Efficient and Lightweight Indexing Approach for Multi-dimensional Historical Data in Blockchain

Bikash Chandra Singh[a], Qingqing Ye[a], Haibo Hu[a,*], Bin Xiao[b]

[a]*Department of Electronic and Information Engineering, The Hong Kong Polytechnic University, Kowloon, Hong Kong, SAR*
[b]*Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong, SAR*

Abstract

In blockchain, stateful data are stored globally and sequentially in the form of key-value pairs. Recently, to improve the query performance of stateful data, blockchain indexing has been studied. However, existing works only consider one-dimensional data and perform poorly when extended to multi-dimensional and historical data. To overcome these issues, in this paper we propose two new blockchain indexing models. The first model is Two-tier Deterministic Appended Only Skip List (TDASL) that improves from *LineageChain* [15, 16] by using an additional indexing layer on top of a skip list to quickly retrieve the state versions and by using prefixes to query multi-dimensional state versions. The second model is Predefined Partitioned *B*-plus Tree (PPBPT), which paves the way of adopting *B*-plus tree in blockchain by addressing the challenge of its heavy reconstruction cost upon updates. To do so, *PPBPT* copies a predefined *B*-plus tree, which is used for generating indexes for blockchain historical data, thereby reducing reconstruction costs. We conduct extensive experiments to verify the effectiveness of the proposed approaches under various parameter settings.

Keywords: Blockchain, Distributed Ledger, Index, Blockchain State Query

1. Introduction

Blockchain is a decentralized ledger system that allows mutually distrusting parties to maintain a common, immutable ledger without a central authority [22]. In essence, the ledger itself is a chain of blocks that stores ordered list of transactions and are linked together with cryptographic hash. Since this ledger is distributed to nodes across a network that do not trust each other, they run a consensus protocol (i.e., PoW, PoS) to consistently append new blocks to the ledger. This guarantees blockchain to support various cryptocurrencies such as Bitcoin [11] and Ethereum [20], for its transparency, fault tolerance, traceability, temper resistance and reliability. In fact, there are many other systems such as e-commerce [7], supply chain management [17], and heath-care [9] that can benefit from such features. As such, in recent years blockchain has gained tremendous popularity in these business sectors.

However, as more transactions occur in the system, the amount of data on the blockchain grows substantially. For example, a report shows that the Bitcoin Satoshi Version (BSV) Scaling Test Network (STN) can process more than 9,000 transactions per second. Despite being a testing network, STN shares most of its technical capabilities with the BSV, the mainnet to scale up its on-chain.[1] As such, in order to handle such a large volume of data, it is essential to design indexing approach to efficiently find the blockchain state. However, several studies have shown that the existing blockchain concepts are struggling when querying its data [1, 19, 23].

Furthermore, new blockchains such as Ethereum and Hyperledger Fabric adopt the concept of smart contract [10], a set of automated executable codes or rules that operate on the blockchain. A smart contract has its states stored into the blockchain, and update them by transactions. The current state is known as the world state, whereas the

[*]Corresponding author
Email address: `haibo.hu@polyu.edu.hk` (Haibo Hu)
[1]https://www.prnewswire.com/news-releases/9-000-transactions-per-second-bitcoin-sv-hits-new-record-301217145.html

blockchain ledger stores the complete history of transactions. As such, the use of smart contract makes the blockchain grow even larger. In order to efficiently query these large amounts of data, many studies have attempted to build indexing approaches for blockchains [15, 14, 19, 6, 21, 16]. Unfortunately, these existing approaches can only query one-dimensional value while in practice data stored in a blockchain are usually multi-dimensional.

Also, values stored in blockchains have different update frequency. For example, in E-commerce each user account is associated with multiple attributes, such as account balance and product ratings (a.k.a., reputation scores). While the former is frequently updated by transactions, the latter is not. As such, the world state of the blockchain is frequently updated for balance rather than reputation scores. If one only wants to query the latter, she does not have to search all the states of the blockchain, but only need to traverse those states when the reputation scores are created. Unfortunately, existing blockchain indexing models [15, 14, 19, 6, 21, 16] do not support such query types.

To solve both issues, in this paper we propose two index schemes, namely, *TDASL* (Two-tier Deterministic Appended Only Skip List) and *PPBPT* (Predefined Partition *B*-plus Tree), for blockchain to support historical queries on multi-dimensional data. The former extends *LineageChain* [15, 16], an state-of-the-art index that can query one-dimensional blockchain data. *TDASL* uses a deterministic skip list to index the multi-dimensional historical data versions, and puts an extra layer on top of it to retrieve historical data. Furthermore, *TDASL* is more generic than *LineageChain* as it is implemented on the chaincode (i.e., smart contract) instead of the Hyperledger Fabric storage system.

While *TDASL* can perform update efficiently on the appended-only skip list, there are many blockchain applications where the query performance is essential. A recent study has adopted the concept of *B*-plus tree in blockchain and proposed EBTree [21]. However, such approach suffers from huge reconstruction costs due to the frequent generation of blockchain data. Inspired by Partitioned *B* Tree (PBT) [14], in this paper we propose a second index Predefined Partition *B*-plus Tree (*PPBPT*) to index the historical data of the blockchain. *PBT* requires less reconstruction costs because it maintains the partitions of the *B*-plus tree, and writes any changes recorded to the corresponding partition. However, *PBT* suffers from two issues when adopted in blockchain. First, it still has a considerable reconstruction cost when rebuilding the *B*-plus tree in those partitions of newly generated blocks. Second, *PBT* can not index multi-dimensional blockchain state version.

PPBPT addresses both issues by creating a predefined *B*-plus tree and use this tree to index the multi-dimensional historical data versions. More precisely, it is a predefined *B*-plus tree with a specific height and order, and it uses consecutive integer numbers as the keys (i.e., nodes). We treat these keys as seat numbers, and each seat reserves a blank space to store a multi-dimensional state version. When the first *B*-plus tree is full with state versions, a new *B*-plus tree with a new partition number will be created by copying the predefined *B*-plus tree and store the newly generated state versions into its blank space associated with the keys. The advantage is that we do not need to rebuild a new *B*-plus tree just to store the new state versions, instead we only duplicate the predefined *B*-plus tree to make a new one. In this way, the reconstruction cost of the *B*-plus tree can be lower than *PBT* in the blockchain.

