ColChain: Collaborative Linked Data Networks

Christian Aebeloe
Aalborg University
caebel@cs.aau.dk

Gabriela Montoya
Aalborg University
gmontoya@cs.aau.dk

Katja Hose
Aalborg University
khose@cs.aau.dk

ABSTRACT

One of the major obstacles that currently prevents the Semantic Web from exploiting its full potential is that the data it provides access to is sometimes not available or outdated. The reason is rooted deep within its architecture that relies on data providers to keep the data available, queryable, and up-to-date at all times – an expectation that many data providers in reality cannot live up to for an extended (or infinite) period of time. Hence, decentralized architectures have recently been proposed that use replication to keep the data available in case the data provider fails. Although this increases availability, it does not help keeping the data up-to-date or allow users to query and access previous versions of a dataset. In this paper, we therefore propose ColChain (COLlaborative knowledge CHAINs), a novel decentralized architecture based on blockchains that not only lowers the burden for the data providers but at the same time also allows users to propose updates to faulty or outdated data, trace updates back to their origin, and query older versions of the data. Our extensive experiments show that ColChain reaches these goals while achieving query processing performance comparable to the state of the art.

ACM Reference Format:
Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2021. ColChain: Collaborative Linked Data Networks. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3450037

1 INTRODUCTION

The increasing popularity of the Semantic Web and its Web of Data has over the past few years led to a rapid increase in the amount of data published as Linked Open Data (LOD) – spanning a broad range of topics, such as life sciences [12], geography [33], and general knowledge [36]. Such data is made available through either raw data dumps, SPARQL endpoints, or dereferenceable URIs. Yet, the current architecture of the Web of Data requires that we rely on the individual data providers to maintain access to their datasets. While this is a practical and simple solution, it causes several issues that significantly limit the general applicability of technologies relying on the Web of Data as their backbone. As highlighted in several recent studies [4, 7, 34, 35], providing access to such datasets is a significant burden for the data providers, which often results in downtime of public SPARQL endpoints [7, 34]. Another burden for the data provider is the responsibility to keep the data up to date. Current architectures do not support mechanisms to update faulty or outdated data in a community-driven way. Typically, it is the data provider publishing a new version of the entire dataset while taking the old version offline, which then becomes unavailable. However, for many applications it is helpful to access previous versions of the dataset and also trace back updates to ensure data quality [2].

In recent years, decentralized architectures [4, 10, 21] have been proposed to lift some of the burden from the data providers. For instance, RDFPeers [10] uses a structured overlay over a Peer-to-Peer (P2P) network that relies on Dynamic Hash Tables (DHTs) to determine data placement. However, in situations where nodes frequently leave or join the network, such systems have to go through a costly adjustment of the overlay and redistribute the data. Therefore, approaches such as Piqnic [4] rely on unstructured P2P systems instead, where there is no global control over which peer stores which portion of the data. Nevertheless, in both flavors of P2P architectures, replication is used to increase data availability in case of peer failures. However, none of these systems efficiently supports updates, in particular not community-driven ones.

Blockchains [15, 27, 32, 37] are chains of data blocks that represent global ledgers. They rely on the consensus of participating nodes and facilitate updates by adding new blocks to the chain. A new block is linked to the existing chain using hashes to prevent changes without reaching a consensus. Using blockchain technology for the Web of Data could thus allow users to collaborate on new updates to the data, keep the published data up-to-date, and improve its quality by correcting mistakes in community-driven efforts. However, blockchains typically replicate the chain on all participating nodes [37] to ensure immutability and persistance [32]. While this increases availability and security, it also requires that every node has to provide a considerably large amount of resources to store multiple large knowledge graphs.

In summary, availability as well as writability issues make it increasingly difficult to trust and rely on the Web of Data since it is sometimes impossible to (i) access the data, (ii) trace back faulty data and updates to the origin, (iii) ensure consistent query answers over longer periods in time by accessing older versions of a dataset, and (iv) update faulty or outdated data. Hence, in this paper we propose ColChain (COLlaborative knowledge CHAINs), a system that increases data availability while enabling users to provide updates and queries over previous versions. In particular, ColChain divides the P2P network into smaller *communities* to increase availability, store updates in chains, and rely on community-wide consensus for updates. Furthermore, with the information in the blockchain, ColChain can trace back updates and access previous versions of the dataset. As such, similar to Wikidata [36] which lets users publish manual updates, ColChain relies on user-provided updates to keep data up to date. In ColChain, however, a consensus is required before an update is applied to the chain, making malicious updates and faulty data less likely. Furthermore, ColChain improves consistency and reproducibility of scientific studies that used query

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

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3450037

logs such as LSQ [29] in which queries were executed over older versions of well-known datasets. ColChain is *collaborative*, meaning users collaborate on the state of the data by suggesting updates, whereas updates are *consensual*, meaning users form a consensus on each update individually. In summary, this paper makes the following contributions:

- A formal definition of ColChain, a novel decentralized architecture that increases data availability while allowing nodes to trace back updates to the original source as well as propose updates to existing data.
- An approach to process SPARQL queries over previous versions of the datasets published in a ColChain network.
- An extensive evaluation of the performance and overhead of query processing in a ColChain network using a large-scale benchmark (LargeRDFBench [17]).

This paper is organized as follows. Section 2 discusses related work while Section 3 introduces preliminaries. Section 4 presents ColChain and Section 5 outlines consensual updates. Section 6 describes query processing in ColChain. Section 7 then presents experimental results and Section 8 concludes the paper.

2 RELATED WORK

In this section, we discuss approaches to increase the availability of knowledge graphs and the pitfalls of such approaches. Furthermore, we discuss approaches that can accommodate collaborative or consensual updates to data fragments and their shortcomings.

2.1 Client-Server Architectures

SPARQL endpoints remain among the most popular interfaces for querying RDF datasets. SPARQL endpoints are centralized servers that provide an HTTP interface to process SPARQL queries. However, such endpoints are expensive to maintain and experience downtime, leaving the data inaccessible [7, 34]. SaGe [24] increases the availability of the centralized server by suspending queries after a fixed time quantum to avoid long-running queries exhausting the server resources. However, SaGe still processes entire queries on the server, and can thus still suffer downtime.

Several recent approaches [3, 8, 9, 24, 31, 35] attempt to increase the availability of servers by lowering their query processing load. Triple Pattern Fragments (TPF) [35] shift most of the query processing burden from the server to the client, lowering the server load and increasing availability. This is done by ensuring that the server only has to process individual triple patterns, whereas joins and other SPARQL operators are processed by the client. However, while TPF does decrease the load on the server, it incurs in high network usage and high client load, and the performance of TPF depends strongly on factors such as the type of triple pattern and fragment cardinality [18]. Derivatives of TPF [3, 8, 9, 16, 25] therefore aim to further reduce the server load by, for example, sending bulks of previously obtained bindings to the server [16], or increasing the overall throughput by taking advantage of the characteristics of the queries [25]. SPF [3] processes star-shaped subqueries on the server, whereas Smart-KG [8] ships predicate-family partitions to the client. WiseKG [9] combines the approaches presented by SPF and Smart-KG and determines dynamically which strategy is the most cost-efficient for a given subquery. Nevertheless, such systems still rely on centralized servers that are subject to failure.

Federated systems [1, 26, 30, 31] process queries over multiple SPARQL endpoints and make it feasible to exploit the resources of multiple servers during query processing. Nevertheless, since federated systems rely on a fixed set of SPARQL endpoints that are subject to failure, they do not provide a reliable solution to the availability issue. Col-Graph [19] lets consumers create updates to datasets available over federated sources by using `CONSTRUCT` queries to create fragments which they can then change and expose via SPARQL endpoints. This, however, still requires significant resources on the part of the consumers.

In any case, systems that rely on a central server or a fixed set of central servers do not completely address the availability issue; while reducing the load on the server reduces the chance of server failure, the data will still be unavailable in the event of failure.

2.2 Decentralized Architectures

Decentralized architectures have over the past few years been gaining attention in the Semantic Web community [4, 8, 22, 24, 35]. Several approaches propose to use Peer-to-Peer (P2P) architectures to query RDF data [4, 10, 20, 21]. Such approaches rely on the replication of data over several nodes to increase the availability of the data. Some of these approaches [10, 20, 21] apply a structured overlay such as Dynamic Hash Tables (DHTs) over the network and enforce data placement. While this can be exploited to make query processing relatively efficient, it also makes the networks vulnerable to churn (when nodes frequently leave and join the network) as the overlay then has to be computed again and the data has to be redistributed each time a node leaves or joins the network. Instead, [4] proposes an unstructured network, where connections between nodes are random and replication of data is managed by the data provider. Processing queries in such a network usually relies on flooding the network. This is generally inefficient; however, the query processing performance of such approaches were increased using decentralized indexes [5, 11].

