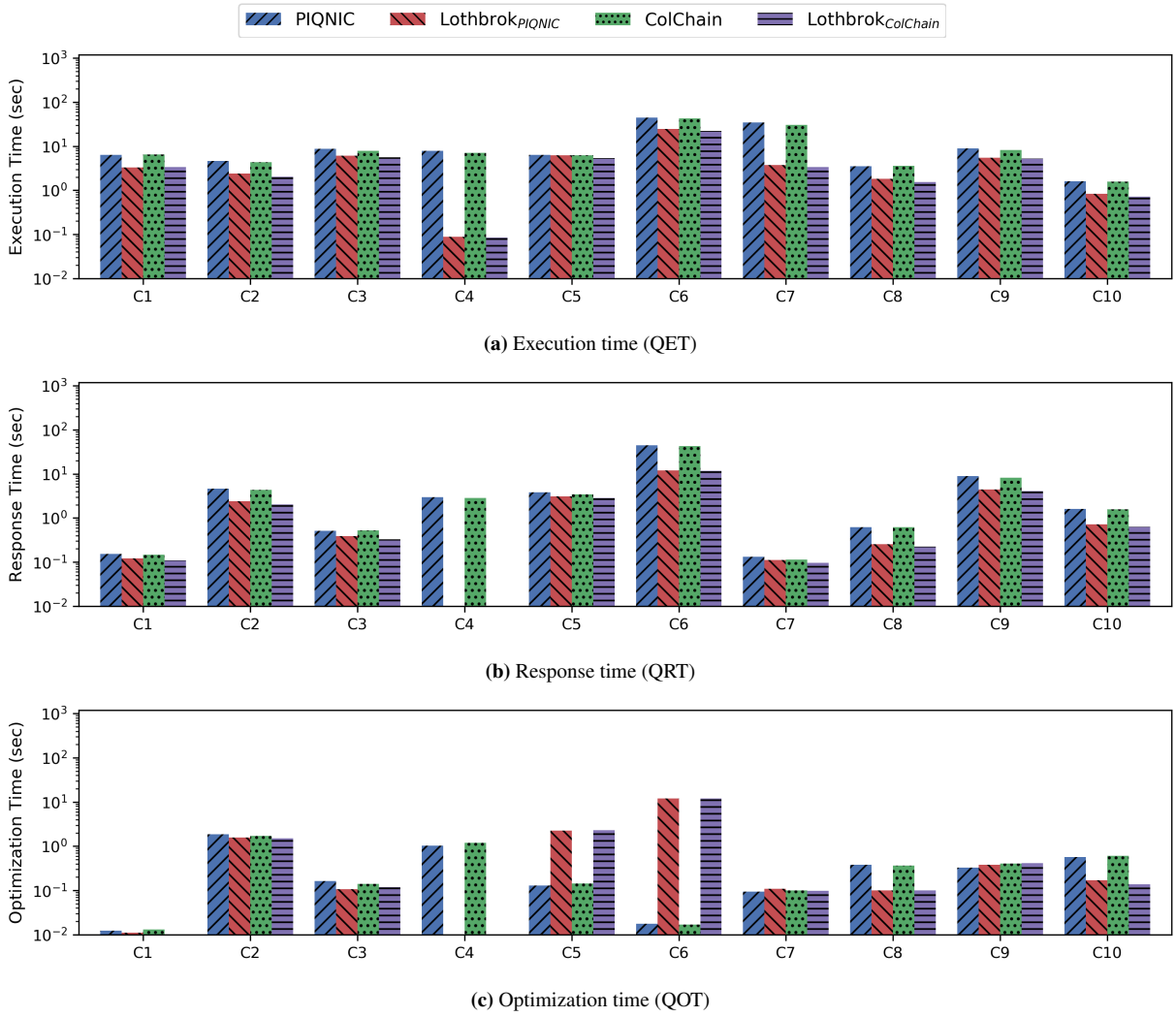
(f) Number of transferred bytes (NTB) over `watdiv1000M`

Fig. 20. Number of requests (REQ) and number of transferred bytes (NTB) for the WatDiv datasets and star queries.

