Pistis: A Decentralized Knowledge Graph Platform Enabling Ownership-Preserving SPARQL Querying

Enyuan Zhou
PolyU SZRI
21038299r@connect.polyu.hk

Song Guo
HKUST
songguo@cse.ust.hk

Zicong Hong
HKUST
congcong@ust.hk

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

Yang Xiao
Xidian University
yxiao@xidian.edu.cn

Jinwen Liang
HK PolyU
jinwen.liang@polyu.edu.hk

Dalin Zhang
Aalborg University
dalinz@cs.aau.dk

ABSTRACT

Decentralized Knowledge Graph (DKG) platforms allow the sharing of knowledge with multiple owners. While data owners can share their data with others by encrypting their data before sharing it, this naïve approach prevents data encrypted by different owners from being queried together, as it compromises query verifiability, an essential DKG platform feature. We propose Pistis, the first DKG platform capable of preserving ownership while also enabling verifiable SPARQL queries. Two novel techniques facilitate this: *owner-managed end-to-end encryption* and *collaborative query verification*. In Pistis, data owners thus encrypt their data individually and collaborate to construct an authenticated data structure (ADS) with a global key by means of secret sharing and secure multi-party computation. Then, by indexing KG data as ciphertext over the ADS, Pistis offers a cryptographic scheme called VO-SPARQL that facilitates verifiable queries on encrypted KG data with multiple owners. Pistis provides succinct proofs for two-stage SPARQL queries, including subgraph queries based on the ADS and aggregation on encrypted intermediate results based on a key-aggregate cryptographic primitive. A theoretical analysis and an empirical study provide detailed insight into the performance of Pistis while offering provable security.

PVLDB Reference Format:
Enyuan Zhou, Song Guo, Zicong Hong, Christian S. Jensen, Yang Xiao, Jinwen Liang, and Dalin Zhang. Pistis: A Decentralized Knowledge Graph Platform Enabling Ownership-Preserving SPARQL Querying. PVLDB, 18(11): 4602 - 4615, 2025.
doi:10.14778/3749646.3749716

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

1 INTRODUCTION

As one of the bold and innovative aspects of Web 3.0 [30, 46, 76, 87], decentralized knowledge graph (DKG) platforms represent a new knowledge management technology that offers global platforms where everyone can share, manage, and exploit knowledge from diverse data sources on the Web [3–5, 52, 88]. Moreover, DKG platforms promise high data availability and resilience, as their data is distributed across multiple nodes rather than being centralized in a single location [5, 21, 32]. However, the decentralized nature of DKG platforms expose them to malicious threats, especially *Byzantine Attacks* [12, 24, 74] and *Loss of Ownership* [27, 34, 65]. Such attacks can compromise the reliability of DKG platforms and call for robust defense strategies.

A DKG platform must support three key functionalities, *data integrity*, *query verifiability*, and *data ownership*, which are crucial in outsourced database scenarios [35, 49, 71]. Specifically, data integrity ensures that the data stored in a platform is not manipulated or modified without authorization [13, 37]. Query verifiability ensures that query results can be verified as correct, complete, and fresh, preventing malicious query executions that could compromise the integrity of a platform [42, 44, 71, 72, 84–86]. Data ownership affords data owners complete control over their KG data, including the ability to decide with whom to share raw data (e.g., patients may share their data with pharmacists) [7, 35, 58, 64, 69]. Offering this functionality is essential to ensure DKG platforms' operation and facilitate their wider adoption.

Motivation. Real-world applications increasingly require decentralized knowledge collaboration with both verifiable query results and data ownership. In **global supply chains**, consider a DKG shared by Unilever (manufacturer), Maersk (logistics), and Walmart (retailer). Unilever shares RDF-encoded product batches with supplier relationships, while Maersk provides shipping record triples with environmental data. Walmart performs KG-backed authenticity checks. Without data ownership, Maersk could exploit property graph patterns to access Unilever's manufacturing knowledge to gain unfair advantages in pricing and negotiations. Without query verifiability, Maersk could falsify edge attributes (e.g., temperature data or routes), compromising Walmart's quality checks and potentially allowing unsafe products to reach consumers. Similar challenges emerge in **Industry 4.0** where factories must share machine data without revealing proprietary processes, and in **decentralized science** where researchers need to validate results while protecting patient data.

Most existing DKG systems, such as RDFPeers [21] and PIC-NIQ [4], primarily focus on improving query efficiency, while overlooking the essential requirements of data integrity, query verifiability, and data ownership protection. Some emerging DKG platforms adopt blockchain technology to ensure data integrity and query

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749716

verifiability [5, 52, 88]. For example, ColChain [5] adopts blockchain to allow all nodes to collaborate on data updates and maintain a trusted historical record of all DKG data, which ensures data integrity. VeriDKG [88] provides query verifiability of DKG through blockchain consensus-building indexes and cryptographic hashing. This allows participants to confirm that the query results they receive are accurate and have not been tampered with. However, enabling this functionality jeopardizes data ownership, as most blockchain systems store data in plaintext. This gives every DKG platform participant permanent access to the data, making it impossible for data owners to maintain control over their sensitive information. Existing DKGs cannot guarantee query verifiability and data ownership simultaneously due to the following dilemma.

The Query Verifiability versus Data Ownership dilemma. A straightforward way to enable data ownership is for owners to encrypt their data locally before sharing it on the blockchain. This ensures only those with decryption keys can access the data, preserving ownership. However, while effective for ownership, this strategy introduces a challenge: it limits others' ability to interact with the data. In a DKG platform, where data is interconnected, encryption by multiple owners prevents others from reading raw data or executing queries across datasets. This poses a problem for applications requiring verifiable queries over a multi-owner encrypted KG, as encryption obstructs such verification.

Contributions. We propose Pistis, the first DKG platform ensuring data integrity, ownership, and query verifiability via two innovations: *owner-managed end-to-end encryption* and *collaborative query-verification*, which we integrate into blockchain-based DKG management. Specifically, data owners each encrypt their data independently and then collaborate to build a blockchain-maintained authenticated data structure (ADS) to share their ciphertext in the DKG. The ADS is a structured encrypted Merkle trie managed by a global key via a new orchestration of secret sharing and secure multi-party computation. Using this ADS as an index, any participant can query KG data in ciphertext. Data owner can determine to whom to disclose their raw data in query results by managing a private key, thus ensuring data ownership. Next, to perform SPARQL queries (i.e., standard KG queries) on the multi-owner encrypted KG data while offering query verifiability, we propose a new cryptographic scheme, VO-SPARQL (i.e., $\underline{V}$erifiable and $\underline{O}$wnership-preserving SPARQL), that facilitates succinct verification proofs for two-stage SPARQL queries, including subgraph queries based on the ADS and aggregation queries on encrypted intermediate results based on a key-aggregate cryptographic primitive. Our contributions are summarized as follows.

- We present Pistis, the first DKG platform to preserve data ownership and support verifiable SPARQL queries. Through its new VO-SPARQL cryptographic scheme that encompasses two novel techniques, Pistis supports SPARQL query results with succinct verification proofs and offers data ownership guarantees.
- We present an owner-managed end-to-end encryption scheme with a new ADS called encrypted Merkle semantic trie (EMST). This scheme allows data owners to encrypt data it enables to prevent loss of control, and it enables subgraph querying for the data encrypted by different owners, enabling data ownership.
- We present a collaborative query verification scheme called VO-SPARQL that enables SPARQL query verifiability in DKG through two steps while ensuring data ownership. VO-SPARQL includes EMST's workflow (i.e., initialization, update, and query) and a key-aggregate cryptographic primitive allowing one party to perform verifiable data aggregation on encrypted intermediate results queried from the EMST to obtain final results.
- We present a comprehensive security analysis, and we report on an empirical performance study of a Pistis prototype implementation on the widely-used benchmark largeRDFBench. The results show that Pistis is successful at enabling the targeted functionalities with practical performance compared to existing approaches.

2 RELATED WORK

2.1 Decentralized Knowledge Graph

As web content grows, enabling users to effectively access and navigate structured knowledge is increasingly important. KGs represent real-world facts as graphs, with nodes as entities and edges as semantic relationships. Most KGs remain centralized, relying on SPARQL endpoints [14, 39], limiting users to data consumption and hindering broader knowledge sharing. Collaborative platforms like Wikidata [67] support public editing but still store structured data centrally, risking unavailability, tampering, and single points of failure. To improve availability and query performance, prior work [4, 6, 21, 56, 59, 75] explores distributed storage and indexing. RDFPeers [21] and PIQNIC [4] use distributed hash tables and Bloom filter-based indexes for source selection. Other approaches integrate blockchain to enhance trust and traceability: ColChain [5] logs RDF updates in blockchain shards for tamper detection, while VeriDKG [88] builds ADS on blockchain to support verifiable querying. While VeriDKG enables query verifiability, it still leaves room for attackers to compromise data ownership. Decentralized platforms like Solid [63] and ActivityPub [2] promote user-centric data ownership and interoperability but lack support for verifiable querying. Our work addresses these gaps, resolving the core dilemma between query verifiability and data ownership in DKGs.

2.2 Verifiable Query Processing

Numerous studies have explored verifiable query processing in outsourced databases [11, 77, 81, 85, 86, 89]. IntegriDB [86] uses ADS to verify SQL query integrity, while vSQL [85] applies interactive proof systems for dynamic query verification. However, these systems incur significant cryptographic overhead. VeriDB [89] relies on trusted hardware (e.g., Intel SGX) for verification, which is impractical in a decentralized scenario. SQL Ledger [11], LedgerDB [77] and GlassDB [81] achieve verifiable query processing by maintaining tamper-evident logs to trace database updates. However, they target specific data formats and do not support the rich semantics and interlinked structures inherent to KGs. Blockchain and ADS-based approaches like vChain [72] and vChain+ [68] support verifiable keyword and SQL queries with integrity proofs. Li *et al.* [43, 44] propose verifiable queries over graph structures via blockchain, but do not address semantic richness or data ownership. In summary, prior work lacks support for verifiable queries over semantically rich, decentralized KGs with ownership control—challenges that Pistis is designed to meet.

3 PRELIMINARIES

3.1 Knowledge Graph

The Resource Description Framework (RDF) is the standard format for KGs on the web. An RDF graph consists of a set of RDF triples [9, 55], and a KG $\mathcal{G}$ is often represented by an RDF graph, where nodes are entities and directed edges are relations. Each RDF triple consists of a subject, a predicate, and an object.

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

For example, (*Alice*, *worksAt*, *Meta*) and (*Bob*, *knows*, *Alice*) 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 $\mathcal{G}$. In this paper, all subgraphs of a KG are triple fragments.

The de facto query language for KGs is SPARQL [60], and each such query 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 $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.