Decentralized systems allow providers to freely upload data to the network. By relying on replication of the data, such systems increase the availability of the data in the event of node failures. However, there is little to no way for consumers to ensure that they have access to up-to-date data, process queries over previous dataset versions, or trace back updates to the original source. Colledge [23] therefore presented a vision of collaborative networks where heterogeneous data providers are connected with consumers. Inspired by this vision, we propose a collaborative network of peers that accommodates consensual updates to data fragments.

2.3 Blockchains

Blockchains [27] define a global ledger of blocks that all nodes store. They rely on the consensus of participating nodes on the state of the ledger. Using blockchains over decentralized knowledge graphs has, to the best of our knowledge, only briefly been researched [15, 32]. However, such systems require the entire chain to be stored on all nodes [37], pack structured data into blocks of a fixed size, and guarantee immutability of the data itself.

Differently, in relational database systems, multiple previous studies [6, 13] have proposed partial solutions to these limitations.

BlockchainDB [13] provides a partitioned database layer on top of an existing blockchain, meaning not all nodes store the entire dataset. CAPER [6] defines the chain as a directed acyclic graph in such a way that a node only has to maintain a local view over the entire chain. These approaches, however, only present partial solutions to the limitations mentioned above; they either limit node autonomy, still require the entire chain to be stored on all nodes, or enforce shared relational schema on each node. Furthermore, they view the data within a system as one big dataset rather than several separate datasets owned by different providers.

In this paper, we propose an approach that builds upon blockchains and unstructured P2P systems that structures the network in communities and puts community-based ledgers on top of partitioned knowledge graphs such that each node only stores small subsets of the data and chain.

3 PRELIMINARIES

In this section, we briefly define knowledge graphs followed by the fundamentals of unstructured P2P networks and indexing in such network that ColChain builds upon.

3.1 Knowledge Graphs

The standard format for encoding knowledge graphs is RDF[1]. RDF structures data as triples describing edges in a *knowledge graph*, i.e., a triple consists of a subject, a predicate, and an object.

Definition 1 (RDF Triple). *Given the infinite and disjoint sets U, B, and L, describing the set of all URIs/IRIs, blank nodes, and literals, an RDF triple is a triple of the form $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$, where s, p, o are called the subject, predicate, and object.*

A *knowledge graph* $\mathcal{G}$ is a set of RDF triples. The de facto query language for querying over knowledge graphs is SPARQL[2]. A SPARQL query consists of a set of *triple patterns*.

Definition 2 (Triple Pattern). *Given the infinite and disjoint sets U, B, L, and V, describing the set of all URIs/IRIs, blank nodes, literals, and variables, a triple pattern is a triple of the form $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.*

We say that a triple pattern p matches a knowledge graph $\mathcal{G}$ iff there exists a mapping from the variables in p to nodes or edges in $\mathcal{G}$ such that applying the mapping to p yields an RDF triple in $\mathcal{G}$. A Basic Graph Pattern (BGP) P is a set of (conjunctive) triple patterns. A SPARQL query consists of BGPs combined with operators. such as `UNION` or `OPTIONAL`. The answer to a BGP P over a knowledge graph $\mathcal{G}$ is given as a *solution mapping*, defined as follows.

Definition 3 (Solution mapping [35]). *Given a BGP P and a knowledge graph $\mathcal{G}$, U, B, L are the sets of URIs, blank nodes, and literals in $\mathcal{G}$, and V is the set of variables in P. A* solution mapping *μ is a partial mapping $\mu : V \mapsto (U \cup B \cup L)$.*

Given a triple pattern tp and a solution mapping μ, the notation $\mu[tp]$ denotes the triple (pattern) obtained by replacing variables in tp according to the bindings in μ. A triple t is said to be a *matching* triple to tp if there exists a solution mapping μ such that $t = \mu[tp]$. Furthermore, $dom(\mu)$ returns the *domain* of μ, i.e., the set of variables that are bound in μ and $vars(tp)$ returns the variables in tp.

3.2 ColChain Peer-to-Peer Layer

This section defines the basics of unstructured P2P networks and decentralized indexing [4, 5] that ColChain builds upon.

An unstructured P2P network consists of a set of (possibly heterogeneous) interconnected nodes. Each node in the network maintains a local datastore (a set of knowledge graphs). Furthermore, due to the lack of global knowledge in such a network, each node maintains a partial view over the network, i.e., a set of remote nodes to interact with. By replicating datasets across several nodes, such a network can ensure the availability of the data even if the data provider fails. However, since knowledge graphs today can contain several billions of triples, and nodes in a P2P network are relatively resource restricted, uploaded knowledge graphs are divided into smaller disjoint *fragments*. Fragments can be obtained from a knowledge graph by applying a *fragmentation function*, defined as follows.

Definition 4 (Fragmentation function). *A fragmentation function F is a function that maps from a knowledge graph $\mathcal{G}$ to a set of fragments, i.e., $F : \mathcal{G} \mapsto 2^{\mathcal{G}}$ such that $\forall f_1, f_2 \in F(\mathcal{G}) : f_1 \cap f_2 = \emptyset$.*

An example of a fragmentation function is the very coarse-granular function $F_C(\mathcal{G}) = \{\mathcal{G}\}$ that does not split up the original dataset. Another example is the slightly more fine-granular predicate-based fragmentation function that creates a fragment for each predicate in the knowledge graph [4]. This means that fragments can be adapted to various characteristics of each knowledge graph. Each fragment has an identifier denoted u_f. To achieve replications of fragments over several nodes, the uploading node selects a neighbor and propagates the fragment in a chain. Furthermore, updates to fragments are only possible if the owner node issues an update and propagates it throughout the network until all replicas are reached.

To speed up query processing, using routing indexes can help limiting the number of nodes to query by identifying which nodes are relevant to a given subquery. As a result, the network overhead is reduced and query processing performance increased. In order to process queries over fragments not in the local datastore, nodes must include indexes for fragments in their distributed index. A distributed index is defined as follows.

Definition 5 (Distributed Index). *Let n be a node, $\mathcal{N}$ be the set of nodes within a network, $\mathcal{T}$ be the (infinite) set of possible triple patterns, and $\mathcal{F}$ be the (finite) set of fragments that n has access to. A* distributed index *on n is a tuple $I_n = (\nu, \eta)$ with $\nu : \mathcal{T} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^{\mathcal{N}}$. For a triple pattern t, $\nu(t)$ returns the set of fragments in $\mathcal{F}$ that t matches. For a fragment f, $\eta(f)$ returns the nodes on which f is located.*

Definition 6 (Node Mapping). *For any BGP P and distributed index I, there exists a function $match(P, I)$ that returns a* node mapping *$M : T \mapsto 2^{\mathcal{N}}$ where T is the set of triple patterns in P, such that $M(t)$ returns the indexed nodes that have fragments matching t.*

To include indexes for remote fragments, nodes have to share particular parts of their distributed index with other nodes. In

[1] https://www.w3.org/TR/rdf11-concepts/
[2] https://www.w3.org/TR/sparql11-overview/

order to facilitate this, the nodes compute or download *slices* for each fragment not in their local datastore and combine these to form a distributed index. In simple terms, a slice is the part of a distributed index that describes a particular fragment. Formally, a slice is defined as follows.

Definition 7 (Index Slice). *Let f be a fragment. A* 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, and $\eta'(f)$ returns the set of all nodes that contain f in their local datastore. The function $s(f)$ returns the slice of f.*

For instance, in [5], the function ν' within the fragment slice definition is implemented as a partitioned bitvector that summarizes the subject and object values of a particular fragment, while η' is implemented as a dictionary that maps fragments to the nodes that have the fragment. Fragment slices can be combined, using the $\oplus$ operator, into a distributed index, Prefix-Partitioned Bloom Filter Index in [5]. For example, slices $s_{f_1} = (\nu'_1, \eta'_1)$ and $s_{f_2} = (\nu'_2, \eta'_2)$ are combined into index $I_n = (\nu'_1 \oplus \nu'_2, \eta'_1 \oplus \eta'_2)$.[3] The distributed index is then used to check the overlap of fragments during query time to determine which combinations of fragments produce join results. Distributed indexes are composed of the slices of all the accessible fragments, i.e., both locally and remotely available fragments. For a local fragment, the node computes ν' and η', while for a remote fragment, the node retrieves ν' and η' from nodes that have the fragment locally available. Given a set of slices S, the index of S, $I(S)$ can be computed as follows.

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

Query processing in such a setup uses the following general steps:

(1) Use indexes to determine which nodes to process each triple pattern over, i.e., retrieve node mappings (Definition 6).
(2) Determine the join order of the triple patterns using, for example, variable counting or cardinality estimations.
(3) Process each triple pattern in the determined order, at each step using previously obtained bindings to limit the intermediate results.