Nevertheless, the slightly worse performance for LOTHBROK over `watdiv-path` is compensated by the significantly improved performance over the other query loads, so we still see a performance increase for the `watdiv-union` query load. As such, our experimental results show that LOTHBROK is generally able to increase performance over queries with star-shaped subqueries (i.e., all other queries than path queries) significantly and that the increase in performance depends on the shape of the query; queries with fewer but larger star patterns (cf. Figure 16c) show a bigger performance increase than queries with many but small star patterns.

7.4. Network Usage

Figure 20 shows the network usage when processing WatDiv queries over each WatDiv dataset in terms of the number of requests (Figures 20a-20c) and the number of transferred bytes (Figures 20d-20f) in logarithmic scale. LOTHBROK incurs a significant lower network overhead for all query loads except `watdiv-path` despite the larger number of relevant fragments as discussed in Section 7.3. This is caused by LOTHBROK having to send significantly fewer requests for each star pattern since a star pattern can be processed entirely over the relevant fragments, even if there are more fragments (and thus nodes) to send the requests to. Again, the query loads with a smaller number of star patterns see a larger decrease in network usage since larger parts of the queries can be processed by individual nodes. Since the queries in the `watdiv-path` query load do not benefit from the star pattern-based query processing, the network usage is slightly higher; however, even still, the `watdiv-union` shows an improvement in the network usage for LOTHBROK. These results are in line with the experiments shown in Sections 7.2 and 7.3 and support the hypothesis that LOTHBROK increases performance by lowering the network overhead when processing queries, compared to state-of-the-art systems such as PIQNIC and COLCHAIN.



(a) Execution time (QET)

(b) Response time (QRT)

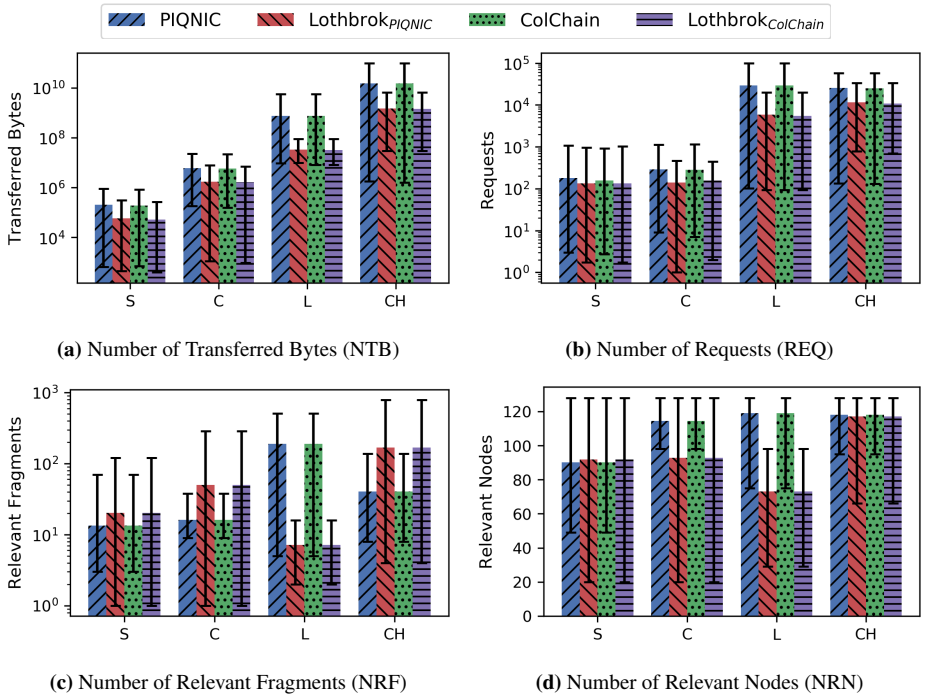
(c) Optimization time (QOT)

Fig. 21. Query execution time (a), response time (b), and optimization time (c) for the C query load over LargeRDFBench.

7.5. Performance of Individual Queries

In these experiments, we ran the LargeRDFBench queries three times on each system sequentially to test the performance of those individual queries and report the average results. Figure 21 shows the execution time (Figure 21a), response time (Figure 21b), and optimization time (Figure 21c) for the C query load over LargeRDFBench in logarithmic scale. Similar figures for the other LargeRDFBench query loads are provided on our website[3]. The results in Figure 21 are similar to the remaining query loads; we show the C query load since this query load had the most diversity in the performance across the queries.

