VeriDKG

A Verifiable SPARQL Query Engine for Decentralized Knowledge Graphs

Zhou, Enyuan; Guo, Song; Hong, Zicong; Jensen, Christian S.; Xiao, Yang; Zhang, Dalin; Liang, Jinwen; Pei, Qingqi

Published in:
Proceedings of the VLDB Endowment

DOI (link to publication from Publisher):
10.14778/3636218.3636242

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

VeriDKG: A Verifiable SPARQL Query Engine for Decentralized Knowledge Graphs

Enyuan Zhou
Hong Kong Polytechnic University
en-yuan.zhou@connect.polyu.hk

Song Guo*
Hong Kong University of Science and Technology
songguo@cse.ust.hk

Zicong Hong†
Hong Kong Polytechnic University
zicong.hong@connect.polyu.hk

Christian S. Jensen†
Aalborg University
csj@cs.aau.dk

Yang Xiao†
Xidian University
yxiao@xidian.edu.cn

Dalin Zhang*
Aalborg University
dalinz@cs.aau.dk

Jinwen Liang
Hong Kong Polytechnic University
jinwen.liang@polyu.edu.hk

Qingqi Pei‡
Xidian University
qqpei@mail.xidian.edu.cn

ABSTRACT

The ability to decentralize knowledge graphs (KG) is important to exploit the full potential of the Semantic Web and realize the Web 3.0 vision. However, decentralization also renders KGs more prone to attacks with adverse effects on *data integrity* and *query verifiability*. While existing studies focus on ensuring data integrity, how to ensure query verifiability - thus guarding against incorrect, incomplete, or outdated query results - remains unsolved. We propose VeriDKG, the first SPARQL query engine for decentralized knowledge graphs (DKG) that offers both data integrity and query verifiability guarantees. The core of VeriDKG is the RGB-Trie, a new blockchain-maintained authenticated data structure (ADS) facilitating correctness proofs for SPARQL query results. VeriDKG enables verifiability of subqueries by gathering global index information on subgraphs using the RGB-Trie, which is implemented as a new variant of the Merkle prefix tree with an RGB color model. To enable verifiability of the final query result, the RGB-Trie is integrated with a cryptographic accumulator to support verifiable aggregation operations. A rigorous analysis of query verifiability in VeriDKG is presented, along with evidence from an extensive experimental study demonstrating its state-of-the-art query performance on the largeRDFbench benchmark.

PVLDB Reference Format:
Enyuan Zhou, Song Guo, Zicong Hong, Christian S. Jensen, Yang Xiao, Dalin Zhang, Jinwen Liang, and Qingqi Pei. VeriDKG: A Verifiable SPARQL Query Engine for Decentralized Knowledge Graphs. PVLDB, 17(4): 912 - 925, 2023.
doi:10.14778/3636218.3636242

*Corresponding authors.
†Three authors contributed equally to this research.
‡The other affiliation of this person is Guangzhou Lianrong Information Technology Co. Ltd.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636242

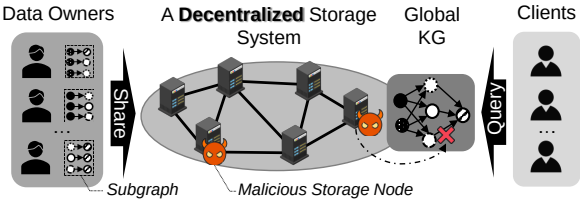


Figure 1: Illustration of a DKG system.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/garlicZhou/veriDKG-RGB-Trie.

1 INTRODUCTION

The Web 3.0 [23, 31, 40] envisions a future Web where the Semantic Web and the Web of Data play increasingly important roles [38, 54, 60, 67]. For the Semantic Web to meet the expectations, it is desirable, or necessary, to be able to support a decentralized knowledge graph (DKG) [1, 3, 8, 69]. For example, DBpedia[1] and Wikidata[2] provide free knowledge base with 9500 and 100 million linked data items contributed by communities of volunteers, empowering diverse applications from research to recommendation systems. As illustrated in Figure 1, a DKG is stored, managed, and queried using a decentralized infrastructure that facilitates the storage of subgraphs at multiple storage nodes. This infrastructure enables data owners to share their linked data as subgraphs of the global KG stored by the infrastructure through the public SPARQL endpoints [15, 30] or dereferenceable URIs it provides.

However, decentralized systems are more vulnerable to attacks and faults (e.g., *Byzantine fault* [21]) and therefore need means of ensuring *data integrity* and *query verifiability* in order to facilitate trustworthiness. In the context of a commercial peer-to-peer database, compromised nodes can manipulate transactions and forge

[1]https://www.dbpedia.org/
[2]https://www.wikidata.org/

data, posing risks to financial transactions and compromising business data. Data integrity ensures the storage nodes cannot tamper with their data [6, 13, 37], while query verifiability ensures that query results are complete, sound, and fresh [65, 72–74]. Without any measures taken, a malicious storage node in DKG can modify its data or return incorrect, incomplete, or outdated results that do not match user requests, violating data integrity and query verifiability. Taking Figure 1 as an example, a malicious storage node (comparable to a travel agency) can intentionally conceal relationships within the KG (suitable or low-cost options) and selectively provide incomplete results to clients (to prioritize its travel packages).

Unfortunately, most of the existing DKG infrastructures tend to prioritize query efficiency over trustworthiness [2, 3, 8, 17], although recent studies exist that focus on data integrity in DKGs [4, 47, 55]. For example, ColChain [4] enables data integrity in Byzantine environments by establishing storage nodes that maintain duplicate, immutable copies of subgraphs through blockchain consensus. However, these studies still assume that the subquery and communication in DKGs are trustworthy, which may not hold in real-world settings. Therefore, it is highly relevant to provide mechanisms that make it possible to verify that the subquery and the aggregation result are correct.

The standard approach to enable the verification of query results is to maintain an authenticated data structure (ADS) [57] that enables the detection of incorrect query results computed by an untrusted party on an outsourced database. While ADSs can be extended to decentralized infrastructures, existing ADSs are not designed for the querying of DKGs and cannot be applied readily to this setting, for two main reasons.

- **Decentralized data storage.** Due to the decentralized storage of DKG data, DKG query processing is done in two steps: executing subqueries to obtain intermediate results from individual storage nodes and aggregating the intermediate results to get final query results [4]. In contrast, existing ADS schemes assume that all data is stored in a single node when building and maintaining an ADS.
- **Semantic richness.** Because of its semantic richness, KG data that is both diverse and exhibits complex relationships is challenging for existing ADSs to capture. Therefore, existing ADSs can support neither verification of local query processing nor verification of global query processing where local results are aggregated.

Therefore, in this paper, we design a new ADS-based SPARQL [9, 50] query verification scheme and enable the use of blockchain for its generation and management to ensure data integrity and query verifiability. To contend with the decentralized data storage, our main idea is to design an ADS that can accommodate essential metadata of subgraphs stored on different nodes to achieve a global index, thus relaxing the requirement for a node to hold all data to build an ADS. To contend with the semantic richness, we ensure that it is possible to embed the semantic information required for KG query into the ADS. Specifically, we use keyword prefixes and an RGB color model to represent all semantic information with minimal cost. With the resulting ADS, DKG can be verified in a divide-and-conquer manner and the data integrity can also be ensured by blockchain. Specifically, for the step of finding local subgraphs, we implement the global index as a new variant of the Merkle tree that can provide the location of any subgraph and give verification proof. For the step of aggregation, we provide a cryptographic accumulator [18, 46] and combine it with the ADS, thus allowing nodes to perform verifiable aggregation operations on intermediate results from different nodes.

Our contributions can be summarized as follows.

- We propose VeriDKG, a novel DKG system that enables SPARQL query verifiability. To the best of our knowledge, this is the first of its kind.
- We propose a verification framework for DKGs that relies on a novel ADS, the RGB-Trie, to process SPARQL queries in a divide-and-conquer manner with correctness proofs.
- We implement the RGB-Trie as a Merkle prefix tree with an RGB color model to enable the capture of the necessary semantic information and combine it with a cryptographic accumulator technique to support verifiable aggregation on intermediate results from multiple storage nodes.
- We provide a rigorous security analysis and report on experiments with a prototype of VeriDKG. The results demonstrate that the system can achieve state-of-the-art query performance on the largeRDFbench benchmark while supporting data integrity and query verifiability.

2 RELATED WORKS

2.1 Decentralized Knowledge Graphs

Decentralized data management involves distributing data across multiple nodes, enabling collaborative storage, retrieval, and processing [20, 22, 28]. Knowledge graphs, key components of the Semantic Web, organize information into nodes (entities) and edges (relationships). By decentralizing knowledge graph management, diverse stakeholders can contribute and access information, fostering a more open and inclusive web ecosystem. Unlike traditional centralized KGs [7, 11, 56], some DKG studies eliminate the central server and allow data owners to share their data in a logically global KG. For instance, RDFPeers [17] places a KG in a peer-to-peer network and uses a Dynamic Hash Table (DHT) technique to build an index. PIQNIC [2] splits a KG into fragments based on predicates and uses a flooding mechanism to find nodes storing relevant fragments. However, the query processing capability of PIQNIC is less efficient than that of a centralized server. Aebeloe *et al.* [3] add two indexing schemes in a peer-to-peer storage architecture to find nodes that store relevant data.

In summary, the above studies are committed to improving data availability and query efficiency, without paying attention to the security of DKG. However, a decentralized network is vulnerable to Byzantine faults in practice, which means that malicious storage nodes can tamper with KG data, providing wrong or no query results. Unlike previous works, our work focuses on trustworthiness of DKG in the Byzantine environment.

2.2 Blockchain for Knowledge Graphs

Blockchain is a tamper-proof and decentralized ledger for reliable and auditable data storage [43, 63] and has been used widely to enable trustworthy database management [25, 32, 44, 49, 51, 68, 75]. Several works have explored using blockchain to store linked data

in KGs, which allows untrusted nodes to collaborate on data updates and to keep a trusted historical record of it [19, 55, 62]. However, a major challenge of using blockchain in KGs is that it requires every node to maintain a full copy of all data, which results in poor scalability and high hardware requirements.