4 COLCHAIN

In this section, we provide a brief overview of a ColChain network and the architecture of a ColChain node, followed by a formal definition of ColChain.

4.1 Design and Overview

In conventional blockchains [27], it is up to the nodes to agree on the current status of the chain [37]. To allow for consensual updates to knowledge graphs and to overcome the limitations posed by blockchains (Section 2.3), we make six design choices:

(1) Knowledge graphs are divided into smaller fragments according to a specific fragmentation function[4].
(2) The network is divided into *communities* of nodes.
(3) Rather than a common ledger (henceforth called a *chain*), there is a distinct chain associated with each fragment.
(4) The chains contain updates to fragments, i.e., inserting or removing triples, as well as provenance information.
(5) We distinguish between *participants* (storing fragments and validating updates) and *observers* (storing index slices).
(6) Participants are allowed to upload new fragments and propose updates to fragments within the community; if there is a community-wide consensus, the updates will be applied.

By dividing the entire network into smaller communities of nodes (defined in Section 4.2) and using fragmentation functions (Definition 4), we avoid storing large knowledge graphs on each node. Furthermore, replicating fragments on participants in a community ensures that the data will be available even if the uploading node fails. By also letting the nodes participate in or observe any community (participants and observers are defined in Section 4.2) and join multiple communities at the same time, we ensure that nodes have the autonomy to decide for themselves what data to store. Furthermore, by relying on the collaboration of participating nodes, ColChain supports consensual updates to the published fragments. Last, by attaching chains of updates (including timestamps describing the point in time the update was applied) to the fragments themselves, nodes are able to roll-back fragments to a specific point in time and process queries over previous versions.

Since ColChain builds on top of unstructured P2P networks [4, 5], each node has a limited view over the entire network. However, in contrast to such networks, where the local view consists of a set of random neighbors, the local view of a ColChain node is entirely delimited by the communities the node participates in or observes. That is, a ColChain node is connected to all other nodes that participate in or observe at least one of the communities that it participates in or observes. Furthermore, fragments published within a community are replicated on all participants along with their update chain and index slice; observers index the fragments in a community but do not store the fragment itself. Fragments can only be published in one community. ColChain satisfies ACID; that is, it is ensured that transactions are atomic, contain only valid RDF triples, and are serialized in the order of the update chains, and that fragments are stored in permanent data storage.

Any node can create new communities and upload data to them; bootstrapping a ColChain network relies on some of the nodes to create communities and the data providers to upload datasets to these communities (only participants can upload fragments). To discover new communities, nodes ask other nodes for a list of communities they participate in. In order to gain access to the data within a community, a node has to either participate in (and replicate the fragments) or observe (and index the fragments) the community. After a node has joined a community, it expands its local view over the network to include the other participants and observers in the community; when leaving a community, it shrinks its local view to exclude the other nodes in that community but not in any of the other communities that it participates in or observes. As such, nodes are able to choose the fragments available in their local view and the fragments they store, but they do not choose the nodes that are part of their local view.

Figure 1 shows such a typical ColChain network with two communities (C_1 and C_2), each giving access to several fragments. The communities are defined by the nodes that participate in and

[3] $\oplus$ is defined in [5] 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; $(f \oplus g)(x) = g(x)$ if g is defined at x.
[4] By default ColChain uses the predicate-based fragmentation function [4]

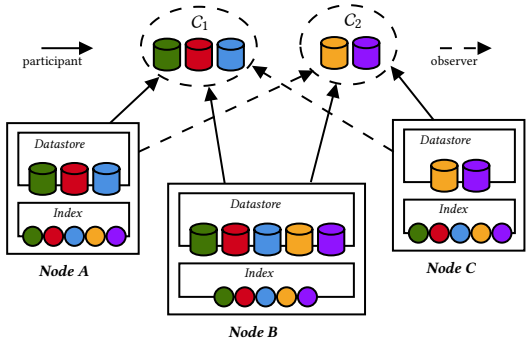


Figure 1: A typical ColChain network.

observe them and the fragments published within them. Figure 1 visualizes which fragments each community gives access to, as well as participating and observing nodes. In this example, there are three nodes (A, B, and C). Node A participates in C_1 and observes C_2, while node C participates in C_2 and observes C_1. Node B participates in both communities. Since nodes A and C do not participate in both communities, they do not replicate all fragments within the network; thus, to answer queries that require data from both communities, they have to reach out to a node participating in the community they observe. Despite not participating in both communities, nodes A and C can reach complete query answers since they, at least, observe both communities, thus indexing all fragments and participants.

Updates in ColChain are structured in chains of so-called *transactions*; a transaction is a set of bundled operations (either addition or deletion of a triple). While ColChain relies on the consensus of participants to accept transactions, it allows the *owner* nodes of fragments (i.e., the uploading nodes) to enforce transactions without consensus. Owners can also veto transactions proposed by participants. Therefore, we use the signature scheme RSA [28]; when proposing a transaction, the node will attach a signature based on its private key. Participants then validate this signature with the owner's public key to validate ownership over fragments.

Figure 2 shows the general architecture of a ColChain node (Section 4.2 provides a formal definition). Such a node combines the conventional data storage and blockchain layers in its local datastore. In particular, each fragment in the local datastore is directly associated with a chain of updates called a *secure hash chain*, i.e., each element in the chain is linked to the previous element using its hash value [27]. The SPARQL query processor is able to process full SPARQL queries and triple pattern requests. To process SPARQL queries, ColChain first uses the index to identify the set of relevant nodes and processes the query according to the method described in Section 6. A node has two interfaces; one for users to issue queries, manage communities, and propose updates, i.e., a Web interface, and one for communication between nodes, i.e., the node interface. Each interface communicates with the community manager to manage communities and updates to fragments, and the query processor for SPARQL queries and triple pattern requests.

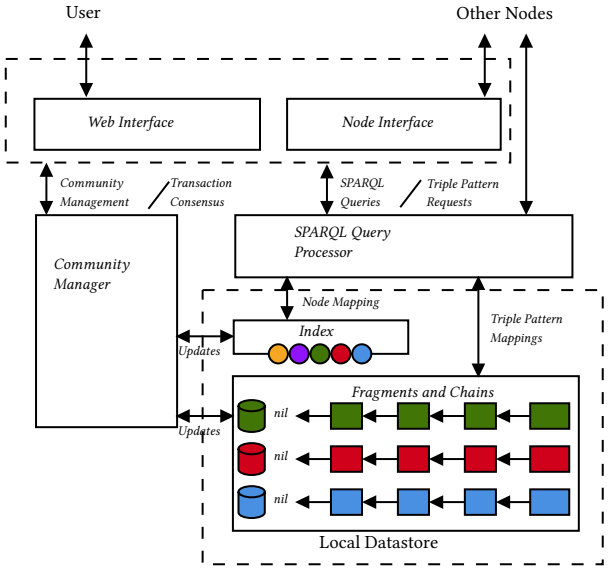


Figure 2: Architecture of a ColChain node.

4.2 Formal Definition of ColChain

A ColChain network consists of a set of nodes $\{n_1, n_2, \ldots, n_n\}$. Each node in such a network contains a local datastore with fragments from zero or more communities.

Definition 8 (Node). *A node n is a triple $n = (K, a_n, u_n)$ where*

- *$K = (\kappa_n, \rho_n)$ is a key-pair such that κ_n is n's private key and ρ_n is n's public key*
- *a_n is n's address*
- *u_n is a valid URI and n's unique identifier*

Given the definition of a node, we now define the *state* of a node.

Definition 9 (Node State). *Let n be a node. n's* state *is $\mathcal{S}_n = (\Sigma, S, I_n, \mathcal{M})$ where*

- *$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ and $\forall \sigma_i \in \Sigma : \sigma_i = (\mathcal{X}_i, f_i)$ such that*
  - *$\mathcal{X}_i$ is a secure hash chain*
  - *f_i is a fragment*
  - *All updates represented in $\mathcal{X}_i$ have been applied to f_i*
  - *All f_j, f_k such that $1 \leq j, k \leq m$ and $j \neq k$ refer to different (unique) fragments in n's local datastore*
  - *σ_i is said to be an* entry *in n's local datastore*
- *S is a set of index slices and $\forall \sigma_i \in \Sigma : s(f_i) \in S$*
- *I_n is n's distributed index, $I_n = I(S)$, consisting of index slices from local and remote fragments*
- *$\mathcal{M}$ is a set of* metadata *triples*

The metadata triples describe a node's local view over the network. These triples include the metadata of all the communities that the node participates in or observes (described in Definition 11). They include, for instance, other nodes that have joined the community. Since the metadata is structured as a set of triples (i.e., a knowledge graph), it can be stored and managed similarly to fragments. Note, however, that since the number of metadata triples is small, it can be stored in memory. Moreover, versioning and consensus over metadata updates is out of the scope of this paper,

and therefore there is no update chain associated to the metadata. Note also that the set of index slices S might contain slices from fragments not included in the node's local datastore. This is the case for fragments in communities observed by the node.