To summarize, we make the following contributions in this paper:

- We propose a novel index *TDASL* for querying historical data in blockchain. It exploits a deterministic skip list, and builds a search layer on the top of the skip list to quickly search older versions of historical data. *TDASL* works efficiently in a dynamic data generation environment.
- We also propose another indexing *PPBPT* based on partitioned *B*-plus tree for querying historical data of blockchain. This index fits well where less dynamic data is generated.
- We adapt both *TDASL* and *PPBPT* to query multi-dimensional historical data into blockchain.
- Through extensive experimental results, we show the feasibility and efficiency of both indexes against state-of-the-art index *LineageChain* under various parameter settings.

The rest of the paper is organized as follows. Section 2 provides a review of recent indexing approaches for blockchain data. In Section 3, we briefly introduce the storage model of Hyperledger Fabric and Ethereum blockchains and describe the basic concept of the existing model, *LineageChain*. Section 4 describes the problem statement. The proposed methods *TDASL* and *PPBPT* are extensively described in Section 5 and in Section 6, respectively. In Section 7, we discuss the implementation process of the proposed models and show the experimental results. Finally, Section 8 outlines the conclusion of this paper and shows the future work.

2. Related Work

Researchers have proposed many solutions to implement indexing methods that can search historical data on the blockchains. Below we discuss the notable work done in this direction and compare it with the models we propose in this paper.

Recent research has attempted to integrate database design with blockchain and vice versa. As a result, blockchain is becoming an increasingly important topic in database research [12, 18, 2, 3]. Therefore, on the one hand, some studies considered to start with a database and add blockchain features on top of it. For example, in [12], the authors presented a blockchain relational database with the help of PostgreSQL. In particular, they made extensive modifications to PostgreSQL to integrate the blockchain features in order to support relational database functions. Likewise, studies in [18, 8] proposed approaches that mount the blockchain features on top of databases. In such a system, each peer in the blockchain has its own local database, running transactions through a consensus protocol to update the database, and the ledger acts as a secure, shared log storing transactions.

On the other hand, some are considered to have started with a blockchain and built database features on top to it. For example, in [2, 3], the authors installed a database layer on top of a blockchain such as Hyperledger Fabric or Ethereum. However, this approach only provided a put() / get() query interface to the database, which has relatively limited functionality to support various blockchain systems. J. Gehrke *et al.* in [4] proposed similar approach by considering cloud infrastructure. In particular, they used system logs to synchronize the instances into database. Therefore, this solution first requires the underlying database system to support log shipping. So this system cannot be compatible with other existing databases.

S. Wang*et al.* at [19] designed a storage engine for the Hyperledger blockchain to support forkable applications. To this end, the authors completely replaced Hyperledger's storage engine with a new storage system called *ForkBase* that supports version tracking. Also, the authors of [15, 16] developed *LineageChain* using the *ForkBase* storage system. They implemented *LineageChain* on top of Hyperledger, and used *ForkBase* to support the indexing model. *LineageChain* is basically a skip list indexing approach designed for blockchain. However, one of the major problems with *LineageChain* is that this approach requires a huge amount of time to query historical data of older versions in the blockchain. In addition, *LineageChain* and other existing blockchain data indexing approaches [14, 19, 6, 21] can query for one-dimensional values associated with key-value pairs. But the fact is, in some cases, we need to look up multi-dimensional values associated with key-value pairs. These queries are not possible with traditional blockchain indexing models [15, 14, 19, 6, 21, 16]. To solve this problem, in this article, we propose an indexing approach called *TDASL*, which allows us not only to quickly query old versions of data, but also can query multi-dimensional state values. Also, instead of replacing the Hyperledger Fabric storage system as did in *LineageChain*[15, 16], we use chaincode (i.e., smart contract) to implement *TDASL* in Hyperledger Fabric.

However, some studies developed *B*-plus tree for the blockchain to promote the benefits of the *B*-plus tree for the blockchain. For example, in [21], the authors designed EBTree, an index based on *B*-plus tree. Indeed, the *B*-plus tree incurs huge reconstruction costs due to the frequent generation of blockchain data. However, the paper in [14] gave some ideas based on the partitioned *B* tree (PBT), which can be better adapted to blockchains than the traditional *B*-plus tree. In fact, *PBT* requires less reconstruction cost because it maintains the partitions in the *B*-plus tree, and *PBT* writes all changes to the corresponding partition. However, inspired by *PBT*, we proceed a step further in order to reduce the cost of rebuilding *B*-plus to index historical blockchain data. Therefore, in this article, in addition to *TDASL*, we propose another index called the Predefined Partition *B*-plus Tree (*PPBPT*). This indexing method can also query multi-dimensional historical data in blockchain.

3. Preliminaries

In this Section, we provide a brief description of the storage model of Hyperledger Fabric and Ethereum blockchains. Also, we briefly describe the existing blockchain index *LineageChian*, which provides the basis for our proposed model *TDASL*.

3.1. Storage Model

Hyperledger Fabric and Ethereum use an account-based data model. This data model assumes that each account can store a value (e.g., balance), and a valid transaction associated with the account can update the status of that account, that is, it can modify the value and create a new state in world state. More precisely, the world state can be randomly updated by smart contracts (i.e., smart contracts can submit transactions) and contains the latest state of each account, while at the same time, executed transactions are included into blocks and the blockchain ledger establishes a cryptographic chain among blocks. This prevents anyone from modifying the committed transactions. Now, we briefly explain the storage models of Hyperledger Fabric and Ethereum blockchains.

Data Model for Hyperledger Fabric. Hyperledger uses Merkle Bucket Tree (MBT) for indexing key-value pairs [23]. MBT consists of a combination of a Merkle tree and a hash table. More particularly, the hash table puts the key-value pairs into the buckets that get position at the bottom level of MBT, and the keys are sorted into each bucket. On the other hand, MBT uses the Merkle tree on top of hash table to form a tree using the buckets' cryptographic hashes in order to prove the existence of a key-value pair. More precisely, MBT computes the hashes of each bucket, uses these hashes to create the internal nodes of the Merkle tree, and continues to obtain the root node of Merkle tree. The number of children of an internal node refers to Fan-out and the cordiality of the bucket set refers to capacity. The capacity and fan-out are predefined and can not be changed for the duration of their lifetime.