The work most relevant to us is ColChain [4], a community-based DKG that splits and stores KG data into multiple blockchain shards [24, 35] for different KG communities. ColChain allows nodes to trace the update record of each KG triple item by requesting related KG fragments from different shards, and performing the data aggregation operation locally to obtain the final query results. However, ColChain cannot guarantee the verifiability of queries because it uses untrusted indexes to locate KG data and doesn't provide any verification proof. In comparison, our work adopts the similar community-based DKG architecture and considers the challenges of query verifiability.

2.3 Verifiable Query Processing

Verifiable query processing ensures query verifiability in outsourced databases, typically employing cryptographic algorithms to generate proofs in untrusted environments. Many cryptography-based solutions, such as those in [73, 74], rely on ADSs maintained on untrusted servers. These structures often involve fundamental components like Cryptographic Hash Functions and Merkle Trees, with variants supporting complex queries in blockchain databases, like GEM^2-tree [71], vChain [66], and vChain+[61]. Zhang *et al.*[70] propose a hybrid-storage architecture for blockchains, enhancing scalability via a Chameleon index-based ADS. However, prior work lacks consideration for KG verifiable query requirements and cannot support verifiable SPARQL queries in DKG.

VeriDKG draws inspiration from verifiable query processing in outsourced databases. It centers on achieving verifiable KG queries via a novel ADS-based verification framework. This pioneering work introduces ADS-based schemes into DKG scenarios, facilitating extensive entity and relationship querying on the web.

3 PRELIMINARIES

3.1 Knowledge Graph

Resource Description Framework (RDF) is the standard format for encoding KGs[3]. A KG $\mathcal{G}$ includes a set of RDF triples [39], each consisting of a subject, a predicate, and an object, as defined below.

Definition 1 (RDF Triple)**.** An RDF triple is in the form of $\langle s, p, o\rangle$ representing a directed labelled edge $s \xrightarrow{p} o$, where s, p, and o denote subject, predicate, and object, respectively. Given infinite and disjoint sets U represents all URIs/IRIs, L represents all literals (text or string, etc.), and B represents all blank nodes, an RDF triple $\langle s, p, o\rangle \in (U \cup B) \times U \times (U \cup B \cup L)$.

For example, $\langle Dr.\ Smith,\ work\ in,\ Mount\ Elizabeth\ Hospital\rangle$ and $\langle Dr.\ Smith,\ married\ with,\ Mr.\ Jason\rangle$ are two RDF triples.

Definition 2 (Triple Fragment)**.** A triple fragment $f \subseteq \mathcal{G}$ is a finite set of RDF triples in a KG $\mathcal{G}$.

[3]https://www.w3.org/RDF/

Table 1: KG

Knowledge graph $\mathcal{G}$		
$\langle a, p_1, b\rangle$	$\langle a, p_3, d\rangle$	$\langle b, p_2, a\rangle$
$\langle e, p_2, f\rangle$	$\langle b, p_1, d\rangle$	$\langle b, p_3, c\rangle$
$\langle d, p_3, f\rangle$	$\langle c, p_1, e\rangle$	$\langle c, p_2, d\rangle$

Table 2: TPF

f_1
$\langle a, p_1, b\rangle$
$\langle b, p_1, d\rangle$
$\langle c, p_1, e\rangle$

SPARQL[4] is the de facto query language for KGs. It comprises a set of triple patterns, as defined below.

Definition 3 (Triple Pattern)**.** Given the sets U, L, and B in **Definition 1** and a set of all variables V, a triple pattern is a triple of the form $\langle s, p, o\rangle \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.

A SPARQL query contains multiple basic graph patterns (BGPs), each comprising a set of (conjunctive) triple patterns, which are combined with some graph patterns and operators, such as `JOIN`, `FILTER`, `OPTIONAL`, and `UNION`. For example, given two triple patterns $\langle ?who,\ married\ with,\ Mr.\ Jason\rangle$ and $\langle ?who,\ work\ in,\ ?where\rangle$, a SPARQL query that searches for the workplace of Mr. Jason's wife (i.e., inner join the two triple patterns on their subjects and select the results on the second triple pattern's object) and the person is younger than 30 (this condition is optional) can be

```
SELECT ?address WHERE {
  ⟨?who, married with, Mr. Jason⟩,
  ⟨?who, work in, ?address⟩,
  OPTIONAL ⟨?who, age, ?age FILTER (?age < 30)⟩. }
```

Definition 4 (Tripple Pattern Fragment)**.** Let f is a triple fragment, tp is a triple pattern, f is the triple pattern fragment of tp iff for every RDF triple $t1 \in f$, $t1$ matchs tp.

For example, given a KG $\mathcal{G}$ in Table 1, Table 2 shows the triple pattern fragment (TPF) f_1 which matches the triple pattern $\langle ?s, p_1, ?o\rangle$.

Decentralized Knowledge Graph. In a conventional KG, the entire KG is stored on a centralized trusted server that can execute SPAQRL queries on its local storage. However, in a DKG, the entire KG is divided into multiple subgraphs and distributed to different KG communities [4], each of which is a series of nodes that store the same subgraph. Note that a node can belong to multiple communities, but a community only stores a subgraph.

3.2 Verifiable Query Methods

Cryptographic Hash Function. A cryptographic hash function, $hash(\cdot)$, converts messages of varying lengths into fixed-size digests, ensuring that $hash(m)$ is computationally indistinguishable from random data. In simpler terms, finding a message m that matches a given hash $h = hash(m)$ is highly challenging.

Merkle Tree. Merkle tree [41] is a binary tree that stores hash values. It is a data structure used to quickly verify the integrity of large-scale data. Its leaf nodes are composed of hash values of records, and non-leaf nodes are generated upward through hash operations until the root node named Merkle root is generated. Therefore, a Merkle tree including n records can be used to verify the records in the leaf nodes in $O(log(n))$ time complexity.

Verifiable Set Operation. Verifiable set operation (VSO) supports various verifiable set operations, such as intersection (denoted

[4]http://www. w3. org/TR/rdf-sparql-query/

as ∩), union (denoted as ∪), and difference (denoted as \) [18, 46]. A VSO includes four steps as follows.

- KeyGen $(s) \rightarrow (sk, pk)$: The input is a random value $s \in \mathbb{Z}_p$, which is based on a bilinear-map accumulator primitive [45]. The outputs is a pair of a secret key $sk = s$ and a public key $pk = (g^s, \cdots, g^{s^q})$, where g is the generator of a cyclic multiplicative group $\mathbb{G}$ and q is an upper-bound on the cardinality of sets in the algorithm.
- Setup $(X, pk) \rightarrow acc(X)$: The input is a pair consisting of a set $X \subset \mathbb{Z}_p$ and pk. The output is an accumulated value $acc(X) = g^{\prod_{x \in X} (x+s)}$.
- Getproof $(X_i, X_j, opt, pk) \rightarrow (X^*, \pi)$: The inputs include two sets X_i and X_j, a set operation $opt \in \{\cap, \cup, \setminus\}$ and pk. The function returns the set operation result of these two sets X^* with a proof π.
- VerifyProof $(acc(X^i), acc(X^j), \pi) \rightarrow \{accept, reject\}$: The inputs are two accumulated value $acc(X^i)$ and $acc(X^j)$ and the proof π. The function then returns the validation result.

The unforgeability for verifiable set operations has been proved under the q-Strong Bilinear Diffie-Hellman assumption [12].

Blockchain. A blockchain is a public and tamper-proof ledger composed of a sequence of blocks, each storing transactions, and is maintained by multiple mutually distrusting nodes. To agree on the order of valid transactions among nodes, the nodes run a consensus (e.g., Proof of Work [43], Proof of Stake [63], and PBFT [5]) in the blockchain. In each block, the transactions are organized as a Merkle tree, and the Merkle tree root is stored in the block header.

4 VERIDKG: FRAMEWORK

4.1 System Model & Threat Model

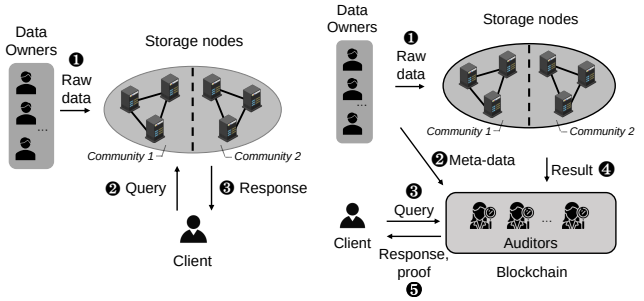
Basic Model. In a DKG system, there are three types of participants.

Data owners generate raw RDF data as triple fragments and share their data in the DKG. Due to limited resources, the data owners outsource their data to storage nodes. **Storage nodes** offer DKG storage and data management services, divided into KG communities, each responsible for specific triple fragments. They can collaborate across communities when executing SPARQL queries and facilitate addressing relevant nodes for a requested RDF triple. **Clients** query the global KG by issuing SPARQL requests to the storage nodes.

As shown in Figure 2a, data owners outsource their data to the storage nodes (❶), and a client can send its query request to any storage node (❷) and get the query result (❸). Most DKG systems have a data replication mechanism [2, 3] to ensure query services normally run when some storage nodes encounter failure.

Threat Model. Different from existing DKGs [2, 3] that assume that storage nodes are trustworthy, in VeriDKG, the storage nodes are not trusted. Each storage node may forge or tamper with query results or return outdated information for various reasons, such as program glitches, security vulnerabilities, and commercial interests. We assume that the data owners and clients are trusted. (A discussion for malicious data owners is provided in § 8.) A similar threat model can be found in outsourced databases [42, 64, 74].

System Model. Our VeriDKG introduces a new role of participants to build a verifiable SPARQL query engine on DKG as follows.



(a) A typical DKG (b) Our VeriDKG

Figure 2: Comparison between the existing DKG and our VeriDKG.