Definition 10 (Community). *A community C contains two sets of nodes called participants and observers (defined in Definitions 12 and 13) and a set of fragments, each owned by a node, i.e., $C = (N, F_C, v, u_C)$ where*

- *$N = (P_C, O_C)$ such that P_C and O_C are the sets of participants and observers, respectively*
- *F_C is a set of fragments*
- *v is an* ownership-mapping *function such that $\forall f \in F_C : v(f) = \rho_f$ where $\exists n \in P_C \bigcup O_C : \rho_f = \rho_n$*
- *u_C is a valid URI and C's unique identifier*

The ownership-mapping function v maps fragments to the public key of the owner node. v is used for validating ownership over fragments during consensus (Section 5.2). Given a fragment f, assume that u_f is a valid URI and a unique identifier for f. We now define the state of a community as follows[5].

Definition 11 (Community State). *Let C be a community. C's* state *is $\mathcal{S}_C = (\Phi, \mathcal{M})$ where*

- *$\Phi = \{\phi_1, \phi_2, \ldots, \phi_m\}$ and $\forall \phi_i \in \Phi : \phi_i = (\mathcal{X}_i, f_i)$*
  - *$\mathcal{X}_i$ is a secure hash chain*
  - *f_i is a fragment*
- *$\mathcal{M}$ is a set of* metadata *triples such that*
  - *$\forall n \in P_C : \langle u_C, \texttt{cc:participant}, u_n \rangle \in \mathcal{M}$*
  - *$\forall n \in O_C : \langle u_C, \texttt{cc:observer}, u_n \rangle \in \mathcal{M}$*
  - *$\forall f \in F_C : \langle u_C, \texttt{cc:fragment}, u_f \rangle \in \mathcal{M}$ and $\langle u_f, \texttt{cc:author}, C.v(f) \rangle \in \mathcal{M}$*

The metadata of a community, $\mathcal{M}$, contains triples detailing all the participating and observing nodes, as well as the fragments within that community and their ownership. Upon joining a community, nodes add these triples to their local metadata,i.e., nodes expand their local view of the network.

Nodes can both participate in or observe a community. Given the definition of a community state, we thus define participants and observers as follows.

Definition 12 (Participant). *Given a community C and a node n, n is said to be a* participant *in C iff for all $\phi = (\mathcal{X}, f) \in \mathcal{S}_C.\Phi$ it is the case that $(\mathcal{X}, f) \in \mathcal{S}_n.\Sigma$, $s(f) \in \mathcal{S}_n.S$, and $\forall t \in \mathcal{S}_C.\mathcal{M} : t \in \mathcal{S}_n.\mathcal{M}$.*

A participant is a node that, in its local datastore, contains replicas of the fragments and chains within the community. Note that a node is not limited to participate in just one community. Instead, nodes are free to participate in any community, and as many as they like. By participating in communities, nodes gain more efficient access to the data provided by that community since it is then available from the local datastore. Furthermore, to avoid routing attacks, and to still ensure complete query processing, nodes can also *observe* communities. This gives them access to the data within the community without having to replicate all the fragments.

[5]The prefix cc : describes the URI http://colchain.org/properties#.

Definition 13 (Observer). *Given a community C and a node n, n is said to be an* observer *in C iff for all $\phi = (\mathcal{X}, f) \in \mathcal{S}_C.\Phi$ it is the case that $s(f) \in \mathcal{S}_n.S$, and $\forall t \in \mathcal{S}_C.\mathcal{M} : t \in \mathcal{S}_n.\mathcal{M}$.*

Even though an observer does not store replicas of the fragments itself, it can still access the fragments during query processing by sending requests to community participants. This is done by downloading the index slice of the fragments within the community, and include those in the index. During query processing, the node will then ask participants for the relevant data based on the node mapping provided by the index. Observers are thus able to obtain complete query answers since observing communities requires far less resources than participating.

5 CONSENSUAL UPDATES

In this section, we outline how ColChain enables consensual updates. We provide a formalize updates to fragments and discuss our approach for reaching consensus on proposed updates.

5.1 Updates

In ColChain, there are two types of updates that need to be accounted for; updates to fragments (for participants) and updates to the metadata (e.g., when a node leaves a community). ColChain represents updates to both fragments or metadata as *transactions* that consist of *operations*. To provide access to previous dataset versions, ColChain organizes and stores the transactions done to a fragment over time as a chain. However, since providing access to previous network configurations (e.g., former community members) is out of the scope of this paper, ColChain does not keep the transactions done to the metadata. For ease of presentation, we therefore focus on updates to fragments in this section. Operations can be insertions or deletions and thus updates are in this paper described as deletions followed by insertions. Formally, an operation is defined over a fragment as follows.

Definition 14 (Operation). *An operation o over a fragment f is a tuple $o = (\alpha, t)$, where α describes whether the operation is an insertion ($\alpha = +$) or deletion ($\alpha = -$) of a triple, and t is the triple that is to be inserted or deleted.*

Applying an operation o to a fragment f, denoted $o(f)$, results in a state where the fragment either does not contain the triple (if $\alpha = -$), or contains the triple (if $\alpha = +$). An operation thus represents a change to a fragment in the local datastore. As operations typically occur in sets, we combine updates into *transactions*. Transactions are defined as follows.

Definition 15 (Transaction). *A transaction γ is a triple $\gamma = (O, f_i, \lambda)$ where $O = \{o_1, o_2, \ldots, o_n\}$ is a set of operations, f_i is a fragment, and $\lambda = (u_n, \alpha_\gamma, \iota)$ is provenance information, where u_n is the unique identifier of the node proposing the transaction n, α_γ is a signature obtained using n's private key, and ι is a timestamp.*

Note that while blocks in conventional blockchains [27] have a fixed size, transactions in ColChain can have varying sizes, i.e., any number of operations. This allows for more efficient storage of updates in situations where updates to knowledge graphs contain different numbers of operations.

Applying all operations in a transaction γ to a fragment f is denoted by $[[f]]_\gamma$. Notice also that a transaction can be applied to the metadata triples. This is denoted $[[\mathcal{M}]]_\gamma$.

Table 1: Fragment f

Fragment f		
$\langle \mathbf{a}, \mathbf{p_1}, \mathbf{c} \rangle$	$\langle a, p_2, d \rangle$	$\langle b, p_2, d \rangle$
$\langle b, p_3, e \rangle$	$\langle c, p_1, d \rangle$	$\langle c, p_3, a \rangle$
$\langle f, p_4, d \rangle$	$\langle d, p_4, f \rangle$	$\langle e, p_5, g \rangle$

Consider fragment f in Table 1 and the following transaction:

$$\gamma = (\{(+, \langle a, p_1, b \rangle), (-, \langle a, p_1, c \rangle)\}, f, \lambda)$$

The result of applying the transaction γ to the fragment f, $[[f]]_\gamma$, is the fragment that exchanges the triple $\langle a, p_1, c \rangle$ with $\langle a, p_1, b \rangle$, and can be seen in Table 2.

Table 2: The result after applying γ to f, $[[f]]_\gamma$

Fragment $[[f]]_\gamma$		
$\langle \mathbf{a}, \mathbf{p_1}, \mathbf{b} \rangle$	$\langle a, p_2, d \rangle$	$\langle b, p_2, d \rangle$
$\langle b, p_3, e \rangle$	$\langle c, p_1, d \rangle$	$\langle c, p_3, a \rangle$
$\langle f, p_4, d \rangle$	$\langle d, p_4, f \rangle$	$\langle e, p_5, g \rangle$

State transition functions describe how ColChain allows for fragment updates. Transition functions are defined for fragment updates (on participants) as well as fragment slice updates (on both participants and observers). These functions are, upon achieved consensus (described in Section 5.2), triggered on participants and observers. First we define the transition function for the fragments in a state as follows.

Definition 16 (Fragment Transition Function). *Given a transaction γ, a state $\mathcal{S}_n$ of a node n, and a secure hash function H, the* fragment transition function *is for all $\sigma_i = (\mathcal{X}_i, f_i) \in \mathcal{S}_n.\Sigma$ defined as follows.*

$$\tau_\Sigma(\sigma_i, \gamma) = (\tau_\mathcal{X}(\mathcal{X}_i, \gamma), \tau_f(f_i, \gamma)) \tag{2}$$

Where

$$\begin{aligned} \tau_\mathcal{X}(\mathcal{X}_i, \gamma) = \mathcal{X}_i \cup \{x_{i+1}\} \; s.t. \\ x_{i+1} = (h, \gamma) \\ h = (H(\gamma), y) \\ y = \{H(x_j) \mid j < i\} \end{aligned} \tag{3}$$

And

$$\tau_f(f_i, \gamma) = [[f_i]]_\gamma \tag{4}$$

The value h in Equation 3 can be seen as a *header* that contains a hash value of the transaction and links to previous headers (and thus transactions), forming a chain. Since the metadata $\mathcal{M}$ is a set of triples, applying a transaction γ to $\mathcal{M}$ is given by applying $\tau_f(\mathcal{M}, \gamma)$ (τ_f as defined in Equation 4) on the nodes within the community. Given a transaction γ and a state $\mathcal{S}_n$ of a node n, we therefore define a *slice transition function* as follows.