Data Model for Ethereum. Ethereum uses Merkle Patricia Trie (MPT) [20, 23] for its data model. MPT is a radix tree with cryptographic authentication. MPT consists of four types of nodes, namely extension, branch, leaf and null: (i) The extension node (EN) contains a shared nibble and a pointer to the next node and the shared nibble refers to the common sequential character(s) of the keys from left to right; ii) The branch node is an array of 16 elements and a value and Each element indexes a single corresponding child node and stores a nibble; iii) A leaf node refers to child node and it consists of two items: remaining character(s) of the key and the values; iv) A null node does not contain any content indicating the end of the path.

3.2. LineageChain

Recently, in [15, 16], the authors proposed *LineageChain* which provides an indexing approach for account-based data model that utilizes skip list for querying blockchain data. The main idea is that LineageChain captured data provenance during smart contract execution and produced a Directed Acyclic Graph (DAG) based on the states of blockchain. Then, in order to support queries, *LineageChain* built a skip list index on top of the DAG, namely Deterministic Append-only skip List (DASL). Usually, in the skip list, there are multiple linked lists L_i, where $i = \{0, 1, 2, ..., \}$. In order to design the content of L_i, *LineageChain* considered the sequence of blockchain state version numbers $V_k = \langle v_0, v_1, ...\rangle$ for identifier k, where $v_i < v_j$ for all $i < j$. The state version V_k belongs to the linked lists L_i based on the interval j and each interval has the range of $R^i_j = [jb^i, (j+1)b^i]$, where b is the base of *DASL*. More particularly, only the smallest state version belongs to the linked lists L_i, if it takes place in the corresponding interval. Based on this process, *LineageChain* designed the skip list *DASL*.

However, if *DASL* has a large number of state versions, then *LineageChain* takes more times to search the older state versions. In particular, using this method, queries that read historical data backwards take longer, thereby delaying the execution of other transactions. It is worth noting that malicious users could use this opportunity to carry out a denial of service attack. *LineageChain* attempted to mitigate this kind of attacks by increasing the transaction fee of *gas*. However, this mechanism also applies to the rational users who actually need to access too old state versions in the blockchain. In addition, *LineageChain* mainly considered the one-dimensional value associated with the blockchain account. Therefore, it is not possible to use this approach to query of multi-dimensional state version.

4. Problem Statement

This paper studies the indexing problem for multi-dimensional blockchain historical data. In fact, the historical record of the multi-dimensional blockchain state is sequentially generated by blockchain transactions. In order to distinguish and track the dimensions of the blockchain states we use prefix fields. More particularly, we build the indexing approaches with the multi-dimensional blockchain states and prefix values in order to use the indexing models to perform the search and insert operations of the blockchain historical data. As such, in the following, we define the multi-dimensional blockchain state, the transaction that updates the state accordingly, the prefix that can track the multi-dimensional blockchain state based on its dimension and two basic operations (i.e., insert and search).

Definition 4.1. Multi-dimensional Blockchain State. *Let a set of values $U = \{val_1, val_2, .., val_n\}$ be associated with a blockchain account k. A multi-dimensional state $S_{k,v_i,U}$ is composed of a set of values U associated with an identifier k and a state version v_i. The dimension of the state $S_{k,v_i,U}$ is equal to $|U|$.*

Symbol	Description
k	a blockchain account
U	the set of values associated with k
$S_{k,v,U}$	a multi-dimensional blockchain state
v_i	the i-th state version
txn	a blockchain transaction
v^l	latest state version
b	block in blockchain
d	distance between states in blockchain
h	height of a node in $TDASL$
b	base number in $TDASL$
N	number of nodes in a B-plus tree

Table 1: Notations

We consider that the blockchain world state stores key-multidimensional value pairs. Therefore, we define each entry in the world state as a tuple $(key, (val_1, val_2, .., val_n))$, where key be a unique identifier of the blockchain state, $(val_1, val_2, .., val_n)$ be the set of different types of values associated with a key key, transactions can update these values and create new state. Therefore, we retrieve these entries from the world state to chaincode with an additional field v (i.e., version) for creating the historical blockchain states for each account. So, each entry in the chaincode is a tuple of $(key, v, (val_1, val_2, .., val_n))$, where v be the version number of the state and v is treated as an integer number. It can start at 0 and increments by 1 in order. The main notations are listed in Table 1.

Definition 4.2. Transaction. *Let a set of values $U = \{val_1, val_2, val_3, ..\}$ be associated with a blockchain account k. A transaction txn be valid if and only if it updates at least one value $u \in U$ of the blockchain state $S_{k,v_i,U}$, resulting a new state $S_{k,v_{i+1},U'}$, where $S_{k,v_i,U} \neq S_{k,v_{i+1},U'}$, $v_i < v_{i+1}$ and $U \neq U'$.*

In our assumption, we divide transactions into two categories: i) partially updated transaction: A transaction txn refers to a partially updated transaction if and only if it updates at least one but not all values in $U = (val_1, val_2, ...val_n)$ associated with a key key, thereby generating a new state; ii) fully updated transaction: A transaction txn refers to a full value updated transaction if and only if it updates all values in $U = (val_1, val_2, ...val_n)$ associated with a key key, resulting a new state.

Example 4.1. *Figure 1 shows the overall processing steps of how the multi-dimensional values in the world state is updating based on the transactions. We can see that the transaction $txn1$ modifies only one value and changes the state from $S_{(k1,v1,(B_2,R_0,..)}$ to $S_{(k1,v2,(B_3,R_0,..)}$ for the identifier $k1$. While $txn1$ modifies all values of $k2$, thereby updated the state from $S_{(k2,v1,(B_8,R_{10},..)}$ to $S_{(k2,v2,(B_9,R_{11},..)}$. This shows that the transaction $txn1$ can be part of both categories with respect to its associated key. We can also see in Figure 1 that $txn3$ modifies only one value of the associated key $k4$. Therefore, it refers to a partially updated transaction.*

Definition 4.3. Prefix. *Let a set of values $U = \{val_1, val_2, val_3, ..\}$ be associated with a blockchain account k. Prefix is defined as a field in a state entry consists of a number of $|U|$ entities and each entity counts the changing sequence of its corresponding value exist in $U = \{val_1, val_2, val_3, ..\}$ updated by the blockchain transactions. More specifically, $P = \bigcup_{j=1}^{|U|} p_j$, where p_j refers to j-th prefix value of j-th value of U updated by transaction.*

By categorizing transactions, we can identify which specific values are being modified by these transactions. To do this, we use $prefix$ as an identifier that can identify the specific value changed by a transaction, and we also use $prefix\_value$ as a counter that can count the changing sequence of that value associate with a specific key.