A SPARQL query contains *basic graph patterns* (BGP) [75, 83], each of which consists of a set of (conjunctive) triple patterns combined with operators such as UNION or JOIN. For example, given two triple patterns (?*who*, *knows*, *Alice*) and (?*who*, *works at*, ?*address*), a SPARQL query that searches for the workplace of someone who knows Alice (i.e., inner join the two triple patterns on their subjects and select the results on the second triple pattern's object) can be formulated as follows.

SELECT ?*address* *WHERE* {
  (?*who*, *knows*, *Alice*), (?*who*, *worksAt*, ?*address*). }

Definition 4 (Triple Pattern Fragment)**.** Let f be a triple fragment and tp be a triple pattern. Then f is the triple pattern fragment of tp iff for every RDF triple $t \in f$, t matches tp.

Decentralized knowledge graph. In a traditional KG, the complete graph resides on a centralized and trusted server capable of executing SPARQL queries using its local storage. In contrast, in a DKG, the graph is partitioned into subgraphs that are distributed among DKG communities [1, 5, 57]. Each community comprises participants responsible for storing identical subgraphs. It is important to note that a participant can be part of multiple communities, while a community exclusively retains a specific subgraph.

3.2 Cryptographic Building Blocks

Our system leverages basic cryptographic primitives to encrypt data and ensure tamper-proof digests. To support secure SPARQL query execution, we combine secret sharing, secure multi-party computation, and structured encryption to construct a verifiable global index. We further implement verifiable set operations to prove the correctness of encrypted data aggregation. Blockchain acts as a decentralized ledger to record index updates and prevent data manipulation.

Basic cryptographic primitives. An asymmetric encryption scheme `PKE = (Gen, Enc, Dec)` consists of three algorithms: 1) `Gen` is a probabilistic algorithm whose input is a security parameter κ, and its output is a a key-pair (Pk, Sk), where Pk is a public key and Sk is a secret key; 2) `Enc` is a probabilistic algorithm that generates a ciphertext ct given a key Pk and a message m; 3) `Dec` is a deterministic algorithm that takes a key Sk and a ciphertext ct and returns m. We employ the widely used `RSA` algorithm as the asymmetric encryption building block. A pseudo-random function (PRF) F_K and a pseudo-random permutation (PRP) P_K are computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. A cryptographic hash function, $hash(\cdot)$, takes an arbitrary-length message m as input and outputs a fixed-length hash digest $hash(m)$, ensuring that $hash(m)$ is computationally indistinguishable from random data. A Merkle tree [47] uses cryptographic hashes to efficiently verify data integrity. Its extension, the Merkle trie, structures data by prefixes, enabling scalable verification in dynamic key-value stores [31, 82].

Structured encryption. Structured encryption [25, 36] is a generalization of index-based symmetric searchable encryption, which gives an idea of generalizing searchable encryption to arbitrarily-structured data. A semi-dynamic structured encryption scheme for a data structure DS is Σ_{DS} = `(Init, QueryToken, Query, AddToken, Add)` that contains five algorithms as follows.

- `Init`$(1^k) \rightarrow$ (`EDS`, K): The input is a security parameter 1^k. The output is an encrypted data structure `EDS` and a secret key K.
- `QueryToken`$(K, Q) \rightarrow$ `QTK`: The input is a secret key K and a query Q. The output is a query token.
- `Query(EDS, QTK)` $\rightarrow$ `ct`: The input is an encrypted data structure `EDS` and a query token `QTK`. The output is a ciphertext ct.
- `AddToken`$(K, it) \rightarrow$ `ATK`: The input is a secret key K and a new item it. The output is an add token.
- `Add(EDS, ATK)`: The input is an encrypted data structure `EDS` and an add token `ATK`. The output is an updated `EDS`.

Secret sharing. Secret sharing is an ideal scheme to outsource a secret by distributing it among a group. Each participant can only hold a part of the secret and all participants need to combine their respective sub-secrets to recover the original secret. In particular, a (t, n)-threshold secret sharing scheme [66] `SS = (Share, Recover)` contains two functions: 1) `SS.Share` shares the secret with n participants and sets a threshold t for secret recovery. 2) `SS.Recover` takes as input t out of n shares and outputs the secret. For example, $(K_1, K_2) \leftarrow$ `SS.Share`$(K, 2, 2)$ means the secret K is split into two shares K_1 and K_2, which must be combined as $K \leftarrow$ `SS.Recover`(K_1, K_2) to recover K.

Secure multi-party computation. Secure multi-party computation (MPC) [79] is a cryptographic framework that allows untrusted parties to perform computations jointly without revealing any input data. Two-party secure computation (2PC) is special case of MPC that allows two parties to jointly compute an arbitrary function on their input without sharing their input with the opposing party. We use $\mathcal{F}_{2PC}^{f}$ to represent 2PC, where f is a function that takes multiple inputs. We only consider the semi-honest attacks in

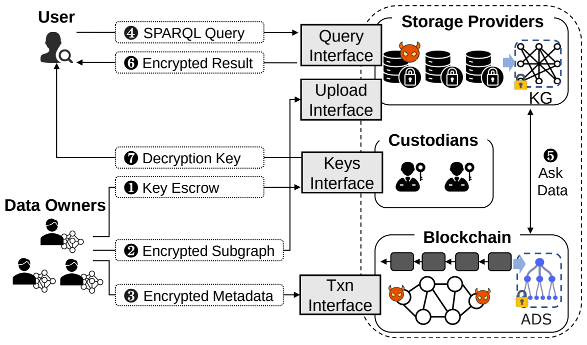


Figure 1: Architecture overview of Pistis.

2PC. Thus, ideal functionalities can be instantiated with standard semi-honest protocols, such as ABY [26] and MP-SPDZ [38]. For example, the process of using 2PC to generate and then split a key K between two participants can be represented as $(K_1, K_2) \leftarrow \mathcal{F}^{f}_{2PC}(r_1, r_2)$, where r_1 and r_2 are secrets held separately by the participants, $K = (r_1 \oplus r_2)$, and $f(r_1, r_2) = \texttt{SS.Share}((r_1 \oplus r_2), 2, 2)$.

Verifiable set operation. A verifiable set operation (VSO) [23, 53] transforms a set X into a fixed-size digest $acc(X)$, allowing users to delegate set computation tasks to an untrusted third party. This includes operations such as intersection ($\cap$), union ($\cup$), and difference ($\setminus$). These set operations can be executed in a nested manner and can be validated using the accumulated values of the input sets. A VSO includes four steps:

- $\texttt{KeyGen}(s) \longrightarrow (Sk, Pk)$: The input is a random value $s \in \mathbb{Z}_p$. The output is 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.
- $\texttt{Setup}(X,\ Pk) \longrightarrow acc(X)$: The input is a set $X \subset \mathbb{Z}_p$ and Pk. The output is an accumulated value $acc(X)$.
- $\texttt{Getproof}(X_i, X_j, OP, Pk) \longrightarrow (X^*, \pi)$: The input is two sets X_i and X_j, a set operation $OP \in \{\cap, \cup, \setminus\}$, and Pk. The function returns the set operation result of these two sets X^* with a proof π.
- $\texttt{Verify}\ (acc(X^i), acc(X^j), \pi) \longrightarrow \{accept, reject\}$: The input is two accumulated values $acc(X^i)$ and $acc(X^j)$ and a proof π. The function returns the validation result.

The unforgeability of VSO has been proved under the q-Strong Bilinear Diffie-Hellman (q-SBDH) assumption [18].

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 untrusted nodes [45, 61]. To agree on the order of valid transactions among nodes, the nodes employ a consensus mechanism (e.g., Proof of Work [50], Proof of Stake [70], and Practical Byzantine Fault Tolerance [10]) 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 PISTIS: OVERVIEW

4.1 System Model

As shown in Figure 1, Pistis provides KG data management services to two types of participants as follows.

Data owners generate and share their RDF data as triple fragments, collectively forming a KG. They outsource data to Pistis due to resource limitations, but they still need to control their raw data and decide what to disclose. **Users** access the KG by submitting SPARQL queries to Pistis, which involve triple fragments from one or more data owners.

To meet the demand of data owners and users, Pistis is a DKG comprised of the following three roles of nodes.

Blockchain plays the role of a trust anchor in Pistis by building a public and immutable ledger via consensus. The ledger records data owners' RDF data metadata for a global index used to query and authenticate. **Storage providers** involve a distributed data storage protocol similar to IPFS [16]. They can provide efficient, usable, and cheap off-chain storage. Each storage provider stores a subgraph of the KG, and an RDF triple of a subgraph can be backed up to multiple storage providers with an address that can locate it. Storage providers can collaboratively execute a SPARQL query on multiple subgraphs. **Custodians** are responsible for managing secret keys and generating query-relevant tokens in Pistis. The number of custodians depends on the choice of MPC adopted in Pistis. In the following, we consider the case of 2PC, thus the system will set up two custodians. The custodian selection policy will be discussed in Section § 7.

4.2 Threat Model

In Pistis, the data owners and users are honest. The custodians are semi-honest, do not collude with each other and strictly follow the protocol's instructions (two non-colluding and semi-honest servers is a common principle in cryptographic protocols and security systems, particularly in the decentralized scenario [19, 36]). Blockchain nodes and storage providers can be malicious for various reasons, such as program glitches, security vulnerabilities, and commercial interests. The proportion of malicious blockchain nodes will not exceed the fault threshold of the blockchain (e.g., 1/2 in Proof of Work or 1/3 in Practical Byzantine Fault Tolerance). Moreover, to ensure data availability, for each RDF triple, at least one storage node storing it is honest [5, 41, 54, 88]. Each participant does not maliciously communicate with the others in violation of the peer-to-peer network. Each adversary is computationally bounded and cannot break standard cryptographic primitives, e.g., finding hash collisions or forging digital signatures.

There are two types of adversarial attacks in Pistis: (*i*) ***Data breaches*** [65, 69]: the semi-honest custodians, malicious blockchain nodes, or malicious storage providers may try to independently infer or learn sensitive information about data owners' data without authorization due to various interests. (*ii*) ***Data tampering*** [33, 82]: the malicious blockchain nodes or storage providers can launch data tampering attacks. They can behave arbitrarily, e.g., forge or tamper with their local data and query results, or provide outdated information. It is a stronger adversarial attack than data breaches.

4.3 Workflow

The workflow of Pistis relies on two key designs, including an encrypted Merkle semantic trie (EMST, refer to Section § 5.1), and a verifiable and ownership-preserving SPARQL query scheme (refer to Section § 5.2). As shown in Figure 1, Pistis consists of the following five phases.

Phase 1: *Initialization.* Each data owner generates a pair of public and private keys to encrypt and decrypt its own RDF data, respectively. After that, it uses secret sharing to share its private key with the two custodians through the keys interface, and the custodians collaboratively generate a global key used to build the EMST later (Figure 1-❶).

Phase 2: *Data outsourcing.* A data owner uses its private key to encrypt its data and then outsources the encrypted data to storage providers through the upload interface (Figure 1-❷).