Auditors compose a blockchain system as a trust anchor in the DKG. The blockchain guarantees that the auditors can collectively build a public and immutable ledger via consensus. We assume that the proportion of malicious auditors will not exceed the fault threshold of blockchains (e.g., 1/2 in PoW or 1/3 in PBFT).

4.2 System Workflow

As illustrated in Figure 2b, the data owners, storage nodes, auditors and clients interact in VeriDKG as follows.

Phase 1: *Data Outsourcing.* Each data owner outsources its RDF data to the storage nodes (❶). At the same time, the data owner also calculates the hash of each RDF triple, and proposes a transaction containing the metadata (i.e., the hash, index information, and address of the triple in the storage network) for each triple.

Phase 2: *ADS Generation.* The data owner then submits the proposed transaction with metadata to the blockchain (❷). To commit the transaction, all the auditors run a consensus to build an ADS with an index function for all RDF data stored on the storage node network. The ADS maintained in the blockchain will be updated based on each newly committed transaction. The details will be described in § 5.

Phase 3: *Query Processing.* A client can issue a SPARQL query to any auditor (❸). The auditor converts the query to a set of triple patterns, uses the ADS to search for the triple pattern fragments of each triple pattern (namely triple pattern queries), and gets the triple pattern fragments from storage nodes (❹). All the triple pattern fragments are aggregated on the auditor for the final results. The auditor also generates proofs using the ADS, discussed in § 6.

Phase 4: *Query Verification.* The final query results and their corresponding proofs are sent to the client from the auditor, and the client verifies the query results (❺), which is discussed in § 6.

4.3 Goals

The query verification in **Phase 4** should guarantee that the query results returned from the storage nodes satisfy three security criteria: 1) **Soundness**: None of the RDF triples returned as results have been tampered with and all of them satisfy the SPARQL query conditions; 2) **Completeness**: No valid result (e.g., RDF triples and their members) is missing from the query results; 3) **Freshness**: The query results are based on the latest version of the DKG.

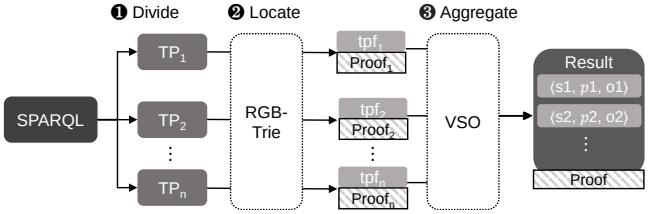


Figure 3: The process of verifiable SPARQL query execution in VeriDKG.

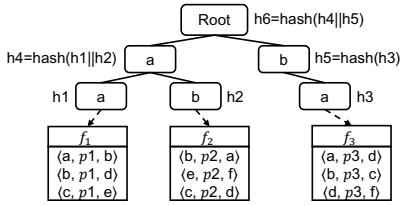


Figure 4: An example of an MPT in the strawman system.

4.4 Roadmap

As mentioned before, executing a SPARQL query in DKGs involves three steps: 1) dividing the SPARQL query into multiple triple patterns, 2) locating the relevant triple pattern fragments matching each triple pattern, and 3) aggregating these fragments for final results. To enable verifiable SPARQL queries in VeriDKG, we introduce a novel ADS for verifiable triple pattern queries and utilize a verifiable set operation-based approach for aggregation. In particular, as shown in Figure 3, in the locating process, VeriDKG employs the RGB-Trie, a Merkle tree variant, to find relevant intermediate results (triple pattern fragments) for all triple patterns alongside their proofs. In the aggregating process, most aggregation operators are converted to some set operations, and a storage node in VeriDKG can perform verifiable set operations on the intermediate results, and returns the final query results with a verification proof to clients. In the following, we will introduce the design of RGB-Trie in Section § 5 and the entire query process in Section § 6.

5 RGB-TRIE: ADS FOR VERIDKG

In this Section, we first introduce a strawman scheme of the ADS design in VeriDKG, which is a Merkle prefix tree for verifiable triple pattern queries with some limitations. We then give the detailed design of RGB-Trie, including its structure and management.

5.1 Strawman

To support verifiable triple pattern queries, we first describe a strawman ADS for **Phase 2** only based on a Merkle prefix tree (MPT) like Merkle Patricia Tree [63], an ADS for verifiable prefix-based keyword queries adopted by many blockchain systems. In the strawman, the auditors can build an MPT on the blockchain, which treats the value given in the triple pattern as a keyword. The internal nodes of the MPT stores the index information (i.e., keywords) of RDF triples, and the MPT's leaf nodes point triple fragments, each containing some RDF triples with the same keyword. For example, for KG $\mathcal{G}$ in Table 1, we let $p_1 = aa$, $p_2 = ab$, and $p_3 = ba$, the MPT to query $\mathcal{G}$ can be shown in Figure 4. For three triple patterns $\langle ?s, p_1, ?o\rangle$, $\langle ?s, p_2, ?o\rangle$, and $\langle ?s, p_3, ?o\rangle$, f_1, f_2 and f_3 are their corresponding triple pattern fragments through the MPT. Given a triple pattern fragment f_1' from a storage node, to verify whether it corresponds to triple pattern $\langle ?s, p_1, ?o\rangle$, a client can recover a Merkle root ($h6'$) based on the hash of f_1' ($h1'$) and a Merkle proof ($h2$ and $h5$), and then verify whether $h6$ and $h6'$ are the same.

Unfortunately, the strawman falls short of enabling verifiable triple pattern queries due to two limitations as follows.

First, it only supports verifiable queries for limited triple patterns, i.e., those where only one of the three features in triples is given. For example, the MPT in Figure 4 supports verifiable queries for triple patterns with a given predicate. It cannot support verifiable queries for triple patterns with a given subject or object.

Second, it cannot support verifiable aggregation of intermediate results, but a SPARQL query requires performing aggregation operations (e.g., UNION or JOIN) on triplet pattern fragments according to § 3.1. For example, if a SPARQL query requires the join of f_1 and f_3 on the subject, the MPT in Figure 4 cannot provide any proof.

Therefore, we propose a new ADS called *RGB-Trie*, a variant of MPT with two new characteristics. 1) It includes an RGB color model on MPT nodes for verifiable queries for any triple patterns, and 2) it integrates an accumulated value design with the MPT for verifiable set operations for aggregation on triple patterns.

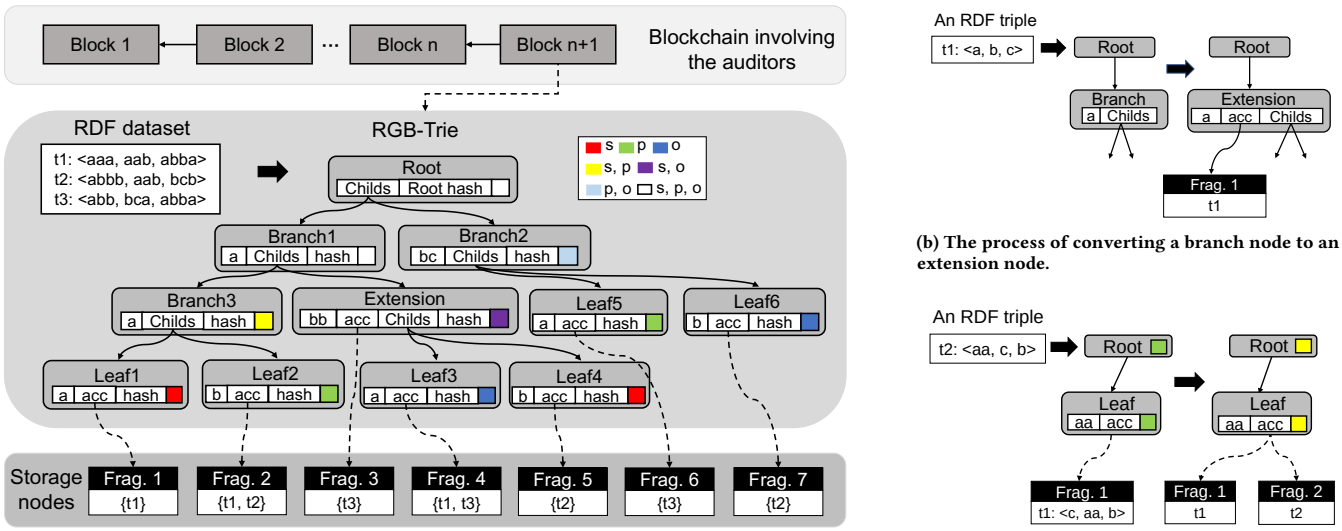
5.2 Trie Structure

As shown in Figure 5a, RGB-Trie comprises four types of nodes, i.e., a root node, branch nodes, extension nodes, and leaf nodes, which are described as follows. In the figure, an RDF dataset with three RDF triples is inserted into an RGB-Trie, and the features in the triples may have all or part of the same prefix. It's worth noting that the prefix for each entity in an RDF triple can encompass various elements such as property paths[5], named graphs[6], and other components found within the KG.

Root node. The top layer of RGB-Trie is a root node, which is a Merkle root maintaining a consistent snapshot of global KG. Based on the characteristics of MPT, the hash value inside a root node changes when any node in RGB-Trie is modified.

Branch/Extension node. The middle layer of RGB-Trie includes branch nodes and extension nodes. Each branch node connects its predecessor (parent node) and successors (child node) through its own value properties. It stores the common prefix of its child nodes and has at least two children. An extension node is a special branch node and can represent the termination of a query path. It has some pointers to point to triple pattern fragments. A branch node can be transformed into an extension node by inserting some RDF triples with a keyword whose matching path ends at this node to RGB-Trie. We illustrate the process of converting a branch node to an extension node in Figure 5b. When an RDF triple with a subject a is inserted into the RGB-Trie, a branch node with the common prefix a is transformed to an extension node that has a pointer to point the inserted RDF triple.

[5]https://www.w3.org/TR/sparql11-property-paths/
[6]https://www.w3.org/2009/07/NamedGraph.html



(a) RGB-Trie.

(b) The process of converting a branch node to an extension node.