Example 4.2. *Figure 2 shows the evaluation of state history of the identifier $k2$ derived from Figure 1. For simplicity, we consider two state values, one is balance (B) and the other is reputation (R) associated with blockchain keys. Since we consider two state values in this example, the bits of the prefix are two in order to distinguish the state value. More precisely, 01 represents reputation value, and 10 represents balance value. In addition, we keep the corresponding*

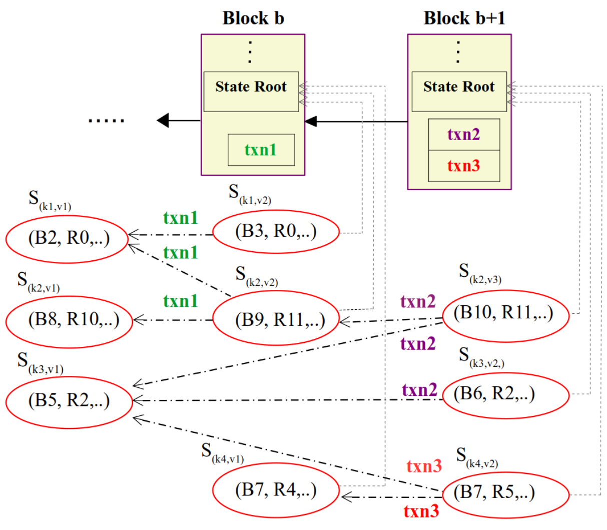
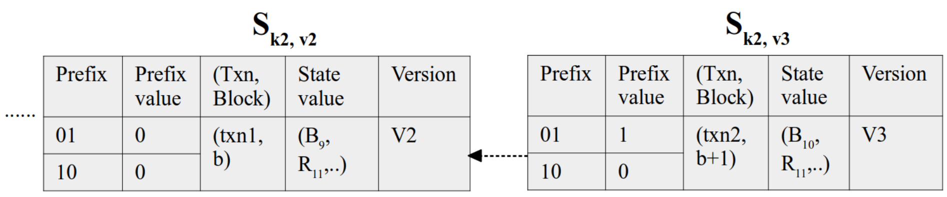


Figure 1: States update based on transactions



$S_{k2,\,v2}$

Prefix	Prefix value	(Txn, Block)	State value	Version
01	0	(txn1, b)	(B_9, R_{11},..)	V2
10	0			

$S_{k2,\,v3}$

Prefix	Prefix value	(Txn, Block)	State value	Version
01	1	(txn2, b+1)	(B_{10}, R_{11},..)	V3
10	0			

Figure 2: Evaluation state history for identifier $k2$

counter of the prefix, which is used to calculate whether the committed transactions change that particular value. It can be seen from Figure 2 that in state $S_{k2,v2}$, the transaction txn1 in block b updates two values (B,R), so the prefix value is 0 *in both cases. In the next state $S_{k2,v3}$, transaction txn2 updates the balance but does not change the reputation, therefore the prefix* 01 *has a value that is increased by 1, indicating that the reputation has same value in the current state $S_{k2,v3}$ and the previous state $S_{k2,v2}$. On the other hand, the prefix* 10 *has* 0 *in this state $S_{k2,v3}$, indicates that balance is updated from the previous state $S_{k2,v2}$ by the transaction txn2 in block* $(b + 1)$.

It is worth noting that the value of a prefix is constantly increasing unless its corresponding value is updated by a transaction. This happens when blockchain transactions update other values to generate new states. Therefore, by looking at the prefix values, we can track the state versions in order to query the corresponding updated value (e.g., reputation or balance). However, users need to perform some operations using the indexing approach. In this paper, we focus on two fundamental operations: state insertion and state search.

Definition 4.4. State Insertion. *Let a set of states of an account k be indexed as $I_{i-1} = I(S_{k,v_1,U_1,P_1}, S_{k,v_2,U_2,P_2}, \dots\dots\dots, S_{k,v_{i-1},U_{i-1},P_{i-1}})$. Let S_{k,v_i,U_i} be a multi-dimensional state and the set of its associated prefix value $P_i = \bigcup_{j=1}^{|U|} p_j$ generated by a transaction txn. More specifically, the state version with its prefix values defined as S_{k,v_i,U_i,P_i}, be inserted for indexing as $I(I_{i-1}, S_{k,v_i,U_i,P_i})$, where the index function $I()$ performs the insertion operation for the newly generated state.*

Definition 4.5. State Search. *Let a query $Q(v, u, r)$, where the query state version v, the value u and the number of states r, returns all state versions $Q(v, u, r) = \{o \in I : \exists u \in U \parallel \forall u \in U, |o| \leq r\}$, where o is the resulting states, I is the indexing database contains a set of multi-dimensional states and U be a set of values.*

Now, we present our proposed indexing approaches: i) *TDASL* (Two-tier Deterministic Appended only Skip List) (see Section 5 for details), ii) Predefined Partitioned *B*-plus Tree (PPBPT) (see Section 6 for details).