While, in our experiments, LOTHBROK provides an improvement for the execution time (Figure 21a) across all the queries in LargeRDFBench, the improvement varies based on the query shape in line with the findings of [10, 11] and the query shape experiments shown in Section 7.3. For instance, query C4 consists of one highly selective star pattern with 6 unique predicates. LOTHBROK is thus able to answer C4 with one request to the only fragment with that predicate combination, while PIQNIC and COLCHAIN have to send at least one request per triple pattern. Hence, LOTHBROK has around two orders of magnitude better performance for this particular query. On the other hand, query C5 consists of four star patterns, two of which contain only one triple pattern with one of them being



(a) Number of Transferred Bytes (NTB)

(b) Number of Requests (REQ)

(c) Number of Relevant Fragments (NRF)

(d) Number of Relevant Nodes (NRN)

Fig. 22. Number of Transferred Bytes (NTB) (a), Number of Requests (REQ) (b), Number of Relevant Fragments (NRF) (c), and Number of Relevant Nodes (NRN) (d) for each LArgeRDFBench query load.

the very common `rdfs:label` predicate. As a result, LOTHBROK has more than twice the number of relevant fragments for C5 compared to both PIQNIC and COLCHAIN. Nevertheless, LOTHBROK still has slightly improved performance for C5 compared to PIQNIC and COLCHAIN since the query still contains two star patterns with three triple patterns each, meaning the increased optimization and communication overhead that the additional relevant fragments entail is offset by the benefits of processing the star patterns over the individual fragments. The response times (Figure 21b) show a similar comparison between the systems as the execution times (Figure 21a) with the exception of query C4. Again, the reason being that LOTHBROK can process this query with a single request, and therefore the first result is obtained immediately after receiving the response to the request.

However, the optimization times (Figure 21c) differ quite significantly depending on the number of relevant fragments to the query. For instance, queries like C5 and C6 (that contain a star pattern consisting of a single triple pattern with a very common predicate) incur a significant number of relevant fragments for LOTHBROK (286 for C5 and 144 for C6) and thus a higher optimization time. This is the case, since a higher number of relevant fragments means a higher number of SPBFs have to be intersected which represents an overhead. In all of these cases, however, the benefits of processing entire star patterns over the fragments, in terms of decreased network overhead mean that the overall execution time is still lower for LOTHBROK. This is especially the case for C6, which contains a star pattern with 6 triple patterns that in PIQNIC and COLCHAIN have to be processed individually. On the other hand, queries like C4 that contain few very selective star patterns have a low optimization time for LOTHBROK, since each star pattern have very few relevant fragments. In the case of C4, PIQNIC and COLCHAIN have a relatively high number of relevant fragments due to one of the predicates being the common `owl:sameAs` predicate that occurs in multiple datasets. As a result, PIQNIC and COLCHAIN have a significantly higher optimization time for this query compared to LOTHBROK.

Figure 22 shows the number of transferred bytes (Figure 22a), the number of requests (Figure 22b), the number of relevant fragments (Figure 22c), and the number of relevant nodes (Figure 22d) for each LargeRDFBench query load in logarithmic scale. We provide figures displaying each measure in Figure 22 for each individual LargeRDFBench query on our website[3]. As with the experiments shown in Section 7.4, LOTHBROK clearly incurs a lower network

usage than both PIQNIC and COLCHAIN, both in terms of data transfer (Figure 22a) and the number of requests made (Figure 22b). This, together with the performance experiments, shows that LOTHBROK is able to reduce the network overhead significantly across all query loads and, in doing so, increase the performance overall.

Interestingly, while for most query loads, LOTHBROK has a higher number of relevant fragments (Figure 22c) in line with the experiments presented in Section 7.3, for the L query load, LOTHBROK has a lower number of relevant fragments in most queries. The reason is that the queries in this query load mostly use data from the quite structured `linkedTCGA` datasets which contain few similar characteristic sets, thus incurring a low number of relevant fragments per star pattern. On the other hand, for PIQNIC and COLCHAIN, the fact that some star patterns with a low number of triple patterns include common predicates like `rdf:type` increases the number of relevant fragments. The number of relevant nodes (Figure 22d) shows a similar trend to the number of relevant fragments since each fragment is replicated across 20 nodes; in some cases, however, where two relevant fragments are simultaneously replicated by some of the same nodes, the actual number of relevant nodes will be a bit lower than when the relevant nodes replicate exactly one relevant fragment.