(c) The process of changing the node color.

Figure 5: The structure of RGB-Trie and two cases of RGB-Trie node operation (Frag. means triple fragment.)

Leaf node. RGB-Trie's bottom layer comprises leaf nodes, each with pointers to address sets for triplet pattern fragments and their accumulated values. Additionally, each leaf node holds hash values for the referenced fragments. These attributes are pivotal in enabling RGB-Trie to furnish verification proofs for SPARQL queries (refer to § 6).

The design above creates an index for triple fragments linked to an input word. However, in KG, each triple has three features (subject, predicate, object), and each index fragment corresponds to just one feature. This necessitates three separate tries, tripling the storage and computation overhead as all three must update simultaneously with each transaction. To address this, we merge these tries into one using an RGB color model as follows.

Definition 5 (RGB Color Model)**.** Given a node n in RGB-Trie, its color field is defined as red, green or blue if its search target feature is subject s, predicate p or object o, respectively. If n points to two different features, e.g., $s \wedge p$, $s \wedge o$, or $p \wedge o$, its color field is defined as yellow, magenta, or cyan, respectively. The color field is set to white when n has simultaneous access to the three features.

Example. *An example of node color change is shown in Figure 5c. The color of a leaf node is green because the fragment it points to is a triple pattern fragment for the triple pattern* $\langle ?s, aa, ?o \rangle$ *and its index value is* ***aa****. If a new RDF triple with a subject* ***aa*** *is inserted into this leaf node, its color is changed from green to yellow because yellow is a mixed color of green (for predicate) and red (for subject). After the node color change process is over, the leaf node will store two different pointers, two hash values, and two accumulated values.*

After adding the RGB color model to MPT, the RGB-Trie can convert any form of triple pattern query into keyword queries with different color combinations, which is described in § 6. We also give a cardinal Rule of *color mixing* for the RGB color model in RGB-Trie to improve its query performance. Details are shown in § 5.3.

5.3 Operations of RGB-Trie

At the beginning of VeriDKG, an empty RGB-Trie is stored in the genesis block (the first blockchain block). Along with each new RDF data outsourced to the storage nodes (Figure 2b-❶), the auditors change the RGB-Trie according to the newly committed transaction about the new RDF data (Figure 2b-❷). There are three kinds of operations, i.e., *insert*, *update*, and *delete*, introduced as follows.

5.3.1 Insert Operation. After receiving a transaction with an address points to an RDF triple $\langle s, p, o \rangle$, the auditors add three items $[s, red]$, $[p, green]$, $[o, blue]$ to an item list L. For each item $item_i$ in L, the auditors search for an insertion point $node_p$ of the root node of RGB-Trie by traversing its child nodes.

For each item $item_i$, if the auditors find $node_p$, it inserts $item_i$ to $node_p$ through a recursive insertion algorithm. If they cannot find $node_p$, auditors create a new child node $node_{new}$ of the root node of RGB-Trie and insert $item_i$ to $node_{new}$. In the insert processing, if at any time the next character to be matched in the item does not match the characters stored in all children of an inserted node, the RGB-Trie will create a new child node of the inserted node to store the remaining unmatched part of the item. If $node_p$ has more than one character (i.e., slices of character, which is a variable-length character array), and the unmatched part of the item is only partly the same as the slices, $node_p$ will split (i.e., keep the matched part of the slices in the node and create two new child nodes to store the remaining part of $item_i$ and the remaining part of the slices).

Example. *Consider Figure 5a as an example. Suppose* $t3 = \langle abb, bca, abba \rangle$ *is a new triple. When it is inserted into the tree, the right child node of node Branch*1 *is converted to an extension node with a new triple fragment Frag.*3*. The node Branch*2 *will only keep prefix* ***bc*** *and split with two new child nodes, including a leaf node Leaf*6 *with Frag.*7 *and a leaf node Leaf*5 *with prefix* ***a*** *and a new triple fragment Frag.*6*. And t*3 *will also be inserted into Frag.*4*.*

Finally, after the insertion process, the auditors update RGB-Trie's hashes and colors and change it to a new state. Different nodes have different methods to update their hash value. Let $||$ be the concatenation operator of multiple values, $hash(\cdot)$ is the cryptographic hash function, and the hash update processing of different types of nodes in RGB-Trie are as follows:

For a leaf node n_l, the hashing process is:

- f_n = the triple pattern fragment(s) that n_l points;
- c_n = the keyword related index content in n_l, i.e., a segment of the keyword.
- $h_n = hash(c_n||hash(f_n))$, it is the hash value of n_l.

For an extension node n_e, denotes its child nodes is $\{cn_1, cn_2 \ldots cn_k\}$, the hashing process is:

- f_n = the triple pattern fragment(s) that n_e points;
- c_n = the keyword related index content in n_e, i.e., a segment of the keyword.
- $ch_n = hash(cn_1||cn_2 \ldots||cn_k)$
- $h_n = hash(c_n||hash(f_n)||ch_n)$, it is the hash value of n_e.

For a branch node n_b, denotes its child nodes is $\{cn_1, cn_2 \ldots cn_k\}$, the hashing process is:

- c_n = the keyword related index content in n_b, i.e., a segment of the keyword.
- $ch_n = hash(cn_1||cn_2 \ldots||cn_k)$
- $h_n = hash(c_n||ch_n)$, it is the hash value of n_b.

Besides, RGB-Trie updates the color of all nodes on this path at the same time. It follows the RGB additive color mixing rule. The color of a node is determined by a mixture of the index field in which it stores its content and the color of its children, and the specific descriptions are shown as follows:

Basic color mixing rule. Under this rule, each node in the RGB-Trie has a triple (R, G, B), and each element of the triple is a binary value. We use 1 to indicate that the color exists, and 0 to indicate that the color does not exist. For example, in Figure 6, the color of the root node is white and represented as $(1, 1, 1)$, because all child nodes of the root node contain all items of the RDF triple. To update the color of a non-leaf node on the RGB-Trie, the node needs to OR the color triples of all its child nodes.

Gradient color mixing rule. Under this rule, each node in the RGB-Trie has a triple (R, G, B), and each element of the triple is an 8 bits value. We use $(255, 0, 0)$ to represent color red, $(0, 255, 0)$ to represent color green and $(0, 0, 255)$ to represent color blue. To update the color of a non-leaf node on the RGB-Trie, it needs to calculate the proportion of different items of RDF triples (i.e., s, p, and o) in its child nodes. For example, in Figure 6, the color triple of the root node can be calculated as $(255 * (3/5), 255 * (2/5), 255 * (1/5))$, because there are 5 RDF triples in the whole KG, including 3 pointed by subject, 2 pointed by predicate and 1 pointed by object.

The two color mixing rules offer distinct advantages and drawbacks. The basic rule simplifies color calculations and ensures precise and comprehensive query results. In contrast, the gradient color mixing rule empowers clients to decide query termination based on color proportions, enhancing efficiency but potentially reducing recall. Importantly, the recall reduction doesn't compromise query verifiability, as it aligns with client preferences. We will assess the query performance of these rules in § 9.

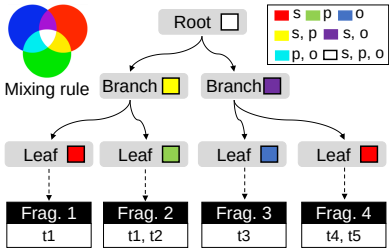


Figure 6: The color mixing rule in RGB-Trie.

5.3.2 Update & Delete. When a data owner wants to update or delete an RDF triple stored in storage nodes, it asks the storage nodes to update or delete that triple, and sends a new transaction consisting of the triple and a field marked for update or deletion to the auditors. When the auditors receive the transaction, they will update or delete the triple item and update the nodes in the RGB-Trie to a new state. Under these two operations, the update process of the RGB-Trie is similar to the insert operation.

5.4 Cost Analysis

Here, we provide the time and space complexities associated with the RGB-Trie. **Time complexity:** When inserting an item with a word of length L and a color, the auditor traverses the RGB-Trie recursively level by level, which incurs L comparisons in the worst case. Updating RGB-Trie's hashes and colors upon an insertion involves reversing the order of traversal and L executions. Thus, insertion has a time complexity $O(L)$, with L being the length of the word to be inserted. Similarly, update, deletion and querying also have time complexity $O(L)$.

Space complexity: The space consumption of an RGB-Trie depends on its total number of nodes, which is related to its depth and number of leaf and non-leaf nodes. An RGB-Trie is constructed from a dataset of RDF triples. We denote the number of distinct items among all RDF triples in the input dataset by N, and we denote the cardinality of the character set of the input dataset by D. We can then determine an upper bound on the number of children of each non-leaf node. We start by observing that in an RGB-Trie, each item is stored either in a non-leaf node or in a leaf node. Further, each non-leaf node has at least 2 and at most D child nodes (assuming $D \geq 2$). Next, in the worst case, an RGB-Trie is a fully balanced binary tree with all items being stored in leaf nodes. Thus, the leaf layer has N nodes, each non-leaf layer L has 2^L nodes, and the RGB-Trie has $\lceil log_2 N \rceil$ layers. The maximum total number of nodes that an RGB-Trie can contain is therefore $1 + 2 + 2^2 + \ldots + 2^{log_2 N - 1} + N = 2N - 1$. Consequently, the space complexity of the RGB-Trie is $O(N)$.

6 QUERY PROCESSING AND VERIFICATION

As described in § 3.1, each SPARQL query combines some triple patterns with aggregation operations. We achieve a verifiable SPARQL query process in two steps, i.e., *verifiable triple pattern query* and *verifiable fragments aggregation*, as follows.