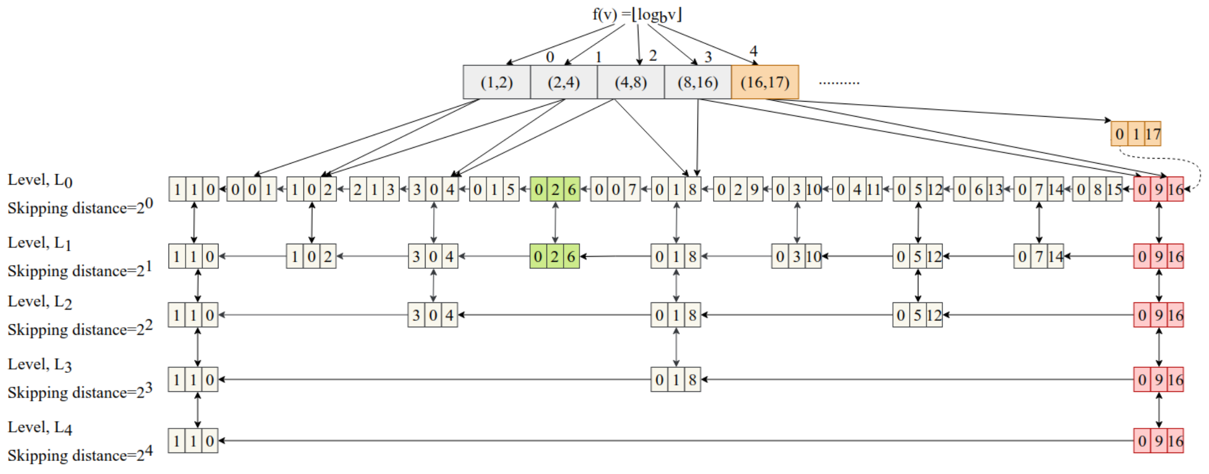


Figure 3: Two-tier Deterministic Appended only Skip List (TDASL) indexing approach

5. Two-tier Deterministic Appended Only Skip List (TDASL)

In practice, some applications in the blockchain generate data with a high frequency. As such, it is essential to cope this feature with the indexing model. To do so, this approach can work well as it can update its index efficiently with frequently generated data. More specifically, *TDASL* uses a skip list, so it can quickly update its index by appending state versions to the skip list.

However, the blockchain state versions are generated in sequence. For example, suppose that $S_{k2,v2,(B_9,R_{11},..)}$ and $S_{k2,v3,(B_{10},R_{11},..)}$ are two distinct states generated sequentially with state versions, namely, $v2$ and $v3$. It is obvious that $v2 < v3$. With this fact in mind, we design *TDASL* with the latest state appended to its previous state. In particular, *TDASL* is appended only approach as like in [15]. However, *TDASL* has two-tier index: i) the upper tier is used for finding the searching range of the bottom tier that contains the query state; ii) the bottom layer is a variant version of skip list [13] that uses the concept of deterministic skip lists to index and query the blockchain states. Figure 3 depicts the entire procedure of the *TDASL* approach. The nodes that appear in *TDASL* are the state entries (shown in Figure 2). For simplicity, we consider that the nodes in *TDASL* consists of state version, prefix values for reputation and balance, respectively.

In order to design the bottom tier, we leverage the deterministic skip list. As we know, the skip list consists of multiple lists (levels) and a node may belong to multiple levels. The Figure 3 illustrates that each level of the skip list has different skipping intervals to select nodes. In general, the skipping interval is b^i for the i-th level L_i, where b is the base number (e.g., b=2). Each level probably has several nodes, based on the size of the nodes. For example, in Figure 3, if we consider the second level of the skip list, then it always proceeds to select the nodes in the list with a skip distance of $2^2 = 4$. Therefore, the possible selected nodes in the second level of the skip list are: {4,8,12,16}. Similarly, if we proceed to fourth level of the skip list, the skip distance is 2^4, so the selected nodes in this list be {16}. It is worth noting that the first node (state version) 0 we keep for all levels. Finally, the fifth level cannot be formed in figure 3 because so far there are not enough nodes in the skip list to make this level.

The upper tier helps to find the appropriate location in the skip lists (i.e., bottom tier) where the searching node could be appeared, that can speed up the search process in the skip list. To do so, we embodied some entries as shortcuts in the routing path. However, it is challenging to determine which entries should be promoted to the upper layer from the skip list. To this end, we consider the number of entries in the top tier based on the number of levels in the skip list. For example, if there are N levels in the skip list, the number of entries in the top tier is N. Each entry consists of two routing nodes, which are selected based on their heights in the skip list.

For example, we can see from Figure 3 that *TDASL* has five levels in the skip list. Thus, it has five entries at the top tier. For the first entry, we assign i=0, and j=1, therefore we get the nodes for the first entry is $(b^i, b^j) = (1, 2)$, if $b = 2$ (we can consider large value for b). In the second entry, we increase the value of i and j by 1, they become

1 and 2, so the nodes of the second entry are $(2^i, 2^j) = (2, 4)$. Likewise, we can select the state versions for the next entries. But for the last entry (that is, fifth entry here), the selected nodes are $(2^4, 2^5) = (16, 32)$. Since the node 32 has not been created yet, so it points to the latest node (that is, 17 in Figure 3), and this entry will be updated until 32 is generated.

Insert. To insert a new node into *TDASL*, we first need to determine which linked lists should include this node. For example, we want to insert state version 16 into *TDASL* as shown in Figure 3. The fact is that L_0 contains all nodes since its skip distance is 2^0. Therefore, we put the new node 16 in L_0, and then if this condition 16 mod $2^l = 0$ is met, we continue to increase the list level for the node 16, where l refers to the list levels $\{0, 1, 2, .., 4\}$ in the skip list. After that, we find the appropriate entry in the top tier to which it belongs to. To do this, we first need to find the index in the top tier with $\lfloor log_2 16 \rfloor$, that is, 4. Since 16 is a new node, there is no indexing with 4 yet at the upper layer. Therefore our system updates the fourth routing entry with 16 and generates a new entry in the top tier with 16 as its fifth routing entry, and this entry nodes are pointed with 4. In addition, we adjust the prefix values of node 16 with its previous node 15. In particular, since the version 16 has prefix value 1 for balance and 0 for reputation and its previous version 15 has 8 for balance and 0 for reputation, therefore, the adjusting score will be 9 and 0 in Figure 3. Similarly, we can perform the insertion operation with a node 17 as shown in Figure 3. Note that $\lfloor log_2 17 \rfloor$, refers to 4, and the index points to the exact entry in the top tier. Since there is already a routing node 16, this will be the second routing node in this entry. The thing is that the second routing node will be updated with the next new node until we get a new entry in the top tier.