Definition 17 (Slice Transition Function). *Given a transaction $\gamma = (O, f_i, \lambda)$ and a state $\mathcal{S}_n$ of a node n such that $s_{f_i} \in \mathcal{S}_n.S$, the* slice transition function *is defined as follows.*

$$\tau_S(s_{f_i}, \gamma) = s([[f_i]]_\gamma) \tag{5}$$

After a transaction is applied to a fragment, the participants compute a new index slice for the fragment using the slice transition function. Afterwards, participants and observers replace the slice of that fragment with the new slice in their distributed indexes.

5.2 Consensus Protocol

To obtain a consensus on a proposed update, ColChain relies on the participants to actively vote on a transaction. This means that a transaction is not applied until a majority of participants accepts. While this could take some time (since users are active at different times), it prevents faulty or malicious updates from being applied. Furthermore, the data provider can enforce updates to the fragment (i.e., circumvent the majority).

Consensus is reached by a majority of nodes accepting the transaction. The *acceptance* protocol of a participant is defined as follows. Let $verify(\alpha_\gamma, \rho_f)$ be a function that returns `true` iff the signature α_γ matches the public key ρ_f, i.e., if γ was signed by the owner of f, and $validate(\gamma)$ be a function that returns `true` iff the user actively accepts the transaction. The acceptance protocol uses the following steps:

(1) Let $\gamma = (O, f, \alpha_\gamma)$, $\mathcal{M}$ be the set of metadata triples of n, $\mathcal{S}_n.\mathcal{M}$, and $\sigma_i = (\mathcal{X}_i, f_i) \in \mathcal{S}_n.\Sigma$ be the unique state entry such that $u_{f_i} = u_f$. Find $\langle u_f, \texttt{cc} : \texttt{author}, \rho_f \rangle \in \mathcal{M}$.
(2) If $verify(\alpha_\gamma, \rho_f) = \texttt{true}$, accept γ on σ_i.
(3) If $validate(\gamma) = \texttt{true}$, accept γ on σ_i.
(4) If $verify(\alpha_\gamma, \rho_f) = \texttt{false}$ and $validate(\gamma) = \texttt{false}$, reject γ on σ_i.

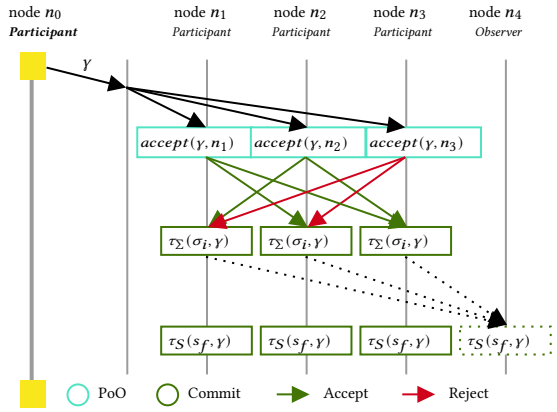


Figure 3: Example flowchart of transaction proposition, validation and application.

In a ColChain network, similarly as in [6], transactions are propagated through the community as shown in Figure 3. First, a node proposes a candidate transaction and calls the acceptance method on each participant in the community.[6] Then, each participant validates the transaction and forwards the result of $accept(\gamma, n)$ to all the other participants. By receiving the acceptance messages from all other participants, nodes are able to, at all times, locally determine when a majority has accepted. In this example, two out of three participants accept the transaction and thus a consensus is reached. After reaching consensus, each participant triggers its

[6] community participants and their addresses are available in the node's metadata

state transition functions with the proposed transaction, thereby committing the transaction. Last, the participant that proposed the update forwards the updated index slice to the observers which will, upon receiving an accept message from a majority of nodes, update their index accordingly. This way, changes to the fragments within the community can be collaboratively determined.

Relying on a majority of the users to actively accept transactions creates a practical limitation on the number and size of transactions a user can vote on in reality. This limitation could be addressed by using different consensus protocols in different situations that make active participation of users more scalable, e.g., by detecting non-trivial changes in large updates or malicious updates. This is part of our future work (Section 8).

6 QUERY PROCESSING

In line with the state of the art [4, 5], a ColChain node processes SPARQL queries by requesting reachable fragments that match the query's triple patterns, while operations like joins are evaluated locally on the querying node. In line with [4, 16] we attach the bindings from previously evaluated triple patterns to the requests when evaluating subsequent triple patterns. This is to reduce the cardinality of the retrieved fragment, i.e., sizes of intermediate results. Furthermore, by using the node's distributed index (as defined in Definition 5), it is possible to limit the number of nodes to query for each triple pattern and avoid having to flood the network, i.e., forwarding requests through several layers of neighbors, which is the basic strategy in unstructured P2P networks [4]. Similarly to [4], a ColChain node has a limited view over the entire network. However, differently from [4], this view is not random but determined by the communities the node has joined.

Recall in Definition 6 the function $match(P, I)$ for a BGP P and index I. The function returns a node mapping M that, given a triple pattern tp, $M(tp)$, returns the set of nodes N with relevant fragments, i.e., fragments that tp matches. Each node n contains a *selector* function that formalizes how triple pattern requests are processed over an entry in n's local datastore. Since ColChain bulks bindings from previously evaluated triple patterns, we define a selector function in line with [16] that takes these bindings into account. Furthermore, to allow for queries over previous dataset versions, we extend the selector function to also take into account a timestamp ι. Let $\sigma_i = (\mathcal{X}_i, f_i)$ be an entry in n's local datastore. $\iota(\sigma_i)$ denotes the fragment obtained by reversing the operations in the transactions encoded within $\mathcal{X}_i$ with a timestamp greater than or equal to ι, i.e., γ_j with $\gamma_j.\lambda.\iota \geq \iota$, i.e., the fragment that was available at ι. The selector function is defined as follows.

Definition 18 (Selector Function). *Given a node n, a triple pattern tp, a finite set of distinct solution mappings Ω, and a timestamp ι, the fragment-based bindings-restricted triple pattern selector for tp, Ω, and ι, denoted $s_{(tp,\Omega,\iota)}$, is for every entry σ in n's local datastore defined as follows.*

$$s_{(tp,\Omega,\iota)}(\sigma) = \begin{cases} \{t \in \iota(\sigma) \mid t \in \iota(\sigma) \wedge \exists \mu : t = \mu[tp] & \text{if } \Omega = \emptyset \\ \{t \in \iota(\sigma) \mid t \in \iota(\sigma) \wedge \exists \mu : t = \mu[tp] \wedge \\ \qquad \exists \mu' \in \Omega : \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

In line with the state of the art, we apply pagination [16, 35], i.e., we group the results into reasonably sized pages (e.g., 100 results per page in [35]) to avoid excessive data transfer. For ease of presentation though, we focus on the case where all answers fit into a single page. Given a node n, a triple pattern tp, a set of solution mappings Ω, and a timestamp ι, we define two additional functions that call the selector function (Definition 18) to process triple pattern requests: $n^c(tp, \Omega, \iota)$ and $n^p(tp, \Omega, \iota)$.

- $n^c(tp, \Omega, \iota)$ returns a cardinality estimation of the result of invoking $s_{(tp,\Omega,\iota)}$ on n.
- $n^p(tp, \Omega, \iota)$ returns the result of invoking $s_{(tp,\Omega,\iota)}$ on n.

Note that in some cases nodes will process triple patterns over fragments available in the local datastore. In this case, the returned node n from the node mapping is the local node, and the node does not actually perform a request to itself; rather it just processes the triple pattern locally and forwards the result to the query processor.