Verifiable Triple Pattern Query. A triple pattern query aims to search for a triple pattern fragment that matches a given triple pattern. Algorithm 1 describes the verifiable triple pattern query

Algorithm 1: Verifiable triple pattern query

1 **Function** *Search for fragments (r,tp_i)*:
  Input : RGB-Trie root node r, a triple patten tp_i
  Output: Triple pattern fragments f, matching path p
2 **foreach** $item_i$ *in* tp_i **do**
3 f_i, p_i = r.nodeSearch($item_i$)
4 **if** f_i != *null* **then**
5 add f_i to f
6 **else**
7 add p_i to p
8 **return** f, p;
9 **Function** *Get proof (node with f_i,$f_i \in f$, p)*:
10 **foreach** *node with* f_i **do**
11 add hash(f_i) to VO;
12 **while** *node.parent != null* **do**
13 add nodeInfo to VO
14 **foreach** $node_i \in node.parent.child$ **do**
15 add $node_i.hash$ to VO
16 node ← node.parent
17 **if** p != *null* **then**
18 add p, Merkle proof of p to VO
19 **return** VO

Algorithm 2: Verifiable data aggregation

1 **Function** *Get result and proof (f, tp, pk)*:
  Input : Triple fragments f, triple pattern list tp, RGB-Trie R, public key pk
  Output: Query result S, verification proof π
2 **foreach** $f_i \in f$ **do**
3 **foreach** $tp_j \in tp$ **do**
4 tpf_{ij} ← R.search (f_i, tp_i) and add tpf_{ij} into tpf_j
5 add tpf_j into tpf
6 $S \leftarrow$ aggregate (all $tpf_i \in tpf$)
7 $\pi \leftarrow$ prove (tpf_i, pk)
8 **return** $\{S, \pi\}$
9 **Function** *Verify proof (R, π, X^*)*:
  Input : Triple pattern fragments tpf, query result S, verification proof $\{X^*, \pi\}$
  Output: verification result
10 **foreach** $tpf_i \in tpf$ **do**
11 add $acc(tpf_i)$ into ACC
12 $result \leftarrow VerifyProof(ACC, \pi)$
13 **return** $result$

process through RGB-Trie. First, the auditor searches in RGB-Trie to find the triple pattern fragments that match the input triple pattern (Lines 1-8). Each given value of the triple pattern, i.e., $item_i$, visits the RGB-Trie to query related triple fragments with their colors independently. RGB-Trie uses the depth-first search method to find a path in RGB-Trie that matches $item_i$, and the path ends at a leaf node or an extension node. Then, if $item_i$ matches successfully, RGB-Trie will add the fragment pointed by the end-point node and its Merkle proof to a verification object (VO) (Lines 9-16). Otherwise, if $item_i$ matches failed, RGB-Trie will add a set of nodes with their Merkle proofs, including the nodes in the partially matched path and all child nodes of the last node in this path, to the VO (Lines 17-18). Finally, the VO is returned to the client.

After receiving the results and proof, the client can verify them by comparing the recovered root node's hash that it calculates with the root hash of RGB-Trie which is kept in the current block header. If they are equal, the verification succeeds.

Example. *To search a triple pattern $tp_i = \langle aaa, ?p, aab \rangle$ in Figure 5a, an auditor should extract two items **aaa** and **aab** with color red and blue, and later search the RGB-Trie to find the related triple pattern fragments. The auditor will get the results $R_{aaa} = \{t_1\}$, and $R_{aab} = \emptyset$, because no nodes in RGB-Trie have both a prefix content of **aab** and possess the blue color. Finally, the auditor takes the intersection of two intermediate results R_{aaa} and R_{aab} and gets the final result $\emptyset$. Thus, the auditor will return the final result of tp_i (i.e., $\emptyset$), the intermediate result $\{t_1\}$ with its Merkle proof, and the non-existing proof of R_{aab} to the client.*

Verifiable Fragments Aggregation. After the triple pattern query, a data aggregation process needs to be executed on an auditor that collects the intermediate results (i.e., triple pattern fragments) to get the final results. A SPARQL begins with an operation `SELECT` followed by a subset of variables in the triple patterns, and then a keyword `WHERE` followed by a set of triple patterns (i.e., a BGP) connected by some graph patterns (e.g., `JOIN`, `LEFT JOIN`, `FILTER`) and operations (e.g., `OPTIONAL`, `UNION`, `ORDER BY`). It is worth noting that most graph patterns and operations can be converted to set operations between subgraphs. For example, for a `JOIN` graph pattern involving the same features of two triple pattern fragments, an auditor treats the features of these two fragments as two sets and constructs a proof for their intersection through VSO. In the following, we will describe the detailed execution process of the verifiable fragments aggregation on the graph patterns and operations that can be converted to set operations and give the solutions for other operations.

In VeriDKG, the auditor is responsible for aggregating the intermediate results from the triple pattern query transfers the aggregation operations in SPARQL query request into accumulator set operations (e.g., intersection, union, complement, and difference) and uses the following algorithm to generate verification proofs of the aggregating operation.

Algorithm 2 describes the process of verifiable fragments aggregation. First, for each triple pattern, an auditor uses the RGB-Trie to search for its related triple pattern fragments (Lines 2-5). After all triple pattern fragments are found, the auditor aggregates them to get the final query result (Line 6). To generate the verification proof of the final result, the auditor uses the accumulated values of all triple pattern fragments and generates a verification proof π (Line 7). The client that receives the query result uses the proof π and the accumulated values to verify the query result locally (Lines 9-13).

Example. *Figure 7 gives an example of fragments aggregation, the SPARQL query selects s fields from two fragments through two triple patterns including a p-fixed triple pattern and an o-fixed triple pattern. The generation proof is $< acc(X^1), acc(X^2), \pi, [1, 1, 0], [1, 0, 1] >$.*

To calculate the accumulated value of each triple pattern fragment, all the auditors collaboratively generate a pair of keys (i.e., a

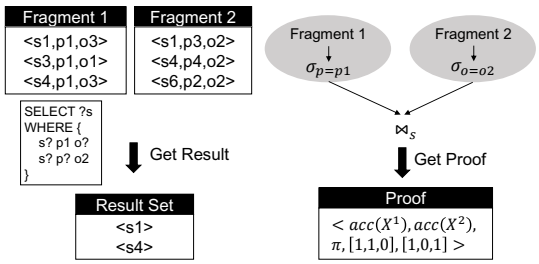


Figure 7: Example for proof generation of two triple fragments 1 and 2, respectively. (σ is an operator to select triples from a triple pattern and $\bowtie$ is an operator to join fragments based on a specified column.

public key and a private key), share the public key in a public way, and share the private key in a private way (e.g., secret sharing [53]). All the accumulated values are stored on the extension nodes and leaf nodes of RGB-Trie.

Verification of Non-set Operations. For other operations that cannot be converted to set operations, the auditor can let clients verify the results themselves or use other cryptographic tools to generate proofs for their results. These operations include some restrictions in the `FILTER` pattern and most of the solution sequence modifiers (e.g., `ORDER BY`, `OFFSET`, `DISTINCT`, `LIMIT`). For restrictions in the `FILTER` pattern, the auditor can use a hash function, a partial path of RGB-trie, VSO, or generate a general zero-knowledge proof to prove that the result satisfies its constraints. Specifically, a matching or no-matching constraint of two words can be verified by their hash values, a regular expression constraint can be verified by VSO and a Merkle proof provided by a partial path of RGB-Trie, and a range proof of whether the query result satisfies a certain size range can be generated through bulletproof [16]. For the solution sequence modifiers, since the sorting criteria are given by the clients, they can verify the query results themselves without proof. A `GROUP BY` clause is used to group query results based on one or more variables, and can also be checked by clients themselves.

Time-window Query. It is straightforward to extend the query algorithm to support time-window multi-version queries because the state of RGB-Trie at each moment is recorded by its root hash which is stored in the block header with a timestamp. Therefore, to query historical data in DKG, a client only needs to send a SPARQL query request with a given time period to an auditor, and the auditor uses the previous version RGB-Trie for the time-window queries. For verification, the client can use the Merkle root corresponding to the time window stored locally to verify the query result.

7 VERIFIABILITY ANALYSIS

VeriDKG enables verifiable SPARQL query in DKGs, which ensures soundness, completeness, and freshness as defined in § 4.3. Next, similar to the common security definition of verifiable query [73, 74], we describe the formal definition of our SPARQL query's security as follows.

Definition 6 (Query verifiability)**.** A SPARQL query is verifiable if the success probability of any polynomial-time adversary $\mathcal{A}$ is negligible in the following experiment:

For a SPARQL query Q, $\mathcal{A}$ is picked as the auditor for executing the triple pattern queries and fragments aggregation, and $\mathcal{A}$ produces result R and proof VO_t for Q. $\mathcal{A}$ succeeds if one of the following results is true: 1) R includes an RDF triple which does not satisfy Q (**correctness**); 2) There exist an RDF triple which is not in R but satisfies Q (**completeness**); 3) R includes an RDF triple not from the latest DKG (**freshness**).

Theorem 7.1. *VeriDKG is verifiable with respect to **Definition 6** if the hash function is a pseudo-random function, the accumulator is secure under the q-SBDH assumption, and the proportion of malicious auditors will not exceed the fault threshold of blockchains.*

Proof. We intuitively prove Theorem 7.1 by three cases, which represent proofs of soundness, completeness, and freshness.

Case 1: This case means a tampered or fake RDF triple t is returned, which does not satisfy the BGPs (i.e., a set of triple patterns) of Q. In this case, if t passes client verification following the soundness in **Definition 6**, it implies that the auditor can obtain two distinct triple pattern fragments sharing the same digest RGB_{root} in the on-chain ADS or two distinct set operation results with the same accumulator proof π. Case 2: This case means an RDF triple t that satisfies the BGPs of Q is missing from R. In this case, if the returned result R verifies with the client under the completeness criterion in **Definition 6**, it suggests that the auditor can acquire a triple pattern fragment lacking some matching triples but sharing the same digest RGB_{root} as the genuine fragment or an incomplete set operation result with the same accumulator proof π as the genuine result. Case 3: This case means the result R involves an old RDF triple t that satisfies q but is not from the latest DKG. In this case, once t passed the verification of the client under the freshness in **Definition 6**, it means that the auditor can get two different triple pattern fragments (i.e., a new and an old) with the same digest RGB_{root} of the on-chain ADS or the auditor can get two different set operation results with the same accumulator proof π.