Our results are similar for all query loads (figures provided on our website[3]) and show that even for the complex queries in query loads C and CH and the queries with a large number of intermediate results in query load L, LOTHBROK presents a significant performance increase because it lowers the communication overhead. For some queries, this is quite significant; for instance the queries C4 and S3 where LOTHBROK increases execution time by up to two orders of magnitude. Furthermore, some queries in the L and CH groups that timed out for PIQNIC and COLCHAIN, such as L3 and CH2, finished within the timeout of 1200 seconds for LOTHBROK. This is in line with the results presented in Section 7.2 and suggests that LOTHBROK is able to complete more queries within the timeout than the state-of-the-art systems.

7.6. Discussion

Our experimental evaluations show that LOTHBROK significantly improves query performance while lowering the communication overhead compared to PIQNIC and COLCHAIN. LOTHBROK does so by distributing subqueries to other nodes such that the estimated network cost is limited as much as possible, and by processing entire star patterns over the individual fragments. In doing so, LOTHBROK decreases the network usage both in terms of the data transfer and number of requests, and increases performance by up to two orders of magnitude compared to the state of the art. Moreover, LOTHBROK does so while providing scalable performance under load; in fact, even when all nodes in the network issue queries concurrently, LOTHBROK maintains efficient query processing.

The only exception to the improved scalability and performance in our experiments is the slightly worse performance for path queries in LOTHBROK compared to existing approaches. This is mainly caused by factors like similarity and coplanarity of fragments having an effect on the performance of LOTHBROK that is not mitigated by star-shaped query optimizations. Specifically, we see an increased number of relevant fragments and lower number of pruned fragments per query incurred by LOTHBROK over the path queries caused by many of the fragments containing very similar characteristic sets. Optimizing queries for such path queries is a difficult problem, corroborated by several previous studies [10, 11, 24, 25, 50], and was out of scope of this paper. Nevertheless, looking into query optimization for path queries is an interesting topic for future work. For instance, one possible solution to the path query optimization problem could be co-location of fragments relevant for common paths on the same nodes, similar to workload-aware fragmentation techniques [21, 22], or storing and using statistics about commonly used paths as part of the query optimization step.

We emphasize that our goal with this work was not to beat client-server systems like SPF [11] or WiseKG [10] in terms of performance, rather we aimed at making query processing in the decentralized setup, where node failures can be tolerated [6], more feasible with high scalability. While this paper specifically aims to optimize queries, the nature of P2P networks means that processing queries comes at a performance cost due to queries having to be split across multiple nodes that is impossible to remove completely. In any case, P2P systems also have the benefit that the query processing effort is divided across several nodes. As such, even when the network incurs heavy load, the performance should stay relatively stable in contrast to the centralized solutions. In fact, our scalability experiments in Section 7.2 clearly support this since they show that the query throughput hardly decreases, even when all the nodes in the network process queries at the same time (Figure 17). On the other hand, the analysis provided by [6, 10]

of SPF and WiseKG (both are client-server systems where all queries have to be evaluated on a single server holding all the data) show a significant decrease in query throughput when the load increases.

8. Conclusions

In this paper, we proposed LOTHBROK a novel query optimization approach for SPARQL queries over decentralized knowledge graphs. LOTHBROK builds upon recent work on decentralized Peer-to-Peer (P2P) systems [6, 7] and introduces a novel fragmentation technique based on characteristic sets [24], i.e., predicate families, as well as a novel indexing scheme that summarizes the sets of subjects and objects in a fragment using partitioned bitvectors. Furthermore, LOTHBROK proposes a query optimization strategy based on cardinality estimation, fragment compatibility, and data locality that is able to delegate the processing of (sub)queries to other, neighboring nodes in the network that hold relevant data. We implemented our approach on top of two recent systems and evaluated LOTHBROK's capabilities over well-known benchmarking suites containing real-world data and queries, as well as the performance of LOTHBROK under load using large-scale synthetic datasets and stress-testing query templates. The experimental results show that LOTHBROK significantly reduces the network overhead when processing queries in a P2P network and, in doing so, increases performance by up to two orders of magnitude.