Phase 3: *Index updating.* When a data owner outsources its encrypted data to the storage providers, it packs some metadata (e.g., the hash and the address) of the encrypted data into a blockchain transaction. The transaction will be submitted through the transaction (Txn) interface and will be committed to the blockchain (Figure 1-❸). The blockchain nodes then update the on-chain global index maintained according to the transaction via a consensus.

Phase 4: *Query processing.* To query the DKG, a user can send a SPARQL query request to any storage provider through the query interface (Figure 1-❹). Next, the storage provider asks blockchain nodes to search for the addresses of the relevant data by the on-chain global index, and aggregate the relevant data (Figure 1-❺). Finally, the storage provider sends the combination of the final query results and their verification proofs to the user (Figure 1-❻).

Phase 5: *Verification.* After receiving the encrypted query results and the corresponding proofs, the user decrypts the results (Figure 1-❼) with the aid of the two custodians and verifies the results based on the proof provided by the blockchain nodes and storage providers.

4.4 Design Goals

Pistis should meet all of the following requirements.

Data integrity. Pistis should ensure that the data outsourced to storage providers has not been tampered with, i.e., the system should provide a proof that $h' = h$, where h' is the data received by users, and h is the data provided by the data owner. **Data ownership.** Pistis should prevent data breaches attacks without the data owner's authorization, i.e., the user knows nothing about the raw data except the query result decrypted by the owner. Likewise, blockchain and storage nodes can only conduct authorized operations on sensitive data without knowing its content. **Query verifiability.** Pistis should prevent data tampering attacks, i.e., ensure the query results' *correctness* (i.e., none of the RDF triples returned as results have been tampered with), *completeness* (i.e., no valid result is missing from the query results), and *freshness* (i.e., the query results are based on the latest version of the DKG).

4.5 Usability

To support usability in a decentralized setting, Pistis provides a client-assisted key management process. Specifically, data owners generate a public/private key pair through a user-friendly local client (e.g., web-based application) that offers an intuitive interface similar to decentralized tools like MetaMask [48]. The client automatically handles key generation and offers simple backup options

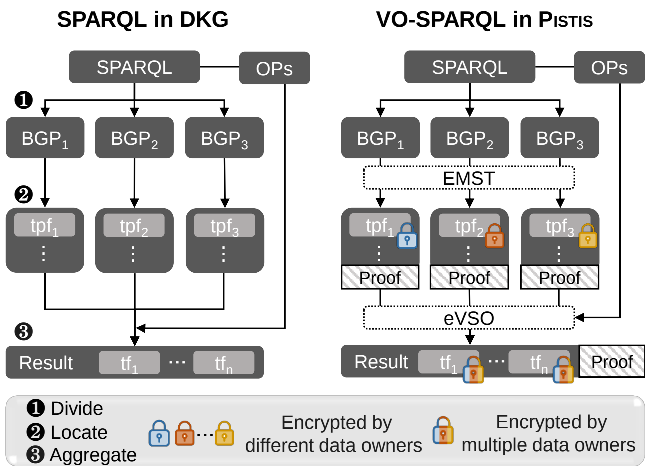


Figure 2: Comparison of SPARQL query in DKG and the verifiable and ownership-preserving SPARQL query in Pistis.

(e.g., mnemonic phrases), eliminating the need for cryptographic expertise. Authentication for subsequent interactions is performed through digital signatures using the owner's private key, enabling secure, decentralized identity verification. In addition, Pistis is compatible with existing KG platforms such as Apache Jena [29] and GraphDB [51], reusing their storage and query interfaces without modifying underlying data management or execution logic.

5 PISTIS: DETAILED DESCRIPTION

Roadmap. As shown in Figure 2, to process a SPARQL query in a DKG, a common workflow [3–5] consists of three steps: 1) dividing the SPARQL query into multiple BGPs, 2) locating the relevant triple pattern fragments matching each BGP, and 3) aggregating all triple pattern fragments with set operators (OPs) to get the final results. Following this workflow, to implement a verifiable and ownership-preserving SPARQL query in DKG, we propose a cryptographic scheme called VO-SPARQL (i.e., verifiable and ownership-preserving SPARQL) in Pistis. In particular, in the locating process, Pistis adopts a structured encryption-based new Merkle tree variant, EMST, and constructs it by blockchain consensus for verifiable and ownership-preserving triple pattern queries. In the aggregating process, Pistis uses an key-aggregate-based new VSO algorithm, eVSO (i.e., verifiable set operation for encrypted triples), for verifiable and ownership-preserving set operations between multiple triple pattern fragments encrypted by different data owners.

5.1 Encrypted Merkle Semantic Trie

Strawman. We first design a strawman ADS (i.e., a basic solution) with the ability of verifiable triple pattern queries. This structure extends a Merkle prefix tree (aka trie) by incorporating hashes of the last prefix bit and values in leaves, as well as hashes in internal nodes, enabling verifiable prefix matching. We call this structure the Merkle Semantic Trie (MST). An MST of depth d consists of three node types: 1) a root node containing the Merkle root hash, 2) $d - 2$ layers of branch nodes storing individual characters and child hashes, and 3) a layer of leaf nodes that store characters, their hashes, and pointers to RDF triple addresses that match the prefix from the root to the leaf.

Example. *Figure 3 (a) illustrates an example of an MST. Specifically, f_1, f_2, and f_3 are triple fragments, each containing RDF triples involving items aa, ab, and ba, respectively. For simplicity, each triple uses the same term for subject, predicate, and object: $f_1 = \{(aa, aa, aa)\}$, $f_2 = \{(ab, ab, ab)\}$, and $f_3 = \{(ba, ba, ba)\}$. Suppose a user issues a triple pattern query like $(?who,\ knows,\ aa)$, aiming to match any subject who knows aa. To verify whether a fragment f_1' returned by a storage provider corresponds to the correct triple pattern fragment (i.e., equals f_1), a user holding only the Merkle root h6 can recompute a candidate root h6′ using the hash of f_1' (denoted h1′) and the Merkle proof $\{h2, h5\}$. If $h6 = h6'$, the verification passes, confirming that the retrieved fragment matches the triple pattern.*

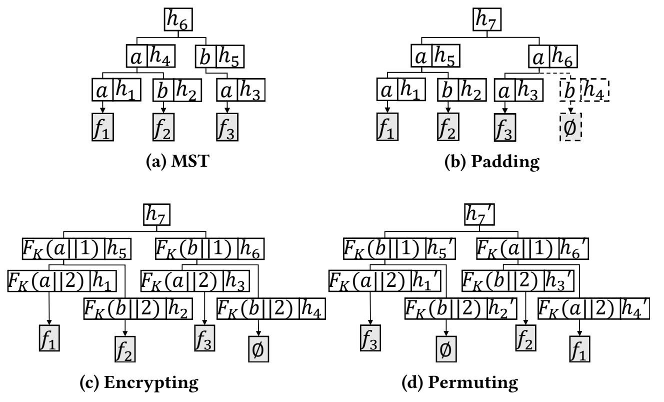
However, the strawman ADS only supports verifiable triple pattern queries over plaintext KG data—even if the index stores only result addresses, plaintext indexing and user queries can still compromise data ownership. A naive solution encrypts each node's content independently while keeping the MST structure unchanged; users then encrypt their query triple patterns with the same key and execute them over ciphertext. While this offers partial privacy, it still leaks access and search patterns (see § 6.1), including the number of children per node, identical ciphertexts for the same characters across layers, and consistent character ordering. For formal definitions of these leaks, see [25]. A fundamental idea is to design a new structured encryption scheme that transforms the MST into an EMST, protecting the above information while keeping the overall structure unchanged.

Challenge. Most of the structured encryption schemes [25, 36] break the correlation of messages and their ciphertext by inducing a random permutation between them to hide the part of access patterns. However, it is challenging to convert an MST to an EMST by this idea since inducing a random permutation for all the nodes in the whole MST will disrupt the tree structure and let it lose the prefix-matching functionality.

Design. Therefore, to convert an MST to an EMST, we need a new random permutation for the MST with its prefix-matching functionality guarantee. The key of the challenge to achieve it is to keep the connection relationship between different layers in case the node position of MST is disturbed. To overcome this challenge, we design a subtree-based random permutation (STRP) algorithm to convert MST to EMST.

a) *STRP algorithm.* The encryption is done by (1) padding the child nodes of each non-leaf node in an MST to be of the same length; (2) For each non-leaf node in the MST, encrypting the character of it using the output of a PRF; (3) For each non-leaf node in the MST, randomly permuting the location of its child nodes using a PRP. The purpose of step (1) is to help us hide the number of data items of non-leaf nodes' child nodes, and the purpose of steps (2) and (3) is to prevent index information leakage and hide the part of access patterns. The formal description of STRP is shown in two protocols AddToken and Add of the pseudo-code of Figure 4.

Example. *An example of the conversion process of EMST is shown in Figure 3. In Figure 3 (b), the dashed box represents the padding part of the unbalanced MST shown in Figure 3 (a). After the padding process, each non-leaf node of the MST has the same number of child nodes. Figure 3 (c) shows the encryption process of MST. In this process, the character with its level cascaded of each non-root node of the MST is encrypted by a PRF. The final EMST with the permutation operation completed is shown in Figure 3 (d).*



h_6
$a|h_4$ $b|h_5$
$a|h_1$ $b|h_2$ $a|h_3$
f_1 f_2 f_3
(a) MST
h_7
$a|h_5$ $a|h_6$
$a|h_1$ $b|h_2$ $a|h_3$ $b|h_4$
f_1 f_2 f_3 $\emptyset$
(b) Padding
h_7
$F_K(a||1)|h_5$ $F_K(b||1)|h_6$
$F_K(a||2)|h_1$ $F_K(a||2)|h_3$ $F_K(b||2)|h_2$ $F_K(b||2)|h_4$
f_1 f_2 f_3 $\emptyset$
(c) Encrypting
h_7'
$F_K(b||1)|h_5'$ $F_K(a||1)|h_6'$
$F_K(a||2)|h_1'$ $F_K(b||2)|h_3'$ $F_K(b||2)|h_2'$ $F_K(a||2)|h_4'$
f_3 $\emptyset$ f_2 f_1
(d) Permuting

Figure 3: The process to convert an MST to an EMST (ϕ is an empty triple fragment, $F_K()$ is a pseudo-random functions with a private key K, and $||$ is a cascading symbol).

b) *Operations for EMST.* As mentioned in Section § 4.3, the EMST will be updated after a series of transactions submitted by data owners when the metadata of their RDF data are committed to the blockchain. The update process involves four different operations on the EMST, including `Insert`, `Change`, `Delete`, and `Query`. In structured encryption, the user needs to use a token (add token or query token, refer to Section § 3.2) to manipulate a structure.