However, all these three cases contradict two assumptions. First, the on-chain ADS digest RGB_{root} is generated by a cryptographic hash function, making it nearly impossible for the auditor to forge another fragment with the same hash value as the genuine one. Second, the unforgeability of verifiable set operations, proven to hold under the q-SBDH assumption [12]. A special case occurs when the auditor returns a null result to the client while the system indeed has the matched query result. This case also contradicts the first assumption as the auditor must provide non-existence proof. □

8 DISCUSSION

Storage Optimization of RGB-Trie. Since RGB-Trie needs to store the index information of all triples, the size of RGB-Trie increases fast as the KG data is continuously added. Storing the latest RGB-Trie in every block will be expensive. Thus, in VeriDKG, auditors only need to update part of the RGB-Trie in a new block with its transactions, and the internal nodes of the two RGB-Tries in two blocks are mostly the same. Based on the node pointers

design in [48], we use a node pointer structure to link the same path between multiple RGB-Tries in multiple blocks.

Limitations and Potential Workaround. When deploying VeriDKG in real-world environments, some limitations and potential workarounds must be considered. First, while VeriDKG effectively protects against malicious behavior between storage nodes, it does not address the possibility of malicious data owners introducing fake or junk data-a challenge seen in widely used AI applications. To mitigate this concern, we assume the integration of a content moderation mechanism (e.g., as proposed in existing studies [33, 36, 58]) to detect malicious data owners. Second, there are several resource-intensive tasks in VeriDKG, such as data storage, blockchain consensus, and query execution. However, incentivizing participants to effectively contribute resources can be complicated and requires careful consideration of the interests of different participants and ways of ensuring fairness. Third, VeriDKG introduces auditors that are responsible for maintaining trust anchors in the DKG. They require higher hardware specifications compared to existing DKG nodes. For example, the Ethereum backend requires a full node with at least 16GB of RAM and a 1TB SSD. Hardware requirements for blockchain node deployment vary widely depending on factors such as network, participation level, and use case. Careful selection of settings by application deployers is critical.

Real-world Deployment. The deployment of VeriDKG involves several main steps. Blockchain nodes, acting as auditors, are initially set up with robust hardware configurations, including substantial RAM and storage capacity. Users access the DKG through responsive interfaces from a variety of devices. Storage nodes, equipped with suitable hardware and data distribution software, host and share the KG data. Additionally, scalability, maintenance strategies, and regulatory compliance are considered to ensure seamless real-world operation. VeriDKG is particularly suited to high-trust data scenarios, such as those found in healthcare and finance, and offers trusted data to improve AI fidelity.

Graph Model Extension of VeriDKG. In addition to RDF triple-based KGs, some recent studies focus on graph model-based knowledge graph management. VeriDKG can be extended to such graph-based DKGs by managing RDF data in a graph database. This involves two steps: storing RDF triples as a graph, where entities are represented as nodes and relations are represented as edges, and converting SPARQL queries into graph patterns, such as paths or trees. Taking Neo4j[7] as an example, we can use tools such as rdf2neo [14] to convert RDF triples into a Neo4j graph and to convert SPARQL queries into Cypher queries.

9 PERFORMANCE EVALUATION

9.1 Implementation

A prototype of VeriDKG is implemented in Java, Go and JavaScript. In the prototype, the blockchain involving the auditors is implemented based on Go-Ethereum [26] and the decentralized storage system is implemented based on IPFS [10]. The prototype has a user-server architecture that is implemented based on Spring Boot framework [59] and the blockchain interfaces and requests are in the form of web3.js [27]. For the proposed ADS, the level of the tree is set to 32, and the cryptographic hash function is SHA-256. The triple fragments stored on storage nodes are separate HDT files [29], which allows the storage nodes to efficiently execute triple patterns. For VSO, our prototype uses library named ate-pairing [34].

[7] https://neo4j.com/

Table 3: Overall comparison of four different systems

Schemes	Query Verifiability	Storage cost of index (KB)	Triple pattern query time (s/ms)
P2P-LI [3]	✗	317982	7.392/3.696
P2P-RGB	✗	5647	36.990/18.495
Colchain [4]	✗	317982	8.374/4.187
VeriDKG	✓	9193	51.116/25.558

9.2 Experimental Setup

Hardware Configuration. We run 16 VeriDKG auditor nodes and 16 storage nodes on 32 64-bit Linux servers (Ubuntu 20.04) with Intel i9-11th CPU and 64GB memory. All nodes are run on separate machines, and we set the bandwidth of connections to 20Mbps.

Baseline. Three state-of-the-art DKGs are considered as three baselines. (1) A DKG with a locational index in every peer [3] (**P2P-LI**). (2) VeriDKG without the blockchain architecture and Merkle tree characteristic, which is a DKG with an untrusted RGB-Trie (**P2P-RGB**). (3) A sharding blockchain-based DKG in ColChain (**Colchain**) [4]. **Colchain** adopt the same consensus of Ethereum and hardware configuration. Besides, the maximum number of shards in **Colchain** is 16, and each shard has 8 nodes which are put into the docker containers.

Datasets and Benchmark. We evaluate the query performance of VeriDKG using six real-world datasets that represent different scenarios and using queries from largeRDFBench [52] benchmark, which is used widely in the Semantic Web community. The datasets include DBPedia-Subset (42,849,609 triples), GeoNames (107,950,085 triples), Jamendo (1,049,647 triples), Linked MDB (6,147,996 triples), DrugBank (517,023 triples), and Semantic Web Dog Food (103,595 triples). The largeRDFBench queries in our evaluation include simple (S), complex (C), and large data (L) categories. Moreover, we develop a variant of simple queries named time-window simple queries to study the time-window query in VeriDKG.

Metrics. We measure the following metrics of VeriDKG: 1) *On-chain Storage Cost (OSC)*: the storage space size of transactions in the blockchain, 2) *Transaction Throughput (TT)*: the number of committed transactions per second, 3) *Triple Pattern Query Time (TPQT)*: the amount of time to receive the triple pattern query results, 4) *Query Execution Time (QET)*: the amount of time to receive the full query results, 5) *Number of Exchanged Messages and Transfer Bytes (NEM & NTB)*: the number of messages exchanged and transferred bytes between nodes, 6) *Proof Generation Time (PGT)*: the amount of time to generate the verification proof of query results, and 7) *Verification Time and Object Size (VT & VOS)*: the amount of time to verify query results and the proof size.

9.3 Experimental Results

9.3.1 Overall Comparison. Table 3 provides an overview of the performance of VeriDKG, comparing its query verifiability, index storage cost, and triple pattern query execution time with those of three baseline systems on the six datasets. Here, the clients send

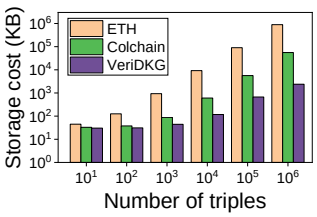


Figure 8: On-chain storage cost (y-axis in log scale.)

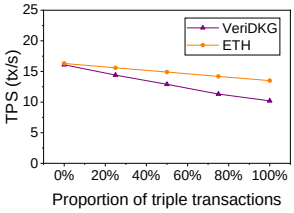


Figure 9: Transaction throughput.

Table 4: Performance of RGB-Trie under different rules. It shows triple pattern query time (TPQT) and recall per 1000 queries for basic and gradient color mixing rule with different proportion of colors required for early termination.

Rule	TPQT	Recall
Basic	51.2s	1
Gradient (5%)	47.5s	0.952
Gradient (10%)	40.7s	0.816
Gradient (20%)	27.8s	0.621
Gradient (30%)	22.3s	0.366

2,000 triple pattern query requests to each system. As shown in Table 3, only VeriDKG enables verifiable SPARQL query results, ensuring that the malicious storage nodes cannot tamper with query results. Considering the index storage cost, the index size of **P2P-LI** and **Colchain** is 317,982 KBs, while **P2P-RGB** and VeriDKG only need 5,647 KBs and 9,193 KBs, respectively. The reasons are that the RGB-Trie compresses the index by combining the same prefixes of keywords and that VeriDKG needs a Merkle characteristic to enable query verifiability. As for the triple pattern query time, **P2P-LI** and **BC-SC** need 7.392s (3.696ms per query) and 8.374s (4.187ms per query) to get the query results because they have the same index, and **Colchain** has to find the triples in the blockchain transactions. **P2P-RGB** and VeriDKG respectively require 36.990s (18.495ms per query) and 51.116s (25.558ms per query) to get the query results because they have the same index, and VeriDKG need more time to generate the verification proofs.

In summary, VeriDKG is the only one DKG that achieved verifiable SPARQL query and has a smaller index size than the locational index-based systems. Although it requires more time to get the triple pattern query results than the existing systems, it can guarantee the verifiability of query results and the extra time is tiny in real-world applications.

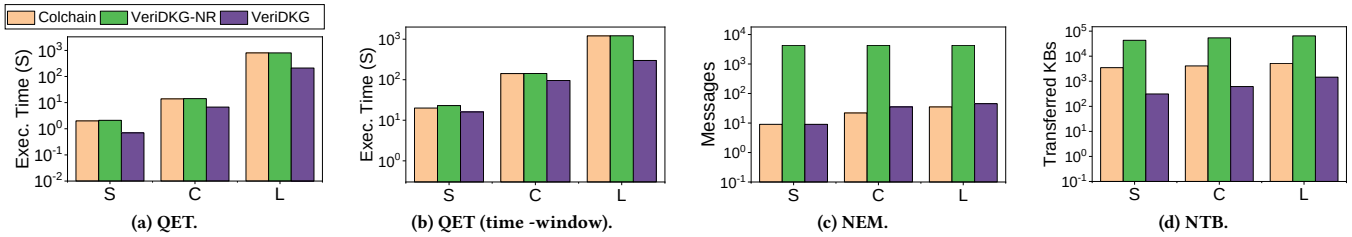
9.3.2 On-chain Cost. To alleviate the on-chain storage pressure, VeriDKG stores the raw data on storage nodes and only keep the metadata on blockchain. We randomly sample triples from the 6 datasets to determine the storage cost (i.e., OSC) of VeriDKG and the other two blockchain-based baseline systems. For **Colchain**, it has 16 shards and each node only stores one shard. Figure 8 shows the average storage size of each node in VeriDKG and the baselines.