While LOTHBROK generally improves performance, our experimental results showed that path queries have slightly worse performance for LOTHBROK than existing approaches. As such, our future work includes looking into the optimization problem for path queries, e.g., by co-locating relevant fragments for common path patterns on the same nodes, similar to workload-aware partitioning techniques [21, 22]. Furthermore, a complete analysis of the effects of graph complexity metrics, like density and centrality, on the fragment skew, query performance, and indexing strategy, as well as an analysis of different fragmentation techniques, e.g., based on SHACL/ShEx shapes [66, 67], is important future work. We also plan to expand the range of supported queries to include aggregation and analytical queries [68, 69], and add support for provenance both for data [70–72], so that the system has information about the origin of the data it uses, as well as for queries [73] so that the system can explain how query answers were computed. Finally, we plan to generalize our approach to other types of distributed graphs, extend the query optimizer to support relevance scores or benefit-based source selection [27], extend our approach with methods for handling inconsistent or conflicting data, analyze the benefits and tradeoffs of the merging procedure presented in Section 4 and indexes presented in Section 4.3 using even more diverse real-world datasets, and investigate the potential benefits of allowing execution plans of any shape (i.e., removing the assumption, given in Section 5, that execution plans are always left-deep).

Acknowledgments. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensen Foundation.

References

[1] K. Hose, Knowledge Graph (R) Evolution and the Web of Data, in: *MEPDaW 2021*, 2021.
[2] D. Vrandecic and M. Krötzsch, Wikidata: a free collaborative knowledgebase, *Commun. ACM* **57**(10) (2014), 78–85.
[3] M. Dumontier, A. Callahan, J. Cruz-Toledo, P. Ansell, V. Emonet, F. Belleau and A. Droit, Bio2RDF Release 3: A larger, more connected network of Linked Data for the Life Sciences, in: *ISWC 2014 Posters & Demonstrations Track*, Vol. 1272, 2014, pp. 401–404.
[4] C.B. Aranda, A. Hogan, J. Umbrich and P. Vandenbussche, SPARQL Web-Querying Infrastructure: Ready for Action?, in: *ISWC 2013*, 2013, pp. 277–293.
[5] P. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan and C.B. Aranda, SPARQLES: Monitoring public SPARQL endpoints, *Semantic Web* **8**(6) (2017), 1049–1065.
[6] C. Aebeloe, G. Montoya and K. Hose, A Decentralized Architecture for Sharing and Querying Semantic Data, in: *ESWC 2019*, 2019, pp. 3–18.
[7] C. Aebeloe, G. Montoya and K. Hose, ColChain: Collaborative Linked Data Networks, in: *WWW 2021*, 2021, pp. 1385–1396.
[8] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt and R. John, UniStore: Querying a DHT-based Universal Storage, in: *ICDE 2007*, 2007, pp. 1503–1504.
[9] R. Verborgh, M.V. Sande, O. Hartig, J.V. Herwegen, L.D. Vocht, B.D. Meester, G. Haesendonck and P. Colpaert, Triple Pattern Fragments: A low-cost knowledge graph interface for the Web, *J. Web Sem.* **37-38** (2016), 184–206.