Lookup. To perform the lookup procedure with *TDASL* approach, we first calculate the index value in the top tier of *TDASL*. For example, we illustrate the lookup process for state version 12. The index value of 12 is $\lfloor log_2 12 \rfloor = 3$ in the top tier, indicating that 12 exists in the range of nodes [8,16]. Then, we measure the distance from the largest node, $d = 16 - 12 = 4$. After that, we find the appropriate level in the skip list to start the search. To do this, we compute $\lceil log_2 d \rceil = \lceil log_2 4 \rceil = 2$, indicating that the search starts from node 16 at the L_2 level. Then we proceed to find the state version 12. Moreover, we want to further search the previous states from the state 12, and these states have updated values of different dimensions (e.g., balance/reputation). Then, in the state version 12, we can see that the prefix value for reputation is 5, means that 5 successive states have the same reputation value as no transactions have updated its reputation so far. In this case, the state version 12 and the reputation prefix 5 implies that the desired state version, $(12 - 5) - 1 = 6$ has a new reputation. In addition, the prefix value for balance is 0, indicating that state 12 has updated balance, and its previous state also has a different balance and this state is, $(12 - 0) - 1 = 11$. So, our desired state versions are 6 and 11 that we have to traverse. To do this, we can repeat the process we did for state version 12.

Algorithm 1: TDASL_append()

```
Input: $v$, be the state version to be appended in TDASL
Output: nodes of the top tier, $top_{tier}$; nodes of bottom tier $level_{nodes}$
1  Let $level_{nodes}$ = []
2  Let $top_{tier}$ = []
3  Let $v_{top-tier} = \lfloor \log_2(v) \rfloor$
4  Let level is initialized to zero
5  if ($v_{top-tier} \in top_{tier}$) then
6      Let update the latest entry of $top_{tier}$ with version $v$ as second node
7      for $i = level; i \leq v_{top-tier}; i++$ do
8          if $v$ mod $2^i == 0$ then
9              $level_{nodes}.append(v)$
10 else
11     Let create a new entry with version $v$ as first node in $top_{tier}$
12     for $i = level; i \leq v_{top-tier}; i++$ do
13         if $v$ mod $2^i == 0$ then
14             $level_{nodes}.append(v)$
15 Return: $level_{nodes}, top_{tier}$
```

Algorithm 2: TDASL_search()

```
Input: $v$, be the state version to be searched in TDASL
Output: true if $v$ be found in TDASL; false otherwise
Let $level_{nodes}$ be the nodes exist at bottom tier
Let $top_{tier}$ be the nodes exist at top tier
Let $n$, be the number of states we want to search
Let $check$ be initialized to zero
Function Search($v$):
    while $check \leq n$ do
        Let $v_{top-tier} = \lfloor \log_2(v) \rfloor$
        if ($v_{top-tier} \in top_{tier}$) then
            Let the nodes range of the $level_{nodes}$ between $f_{node}$ and $s_{node}$
            Let distance, $d = s_{node} - v$
            Let $app_{level} = \lceil \log_2(d) \rceil$
            for $i = app_{level}; i \leq 0; i--$ do
                if $cur-> level_{nodes}() == v$ then
                    Let $prefix$, be the corresponding prefix value that can point to the next state
                    $version = v - prefix$
                    $check++$
                    $Search(version)$
                    $search_{output} = true$
                else if $cur-> level_{nodes}.left() \geq v$ then
                    $cur-> level_{nodes} = cur-> level_{nodes}.left()$
                    $search(cur-> level_{nodes})$
                else
                    $cur-> level_{nodes} = cur-> level_{nodes}.down()$
                    $search(cur-> level_{nodes})$
        else
            $search_{output} = false$
    Return: $search_{output}$
End Function
```

Now, we separately analyze the storage and query costs of *TDASL* in more detail.

Storage overhead. Let v^l be the latest state version. In this case, the top tier requires a maximum of $3\lceil \log_b v^l \rceil$ pointers, and the bottom tier requires $2v^l$. Therefore, the total storage required to build *TDASL* is $3\lceil \log_b v^l \rceil + 2v^l$, which is slightly higher than the storage $2v^l$ required for *LineageChain* [15, 16].

Query time. In *TDASL*, the top tier refers to the nodes that are relatively taller than other nodes. In other words, the nodes do not have equal height in the *TDASL*. The all possible heights for a node can be defined in a set as follows: $N_h = \{h | v^l \% b^h = 0, h \in \{0, 1, 2, 3, ...\}, h \leq \lceil \log_b v^l \rceil\}$. Assume that the latest entry in the top tier consists of $\lceil \log_b v^l / b^1 \rceil$ and $\lceil \log_b v^l / b^0 \rceil$, which are the second-highest and first-highest nodes in the bottom tier that cover the largest area (i.e., almost 50% of the nodes) in *TDASL*. In this case, we need the maximum number of pointers to traverse this area. More precisely, the pointers required for level L_0 is: $(v^l - \frac{v^l}{b}) - 1 = \frac{v^l}{b.b^0} - 1$, if b=2. For L_1, it is, $\frac{v^l}{b.b^1} - 1$. For i-th level, it takes at most $= \left\lceil \frac{v^l}{b.b^i} \right\rceil - 1 = \left\lceil \frac{v^l}{b^{(i+1)}} \right\rceil - 1$. Therefore, the maximum number of pointers is: $\sum_{i=0}^{\lceil \log_b v^l \rceil} \left\lceil \frac{v^l}{b^{(i+1)}} \right\rceil - 1 = \frac{bv^l - 1}{b(b-1)}$, need to traverse to find the desired state version, which is smaller than $\frac{bv^l - 1}{b - 1}$ required by *LineageChain* [15, 16]. In addition, suppose that we want to query the state version vq. *LineageChain* always starts the search from the latest version v^l, so the distance is $D_L = v^l - vq$. Conversely, *TDASL* uses the top tier

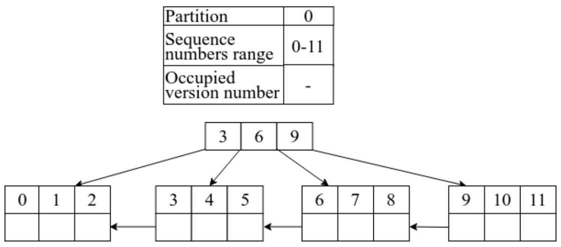


Figure 4: Empty Predefined Partitioned B-plus Tree (PPBPT) with serial numbers

entries to find the searching range in the bottom layer. In this case, $v^l \in V$, where $V = \{\lceil v^l/b^0 \rceil, \lceil v^l/b^1 \rceil, .., \lceil v^l/b^{log_b v^l} \rceil\}$. Therefore, the distance $D_T = v^l - vq$ is less than or equal to D_L. Consequently, $D_T < D_L$, when vq appears in any entry at the top tier of *TDASL* except the latest entry, otherwise D_T is equal to D_L.