To simplify the presentation, we omit token generation when describing the following operations. Algorithm 1 details the `Insert` operation, where each component of an RDF triple I (subject, predicate, object) is individually encrypted with private key K, and each encrypted character is recursively inserted into the EMST. The triple's address *Cid* is then added to the leaf node's *CID* set. Operations `Change` and `Delete` follow a similar process. The main difference is that `Insert` may create a new path in EMST if the corresponding keyword is new, while `Change` and `Delete` operate on existing paths, updating or removing *Cid* from the leaf node.

Recall in Section § 4.3 the EMST can be used to find the relevant data (i.e., the intermediate results of a SPARQL query, which are some triple pattern fragments). Note that in Section § 5.2, the EMST can execute a set of triple patterns (i.e, a BGP) in batches. The pseudo-code of Algorithm 2 shows the detailed processing of the `Query` operation. In the `Query` operation, each prefix in the triple pattern is encrypted by the private key K first, and then is used to match a path in the EMST. If a triple pattern has more than one variable, the results R will contain multiple triple pattern fragments. Regardless of whether the match is successful or not (i.e., null result), the proof corresponding to the result will be returned. In addition, the freshness of Pistis is reflected in the fact that queries are always based on the latest EMST.

Example. *Figure 3 (d) shows the EMST before inserting the triple $(?who,\ worksWith,\ aa)$. To determine where to insert the triple, the data owner first encrypts each character of the object aa using a PRF:*

Algorithm 1: `Insert` operation of EMST

```
1 Function Insert (EMST, I, K, Cid):
      Input  : EMST root node root, the content of an RDF
               triple I, and a private key K
      Output : the root node root′ of a new state of EMST
2     $node \leftarrow root$;
3     foreach $item_i \in I$ do
4         foreach $c_j$ in $item_i$ do
5             $c'_j \leftarrow F_K(c_j \| j)$;
6             if $node.child[P_K(c'_j)] = null$ then
7                 $node.child[P_K(c'_j)] \leftarrow New(node)$;
8             $node \leftarrow node.child[P_K(c'_j)]$;
9         add $Cid$ to $node.CID$;
10        $node.hash \leftarrow hash(node.c'_j \| node.CID)$;
11        while $node\ != root$ do
12            $node.hash \leftarrow hash(node.c'_j \| node.child)$;
13            $node \leftarrow node.parent$;
14    return $node$;
```

Algorithm 2: `Query` operation of EMST

```
1 Function Query (EMST, tp, K):
      Input  : EMST root node root, a triple pattern tp, and a
               private key K
      Output : the query result R with an address set CID, the
               Merkle proof set Mproof of R
2     $node \leftarrow root$;
3     foreach $keyword_i \in tp$ do
4         foreach $character_j$ in $keyword_i$ do
5             $character'_j \leftarrow F_K(character_j \| j)$;
6             add $node.hash$ to $mproof$;
7             if $node.child[P_K(character'_j)] = null$ then
8                 add $\phi$ to R;
9                 add the Merkle proof $mproof$ to $Mproof$;
                  Break;
10            $node \leftarrow node.child[P_K(c'_j)]$;
11        add $CID$ to R;
12        add the Merkle proof $mproof$ to $Mproof$;
13    return R, $Mproof$;
```

$F_K(a\|1)$ and $F_K(a\|2)$. These encrypted values are used to traverse the EMST and locate the corresponding fragment f_1 before insertion. The complete triple is then added to this fragment, resulting in an updated fragment $f'_1 = f_1 \cup \{(?who,\ worksWith,\ aa)\}$. After insertion, the EMST is updated by recalculating hash values from the modified leaf node up to the root, preserving Merkle integrity.

5.2 VO-SPARQL Scheme

In the section, we describe the design of the VO-SPARQL scheme in our Pistis system and its usage. The VO-SPARQL scheme Ω = (`InitGlobal`, `OffchainStore`, `AddToken`, `Add`, `QueryToken`, `Query`, `Aggregate`, `Verify`) consists of eight protocols which we describe at a high-level below. The detailed pseudo-code is shown in Figure 4 and Figure 5.

- `InitGlobal`$_{C_1,C_2}(1^k, 1^k)$:
  1) $\mathbf{C}_1$ randomly samples $r_1 \overset{\$}{\leftarrow} \{0,1\}^k$ and $\mathbf{C}_2$ randomly samples $r_2 \overset{\$}{\leftarrow} \{0,1\}^k$;
  2) $\mathbf{C}_1$ and $\mathbf{C}_2$ execute $(K_1, K_2, \mathsf{EMST}) \leftarrow \mathcal{F}^f_{2PC}(r_1, r_2)$
     where $f(r_1, r_2)$:
     a) $K \leftarrow \Pi.Gen(1^k, r_1 \oplus r_2)$
     b) generate an empty MST and use K to encrypt MST, output EMST to $\mathbf{C}_1$
     c) $(K_1, K_2) \leftarrow \mathtt{SS.Share}(K, 2, 2)$
     d) output K_1 to $\mathbf{C}_1$ and output K_2 to $\mathbf{C}_2$
  3) $\mathbf{C}_1$ sends EMST to blockchain.
- `OffchainStore`$_{DO_i}(t)$:
  1) $\mathbf{DO}_i$ executes `RSA.Gen()` to generate a key-pair (Pk_i, Sk_i), broadcasts Pk_i, splits Sk_i by $(Sk_1, Sk_2) \leftarrow \mathtt{SS.Share}(Sk_i, 2, 2)$, and sends Sk_1 to $\mathbf{C}_1$ and Sk_2 to $\mathbf{C}_2$.
  2) When a data owner $\mathbf{DO}_i$ wants to outsource an item I with an RDF triple (s, p, o) to the storage providers, it uses Pk_i to encrypt all the three elements of the triple separately, and sends $hash(I)$ and $\{\mathtt{RSA.Enc}(Pk_i,\ I.s),\ \mathtt{RSA.Enc}(Pk_i,\ I.p),\ \mathtt{RSA.Enc}(Pk_i,\ I.o)\}$ to the storage providers.
  3) The storage providers return the storage address Cid to $\mathbf{DO}_i$.
- `AddToken`$_{C_1,C_2,DO_i}(K_1, K_2, I)$:
  1) $\mathbf{DO}_i$ parses I as $(I.s, I.p, I.o)$;
  2) for each member $I.m$ in I, do:
     a) $\mathbf{DO}_i$ computes $(p_1, p_2) \leftarrow \mathtt{SS.Share}(I.m, 2, 2)$ and sends p_1 to $\mathbf{C}_1$ and send p_2 to $\mathbf{C}_2$;
     b) $\mathbf{C}_1$ and $\mathbf{C}_2$ execute $\mathsf{ATK} \leftarrow \mathcal{F}^f_{2PC}(K_1, K_2, p_1, p_2)$
        where $f(K_1, K_2, p_1, p_2)$:
        $(I.m) \leftarrow \mathtt{SS.Recover}(p_1, p_2)$;
        for each character c_j in $I.m$, do:
           $c'_j \leftarrow F_K(c_j \| j)$ and add $c'_j \| i \| P_K(c_j)$ to atk_i
     c) add atk_i to ATK
  3) add Cid to ATK and sent ATK to $\mathbf{DO}_i$.
- `Add`$_{BN_l}$(EMST, ATK):
  1) $\mathbf{BN}_l$ parse ATK as $atk_1, atk_2, \ldots, atk_i$;
  2) Let $node$ = EMST.$root$. For each atk_i in ATK, do:
     a) let $j = 1$, $kw = c'_j$ in atk_i. $\mathbf{BN}_l$ do:
        a) if $node.child[P_K(c_j)] = null$,
        b) $node \leftarrow node.child[P_K(c_j)]$, and $i{+}{+}$
     b) $node.add$(ATK.Cid)
  3) $\mathbf{BN}_l$ broadcasts EMST to other blockchain nodes;

Figure 4: VO-SPARQL query scheme in Pistis (part 1).

Parties. Pistis involves: a large (dynamic) set of data owners $\mathbf{DO}_1, \ldots, \mathbf{DO}_\theta$, storage providers $\mathbf{SP}_1, \ldots, \mathbf{SP}_\tau$, blockchain nodes $\mathbf{BN}_1, \ldots, \mathbf{BN}_\rho$, two custodians $\mathbf{C}_1$ and $\mathbf{C}_2$, and a user $\mathbf{Q}$.

Initializing a global index. To initialize the system, the two custodians $\mathbf{C}_1$ and $\mathbf{C}_2$ execute Ω.`InitGlobal` to generate an empty EMST on the blockchain which we call the global index and provides each custodian with a share of the global key. In detail, $\mathbf{C}_1$ and $\mathbf{C}_2$ execute a 2PC function to 1) generate a global key K, 2) generate an empty MST and use K to encrypt MST to EMST , and 3) use a Shamir secret sharing scheme to distribute K to K_1 and K_2 and share them to $\mathbf{C}_1$ and $\mathbf{C}_2$ respectively. And then they send EMST to the blockchain. The global index can support prefix-based triple pattern queries for RDF triples and return their addresses in the storage providers. With the addresses, the storage providers can get

```
• $\texttt{QueryToken}_{C_1,C_2,Q}(K_1, K_2, \texttt{BGP})$:
    1) Q parses BGP as $\{tp_1, tp_2, \ldots, tp_\alpha\}$
    2) For each $tp_i$ in BGP, do:
        a) compute $(q_1, q_2) \leftarrow \texttt{SS.Share}(tp_i, 2, 2)$ and send $q_1$ to $C_1$ and send $q_2$ to $C_2$;
        b) $C_1$ and $C_2$ execute $\texttt{QTK} \leftarrow \mathcal{F}^f_{2PC}(K_1, K_2, q_1, q_2)$
            where $f(K_1, K_2, q_1, q_2)$:
            $tp_i \leftarrow \texttt{SS.Recover}(q_1, q_2)$;
            for each character $c_i$ in $tp_i.keyword$, do: \\ assume that there is one given keyword in $tp_i$
                $c'_i \leftarrow F_K(c_i\|i)$ and add $c'_i\|i\|P_K(c_i)$ to $qtk_i$
        c) add $qtk_i$ to QTK
    3) sent QTK to Q.
• $\texttt{Query}_{SP_l}(\texttt{EMST}, \texttt{QTK})$:
    1) $SP_l$ parse QTK as $qtk_1, qtk_2, \ldots, qtk_i$;
    2) Let $node = \texttt{EMST}.root$. For each $qtk$ in QTK, do:
        a) let $j = 1$, $kw = c'_j$ in $qtk_i$. $SP_i$ do:
            a) if $node.child[P_K(c_j)] = null$, break;
            b) $node \leftarrow node.child[P_K(c_j)]$, and $i++$
        b) add $node.CID$ and $mproof$ to R and $Mproof$
• $\texttt{Aggregate}_{SP_l}(\texttt{R})$:
    1) $SP_j$ parses $R.tpf$ as $\{tpf_1, tpf_2 \ldots, tpf_Y\}$ and extracts all public keys $\{Pk_1, Pk_2, \ldots, Pk_j\}$ from R;
    2) For each $tpf_i$ in R, $SP_j$ caculates $E(tpf_i) \leftarrow \texttt{RSA.Enc}(Pk_1, Pk_2, \ldots, Pk_\varphi, tpf_i)$ and add $E(tpf_i)$ to $ETPF$;
    3) $S \leftarrow$ aggregate(ETPF);
    4) $\pi \leftarrow$ prove(ETPF, $Pk_{SP_j}$);
    5) $SP_j$ sends $\{S, \pi, \texttt{R}, Mproof\}$ to Q
• $\texttt{Verify}_{C_1,C_2,Q}(S, \pi, \texttt{R}, Mproof)$:
    1) Q verifies R by its Merkle proof $Mproof$.
    2) For each $tpf_i$ in R, Q calculates $acc(tpf_i)$ and adds it to $acc$.
    3) Q computes $vr \leftarrow VerifyProof(acc, \pi)$ and verifies $vr$;
    4) $C_1$ and $C_2$ execute $I \leftarrow \mathcal{F}^f_{2PC}(\texttt{R}, Sk_1, Sk_2)$
        where $f(\texttt{R}, Sk_1, Sk_2)$:
        a) $Sk_i \leftarrow \texttt{SS.Recover}(Sk_1, Sk_2)$
        b) $I \leftarrow \texttt{RSA.Dec}(Sk_i, \texttt{R})$
```

Figure 5: VO-SPARQL query scheme in Pistis (part 2).

the ciphertext of relevant RDF triples and execute the aggregation processing to provide final query results to the user.