Algorithm 1 Evaluate a BGP on a ColChain node

Input: A BGP $P = \{tp_1, \ldots, tp_n\}$; a node n; a timestamp ι; a set of solution mappings Ω; a node mapping M
Output: A set of solution mappings

1: **function** *evaluateBGP*(P,n,ι,$\Omega = \emptyset$,$M = \emptyset$)
2: | **if** $M = \emptyset$ **then** $M \leftarrow match(P, \mathcal{S}_n.I_n)$;
3: | **for all** $tp_i \in P$ **do**
4: | | $cnt_i \leftarrow \Sigma_{n_1 \in M(tp_i)} n_1^c(tp_i, \Omega, \iota)$;
5: | | **if** $cnt_i = 0$ **then return** $\emptyset$;
6: | $tp_\epsilon \leftarrow tp_k$ where $tp_k \in P$ and $cnt_k \leq cnt_j \forall tp_j \in P$;
7: | $\phi^\epsilon \leftarrow \bigcup_{n_1 \in M(tp_\epsilon)} n_1^p(tp_\epsilon, \Omega, \iota)$;
8: | $\Omega^\epsilon \leftarrow \Omega \bowtie \{\mu \mid dom(\mu) = vars(tp_\epsilon)$ and $\mu[tp_\epsilon] \in \phi^\epsilon\}$;
9: | $P' \leftarrow P \setminus \{tp_\epsilon\}$;
10: | **if** $P' = \emptyset$ **then return** Ω^ϵ;
11: | **return** *evaluateBGP*($P', n, \iota, \Omega^\epsilon, M$);

Given a BGP P and timestamp ι specifying which fragment versions to process P over, Algorithm 1 defines a recursive algorithm to process P over the fragments reachable for a node n (i.e., covered by n's index). First, we obtain the node mapping (Section 3, Definition 6) from n's index (Section 3, Definition 5) in line 2. Recall that a node mapping given a triple pattern tp, $M(tp)$, returns a set of nodes that contain relevant fragments to tp. Using the previously defined function $n^p(tp, \Omega, \iota)$, we then obtain a cardinality estimation for each triple pattern by summing the cardinality estimations obtained by each node specified by the node mapping we found in the previous step (lines 3-5). The triple pattern with lowest cardinality estimation is then evaluated first by sending it to all sources specified by the node mapping to obtain the bindings for this triple pattern (line 7). The output results from previous triple patterns are then joined with the bindings just obtained (line 8). Last, the evaluated triple pattern is removed from the BGP (line 9) and, if there are any remaining triple patterns, the algorithm is called recursively with the new BGP and set of bindings (line 11).

7 EXPERIMENTAL EVALUATION

We compare ColChain with state-of-the-art decentralized architectures for sharing and querying semantic data in terms of performance. Moreover, we investigate the impact of community sizes on ColChain's performance and updates and study ColChain's performance when processing queries against previous versions.

7.1 Implementation Details

A prototype of ColChain was implemented in Java 8[7]. Both the Web interface and node interface (Figure 2) are implemented as Java 8 Servlets using Jetty[8]. The query processor is implemented as an extension to Apache Jena[9] and can process SPARQL queries that Jena can process (e.g., with `UNION` and `OPTIONAL`). Our implementation uses Prefix-Partitioned Bloom Filter indexes from [5] as distributed index (I_n) and the predicate-based fragmentation function from [4]. The fragments themselves are stored as separate HDT files [14] that allow for efficient processing of triple patterns. The chains of transactions are stored separately from the fragments and if possible in main memory.

7.2 Experimental Setup

Datasets and Queries. We use the datasets and queries from the extended LargeRDFBench [17] benchmark, which comprises 13 interconnected datasets that totals over 1 billion triples. LargeRDFBench includes 40 SPARQL queries divided into 5 different categories: Simple (S), Complex (C), Large Data (L), and Complex and Large Data (CH). We generated chains of 1, 10, and 100 transactions for each fragment. Each transaction has a random number of operations between 10 and 100. We assessed the correctness of query answers by comparing the output to the results obtained using the same data and queries in a standard triple store. In our experiments, all queries that finished returned complete results.

Experimental Setup. We ran our experiments on a network with 128 nodes. We created 4004 fragments from the LargeRDFBench data in total and 200 communities each with between 10 and 31 fragments (randomly assigned). Nodes are assigned randomly to participate and observe each of the communities. The nodes observe all communities they do not participate in, i.e., all the fragments are reachable from all the nodes, and therefore complete query results can be obtained during query processing. We compare the following systems: (i) an unstructured P2P network with flooding [4] (Piqnic), (ii) the same system with decentralized indexes [5] (Piqnic$_{PPBF}$), and (iii) ColChain using the same indexes as Piqnic$_{PPBF}$. In Piqnic, fragments are replicated randomly throughout the network and connections are random. We match the replication factor in Piqnic and Piqnic$_{PPBF}$ (i.e., the number of replicas per fragment) with the number of participants in each community for ColChain. For each experiment, we ran the 40 queries in the benchmark sequentially at three random nodes and report the average over the three runs.

Parameters. To test how the characteristics of a ColChain network affect performance and update overhead time, we ran experiments with 5, 10, and 20 participants per community. Moreover, to assess ColChain in a more realistic setup with varying community sizes, we ran experiments using a Zipfian distribution of the community sizes, i.e., the most popular community has all 128 nodes as participants, after which the nth most popular community has $128/n$, or at least one, participants. In the versioning experiments, we assessed how scalable ColChain is for different chain lengths (i.e., the number of transactions to roll back for a single query) when processing queries over previous versions. Specifically, we processed queries with chain lengths of 1, 10, and 100.

Hardware Configuration. We ran, for each P2P system, 128 nodes on a virtual machine (VM) with 128 vCPU cores with a clock speed of 2.5GHz, 64KB L1 cache, 512KB L2 cache, 8192KB L3 cache and 2TB main memory. Since all clients are run concurrently on the same machine, they are limited to use 1 core and 15GB memory, and the connection between them is limited to 20 MBit/sec.

Evaluation Metrics. We measure the following metrics:

- *Query Execution Time (QET)*: The amount of time (in milliseconds) it takes to obtain the full query answer.
- *Query Response Time (QRT)*: The amount of time (in milliseconds) it takes to obtain the first query answer.
- *Update Overhead Time (UOT)*: The amount of time (in milliseconds) elapsed from when an update is proposed to when it is committed on all targets (assuming a consensus is reached instantly).
- *Version Materialization Time (VMT)*: The amount of time (in milliseconds) it takes to materialize previous versions of requested fragments.
- *Number of Exchanged Messages (NEM)*: The number of messages exchanged between nodes.
- *Number of Transferred Bytes (NTB)*: The amount of data transferred (in bytes) between nodes.

Software Configuration. The number of results that each system is allowed to attach to requests is set to $|\Omega| = 30$ (Section 6), and the page size to 100 (i.e., each system can only send 100 results at a time). The timeout is set to 1,200 seconds (20 minutes). For Piqnic, we use the following values (as recommended by [4]). Time-to-Live (number of hops): 5, Number of Neighbors: 5.

7.3 Experimental Results

Due to space limitations, we will only show the most interesting experimental results in this section. The full experimental results are available at https://relweb.cs.aau.dk/colchain.

Query Processing Performance. Figure 4 shows the query execution time (QET) for each system over the queries in the C and CH query categories. ColChain has similar performance to Piqnic$_{PPBF}$. This is due to the fact that the characteristics of a ColChain network are similar to that of a Piqnic network; in our setup, the number of participants in a community was matched to the replication factor in Piqnic. As such, since ColChain and Piqnic$_{PPBF}$ use the same indexing scheme, they have similar overall performance. Only a few queries, such as C3 and CH7, show a slight difference in execution time for ColChain and Piqnic$_{PPBF}$. This is due to the particular topology of the networks and that the randomly chosen nodes have different fragments available in the local datastore, e.g., for CH7, one of the Piqnic$_{PPBF}$ nodes was able execute the query issuing just 2,058 triple pattern requests, while all the ColChain nodes required at least 2,423 triple pattern requests to gather the fragments that were not available locally at the issuing node. The decerease in execution time this caused the particular node to have was significant enough to still be visible even with the value averaged over three separate nodes. Furthermore, ColChain and Piqnic$_{PPBF}$ have better query processing performance across

[7] The prototype is available at https://github.com/ColChain/ColChain-Java.
[8] https://www.eclipse.org/jetty/
[9] https://jena.apache.org/