Algorithm 1 represents the whole process of inserting new state version in *TDASL* approach. As shown in Algorithm 1, given a new state version v as input, this design adds the lists at the bottom tier and also compute an entry in the top tier, where v belongs to. To do so, we first check whether v belongs to the top-tier top_{tier} or not. If it does (see line 5), then we update the entry with v as the second node, otherwise we create a new entry in top_{tier}, where v be the first node of this entry (see line 11). Later, we add v to the underlying lists in the bottom tier (see lines in 7- 9 or 12-14).

Algorithm 2 depicts the entire process of seeking version v with *TDASL*. For this, we find the entry in the top tier, where v is involved and this entry makes the searching range in the bottom tier to find v (see lines 8-9). Next, we compute the exact location to start the search at bottom tier (see lines 10-11). After that, we proceed to find v at the bottom tier (see lines 12-24). However, when we find v, then we can further look for the state versions for n consecutive times, which contain the new values (reputation or balance). To do this, we use the recursive function *Search()* (see lines 12-18).

6. Predefined Partitioned *B*-plus Tree (PPBPT)

However, there are many blockchain applications that require relatively better query performance rather than update performance of the indexing model. For this kind of blockchain applications, our second approach, Predefined Partitioned B-plus Tree (PPBPT) can be well applied. *PPBPT* implements a variant of the B-plus tree, which can provide relatively better query performance.

Moreover, *TDASL* is a variant of the skip list. In essence, although it has better update performance for its index, it takes more time to find nodes that appear on the baseline of the skip list. On the other hand, a B-plus tree does not have this limitation and can retrieve any node in about the same amount of time as the node gets its position at the same level in B-plus tree. However, a blockchain is a special type of data structure in which blocks are frequently generated. Therefore, it is not suitable to use the existing traditional B-plus tree [23] on the blockchain. More particularly, every new blockchain data can often generate a new rebuild B-plus tree, which significantly increases the cost of reconstruction. However, *PBT* (Partitioned B Tree) [5, 14] attempts to reduce the cost of building a B-plus tree by splitting the entire tree into multiple partitions, and then write any modification of the record exactly once at the eviction time of the corresponding partition. As such, *PBT* still has considerable construction cost.

To address this issue, we propose Predefined Partitioned B-plus Tree (*PPBPT*). The main idea is that we first create a predefined B-plus tree with some fields that represent the metadata of this tree. More precisely, we design a B-plus with a specific height and order, and use consecutive integer numbers as the keys in B-plus tree. We treat these keys as the seat number that we reserved for storing the state versions. Figure 5 depicts the predefined partitioned B-plus tree. Eventually, *PPBPT* consists of a number of B-plus trees and each B-plus tree are separated and represented with individual partition number. Therefore, *PPBPT* has two parts: i) a root head node that contains *PPBPT*'s metadata ii) a predefined B-plus tree with empty space.

Root head node. The root head node contains the metadata of B-plus trees that mainly specifies the identification and properties of the tree. In particular, we consider three fields, i) partition: specifies the partition number of a B-plus tree, ii) range of sequence numbers for state versions: specifies the range of the state versions that are going to take

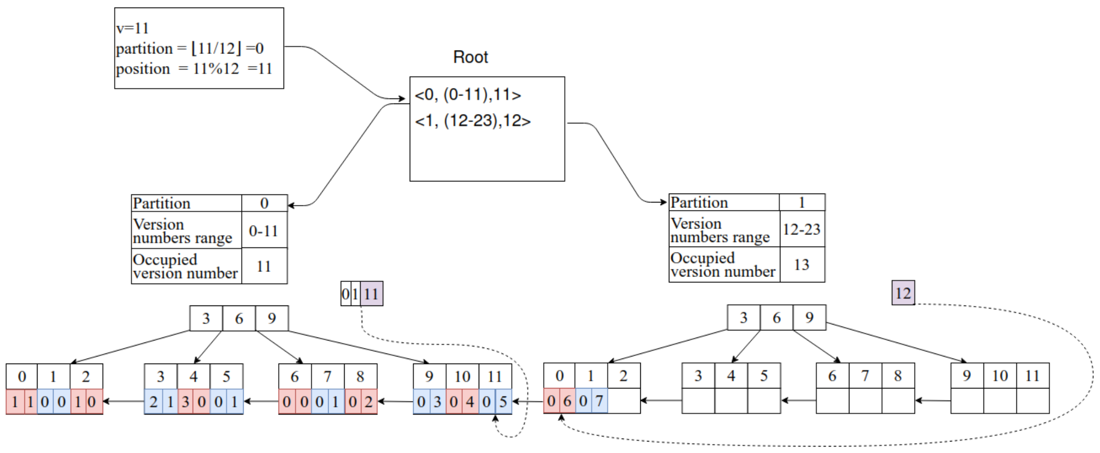


Figure 5: Predefined Partitioned *B*-plus Tree (PPBPT) with state versions

position in corresponding *B*-plus tree, iii) occupied version number: indicates the version number that already take position in the empty space associated with the serial number identical with the version number.

For example, in Figure 5, there are two records in the root head node. The first record is $< 0, (0 - 11), 11 >$, indicating that the partition number of *B*-plus tree is 0, which can store the version numbers from 0 to 11, and the last field 11, indicates this tree is full of state versions as it is equal to the last element of the state versions. On the other hand, for the record $< 1, (12 - 23), 12 >$, it has empty spaces for state versions 13 to 23 since its last field, that is 12 is not equal to the last state version, which is 23.

Predefined *B*-plus tree with empty space. First of all, we define the order m, and height h of the *B*-plus tree. Then we compute the size of the tree, N= $\sum_{l=1}^{h} m^l$. After that, we suppose to consider serial numbers starting from 0 to $N - 1$. Using these numbers, we design a *B*-plus tree with order m and height h. Moreover, we add free space to each serial number for keeping the prefix values associate with the state versions. Initially, these spaces are empty in *B*-plus tree, but it is sequentially filled up with prefix values of versions. For example, Figure 4 shows a predefined *B*-plus tree that contains serial numbers from 0 to 11. Now, in the followings, we explain the insertion and search operations on *PPBPT*.