[10] A. Azzam, C. Aebeloe, G. Montoya, I. Keles, A. Polleres and K. Hose, WiseKG: Balanced Access to Web Knowledge Graphs, in: *WWW 2021*, 2021, pp. 1422–1434. doi:10.1145/3442381.3449911.
[11] C. Aebeloe, I. Keles, G. Montoya and K. Hose, Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns, *CoRR* **abs/2002.09172** (2020). https://arxiv.org/abs/2002.09172.
[12] A. Azzam, J.D. Fernández, M. Acosta, M. Beno and A. Polleres, SMART-KG: Hybrid Shipping for SPARQL Querying on the Web, in: *WWW 2020*, 2020, pp. 984–994.
[13] L. Heling, M. Acosta, M. Maleshkova and Y. Sure-Vetter, Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study, in: *ISWC 2018*, 2018, pp. 86–102. doi:10.1007/978-3-030-00668-6_6.
[14] M. Cai and M.R. Frank, RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network, in: *WWW*, 2004, pp. 650–657.
[15] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2009. http://www.bitcoin.org/bitcoin.pdf.
[16] D. Graux, G. Sejdiu, H. Jabeen, J. Lehmann, D. Sui, D. Muhs and J. Pfeffer, Profiting from Kitties on Ethereum: Leveraging Blockchain RDF with SANSA, in: *ISWC Posters and Demos*, 2018.
[17] M. Sopek, P. Gradzki, W. Kosowski, D. Kuzinski, R. Trójczak and R. Trypuz, GraphChain: A Distributed Database with Explicit Semantics and Chained RDF Graphs, in: *WWW Companion*, 2018, pp. 1171–1178.
[18] Z. Zheng, S. Xie, H. Dai, X. Chen and H. Wang, Blockchain challenges and opportunities: a survey, *IJWGS* **14**(4) (2018), 352–375.
[19] C. Aebeloe, G. Montoya and K. Hose, Decentralized Indexing over a Network of RDF Peers, in: *ISWC 2019*, 2019, pp. 3–20.
[20] A. Ailamaki, D.J. DeWitt, M.D. Hill and D.A. Wood, DBMSs on a Modern Processor: Where Does Time Go?, in: *VLDB 1999*, 1999, pp. 266–277.
[21] A. Akhter, M. Saleem, A. Bigerl and A.-C. Ngonga Ngomo, Efficient RDF Knowledge Graph Partitioning Using Querying Workload, in: *K-Cap 2021*, 2021.
[22] K. Hose and R. Schenkel, WARP: Workload-aware replication and partitioning for RDF, in: *ICDE 2013 Workshops*, 2013, pp. 1–6. doi:10.1109/ICDEW.2013.6547414.
[23] L. Galárraga, K. Hose and R. Schenkel, Partout: a distributed engine for efficient RDF processing, in: *WWW 2014*, ACM, 2014, pp. 267–268. doi:10.1145/2567948.2577302.
[24] T. Neumann and G. Moerkotte, Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins, in: *ICDE 2011*, 2011, pp. 984–994. doi:10.1109/ICDE.2011.5767868.
[25] G. Montoya, H. Skaf-Molli and K. Hose, The Odyssey Approach for Optimizing Federated SPARQL Queries, in: *ISWC 2017*, pp. 471–489. https://doi.org/10.1007/978-3-319-68288-4_28.
[26] Y. Park, S. Ko, S.S. Bhowmick, K. Kim, K. Hong and W. Han, G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching, in: *SIGMOD 2020*, ACM, 2020, pp. 1099–1114. doi:10.1145/3318464.3389702.
[27] K. Hose and R. Schenkel, Towards benefit-based RDF source selection for SPARQL queries, in: *SWIM 2012*, ACM, 2012, p. 2. doi:10.1145/2237867.2237869.
[28] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler and J. Umbrich, Data summaries for on-demand queries over linked data, in: *WWW 2010*, ACM, 2010, pp. 411–420. doi:10.1145/1772690.1772733.
[29] J. Umbrich, K. Hose, M. Karnstedt, A. Harth and A. Polleres, Comparing data summaries for processing live queries over Linked Data, *World Wide Web* **14**(5–6) (2011), 495–544. doi:10.1007/s11280-010-0107-z.
[30] M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra and A. Polleres, Efficiently Joining Group Patterns in SPARQL Queries, in: *ESWC 2010*, 2010, pp. 228–242. doi:10.1007/978-3-642-13486-9_16.
[31] J. Pérez, M. Arenas and C. Gutiérrez, Semantics and complexity of SPARQL, *ACM Trans. Database Syst.* **34**(3) (2009), 16:1–16:45.
[32] M. Saleem, A. Hasnain and A.N. Ngomo, LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation, 2018, pp. 85–125.
[33] G. Aluç, O. Hartig, M.T. Özsu and K. Daudjee, Diversified Stress Testing of RDF Data Management Systems, in: *ISWC 2014*, 2014, pp. 197–212. doi:10.1007/978-3-319-11964-9_13.
[34] J.V. Herwegen, R. Verborgh, E. Mannens and R.V. de Walle, Query Execution Optimization for Clients of Triple Pattern Fragments, in: *ESWC 2015*, 2015, pp. 302–318. doi:10.1007/978-3-319-18818-8_19.
[35] M. Acosta and M. Vidal, Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data, in: *ISWC 2015*, Springer, 2015, pp. 111–127. doi:10.1007/978-3-319-25007-6_7.
[36] G. Montoya, I. Keles and K. Hose, Analysis of the Effect of Query Shapes on Performance over LDF Interfaces, in: *QuWeDa@ISWC 2019*, 2019, pp. 51–66.
[37] G. Montoya, I. Keles and K. Hose, Querying Linked Data: An Experimental Evaluation of State-of-the-Art Interfaces, *CoRR* **abs/1912.08010** (2019). http://arxiv.org/abs/1912.08010.
[38] O. Hartig and C. Buil-Aranda, Bindings-Restricted Triple Pattern Fragments, in: *OTM Conferences*, 2016.
[39] G. Montoya, C. Aebeloe and K. Hose, Towards Efficient Query Processing over Heterogeneous RDF Interfaces, in: *DeSemWeb@ISWC 2018*, 2018.
[40] T. Minier, H. Skaf-Molli and P. Molli, SaGe: Web Preemption for Public SPARQL Query Services, in: *WWW 2019*, ACM, 2019, pp. 1268–1278. doi:10.1145/3308558.3313652.
[41] M. Acosta, M. Vidal, T. Lampo, J. Castillo and E. Ruckhaus, ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints, in: *ISWC 2011*, 2011, pp. 18–34. doi:10.1007/978-3-642-25073-6_2.
[42] A. Charalambidis, A. Troumpoukis and S. Konstantopoulos, SemaGrow: optimizing federated SPARQL queries, in: *SEMANTiCS 2015*, 2015, pp. 121–128. doi:10.1145/2814864.2814886.