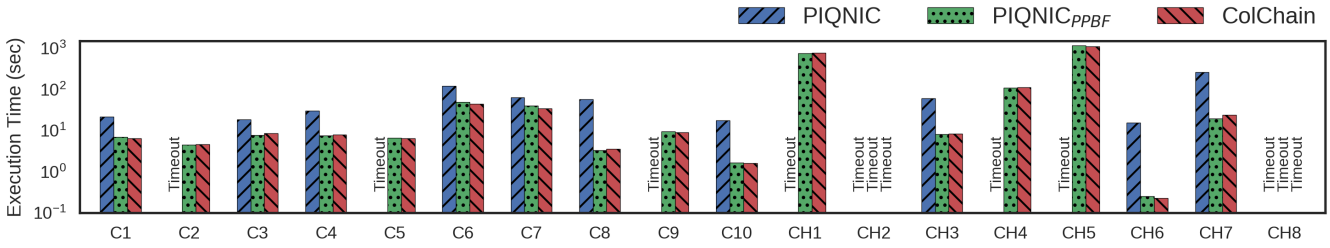
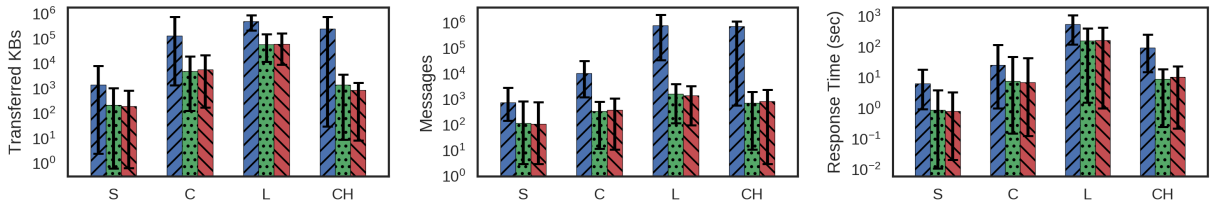


Figure 4: Execution time for queries in the C and CH categories for PIQNIC, PIQNIC$_{PPBF}$, and COLCHAIN *(y-axis in log scale).*



(a) Transferred KBs **(b) Messages** **(c) Response Time**

Figure 5: Number of transferred bytes (NTB) in KBs, number of exchanged messages (NEM), and response time (QRT) for each approach over the different query categories excluding queries that timed out *(y-axis in log scale).*

all queries compared to PIQNIC. This is consistent across not just the C and CH query categories, but all remaining query categories in our experiments as well. COLCHAIN and PIQNIC$_{PPBF}$ timed out for 5 out of the 40 queries, and PIQNIC timed out for 14 out of the 40 queries. The timeouts were in the L and CH query categories and were queries for which even federated systems timed out [17].

Figure 5 shows, for each query category and over each compared system, the number of transferred bytes (Figure 5a), number of exchanges messages (Figure 5b), and the query response time (Figure 5c). Both COLCHAIN and PIQNIC$_{PPBF}$ generate significantly less network load than PIQNIC. This is due to the ability to accurately determine which other nodes to query for specific data, thus removing the need to flood the network. A similar trend can be observed in Figure 5b where COLCHAIN and PIQNIC$_{PPBF}$ incur significantly fewer exchanged messages than PIQNIC. As a result, COLCHAIN is able to maintain a comparatively low response time similar to PIQNIC$_{PPBF}$'s (Figure 5c).

Overall, our results clearly show that COLCHAIN's performance is comparable to state-of-the-art systems without incurring in any additional network load. This was expected since improving query processing performance was not part of our contributions, and emphasizes that the additional functionality of consensual updates and query processing over previous versions does not decrease the performance when comparing to state-of-the-art systems.

Impact of the Size of the Communities. Figure 6 shows the query execution time (QET) for each query in the S query category over each community size. It is clear that in most cases the networks with larger community sizes show slightly better performance overall. This is due to the fact that communities with larger numbers of participants have more fragments replicated at each node, and therefore executing a query naturally incurs in a lower number of requests. In our experiments, the network with five participants per community has between 47 and 332 fragments per node, while the network with 20 participants per community has between 408 and 885. There were a few exceptions, e.g., queries S2 and S5, but this comes naturally from the fact that the additional replicas were placed randomly and were not necessarily located at the node issuing the query. In all cases, the network with five participants in each community is the slowest, and for some queries this margin is quite significant (e.g., up to 7 times slower for S9 compared to the setup with 20 participants in each community). These results are consistent across all query categories and show that there is a correlation between the sizes of the communities and the performance. Finally, we have also considered a configuration where the number of the participants per community varies according to a Zipfian distribution. For this configuration, the results show that query processing is less efficient than in the setup where all communities have 10 or 20 participants. This is the case since in a Zipfian distribution there will on average be fewer participants per communities, resulting in fewer fragments locally available on each node. In fact, in our experiments, nodes have between 19 and 258 fragments available, which is significantly less than in any other setup. Still, the performance is similar to the setup with 5 participants. Moreover, for a few queries such as S1 and S9, the performance is close to the setup with 20 participants. This is a result of many of those fragments being part of a popular community, i.e., there is a high likelihood of such fragments being available in the local datastore of the node issuing the query.

Cost of Processing Queries over Previous Dataset Versions. In order to study COLCHAIN's performance when processing queries against previous dataset versions, we created update chains with 1, 10, and 100 transactions per fragment. We applied these updates to the fragments and saved only the latest version in the local datastore. All queries were evaluated over the initial version of the datasets by rolling back the entire chain and computing the initial version during query processing. Figure 7 shows, for each query

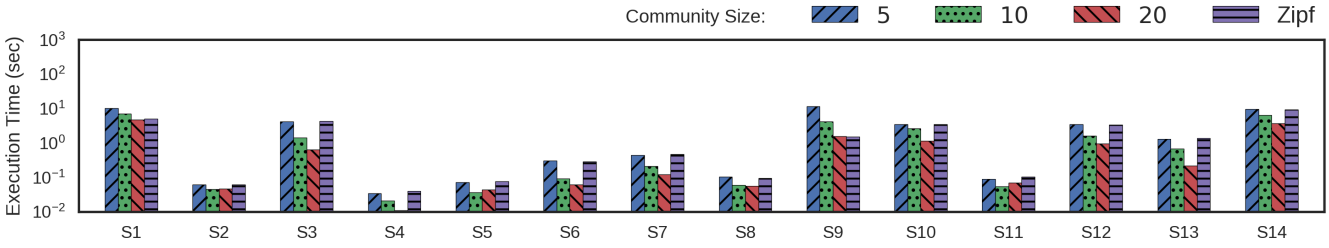
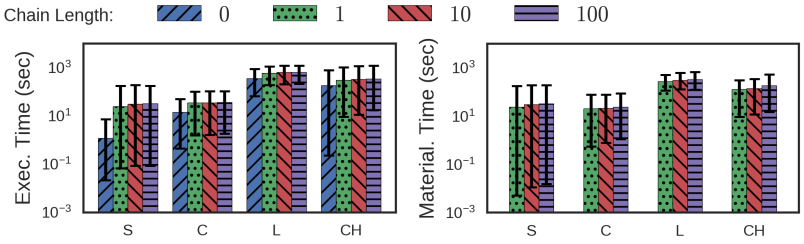


Figure 6: Execution time for queries in the S query category for different community sizes *(y-axis in log scale).*



(a) Execution Time **(b) Materialization Time**

Figure 7: Execution and materialization time for the versioning experiments excluding queries that timed out *(y-axis in log scale).*

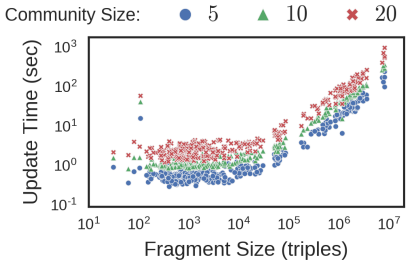


Figure 8: Update Overhead Time over different community sizes *(log scale).*

category the query execution time (Figure 7a) and the version materialization time (Figure 7b). Our experiments show that while processing queries over a previous dataset version clearly takes more time, the length of the chain (i.e., number of transactions to roll back) actually has very little impact. This is a sub-linear effect since it is not necessary to materialize each version in between the current version and the target version; rather it is sufficient to simply combine all the operations in the transactions and materialize the target version by rolling back those operations on the current version. Moreover, it is clearly visible that the materialization time (Figure 7b) does not significantly vary with the number of transactions; instead, it is mostly impacted by the query category. One of the main differences between the different query categories is the number of transferred bytes. In particular, while the S query category requires at most the transfer of 856 kilobytes, the L query category requires at most the transfer of 154 megabytes. Clearly, the materialization time is the main component of the query execution time (cf. Figure 7a and Figure 7b), where up to 94% of the materialization time is spent in decompressing the fragment triples and compressing the triples in the fragment's target version. This suggests that while HDT favors efficient processing of triple pattern requests, using this encoding in the context of writable linked data has clear limitations. Additionally, our experimental setup with fragments of up to 10 million triples clearly exacerbates these limitations. Nevertheless, our experiments still show that it is possible to process queries over previous dataset versions within a relatively low query timeout, i.e., 31 out of 40 queries were executed within a timeout of 20 minutes; all timed out queries are either queries in the L and CH query groups, resulting in larger fragments to be materialized and therefore longer materialization time, or including triple patterns with variables as predicates, resulting in a large number of fragments to be materialized. Notice that these queries time out even when processed by state-of-the-art federated systems [17].