Adding a new data item. To add a new triple item I, data owner $\mathbf{DO}_i$ executes three protocols: Ω.OffchainStore with storage nodes **SP**, Ω.AddToken with custodians $\mathbf{C}_1$ and $\mathbf{C}_2$, and Ω.Add with blockchain nodes **BN**. First, $\mathbf{DO}_i$ runs Ω.OffchainStore to generate an RSA key pair (Pk_i, Sk_i), shares Sk_i with $\mathbf{C}_1$ and $\mathbf{C}_2$, broadcasts Pk_i, and uses it to encrypt each element (subject, predicate, object) of I. $\mathbf{DO}_i$ also signs the ciphertext using Sk_i and sends the ciphertext to **SP** to obtain its address Cid. Next, $\mathbf{DO}_i$ splits the index (prefix) of I into shares p_1 and p_2, sends them to $\mathbf{C}_1$ and $\mathbf{C}_2$, and requests them to run Ω.AddToken. Custodians then use 2PC to: 1) reconstruct the global key K; 2) recover the index from p_1 and p_2; and 3) use PRF and PRP to encrypt and permute keyword characters to form the add token ATK. Finally, all blockchain nodes **BN** run Ω.Add to update the on-chain global index EMST via consensus. Each node $\mathbf{BN}_l$ parses ATK into $\{atk_1, atk_2, \ldots, atk_i\}$, finds the insertion location in EMST for each atk, and adds its address, broadcasting the updated structure.

Querying a global index. When a user **Q** wants to query the DKG with a BGP = $\{tp_1, tp_2, \ldots, tp_\alpha\}$, it first executes Ω.QueryToken in conjunction with $\mathbf{C}_1$ and $\mathbf{C}_2$ to generate a query token. In detail, **Q** splits the BGP into two shares q_1 and q_2 and sends them to $\mathbf{C}_1$ and $\mathbf{C}_2$ separately. $\mathbf{C}_1$ and $\mathbf{C}_2$ use 2PC to securely compute a function that: 1) recovers the global key K from their key shares; 2) recovers the BGP from their pair shares q_1 and q_2; and 3) for each tp_i in BGP, uses PRF and PRP to encrypt and permute all characters of its prefixes, and add them to a query token QTK. Then **Q** sends QTK to a storage provider $\mathbf{SP}_l$ and asks it to execute Ω.Query protocol to search for some relevant triples through the on-chain global index. In detail, $\mathbf{SP}_l$ first parses QTK as $\{qtk_1, qtk_2, \ldots, qtk_\beta\}$, and then for each qtk in QTK, $\mathbf{SP}_l$ searches in EMST through any blockchain node by matching all characters of qtk with all layers of EMST. Finally, if the match is processed successfully, $\mathbf{SP}_l$ adds the addresses of query results of qtk and their relevant Merkle proofs into R and $Mproof$ separately. R and $Mproof$ will be sent to **Q** after the entire query process is over.

Aggregating intermediate results by eVSO. To get the final query results, $\mathbf{SP}_l$ executes the Ω.Aggregate protocol to aggregate the intermediate query results from the global index. This protocol contains an asymmetric key-aggregate-based VSO algorithm eVSO. In the process of eVSO, first, $\mathbf{SP}_l$ fetches and parses R as different triple pattern fragments $\{tpf_1, tpf_2, \ldots, tpf_Y\}$, check signatures of all $tpfs$, and extracts a public key set $Pks = \{Pk_1, Pk_2, \ldots, Pk_\varphi\}$ from R. Then, $\mathbf{SP}_l$ uses Pks to re-encrypt all items of all relevant triples in different fragments. Next, $\mathbf{SP}_l$ aggregates these encrypted triple pattern fragments (ETPF) by executing some set operations for them and gets the final query results S, and generates a verification proof of S. Finally, after the aggregation process is completed, $\mathbf{SP}_l$ sends S, R, and the verification proofs to **Q**.

Verifying query results. After receiving the final results, the user **Q** needs to execute the Ω.Verify protocol to verify the result in three steps. First, it uses the CID in R and their Merkle proofs $Mproof$ to verify whether the storage provider $\mathbf{SP}_l$ is correctly asking the blockchain nodes to perform the triple pattern query through the on-chain global index. Second, it uses the accumulated values and the proof of eVSO to verify whether the $\mathbf{SP}_j$ is correctly performing the aggregation processing on the intermediate query results. Third, $\mathbf{C}_1$ and $\mathbf{C}_2$ use 2PC to recover the Sk_i and decrypt the results, and send the raw data of the results to **Q**.

Cost Analysis. Here we give the time and space complexity of each function involved in VO-SPARQL. Ω.InitGlobal has $O(k)$ time and space complexity, where k is the length of the global key. Ω.OffchainStore has $O(1)$ time complexity and $O(k_R)$ space complexity, where k_R is the length of the RSA key. Ω.AddToken has $O(l)$ time and space complexity, where l is the length of characters in the triple. Ω.Add has $O(l)$ time complexity and $O(1)$ space complexity. Ω.QueryToken has $O(\alpha \times l_q)$ time and space complexity, where α is the number of triple patterns and l_q is the length of characters in the triple pattern. Ω.Query has $O(\alpha \times l_q)$ time complexity and $O(\alpha)$ space complexity. Ω.Aggregate has $O(\alpha \times \varphi)$ time complexity and $O(\alpha + \varphi)$ space complexity, where φ is the number of relevant owners of the query. Ω.Verify has $O(\alpha + \varphi)$ time complexity and $O(\alpha)$ space complexity.

Operations over BGPs. The above content describes how Pistis implements verifiable and ownership-preserving BGP-based SPARQL queries, while SPARQL also includes several operations over BGPs, such as property paths, named graphs, restrictions in the `FILTER` pattern, and solution sequence modifiers (e.g., `ORDER BY`, `OFFSET`, `DISTINCT`, `LIMIT`). To support broader SPARQL expressiveness under the constraints of verifiability and data ownership, Pistis introduces several additional designs, as follows:

To support named graphs and property paths, we propose prefix declaration and query decomposition methods to integrate them into VO-SPARQL. Named graphs use URIs (e.g., *xmlns.com/foaf/0.1/*) to identify triple scopes. To simplify syntax, we adopt W3C-style prefix declarations, e.g.: *PREFIX foaf : <xmlns.com/foaf/0.1/>* with triple patterns like: $(?person, foaf:name, ?name)$ A global named graph declaration document and its hash are stored at the storage providers and on-chain to ensure verifiability. For property paths, we decompose patterns with operators such as $*$, $?$, $|$, $\{\}$ into basic triples. For example: $(?x, knows|colleagueOf, ?y)$ is rewritten as: $(?x, knows, ?y)$ *UNION* $(?x, colleagueOf, ?y)$ enabling per-branch verifiability under VO-SPARQL.

To support `FILTER` restrictions, the storage provider can use hash functions, partial EMST paths, VSO, or zero-knowledge proofs (ZKPs) to prove constraint satisfaction. Matching or non-matching constraints are verified via hashes, regular expressions via VSO and EMST proofs, and range conditions via ZKPs such as Bulletproofs [20]. We empirically evaluate ZKP generation cost in the evaluation section. For the solution sequence modifiers, since the sorting criteria are given by the users, 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 users themselves.

6 SECURITY ANALYSIS

We first formalize and prove the security of our design in the ideal/real-world paradigm [22], and then give a verifiability analysis of queries in Pistis.

6.1 Ideal/real-world Paradigm

The ideal/real-world paradigm is used to define and prove the security of protocols by comparing two scenarios: the ideal world, where a trusted party ensures perfect security and minimal leakage, and the real world, where the protocol is actually implemented without such a trusted entity. The security of a protocol is established by demonstrating that an adversary cannot distinguish between interactions in the real world and those in the ideal world, ensuring that any potential information leakage in the real-world scenario is no greater than what is permissible in the idealized model.

We first define a leakage function $\mathcal{L}$ for Pistis, which describes the information revealed in the query process. The input of the query protocol is a KG $\mathcal{G}$ and a SPARQL query with a BGP. $\mathcal{L}(\mathcal{G}, \mathsf{BGP})$ is defined as follows:

Definition 5 ($\mathcal{L}(\mathcal{G}, \mathsf{BGP})$)**.** The leakage function $\mathcal{L}$ involves access pattern and search pattern.

- **Access pattern.** The access pattern describes the mapping between a submitted token and its corresponding encrypted RDF triples, potentially revealing results from previous queries.
- **Search pattern.** The search pattern reflects whether an token has been added or queried, based on differences between tokens.

$\mathcal{L}$ is always considered as default leaked information in searchable symmetric encryption [36], and the Adaptive $\mathcal{L}-security$ ensures that a query scheme leaks only a predefined amount of information. If the protocol in a system satisfies Adaptive $\mathcal{L}-security$, the data ownership in the system can be guaranteed. Adaptive $\mathcal{L}-security$ in Pistis can be defined as follows.

Definition 6 (Adaptive $\mathcal{L}-security$)**.** Let Ω = (`InitGlobal`, `OffchainStore`, `AddToken`, `Add`, `QueryToken`, `Query`, `Aggregate`, `Verify`) be a VO-SPARQL scheme. Let $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$ and $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ be an adversary and a simulator, respectively, where $q \in \mathbb{G}$. We define the $\mathbf{Real}_{\Omega}^{\mathcal{A}}(1^k)$ experiment and the $\mathbf{Ideal}_{\mathcal{L},\mathcal{S}}^{\mathcal{A}}(1^k)$ experiment as follows.