[43] O. Görlitz and S. Staab, SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions, in: *(COLD2011)*, 2011.
[44] A. Schwarte, P. Haase, K. Hose, R. Schenkel and M. Schmidt, FedX: Optimization Techniques for Federated Query Processing on Linked Data, in: *ISWC 2011*, 2011, pp. 601–616. doi:10.1007/978-3-642-25073-6_38.
[45] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Processing Aggregate Queries in a Federation of SPARQL Endpoints, in: *ESWC*, 2015, pp. 269–285.
[46] L. Heling and M. Acosta, A Framework for Federated SPARQL Query Processing over Heterogeneous Linked Data Fragments, *CoRR* **abs/2102.03269** (2021). https://arxiv.org/abs/2102.03269.
[47] A.L. Jakobsen, G. Montoya and K. Hose, How Diverse Are Federated Query Execution Plans Really?, in: *ESWC 2019*, 2019, pp. 105–110. doi:10.1007/978-3-030-32327-1_21.
[48] G. Montoya, M. Vidal and M. Acosta, A Heuristic-Based Approach for Planning Federated SPARQL Queries, in: *COLD 2012*, 2012.
[49] M. Saleem, A. Potocki, T. Soru, O. Hartig and A.N. Ngomo, CostFed: Cost-Based Query Optimization for SPARQL Endpoint Federation, in: *SEMANTiCS 2018*, 2018, pp. 163–174. doi:10.1016/j.procs.2018.09.016.
[50] A. Gubichev and T. Neumann, Exploiting the query structure for efficient join ordering in SPARQL queries, in: *EDBT 2014*, 2014, pp. 439–450. doi:10.5441/002/edbt.2014.40.
[51] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou and A. Papadakis-Pesaresi, Atlas: Storing, updating and querying RDF(S) data on top of DHTs, *J. Web Semant.* **8**(4) (2010), 271–277. doi:10.1016/j.websem.2010.07.001.
[52] E. Mansour, A.V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga and T. Berners-Lee, A Demonstration of the Solid Platform for Social Web Applications, in: *WWW 2016*, ACM, 2016, pp. 223–226. doi:10.1145/2872518.2890529.
[53] P. Larson, Dynamic Hash Tables, *Commun. ACM* **31**(4) (1988), 446–457. doi:10.1145/42404.42410.
[54] A. Crespo and H. Garcia-Molina, Routing Indices For Peer-to-Peer Systems, in: *ICDCS 2002*, 2002, pp. 23–32.
[55] D. Brickley, R.V. Guha and B. McBride, RDF Schema 1.1, *W3C recommendation* **25** (2014), 2004–2014.
[56] W.W.W. Consortium et al., SPARQL 1.1 overview (2013).
[57] M. Saleem, G. Szárnyas, F. Conrads, S.A.C. Bukhari, Q. Mehmood and A.N. Ngomo, How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks, in: *WWW 2019*, ACM, 2019, pp. 1623–1633. doi:10.1145/3308558.3313556.
[58] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann and S. Auer, Quality assessment for Linked Data: A Survey, *Semantic Web* **7**(1) (2016), 63–93. doi:10.3233/SW-150175.
[59] B. Xue and L. Zou, Knowledge Graph Quality Management: a Comprehensive Survey, *IEEE Transactions on Knowledge and Data Engineering* (2022), 1–1. doi:10.1109/TKDE.2022.3150080.