Update Overhead. To assess how consensual updates in ColChain scale in relation to the size of the community and the size of the fragments, we ran experiments where we applied updates to 338 randomly chosen fragments (out of the 4004 fragments in the network) of varying sizes in communities of different sizes. We measured the update overhead time (UOT), including the time it takes for nodes within the community to forward acceptance messages. In this experiment, the updates are performed by a random non-owner node, and we assume that the participants in the community instantly accept the transactions. Figure 8 shows the update time for different transactions applied to the 338 randomly chosen fragments. While the number of messages sent between nodes should rise polynomially with the community size (from 20 for the communities with five participants to 380 for communities with 20 participants), this has relatively little impact on the overhead of the updates. In line with the versioning experiments, the materialization of the updated file represents the biggest overhead. This highlights the limitations with updates that HDT entails.

7.4 Summary

Our experimental evaluations clearly show that ColChain is able to not only support consensual updates and query processing over previous dataset versions, but achieves comparable query processing performance to state-of-the-art decentralized architectures. The overhead of processing queries over previous dataset versions is mostly affected by the materialization of the older fragment itself, and ColChain is able to handle increasing chain lengths relatively efficiently in comparison. Furthermore, the overhead of consensual updates to fragments is also mostly affected by the materialization of the new version rather than the validation protocols. This highlights that the used RDF encoding has an impact on the performance with regards to versioning and consensual updates, and that a different encoding that allows for efficient updates could

be more suitable for ColChain. Overall, our experimental evaluation clearly shows that ColChain is able to efficiently handle consensual updates, provides an approach to processing queries over previous versions that is scalable in relation to the length of the chain, and achieves good query processing performance in comparison to state-of-the-art decentralized architectures.

8 CONCLUSION

In this paper, we presented ColChain (COLlaborative knowledge CHAINs), a novel decentralized system that allows for users to provide updates to published knowledge graphs while enabling querying previous dataset versions. ColChain divides unstructured Peer-to-Peer networks into smaller communities of nodes and applies community-based chains of updates to data fragments. By relying on the consensus of participating nodes in a community, ColChain is able to let nodes propose and vote on updates to the fragments. Furthermore, by structuring changes to fragments over time as chains, ColChain allows nodes to roll back updates and obtain query answers over previous dataset versions. Our experimental evaluation shows that ColChain achieves comparable performance to state-of-the-art interfaces, while the overhead of processing queries over previous dataset versions is mostly affected by the materialization of the older fragment and the used RDF encoding rather than the validation protocol. As part of our future work, we will investigate how to include measures to detect and avoid malicious activity. Furthermore, we will assess whether sharing the query processing load between nodes could improve query processing performance in ColChain and assess alternative RDF compression techniques to improve efficiency when processing queries over previous versions. We also plan to explore the possibility of using different consensus protocols that make active participation of users more scalable, e.g., by letting fragment owners specify a qualified majority or detecting malicious updates automatically. This could be done by expanding the metadata triples and adding routines for alternative consensus strategies.

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

REFERENCES

[1] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. 2017. Lusail: A System for Querying Linked Data at Scale. *Proc. VLDB Endow.* 11, 4 (2017), 485–498.
[2] Maribel Acosta, Amrapali Zaveri, Elena Simperl, Dimitris Kontokostas, Sören Auer, and Jens Lehmann. 2013. Crowdsourcing Linked Data Quality Assessment. In *ISWC 2013*. 260–276.
[3] Christian Aebeloe, Ilkcan Keles, Gabriela Montoya, and Katja Hose. 2020. Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns. *CoRR* abs/2002.09172 (2020). https://arxiv.org/abs/2002.09172
[4] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2019. A Decentralized Architecture for Sharing and Querying Semantic Data. In *ESWC 2019*. 3–18.
[5] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2019. Decentralized Indexing over a Network of RDF Peers. In *ISWC 2019*. 3–20.
[6] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: A Cross-Application Permissioned Blockchain. *VLDB* 12, 11 (2019), 1385–1398.
[7] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL Web-Querying Infrastructure: Ready for Action?. In *ISWC 2013*. 277–293.
[8] Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. 2020. SMART-KG: Hybrid Shipping for SPARQL Querying on the Web. In *WWW 2020*. 984–994.
[9] Amr Azzam, Christian Aebeloe Ilkcan Keles, Gabriela Montoya, Axel Polleres, and Katja Hose. 2021. WiseKG: Balanced Access to Web Knowledge Graphs. In *WWW 2021*.
[10] Min Cai and Martin R. Frank. 2004. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW*. 650–657.
[11] Arturo Crespo and Hector Garcia-Molina. 2002. Routing Indices For Peer-to-Peer Systems. In *ICDCS 2002*. 23–32.
[12] Michel Dumontier, Alison Callahan, Jose Cruz-Toledo, Peter Ansell, Vincent Emonet, François Belleau, and Arnaud Droit. 2014. Bio2RDF Release 3: A larger, more connected network of Linked Data for the Life Sciences. In *ISWC 2014 Posters & Demonstrations Track*, Vol. 1272. 401–404.
[13] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB - A Shared Database on Blockchains. *PVLDB* 12, 11 (2019), 1597–1609.
[14] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF representation for publication and exchange (HDT). *J. Web Semant.* 19 (2013), 22–41.
[15] Damien Graux, Gezim Sejdiu, Hajira Jabeen, Jens Lehmann, Danning Sui, Dominik Muhs, and Johannes Pfeffer. 2018. Profiting from Kitties on Ethereum: Leveraging Blockchain RDF with SANSA. In *ISWC Posters and Demos*.
[16] Olaf Hartig and Carlos Buil Aranda. 2016. Bindings-Restricted Triple Pattern Fragments. In *OTM 2016 Conferences*. 762–779.
[17] Ali Hasnain, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, and Dietrich Rebholz-Schuhmann. 2018. Extending LargeRDFBench for Multi-Source Data at Scale for SPARQL Endpoint Federation. In *SSWS@ISWC*. 28–44.
[18] Lars Heling, Maribel Acosta, Maria Maleshkova, and York Sure-Vetter. 2018. Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study. In *ISWC 2018*. 86–102.
[19] Luis Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Olivier Corby. 2014. ColGraph: Towards Writable and Scalable Linked Open Data. In *ISWC 2014*. 325–340.
[20] Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Iris Miliaraki, Matoula Magiridou, and Antonios Papadakis-Pesaresi. 2010. Atlas: Storing, updating and querying RDF(S) data on top of DHTs. *J. Web Sem.* 8, 4 (2010).
[21] Marcel Karnstedt, Kai-Uwe Sattler, Martin Richtarsky, Jessica Müller, Manfred Hauswirth, Roman Schmidt, and Renault John. 2007. UniStore: Querying a DHT-based Universal Storage. In *ICDE 2007*. 1503–1504.
[22] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. A Demonstration of the Solid Platform for Social Web Applications. In *WWW 2016 Posters and Demos*. 223–226.
[23] Steffen Metzger, Katja Hose, and Ralf Schenkel. 2012. Colledge: a vision of collaborative knowledge networks. In *SSW 2012*. 1–8.
[24] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. 2019. SaGe: Web Preemption for Public SPARQL Query Services. In *WWW*. 1268–1278.
[25] Gabriela Montoya, Christian Aebeloe, and Katja Hose. 2018. Towards Efficient Query Processing over Heterogeneous RDF Interfaces. In *DeSemWeb@ISWC*.
[26] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. 2017. The Odyssey Approach for Optimizing Federated SPARQL Queries. In *ISWC 2017*. 471–489.
[27] Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf
[28] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* (1978), 120–126.
[29] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: The Linked SPARQL Queries Dataset. In *ISWC 2015*. 261–269.
[30] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. 2014. HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation. In *ESWC 2014*. 176–191.
[31] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. FedX: A Federation Layer for Distributed Query Processing on Linked Open Data. In *ESWC 2011*. 481–486.
[32] Mirek Sopek, Przemyslaw Gradzki, Witold Kosowski, Dominik Kuzinski, Rafal Trójczak, and Robert Trypuz. 2018. GraphChain: A Distributed Database with Explicit Semantics and Chained RDF Graphs. In *WWW Companion*. 1171–1178.
[33] Claus Stadler, Jens Lehmann, Konrad Höffner, and Sören Auer. 2012. LinkedGeoData: A core for a web of spatial open data. *Semantic Web* 3, 4 (2012), 333–354.
[34] Pierre-Yves Vandenbussche, Jürgen Umbrich, Luca Matteis, Aidan Hogan, and Carlos Buil Aranda. 2017. SPARQLES: Monitoring public SPARQL endpoints. *Semantic Web* 8, 6 (2017), 1049–1065.
[35] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *J. Web Sem.* 37-38 (2016), 184–206.
[36] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.
[37] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain challenges and opportunities: a survey. *IJWGS* 14, 4 (2018), 352–375.