$\mathbf{Real}_{\Omega}^{\mathcal{A}}(1^k)$: In the real-world execution every party has access to ideal $\mathcal{F}_{2PC}$ functionalities. At round 0, $\mathbf{C}_1$ and $\mathbf{C}_2$ execute Ω.`InitGlobal` to generate an encrypted index EMST and send it to $\mathcal{A}$, and each data owner $\mathbf{DO}_i$ executes Ω.`OffchainStore` with $\mathbf{SP}$. Then, $\mathcal{A}$ adaptively chooses a polynomial number of commands $(comm_1, \ldots, comm_q)$ of the form $comm_r = (\mathbf{SP}_r, op_r)$, where op_r is either an add operation (Ω.`AddToken`) or a query operation (Ω.`QueryToken`). At round r ($1 \le r \le q$), $\mathbf{SP}_r$ executes op_r by Ω and sends the results to $\mathcal{A}_r$. After q round interactions, $\mathcal{A}$ produces a b bit message as the output.

$\mathbf{Ideal}_{\mathcal{L},\mathcal{S}}^{\mathcal{A}}(1^k)$: In the ideal-world execution every party has access to ideal Ω functionalities. At round 0, $\mathcal{S}_0$ randomly generates an index EMST* and an encrypted KG $\mathcal{G}^*$ by utilizing $\mathcal{L}(\mathcal{G}, \mathsf{BGP})$, and sends EMST* to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively chooses a polynomial number of commands $(comm_1, \ldots, comm_q)$ of the above form. At round r ($1 \le r \le q$), $\mathcal{S}_r$ reviews the previous requests and generates f_r adaptively. If op_j is an add, with $\mathcal{L}(\mathcal{G}, \mathsf{BGP})$, $\mathcal{S}_r$ generates an add token ATK^{r*} with $\mathbf{C}_1$ and $\mathbf{C}_2$ and sends ATK^{r*} to $\mathcal{A}_r$. After that, $\mathcal{A}_r$ updates EMST* by utilizing ATK^{r*}. If op_j is a query, with $\mathcal{L}(\mathcal{G}, \mathsf{BGP})$, $\mathcal{S}_r$ generates an appropriate query token QTK^{r*} with $\mathbf{C}_1$ and $\mathbf{C}_2$ and sends QTK^{r*} to $\mathcal{A}_r$. After that, $\mathcal{A}_r$ searches EMST* by utilizing QTK^{r*}. After q round interactions, $\mathcal{A}$ produces a b bit message as the output.

We say that Ω is adaptively $\mathcal{L}$-secure if for all probabilistic polynomial-time (PPT) semi-honest adversaries $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$, there exists a simulator $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ and a negligible function $negl(k)$ such that

$$\left| Pr\left[\boldsymbol{Real}_{\Omega}^{\mathcal{A}}(1^k) = 1\right] - Pr\left[\boldsymbol{Ideal}_{\mathcal{L},\mathcal{S}}^{\mathcal{A}}(1^k) = 1\right]\right| \le negl(k).$$

Theorem 6.1. *If* SS *and 2PC are secure, and if F and P are pseudo-random, then Ω is adaptively $\mathcal{L}-security$.*

Proof. We create a simulator $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ such that for an adversary $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$, the outputs of $\mathbf{Real}_{\Omega}^{\mathcal{A}}(1^k)$ and $\mathbf{Ideal}_{\mathcal{L},\mathcal{S}}^{\mathcal{A}}(1^k)$ are computationally indistinguishable. The simulator and adversary work as follow.

$\mathcal{S}_0$: $\mathcal{S}_0$ simulates $(K_1, K_2) \leftarrow$ `SS.Share`$(0^k, 2, 2)$ and $\mathcal{F}_{2PC}$, and sends K_1 to $\mathbf{C}_1$ and K_2 to $\mathbf{C}_2$. It then generates an empty EMST.

$\mathcal{S}_r$: For $1 \le r \le q$, if $comm_r$ is an add, $\mathcal{S}_r$ simulates $\mathcal{F}_{2PC}$ to generates an add token ATK^{r*}. For each element in ATK^{r*}, $\mathcal{S}_r$ sets its value as a random string $\{0,1\}^*$, whose length is the same as the output of F, and permutes their positions by P. Then $\mathcal{S}_r$ sends ATK^{r*} to $\mathcal{A}$. If $comm_r$ is a query, $\mathcal{S}_r$ first checks whether the query BGP_r has appeared before with the search pattern in $\mathcal{L}(\mathcal{G}, \mathsf{BGP})$. If it has appeared before, $\mathcal{S}_0$ searches the access pattern in $\mathcal{L}(\mathcal{G}, \mathsf{BGP})$ and gets the same QTK^{r*} that has used before. If is has not has appeared before, $\mathcal{S}_r$ simulates $\mathcal{F}_{2PC}$ to generate a query token QTK^{r*} in the same way as generating ATK and sends QTK^{r*} to $\mathcal{A}$.

Through the two experiments of real-world execution and ideal-world execution, $\mathcal{A}$ obtains some ATK^r and QTK^r from real-world execution and ATK^{r*} and QTK^{r*} from ideal-world execution. Since SS and 2PC are secure, and F and P are pseudo-random, with all but negligible probability, $\mathcal{A}$ cannot recover K_1 and K_2 to reproduce the process of $\mathcal{F}_{2PC}$, thus $\mathcal{A}$ cannot distinguish the values in ATK^{r*} and QTK^{r*} from that in ATK^r and QTK^r, respectively. Therefore, $\mathcal{A}$ cannot distinguish the output of $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{L},\mathcal{S}}(1^k)$ from $\mathbf{Real}^{\mathcal{A}}_{\Omega}(1^k)$ and the scheme Ω is adaptively $\mathcal{L} - security$.

□

6.2 Verifiability Analysis

Definition 7 (Query verifiability)**.** We say a SPARQL query is verifiable if the success probability of any polynomial-time adversary $\mathcal{A}$ is negligible in the following experiment:

- $\mathcal{A}$ selects a set of RDF triples $\mathcal{T}$;
- The EMST generate algorithm constructs an EMST and its digest EMST_{root} based on $\mathcal{T}$;
- $\mathcal{A}$ produces result R and VO_t for the SPARQL query 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**).

The above definition guarantees that the probability, for a malicious storage provider to convince the user with an incorrect, incomplete or outdated result, is negligible. Meanwhile, data integrity is also guaranteed because it can be represented as a query for a single piece of data.

Theorem 6.2. *Pistis is verifiable with respect to **Definition 7** if the cryptographic hash function is a pseudo-random function, the cryptographic accumulator is secure under the q-SBDH assumption, and the computing power of malicious nodes is less than 51% of the blockchain network.*

Proof. We intuitively prove Theorem 6.2 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 of Q. In this case, once t passed the verification of the user under the soundness in **Definition 7**, it means that the adversary can get two different triple pattern fragments with the same digest EMST_{root} of the ADS or the adversary can get two different 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 can pass the verification of the user under the completeness in **Definition 7**, it means that the adversary can get a triple pattern fragment that does not contain some matching triples and has the same digest EMST_{root} of the ADS with the genuine fragment or the adversary can get an incomplete set operation result with the same accumulator proof π of the genuine set operation 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 user under the freshness in **Definition 7**, it means that the adversary can get two different triple pattern fragments (i.e., a new and an old) with the same digest EMST_{root} of the ADS or the adversary can get two different set operation results with the same accumulator proof π.

However, all these three cases contradict two assumptions. The first is that the digest of the on-chain ADS EMST_{root} is generated by the cryptographic hash function, with all but negligible probability, the adversary can forge another fragment with the same hash value as the genuine fragment. The second assumption is the unforgeability for VSO, which has been proved to be held under the q-SBDH assumption [18].

□

7 DISCUSSION

Custodians. The selection of custodians is vital to the security of our protocol. Our scheme employs the standard Function Secret Sharing technique [19], integrated with MPC to manage the secret key. Function Secret Sharing necessitates that the participants in MPC are at least $k \geq 2$ non-collusive custodians. Consequently, Pistis requires a minimum of two custodians, implemented as 2PC. The number of custodians can be expanded by utilizing MPC instead of 2PC. For instance, data owners may choose reputable Certification Authorities (CAs), such as IdenTrust and DigiCert, to serve as custodians within Pistis. The competitive dynamics of the CA industry act as a deterrent to collusion, while their expertise in certificate management ensures they possess sufficient computational resources.

Data misuse and audit. While Pistis protects against data breaches and tampering, a key concern is the misuse of data collection within DKG. Authorized users may access substantial raw data and misuse it for data mining and recommendation systems. To mitigate this, we suggest limiting access frequency with owner-defined policies, establishing consensus-based rules, and employing technical solutions like smart contracts or attribute-based encryption (ABE) [17]. Additionally, some data owners may encrypt illegal content, making it difficult to assess legitimacy from ciphertext. Previous works have proposed using data deduplication to address this issue [15, 78].

8 EMPIRICAL EVALUATION

8.1 Experimental Setup

Hardware configuration. We run 8 data owner nodes, 16 blockchain nodes and 16 storage providers nodes on 16 64-bit Linux servers (Ubuntu 20.04) with Intel i9-11th CPU and 64GB memory. We set the bandwidth of connections between them to 20Mbps [5, 40, 73, 80].

Implementation environment. A prototype of Pistis is implemented in Java, C++, Go and JavaScript. The blockchain module is implemented based on Go-Ethereum [28] and the storage module is implemented based on IPFS [16]. The prototype has a user-server architecture that is implemented based on Spring Boot framework and the blockchain interfaces and requests are in the form of web3.js.

Cryptographic primitives. For all 2PCs, our prototype uses the ABY framework [26]. For the PRP, the prototype uses the AES algorithm, which is a popular symmetric key cryptography algorithm. For the PRF, the prototype uses the HMAC-SHA256 algorithm, which is a type of keyed hash algorithm constructed from the SHA-256 hash function. For MPC, our prototype uses JIFF library [8]. For secret sharing, our prototype instantiates a threshold secret sharing with Shamir secret sharing [66].

Datasets and benchmark. We evaluate the query performance of Pistis using the datasets and queries from largeRDFBench [62] benchmark, which is widely used by the DKG community. LargeRDFBench consists of 13 datasets and more than 1 billion triples in total. The largeRDFBench queries in our evaluation include simple (S), complex (C), and large data (L) categories. In detail, the S query has the lowest average number of triple patterns, at 4.3. C query has the highest number of triple patterns, at 11. L query has 6.2 triple patterns on average, and each triple pattern in L query corresponds to more than 1,000 query results. To fully assess query expressiveness, our evaluation includes not only BGPs but also full SPARQL features such as named graphs and filter expressions.