[60] S. Issa, O. Adekunle, F. Hamdi, S.S. Cherfi, M. Dumontier and A. Zaveri, Knowledge Graph Completeness: A Systematic Literature Review, *IEEE Access* **9** (2021), 31322–31339. doi:10.1109/ACCESS.2021.3056622.
[61] B.H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, *Commun. ACM* **13**(7) (1970), 422–426.
[62] O. Papapetrou, W. Siberski and W. Nejdl, Cardinality estimation and dynamic length adaptation for Bloom filters, *Distributed Parallel Databases* **28**(2–3) (2010), 119–156. doi:10.1007/s10619-010-7067-2.
[63] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres and M. Arias, Binary RDF representation for publication and exchange (HDT), *J. Web Semant.* **19** (2013), 22–41.
[64] S. Duan, A. Kementsietsidis, K. Srinivas and O. Udrea, Apples and oranges: a comparison of RDF benchmarks and real RDF datasets, in: *ACM SIGMOD 2011*, ACM, 2011, pp. 145–156. doi:10.1145/1989323.1989340.
[65] G. Aluç, O. Hartig, M.T. Özsu and K. Daudjee, Diversified Stress Testing of RDF Data Management Systems, in: *ISWC 2014*, 2014, pp. 197–212. doi:10.1007/978-3-319-11964-9_13.
[66] K. Rabbani, M. Lissandrini and K. Hose, Optimizing SPARQL Queries using Shape Statistics, in: *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, OpenProceedings.org, 2021, pp. 505–510. doi:10.5441/002/edbt.2021.59.
[67] K. Rabbani, M. Lissandrini and K. Hose, SHACL and ShEx in the Wild: A Community Survey on Validating Shapes Generation and Adoption, in: *WWW'22 Companion, April 25–29, 2022, Virtual Event, Lyon, France*, 2022.
[68] D. Ibragimov, K. Hose, T.B. Pedersen and E. Zimányi, Optimizing Aggregate SPARQL Queries Using Materialized RDF Views, in: *ISWC*, 2016, pp. 341–359.
[69] L. Galárraga, K.A. Jakobsen, K. Hose and T.B. Pedersen, Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets, in: *ISWC*, 2018, pp. 547–565.
[70] E.R. Hansen, M. Lissandrini, A. Ghose, S. Løkke, C. Thomsen and K. Hose, Transparent Integration and Sharing of Life Cycle Sustainability Data with Provenance, in: *ISWC*, Vol. 12507, 2020, pp. 378–394.
[71] A.B. Andersen, N. Gür, K. Hose, K.A. Jakobsen and T.B. Pedersen, Publishing Danish Agricultural Government Data as Semantic Web Data, in: *JIST*, Vol. 8943, 2014, pp. 178–186.
[72] O. Hartig and et al., RDF-star and SPARQL-star. W3C Draft Community Group. Report. W3C Community, 2021. https://w3c.github.io/rdf-star/cg-spec/2021-12-17.html.
[73] D. Hernández, L. Galárraga and K. Hose, Computing How-Provenance for SPARQL Queries via Query Rewriting, *Proc. VLDB Endow.* **14**(13) (2021), 3389–3401.
[74] R. Taelman, M.V. Sande, J.V. Herwegen, E. Mannens and R. Verborgh, Triple storage for random-access versioned querying of RDF archives, *J. Web Semant.* **54** (2019), 4–28. doi:10.1016/j.websem.2018.08.001.
[75] O. Pelgrin, L. Galárraga and K. Hose, Towards fully-fledged archiving for RDF datasets, *Semantic Web* **12**(6) (2021), 903–925. doi:10.3233/SW-210434.