As shown in Figure 8, by comparing to the original Ethereum blockchain scheme **ETH**, **Colchain** reduces the on-chain storage of each node by about 80%, and maintains this ratio as the amount of data grows. Compared to the two baselines, the storage savings in VeriDKG can increase to over 99% as the amount of data increases. It is because our design only keeps a succinct ADS on-chain, and most of the growing data is stored off-chain. The ADS only needs to update some hash values of the index information. We also compare experimentally the storage consumption of the RGB-Trie with and without optimization. For the VeriDKG system without storage optimization, the total size of the RGB-Trie is 13,296MB. For the VeriDKG system with storage optimization, the total size of the RGB-Trie is 2,241MB. Thus, after optimizing the storage of the RGB-Trie, the storage cost can be reduced by more than 80%.

Moreover, we evaluate the transaction throughput (i.e., TT) of **ETH** and VeriDKG with different proportions of triple transactions, and the results are shown in Figure 9. The result shows a downtrend of TT in both systems when the proportion of triple transactions in all transactions increases. In particular, VeriDKG has a bigger trend in decline, which means that the update of the RGB-Trie has a negative impact on TT. However, VeriDKG still maintains more than 10 transactions per second in the worst case, which is acceptable.

9.3.3 Verifiable Query Performance. Figure 10a shows the query execution time (QET) for different SPARQL queries in **Colchain**, in VeriDKG without the RGB-Trie (**VeriDKG-NR**), and in VeriDKG. Note that there is no global index in **Colchain**. Similar query performance is observed of **VeriDKG-NR** and **Colchain**, and VeriDKG is fastest for all three queries because RGB-Trie can efficiently find query-relevant fragments, which significantly reduces the search space. Figure 10b shows the QET for time-window queries. VeriDKG has the shortest QET among the three systems because, apart from the above reasons, in real-time SPARQL queries, the auditors in VeriDKG only need to backtrack all block headers to search for a certain previous RGB-Trie state by a given timestamp, instead of tracing back the historical records of all query results. Figure 10c shows the average number of exchanged messages (NEM) between nodes in the three systems. VeriDKG has the same number of messages as **Colchain** because both two only need to transfer a fixed number of fragments between nodes. Further, **VeriDKG-NR** needs to transmit larger messages to get the result because it lacks an RGB-Trie and needs to download all fragments from all storage nodes. Figure 10d shows the number of transferred KBs (NTB) in the three systems. The nodes in VeriDKG transmit the least amount of data for each query because VeriDKG only needs to transfer the query related triple pattern fragments to the auditor, which are more compact than the entire RDF dataset.

Comparison of different color mixing rules. As we mentioned in § 5.3, the gradient color mixing rule can improve the query performance of RGB-Trie while decrease its recall (i.e., the fraction of relevant RDF triples that are returned). Thus we test the triple pattern query time (TPQT) and recall of 1000 triple pattern query requests in VeriDKG under different color mixing rules. Table 4 shows the results. The results imply that the gradient color mixing rule can reduce the time of triple pattern query VeriDKG, while relaxing the early termination conditions results in higher query efficiency at the expense of lower recall rates.



(a) QET. (b) QET (time -window). (c) NEM. (d) NTB.

Figure 10: Query performance (y-asix in log scale).

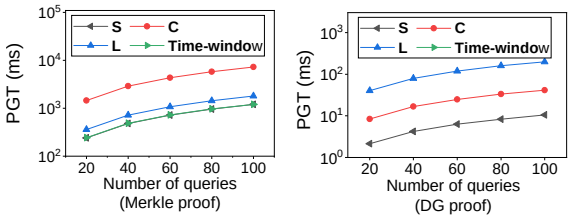


Figure 11: Proof generation time (y-axis in log scale).

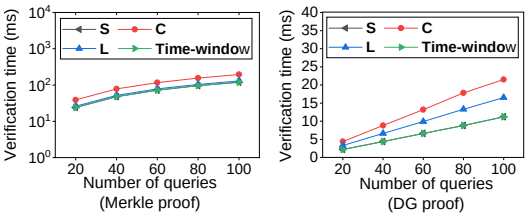


Figure 12: Verification time (left: y-axis in log scale).

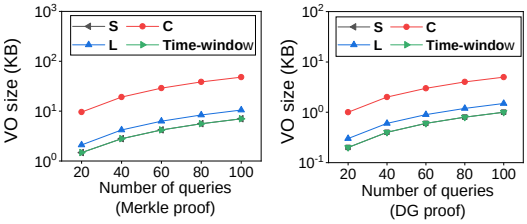


Figure 13: Verification object size (y-axis in log scale).

9.3.4 Verification Cost. All the query results with their verification proofs are generated on an auditor and need to be verified on the client side. Therefore, the proof generation time on the auditor, the size of proof, and the verification time on the client are very important for the availability of VeriDKG. A proof consists of two parts: a Merkle proof and a data aggregation (DG) proof. We evaluate the verification costs of these two proofs as follows.

Figure 11 shows the proof generation time (PGT) of the auditor for four SPARQL query types (S, C, L, and time-window S query). It shows that the complex queries have the longest PGT because they have the highest number of query-related fragments, each of which needs a Merkle proof. On the other hand, the large data queries have the longest PGT for data aggregation proof. Because most of their query-related fragments exceed those of the other queries, they need the most time to generate the accumulated values.

Figure 12 shows the verification time (VT) for proofs of different SPARQL queries, which are the same in Merkle proof, and complex queries have slightly longer Merkle proof VT. This is because all queries need to calculate the same Merkle root hash, while the complex queries need to calculate hashes of more fragments. From the VT results for the data aggregation proof, we can see that the verification is fast and is related only to the number of query-related fragments. Figure 13 shows the VO size for the different query types. It shows that for both Merkle and data aggregation proofs, the complex queries have the largest VOS because they have the highest number of fragments.

10 CONCLUSION

In this paper, we design, implement, and evaluate VeriDKG, which supports verifiable SPARQL query in Web 3.0. We design a new ADS called RGB-Trie for verifiable subgraph locating and combined the tree with cryptographic accumulators for verifiable aggregation for intermediate results. We also do extensive experiments to test the performance of VeriDKG, and the results show that VeriDKG implements verifiable SPARQL with a competitive query performance compared with the state-of-the-art. Compared with the leading-edge DKGs, VeriDKG reduces the index storage overhead by 97%. As for future work, we plan to extend VeriDKG to a privacy-preserving scenario and enable reliable semantic query when the data is encrypted.

ACKNOWLEDGMENTS

This research was supported by fundings from the Key-Area Research and Development Program of Guangdong Province (No. 2021B0101400003), the National Key Research and Development Program of China under Grant 2022YFB3102700, in part by the National Natural Science Foundation of China under Grant 62102295 and Grant 62202358, Hong Kong RGC Research Impact Fund (No. R5060-19, No. R5034-18), Areas of Excellence Scheme (AoE/E-601/22-R), General Research Fund (No. 152203/20E, 152244/21E, 152169/22E, 152228/23E), Shenzhen Science and Technology Innovation Commission (JCYJ20200109142008673). We also appreciate Alibaba Cloud Intelligent Computing LINGJUN to provide the powerful computation ability in the experiments.

REFERENCES

[1] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. 2017. Query optimizations over decentralized RDF graphs. In *Proc. of 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 139–142.

[2] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2019. A Decentralized Architecture for Sharing and Querying Semantic Data. In *Proc. of the European Semantic Web Conference (ESWC)*. 3–18.

[3] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2019. Decentralized Indexing over a Network of RDF Peers. In *Proc. of the International Semantic Web Conference (ISWC)*. 3–20.

[4] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2021. ColChain: Collaborative Linked Data Networks. In *Proc. of the Web Conference (WWW)*. 1385–1396.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proc. of the EuroSys Conference (EuroSys)*. Article 30, 15 pages.

[6] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable Data Possession at Untrusted Stores. In *Proc. of ACM conference on Computer and communications security (CCS)*. 598–609.

[7] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A Nucleus for a Web of Open Data. In *Journal of The semantic web*. 722–735.

[8] Amr Azzam, Christian Aebeloe, Gabriela Montoya, Ilkcan Keles, Axel Polleres, and Katja Hose. 2021. WiseKG: Balanced Access to Web Knowledge Graphs. In *Proc. of the Web Conference (WWW)*. 1422–1434.

[9] Debayan Banerjee, Pranav Ajit Nair, Jivat Neet Kaur, Ricardo Usbeck, and Chris Biemann. 2022. Modern Baselines for SPARQL Semantic Parsing. In *Proc. of ACM SIGIR*. 2260–2265.

[10] Juan Benet. 2014. Ipfs-content Addressed, Versioned, p2p File System. *arXiv preprint arXiv:1407.3561* (2014).

[11] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a Collaboratively Created Graph Database for Structuring Human Knowledge. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*. 1247–1250.

[12] Dan Boneh and Xavier Boyen. 2008. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *J. Cryptol.* 21, 2 (feb 2008), 149–177.

[13] Kevin D Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: A High-availability and Integrity Layer for Cloud Storage. In *Proc. of ACM conference on Computer and communications security (CCS)*. 187–198.

[14] Marco Brandizi, Ajit Singh, and Keywan Hassani-Pak. 2018. Getting the best of Linked Data and Property Graphs: rdf2neo and the KnetMiner use case.. In *SWAT4LS*.

[15] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL Web-querying Infrastructure: Ready for Action?. In *Proc. of the international Semantic Web Conference (ISWC)*. Springer, 277–293.

[16] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *Proc. of 2018 IEEE symposium on security and privacy (SP)*. 315–334.

[17] Min Cai and Martin Frank. 2004. RDFPeers: a Scalable Distributed RDF Repository Based on a Structured Peer-to-peer Network. In *Proc. of the Web Conference (WWW)*. 650–657.