Metrics. We measure the following metrics of Pistis:

- Storage Cost (SC): The storage space size of the index.
- Token Generation Time (TGT): The amount of time it takes to generate an add token or query token.
- Item Add Time (IAT): The amount of time it takes to add a new item to Pistis, including adding to the storage network and blockchain.
- Query Execution Time (QET): The amount of time it takes to receive the full query results.
- Proof Generation Time (PGT): The amount of time it takes to generate the verification proof of query results.
- Verification Time and Verification Object Size (VT & VO): The time to verify query results and the proof size.

8.2 Experimental Results

8.2.1 Overall Comparison. A high-level overall comparison between our Pistis and other state-of-the-art DKG systems is shown in Table 1. In these three DKGs, Pistis is the only one that implements SPARQL queries with data integrity, query verifiability and data ownership. Through the VO-SPARQL scheme, Pistis can give data owners control over data that is outsourced to a decentralized storage system. Moreover, Pistis also achieves double verifiability of raw data and query results in a decentralized byzantine environment. For the storage cost of the index on 13 datasets from largeRDFBench with 1003960176 triples, the size of the index in **RDFPeers** is 105873.5 MBs, in **PIQNIC** and **Colchain** is 3195.2 MBs, in **VeriDKG** is 53.7MBs, while on Pistis is 159.9 MBs. The reason is that EMST compresses the index size by combining the same prefixes of keywords and Pistis needs a Merkle characteristic and prefix encryption to guarantee the query verifiability and data ownership. For the query execution time across the three query types, RDFPeers with a distributed hash table (DHT) have the lowest query time. VeriDKG exhibits verifiability at roughly a 6-fold increase in time cost compared to PIQNIC and Colchain. Meanwhile,

Table 1: Overall comparison of three different systems.

Schemes	Data integrity	Query verifiability	Data ownership	Storage cost of index (MB)	Query execution time (s)
RDFPeers [21]	✗	✗	✗	105873.5	0.90/8.3/274
PIQNIC [3]	✗	✗	✗	3195.2	0.08/0.7/24
Colchain [5]	✓	✗	✗	3195.2	0.08/0.7/25
VeriDKG [88]	✓	✓	✗	53.7	0.45/3.8/110
Pistis	✓	✓	✓	159.9	0.78/6.8/212

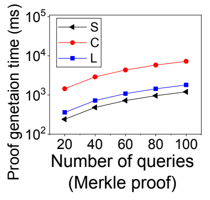
Pistis fulfills all three requirements with an approximately 10-fold increase in time cost over state-of-the-art unprotected systems but remains significantly more efficient than classic method RDFPeers.

8.2.2 Performance Evaluation.

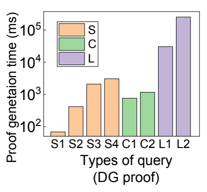
Token generation time. We evaluate the performance of the `AddToken` and `QueryToken` protocols by generating a series of add tokens from the dataset and some different query tokens of different types of queries in the benchmark, and testing their token generation time (TGT) respectively. We vary the size from 10 up to 10 million RDF triples and then test the TGT of them. The results demonstrate that generating an add token of a new triple is independent of the size of requests and takes about 24 milliseconds per triple. For the query token generation time, we set 1000 items for each type of KG query. In the results, generating a simple (S) query token and a large data (L) query token takes about 96ms and 144ms respectively, and a complex (C) query token needs about 240ms because it contains the largest number of triple patterns.

Add time. We evaluate the `Add` protocol by storing RDF triples with storage providers and submitting add tokens to the blockchain. Using all largeRDFBench datasets, each triple is stored on IPFS and packed into a transaction submitted to Ethereum. The item add time (IAT) consists of: 1) IPFS node synchronization and 2) Ethereum block confirmation. Storing a triple in IPFS takes 0.0188s (vs. 0.0031s in **PIQNIC**); block confirmation takes 7.112s (vs. 6.73s in **ColChain**), making it the dominant factor. Pistis thus incurs only modest overhead compared to systems without encryption. We also compare Ethereum and Pistis in terms of block confirmation efficiency: throughput is 15.2 transactions/s for Ethereum and 14.7 for Pistis, indicating the longer add time stems from Ethereum's inherent limits, not our encryption. A scalability test shows that increasing blockchain nodes from 16 to 200 raises block confirmation time from 6.803s to 10.078s—a 10x node increase results in only a 0.5x increase in time.

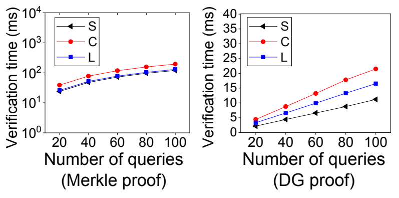
Query execution time. To evaluate the performance of the `Query` and `Aggregate` protocols of Pistis, we compare the query execution time (QET) of Pistis with two baseline systems, including an original Ethereum-based KG (OE-KG) that stores all encrypted RDF triples without any index and a variant Pistis system with a plaintext Merkle prefix tree (Pistis-P). For Pistis, the average QET for these three types of queries is 0.78s, 6.8s, and 212s, respectively. For Pistis-P, the average QET for these three types of queries is 0.63s, 6.5s, and 198s, respectively. For OE-KG, the average QET for these three types of queries is 2.1s, 14.2s, and 815s, respectively. By comparing the query performance of these systems, we can find that Pistis and Pistis-P have better performance than OE-KG because both of them have indexes. Besides, compared with Pistis-P, Pistis



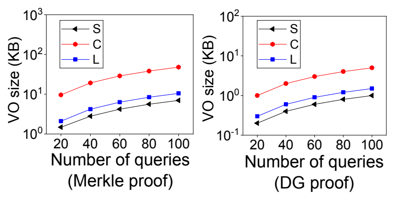
(a) PGT of Merkle proof.



(b) PGT of DG.



(c) Verification time.



(d) Verification object size.

Figure 6: Verification cost of Pistis.

achieves efficient query under ciphertext with small performance loss. With regard to the index size, it should be noted that OE-KE lacks an index. The index sizes of Pistis-p and Pistis are 97.3 MB and 159.9 MB, indicating that Pistis requires an additional 60% of space to store the encrypted index.

Verification cost. We evaluate the performance of the `Verify` protocol by executing SPARQL queries and measuring Proof Generation Time (PGT), Verification Object Size (VO), and Verification Time (VT). For PGT, we assess the Merkle proof generation time for three SPARQL query types, and aggregation proofs for four S queries, two C queries, and two L queries, each with 1000 items. As seen in Figure 6 (a), C queries have the longest PGT due to more query-related fragments. In Figure 6 (b), S and C queries generate aggregation proofs faster, while L queries exhibit longer PGT due to more intermediate results. Figure 6 (c) presents VT for Merkle and aggregation proofs, revealing that C queries have slightly longer VT. This is because all queries compute the same Merkle root hash, but complex queries process more fragment hashes. VT for data aggregation proof is rapid, primarily depending on the number of fragments. Finally, Figure 6 (d) displays the VO size for Merkle and aggregation proofs across different queries. Both figures indicate that complex queries produce the largest VO due to the highest number of fragments. For range conditions in `FILTER` operations, we use Bulletproofs to generate ZKPs, where the PGT, VO, and VT are approximately 30ms, 650bytes, and 5ms, respectively.

End-to-end evaluation. While Pistis preserves full SPARQL functionality, the protected scenario introduces additional overhead for cryptographic token generation, verifiable index traversal, proof construction, and result verification compared to traditional KGs. To provide a more comprehensive performance analysis of Pistis, we conduct an end-to-end evaluation, by simulating a federated scenario, incorporating multiple concurrent data updates and queries. Using predicate-based partitioning, we distributed 100,000 randomly selected triples from the largeRDFBench dataset across the data owner nodes. Each owner submits an update request (e.g., `Insert`, `Change`, or `Delete`) every 10 seconds, with batches of 50 triples per request. Simultaneously, 10 query users issue diverse SPARQL queries at a rate of one query every 5 seconds. The experimental results show that executing 200,000 update operations takes approximately 19,960 seconds, with an average operation time of 0.0998 seconds—comprising 0.024 seconds for token generation, 0.0188 seconds for storage, and 0.057 seconds for block confirmation. For query operations, 1,000 executions require 5,224 seconds, averaging 5.224 seconds per query. Compared to PIQNIC, Pistis adds 0.097s per update and 4.79s per query due to encrypted token generation and blockchain-based verifiability. The extra cost over VeriDKG is modest (0.009s/update, 1.13s/query), while offering ownership and full SPARQL support.

Query performance optimization. Compared to state-of-the-art systems, our solution incurs a 10× higher query overhead, with 73% stemming from proof generation, while query token generation and query processing contribute 12% and 15% respectively - making proof generation the primary bottleneck. To reduce the 10× query latency overhead introduced by verifiability, we explore three practical optimizations. First, **pipelining query and verification** allows asynchronous delivery of results and proofs, reducing perceived latency to 2.7×. Second, **parallel and incremental proof generation** improves end-to-end performance by splitting Merkle and `eVSO` operations across multiple threads, reducing latency to 4.1×. Finally, **reusable proof caching** amortizes repeated cryptographic computations, achieving up to 70% cost reduction on common query patterns. These techniques make verifiable query processing more practical in real-world DKG deployments.

9 CONCLUSION

In this paper, we have studied ownership-preserving SPARQL queries of DKG. To satisfy the technical requirements, we design, implement and evaluate Pistis, an end-to-end encrypted and collaboratively query-verifiable DKG platform with a new cryptographic scheme. The scheme relies on a novel ADS and a key-aggregate cryptographic primitive to query the multi-owner KG data in a verifiable and ownership-preserving manner. Security analysis with an idea/real-world paradigm and experimental evaluations prove the security and availability of our system. In particular, Pistis achieves new functionalities at an overhead of microsecond-level computation time, and kilobyte-level communication costs for a SPARQL query. Our future work will investigate the semantic queries for other types of data in Web 3.0, such as semi-structured data and multimedia data.

ACKNOWLEDGMENTS

This research was supported by fundings from the National Key R&D Program of China (No.2023YFB2703600), NSFC/RGC Collaborative Research Scheme (Grant No. 62461160332 & CRS_HKUST602/24), Hong Kong RGC General Research Fund (152169/22E, 152228/23E, 162161/24E), Research Impact Fund (No. R5060-19, No. R5011-23), Collaborative Research Fund (No. C1042-23GF), Areas of Excellence Scheme (AoE/E-601/22-R), and the InnoHK (HKGAI). We also appreciate the corresponding authors Zicong Hong, Yang Xiao and Dalin Zhang for their valuable guidance.

REFERENCES

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

[2] ActivityPub. 2025. ActivityPub. https://activitypub.rocks/. Accessed: 2025-05-01.

[3] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2019. A decentralized architecture for sharing and querying semantic data. In *Proc. of of the European Semantic Web Conference (ESWC)*. Springer, Springer, Portorož, Slovenia, 3–18.

[4] 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)*. Springer, Springer, Auckland, New Zealand, 3–20.

[5] Christian Aebeloe, Gabriela Montoya, and Katja Hose. 2021. ColChain: Collaborative linked data networks. In *Proc. of the Web Conference (WWW)*. 1385–1396.

[6] Julien Aimonier-Davat, Brice Nédelec, Minh-Hoang Dang, Pascal Molli, and Hala Skaf-Molli. 2024. Fedup: Querying large-scale federations of sparql endpoints. In *Proceedings of the ACM on Web Conference 2024 (WWW)*. 2315–2324.

[7] Ali M Al-Khouri et al. 2012. Data ownership: who owns "my data". *International Journal of Management & Information Technology* 2, 1 (2012), 1–8.

[8] Kinan Dak Albab, Rawane Issa, Andrei Lapets, Peter Flockhart, Lucy Qin, and Ira Globus-Harris. 2019. Tutorial: Deploying Secure Multi-Party Computation on the Web Using JIFF. *2019 IEEE Cybersecurity Development (SecDev)* (2019), 3–3.

[9] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal* (2022), 1–26.

[10] Elli. Androulaki. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proc. of the EuroSys Conference* (Porto, Portugal) *(EuroSys)*. Article 30, 15 pages.

[11] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. 2021. Sql ledger: Cryptographically verifiable data in azure sql database. In *Proceedings of the 2021 international conference on management of data*. 2437–2449.

[12] Balaji Arun and Binoy Ravindran. 2022. Scalable byzantine fault tolerance via partial decentralization. *Proc. of the VLDB Endowment* 15, 9 (2022), 1739–1752.

[13] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*. 598–609.

[14] Amr Azzam, Javier D Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. 2020. SMART-KG: hybrid shipping for SPARQL querying on the web. In *Proceedings of the Web Conference 2020 (WWW)*. 984–994.

[15] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2022. {DUPEFS}: Leaking Data Over the Network With Filesystem Deduplication Side Channels. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 281–296.

[16] Juan Benet. 2014. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).

[17] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-policy attribute-based encryption. In *2007 IEEE symposium on security and privacy (SP)*. IEEE, 321–334.

[18] 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.

[19] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1292–1303.

[20] 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.

[21] Min Cai and Martin Frank. 2004. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *In Proc. of the International Conference on World Wide Web (WWW)*. ACM, New York, NY, USA, 650–657.

[22] Ran Canetti. 2000. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY* 13 (2000), 143–202.

[23] 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.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–130.

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

[25] Melissa Chase and Seny Kamara. 2010. Structured encryption and controlled disclosure. In *Proc. of the International conference on the theory and application of cryptology and information security*. Springer, 577–594.

[26] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation. In *NDSS*.

[27] Jens Ernstberger, Jan Lauinger, Fatima Elsheimy, Liyi Zhou, Sebastian Steinhorst, Ran Canetti, Andrew Miller, Arthur Gervais, and Dawn Song. 2023. Sok: data sovereignty. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 122–143.

[28] Ethereum. 2013. Go Ethereum. https://github.com/ethereum/go-ethereum.

[29] Apache Software Foundation. 2025. Apache Jena. https://jena.apache.org/.

[30] 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.

[31] Zheyuan He, Zihao Li, Ao Qiao, Xiapu Luo, Xiaosong Zhang, Ting Chen, Shuwei Song, Dijun Liu, and Weina Niu. 2024. Nurgle: Exacerbating resource consumption in blockchain state storage via mpt manipulation. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2180–2197.

[32] Lars Heling and Maribel Acosta. 2022. Federated SPARQL query processing over heterogeneous linked data fragments. In *Proc. of the Web Conference (WWW)*. 1047–1057.

[33] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. 2022. Cross-chain deals and adversarial commerce. *The VLDB journal* 31, 6 (2022), 1291–1309.

[34] Siwon Huh, Myungkyu Shim, Jihwan Lee, Simon S Woo, Hyoungshick Kim, and Hojoon Lee. 2023. Did we miss anything?: Towards privacy-preserving decentralized id architecture. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2023), 4881–4898.

[35] Junbeom Hur, Dongyoung Koo, Youngjoo Shin, and Kyungtae Kang. 2017. Secure Data Deduplication with Dynamic Ownership Management in Cloud Storage. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 69–70.

[36] Seny Kamara, Tarik Moataz, Andrew Park, and Lucy Qin. 2021. A decentralized and encrypted national gun registry. In *Proc. of the 2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1520–1537.

[37] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. 2016. Verena: End-to-end integrity protection for web applications. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 895–913.

[38] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

[39] Maria Krommyda and Verena Kantere. 2021. SPARQL-vision: A Platform for Querying, Visualising and Exploring SPARQL endpoints. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM)*. 4730–4733.

[40] Chenxin Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A decentralized blockchain with high throughput and fast confirmation. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 515–528.

[41] Jinyuan Li and David Mazieres. 2007. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems.. In *NSDI*. 10–10.

[42] Siyu Li, Zhiwei Zhang, Jiang Xiao, Meihui Zhang, Ye Yuan, and Guoren Wang. 2024. Authenticated Keyword Search on Large-Scale Graphs in Hybrid-Storage Blockchains. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1958–1971.

[43] Siyu Li, Zhiwei Zhang, Jiang Xiao, Meihui Zhang, Ye Yuan, and Guoren Wang. 2024. Authenticated Keyword Search on Large-Scale Graphs in Hybrid-Storage Blockchains. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1958–1971.

[44] Siyu Li, Zhiwei Zhang, Meihui Zhang, Ye Yuan, and Guoren Wang. 2024. Authenticated Subgraph Matching in Hybrid-Storage Blockchains. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1986–1998.

[45] Zhenni Li, Wensheng Su, Minrui Xu, Rong Yu, Dusit Niyato, and Shengli Xie. 2022. Compact Learning Model for Dynamic Off-Chain Routing in Blockchain-Based IoT. *IEEE Journal on Selected Areas in Communications* 40, 12 (2022), 3615–3630.

[46] 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. *IEEE Transactions on Dependable and Secure Computing* (2021).

[47] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *In Proc. of the conference on the theory and application of cryptographic techniques*. Springer, Springer, Amsterdam, The Netherlands, 369–378.

[48] MetaMask. 2025. MetaMask. https://metamask.io/.

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

[50] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[51] Ontotext. 2025. GraphDB. https://graphdb.ontotext.com/.

[52] OriginTrail. 2025. OriginTrail. https://origintrail.io/. Accessed: 2025-03-01.

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

[54] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. 2012. Score: A scalable one-copy serializable partial replication protocol. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 456–475.

[55] Peng Peng, M Tamer Özsu, Lei Zou, Cen Yan, and Chengjun Liu. 2022. MPC: minimum property-cut RDF graph partitioning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 192–204.

[56] Peng Peng, Lei Zou, Lei Chen, and Dongyan Zhao. 2018. Adaptive distributed RDF graph fragmentation and allocation based on query workload. *IEEE Transactions*

on Knowledge and Data Engineering (2018).

[57] Peng Peng, Lei Zou, and Runyu Guan. 2019. Accelerating partial evaluation in distributed SPARQL query evaluation. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 112–123.

[58] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles (SOSP)*. 85–100.

[59] Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. 2018. Dynamic data exchange in distributed RDF stores. *IEEE Transactions on Knowledge and Data Engineering* (2018).

[60] Eric Prudhommeaux. 2008. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[61] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. 2019. Fine-Grained, Secure and Efficient Data Provenance on Blockchain Systems. *Proc. VLDB Endow.* 12, 9 (may 2019), 975–988. https://doi.org/10.14778/3329772.3329775

[62] 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.

[63] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. [n.d.]. Solid: a platform for decentralized social applications based on linked data. ([n. d.]).

[64] Savvas Savvides, Darshika Khandelwal, and Patrick Eugster. 2020. Efficient confidentiality-preserving data analytics over symmetrically encrypted datasets. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1290–1303.

[65] Hossein Shafagh, Lukas Burkhalter, Sylvia Ratnasamy, and Anwar Hithnawi. 2020. Droplet: Decentralized authorization and access control for encrypted data streams. In *29th USENIX Security Symposium (USENIX Security 20)*. 2469–2486.

[66] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[67] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowledgebase. *Commun. ACM* 57, 10 (sep 2014), 78–85. https://doi.org/10.1145/2629489

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

[69] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. 2022. Operon: An encrypted database for ownership-preserving data management. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3332–3345.

[70] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014).

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

[72] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proc. of the 2019 international conference on management of data (SIGMOD)*. 141–158.

[73] Jie Xu, Qingyuan Xie, Sen Peng, Cong Wang, and Xiaohua Jia. 2023. Adaptchain: Adaptive scaling blockchain with transaction deduplication. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1909–1922.

[74] Hiroyuki Yamada and Jun Nemoto. 2022. Scalar DL: scalable and practical byzantine fault detection for transactional database systems. *Proc. of the VLDB Endowment* 15, 7 (2022), 1324–1336.

[75] Fan Yang, Adina Crainiceanu, Zhiyuan Chen, and Don Needham. 2021. Cluster-Based Joins for Federated SPARQL Queries. *IEEE Transactions on Knowledge and Data Engineering* (2021).

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

[77] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. 2020. LedgerDB: A centralized ledger database for universal audit and verification. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3138–3151.

[78] Zuoru Yang, Jingwei Li, and Patrick PC Lee. 2022. Secure and Lightweight Deduplicated Storage via Shielded {Deduplication-Before-Encryption}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 37–52.

[79] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *Proc. of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. IEEE, 162–167.

[80] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 90–105.

[81] Cong Yue, Gang Chen, Tien Tuan Anh Dinh, Beng Chin Ooi, Zhongle Xie, Xiaokui Xiao, and Meihui Zhang. 2023. GlassDB: An Efficient Verifiable Ledger Database System Through Transparency. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1359–1371.

[82] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of indexing structures for immutable data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 925–935.

[83] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment* 6, 4 (2013), 265–276.

[84] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. 2020. Spitz: a verifiable database system. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3449–3460.

[85] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Proc. of the 2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 863–880.

[86] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1480–1491.

[87] Rui Zhao and Jun Zhao. 2024. Perennial Semantic Data Terms of Use for Decentralized Web. In *Proceedings of the ACM on Web Conference 2024 (WWW)*. 2238–2249.

[88] Enyuan Zhou, Song Guo, Zicong Hong, Christian S Jensen, Yang Xiao, Dalin Zhang, Jinwen Liang, and Qingqi Pei. 2024. VeriDKG: A Verifiable SPARQL Query Engine for Decentralized Knowledge Graphs. In *Proceedings of the VLDB Endowment*.

[89] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*. 2182–2194.