[18] Ran Canetti, Omer Paneth, Dimitrios Papadopoulos, and Nikos Triandopoulos. 2014. Verifiable Set Operations over Outsourced Databases. In *Public-Key Cryptography – PKC 2014*, Hugo Krawczyk (Ed.). 113–130.

[19] Juan Cano-Benito, Andrea Cimmino, and Raúl García-Castro. 2019. Towards Blockchain and Semantic Web. In *Proc. of the international Conference on Business Information Systems*. Springer, 220–231.

[20] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[21] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *Proc. of OSDI*, Vol. 99. 173–186.

[22] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-tran: a high performance distributed graph database with a decentralized architecture. *Proc. of the VLDB Endowment* 15, 11 (2022), 2545–2558.

[23] Usman W Chohan. 2022. Web 3.0: The Future Architecture of the Internet? https://ssrn.com/abstract=4037693. *Available at SSRN* (2022).

[24] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*. 123–140.

[25] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A Shared Database on Blockchains. *Proc. of the VLDB Endowment* 12, 11, 1597–1609.

[26] Ethereum. 2013. Go Ethereum. Retrieved March 20, 2023 from https://github.com/ethereum/go-ethereum

[27] Ethereum. 2016. web3.js - Ethereum JavaScript API. Retrieved March 20, 2023 from https://web3js.readthedocs.io/en/v1.5.2/

[28] Nicholas L Farnan, Adam J Lee, Panos K Chrysanthis, and Ting Yu. 2014. PAQO: Preference-aware query optimization for decentralized database systems. In *Proc. of 2014 IEEE 30th International Conference on Data Engineering (ICDE)*. 424–435.

[29] Javier D Fernández, Miguel A Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF Representation for Publication and Exchange (HDT). *Journal of Web Semantics* 19 (2013), 22–41.

[30] Sébastien Ferré. [n.d.]. Expressive and Scalable Query-Based Faceted Search over SPARQL Endpoints. In *Proc. of ISWC*. 438–453.

[31] Wensheng Gan, Zhenqiang Ye, Shicheng Wan, and Philip S Yu. 2023. Web 3.0: The Future of Internet. In *Proc. of the Web Conference (WWW)*. 1266–1275.

[32] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. 2022. Hybrid Blockchain Database Systems: Design and Performance. *Proc. of the VLDB Endowment* 15, 5 (2022), 1092–1104.

[33] Mitchell L Gordon, Michelle S Lam, Joon Sung Park, Kayur Patel, Jeff Hancock, Tatsunori Hashimoto, and Michael S Bernstein. 2022. Jury learning: Integrating dissenting voices into machine learning models. In *Proc. of CHI Conference on Human Factors in Computing Systems*. 1–19.

[34] Herumi. 2020. High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves. Retrieved March 20, 2023 from https://github.com/herumi/ate-pairing

[35] Zicong Hong, Song Guo, Peng Li, and Wuhui Chen. 2021. Pyramid: A Layered Sharding Blockchain System. In *Proc. of IEEE INFOCOM*.

[36] Shagun Jhaver, Sucheta Ghoshal, Amy Bruckman, and Eric Gilbert. 2018. Online harassment and content moderation: The case of blocklists. *Journal of ACM Transactions on Computer-Human Interaction (TOCHI)* 25, 2 (2018), 1–33.

[37] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-end Integrity Protection for Web Applications. In *Proc. of 2016 IEEE Symposium on Security and Privacy (SP)*. 895–913.

[38] Lukas Klic. 2023. Linked Open Images: Visual similarity for the Semantic Web. *Journal of Semantic Web* 14, 2 (2023), 197–208.

[39] Ora Lassila, Ralph R Swick, et al. 1998. Resource Description Framework (RDF) Model and Syntax Specification. *W3C Recommendation* (1998).

[40] Zhuotao Liu, Yangxi Xiang, Jian Shi, Peng Gao, Haoyu Wang, Xusheng Xiao, Bihan Wen, Qi Li, and Yih-Chun Hu. 2021. Make Web3. 0 Connected. *Journal of IEEE Transactions on Dependable and Secure Computing* (2021).

[41] Ralph C Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Proc. of the conference on the theory and application of cryptographic techniques*. Springer, 369–378.

[42] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. 2006. Authentication and integrity in outsourced databases. *Journal of ACM Transactions on Storage (TOS)* 2, 2 (2006), 107–138.

[43] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-peer Electronic Cash System. *Decentralized Business Review* (2008), 21260.

[44] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. 2019. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proc. of the VLDB Endowment* 12, 11, 1539–1552.

[45] Lan Nguyen. 2005. Accumulators from Bilinear Pairings and Applications. In *Proc. of the 2005 International Conference on Topics in Cryptology (CT-RSA)*. 275–292.

[46] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2011. Optimal Verification of Operations on Dynamic Sets. In *Advances in Cryptology – CRYPTO 2011*, Phillip Rogaway (Ed.). 91–110.

[47] OriginTrail Parachain. 2022. OriginTrail Ecosystem White Paper 2.0. Retrieved March 20, 2023 from https://parachain.origintrail.io/whitepaper

[48] Qingqi Pei, Enyuan Zhou, Yang Xiao, Deyu Zhang, and Dongxiao Zhao. 2020. An Efficient Query Scheme for Hybrid Storage Blockchains Based on Merkle Semantic Trie. In *Proc. of the International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 51–60.

[49] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. FalconDB: Blockchain-based Collaborative Database. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*. 637–652.

[50] Eric Prudhommeaux. 2008. SPARQL Query Language for RDF. Retrieved March 20, 2023 from http://www.w3.org/TR/rdf-sparql-query/

[51] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. 2021. Blockchains vs. distributed databases: Dichotomy and fusion. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*. 1504–1517.

[52] Muhammad Saleem, Ali Hasnain, and Axel-Cyrille Ngonga Ngomo. 2018. Largerdfbench: a Billion Triples Benchmark for sparql Endpoint Federation. *Journal of Web Semantics* 48 (2018), 85–125.

[53] Adi Shamir. 1979. How to share a secret. *Journal of Communications of the ACM* 22, 11 (1979), 612–613.

[54] Dan Sheridan, James Harris, Frank Wear, Jerry Cowell Jr, Easton Wong, and Abbas Yazdinejad. 2022. Web3 Challenges and Opportunities for the Market. *arXiv preprint arXiv:2209.02446* (2022).

[55] Mirek Sopek, Przemyslaw Gradzki, Witold Kosowski, Dominik Kuziski, Rafa Trójczak, and Robert Trypuz. 2018. GraphChain: a distributed database with explicit semantics and chained RDF graphs. In *Proc. of the Web Conference (WWW)*. 1171–1178.

[56] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a Core of Semantic Knowledge. In *Proc. of the Web Conference (WWW)*. 697–706.

[57] Roberto Tamassia. 2003. Authenticated data structures. In *Proc. of European symposium on algorithms*. 2–5.

[58] Kristen Vaccaro, Ziang Xiao, Kevin Hamilton, and Karrie Karahalios. 2021. Contestability For Content Moderation. *Proc. of the ACM on Human-Computer Interaction* 5, CSCW2 (2021), 1–28.

[59] VMvare. 2023. Spring boot. Retrieved March 20, 2023 from https://spring.io/projects/spring-boot

[60] Shicheng Wan, Hong Lin, Wensheng Gan, Jiahui Chen, and Philip S Yu. 2023. Web3: The Next Internet Revolution. *arXiv preprint arXiv:2304.06111* (2023).

[61] Haixin Wang, Cheng Xu, Ce Zhang, Jianliang Xu, Zhe Peng, and Jian Pei. 2022. vChain+: Optimizing Verifiable Blockchain Boolean Range Queries. In *Proc. of IEEE International Conference on Data Engineering (ICDE)*.

[62] Shuai Wang, Chenchen Huang, Juanjuan Li, Yong Yuan, and Fei-Yue Wang. 2019. Decentralized Construction of Knowledge Graphs for Deep Recommender Systems Based on Blockchain-powered Smart Contracts. *Journal of IEEE Access* 7 (2019), 136951–136961.

[63] Gavin Wood et al. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[64] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. 2019. Servedb: Secure, verifiable, and efficient range queries on outsourced database. In *Proc. of 2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 626–637.

[65] Min Xie, Haixun Wang, Jian Yin, and Xiaofeng Meng. 2007. Integrity Auditing of Outsourced Data.. In *Proc. of the VLDB Endowment*, Vol. 7. 782–793.

[66] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vchain: Enabling Verifiable Boolean Range Queries over Blockchain Databases. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*. 141–158.

[67] Sean Yang and Max Li. 2023. Web3. 0 Data Infrastructure: Challenges and Opportunities. *Journal of IEEE Network* 37, 1 (2023), 4–5.

[68] Cong Yue, Tien Tuan Anh Dinh, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Xiaokui Xiao. 2023. GlassDB: An Efficient Verifiable Ledger Database System Through Transparency. *Proc. of the VLDB Endowment* 16, 6 (2023).

[69] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *Proc. of the VLDB Endowment* 6, 4, 265–276.

[70] Ce Zhang, Cheng Xu, Haixin Wang, Jianliang Xu, and Byron Choi. 2021. Authenticated Keyword Search in Scalable Hybrid-storage Blockchains. In *Proc. of IEEE International Conference on Data Engineering (ICDE)*. 996–1007.

[71] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. 2019. GEM2-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain. In *Proc. of IEEE International Conference on Data Engineering (ICDE)*. 842–853.

[72] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. 2020. Spitz: a verifiable Database System. *Proc. of the VLDB Endowment* 13, 12 (2020), 3449–3460.

[73] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *Proc. of IEEE Symposium on Security and Privacy (SP)*. 863–880.

[74] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for outsourced databases. In *Proc. of ACM Computer and Communications Security (CCS)*. 1480–1491.

[75] Chris Liu Ziliang Lai and Eric Lo. 2023. When Private Blockchain Meets Deterministic Database. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*.