Insert. In initial phase, *PPBPT* has only one *B*-plus tree with partition number 0. When this tree is full of nodes (versions), *PPBPT* generates a second *B*-plus tree with a partition number of 1, which is a copy of the predefined *B*-plus tree as shown in Figure 5. For example, in Figure 5, we consider the size of each *B*-plus tree is 12. Now, let us assume that state version 11 is generated by a transaction in blockchain. In this case, *PPBPT* first checks which *B*-plus tree may contain the version 11, and then finds where it belongs to. To do this, *PPBPT* divides the version number 11 by its size (e.g., here it is 12), refers to the partition number 0, and then performs a modular operation, that is, 11 mod 12 = 11 indicates that version 11 will be inserted at position 11 of the *B*-plus tree whose partition number is 0. In this case, *PPBPT* advances from the root node to the *B*-plus tree 0 as shown in Figure 5. Since the location of the version 11 is 11, it moves to the right of 9, then finds the location 11 and stores the adjusted prefix value in the blank space associated with 11. In particular, since the prefix value of version 11 is 1 for balance and 0 for reputation, and its previous version 10 has 9 for balance and 0 for reputation, so the adjusted score is 10 and 0, as shown in the Figure 5. This is the entire process of inserting a new version in *PPBPT*. Moreover, since the *B*-plus tree $< 0, (0 - 11), 11 >$ is full after inserting 11, so *PPBPT* copies the predefined *B*-plus tree with partition number 1, as shown in Figure 5. As such, it does not need to reconstruct the second *B*-plus tree with the upcoming state versions, thereby reducing the construction cost of *PPBPT*. Then it updates the root head node with $< 1, (12 - 23), - >$, indicating that the *B*-plus tree 1 can contain version numbers from 12 to 23.

Lookup. The search operation is quiet similar to insertion operation. For example, suppose that we want to search version 12 in *PPBPT*, as shown in Figure 5. To do this, first, we need to compute the tree's partition number, i.e., $\left\lfloor \frac{12}{12} \right\rfloor = 1$, and its location, 12 mod 12 = 0, indicating that 12 exists at location 0 in tree 1. Then *PPBPT* proceeds to this specific tree and start searching from top of the tree. Since the position 0 is less than 3, it goes to the left of

Algorithm 3: PPBPT_insert()

```
Input: $v$, be the state version to be inserted in $PPBPT$
Output: true if version $v$ is inserted; false otherwise
1  Let $bPlus$ be the predefined $B$-plus tree generated with order $M$ and height $h$
2  Compute $T_s = \sum_{l=1}^{h} M^l$
3  Let $B$-plus tree contains sequence numbers $\{0, 1....(T_s - 1)\}$
4  Let $Partition = []$
5  Let $Root$ be the root head node
6  Compute the partition number, $v_p = \left\lfloor \frac{v}{T_s} \right\rfloor$
7  Compute the position, $v_{position=v \bmod T_s}$
8  Let $cur_{root}$=$Root$
9  Let finish=false
10 if $v_p == cur_{root} -> Partition$ then
11     Let $SN$ be the sequence number in $bPlus_{vp}$ equal to $v_{position}$
12     if $SN$ is empty then
13         SN.UPDATE($prefixes$)
14         $finish = true$
15     else
16         no update required
17         $finish = false$
18 if $v_{position} == (T_s - 1)$ then
19     Let generate $bPlus_{vp+1}$ by copying the $bPlus$
20     Root.UPDATE $((v_p + 1), (T_s \times (v_p + 1), T_s \times (v_p + 1) + (T_s - 1)), -)$
21 Return: $finish$
```

3 and find the location of version 12. Also, we can further search the earlier states from the state 12 based on the updated values of different dimensions (e.g., balance/reputation). To do this, we need to pay attention to the prefix values of the state 12. The state 12 has the prefix value for reputation 6, which refers that 6 successive states have the same reputation value. Therefore, the desired state version $(12 - 6) - 1 = 5$ has a new reputation. In addition, the prefix value of the balance is 0, which means that the state 12 has updated the balance, and its previous state also has a different balance. This state is $(12 - 0) - 1 = 11$. Therefore, we need to traverse state versions 5 and 11 to find different reputation and balance, respectively. To do this, we can repeat the process we did for the state version 12.

Now, we explain in detail the storage and query costs of *PPBPT*.

Storage overhead. In *PPBPT*, the order of each *B*-plus tree is m and height is h, so the size of the *B*-plus tree is $N = \sum_{l=1}^{h} m^l$. Now, suppose v^l be the number of versions to be indexed in *PPBPT*. In this case, *PPBPT* requires the number of *B*-plus trees, $\lceil v^l/N \rceil$, to store this number of versions v^l. In addition, the root head requires to store $\lceil v^l/N \rceil$ records. Therefore, the total storage required to construct *PPBPT* is $(v^l + v^l/N)$, which is less than the storage required by *TDASL* and *LineageChain*.

Query time. The query time for worst, average and best cases for searching a version is, $log_m(v^l - N \times (\frac{v^l}{N} - 1)) + \lceil v^l/N \rceil = log_m N + \lceil v^l/N \rceil$, where v^l be the latest state version and N is the size of each *B*-plus tree in *PPBPT*.

Algorithm 3 depicts the entire process of inserting nodes into *PPBPT* approach. At initial phase, it generates a predefined *B*-plus tree *bPlus* and root head node *Root* (see lines 1-5 in Alogrithm 3). Then, it computes the partition number of the *B*-plus tree and position where the version v belongs to (see lines 6-7). Next, Algorithm 3 proceeds to check the computed partition number into the root head node *Root* and goes to the specific *B*-plus tree to search the sequence number that is equal to $v_{position}$ and insert prefix values of the version v into that place (see lines 10-13 in Algorithm 3). Furthermore, if v's position number is equal to the last serial number of *B*-plus tree then Algorithm 3 automatically generate the next *B*-plus tree by coping the predefined tree as the current *B*-plus becomes full of size with v (see lines 18-20). As a result, *PPBPT* needs less rebuilding cost to make the *B*-plus tree with its state versions.

Algorithm 4 describes the search operation of *PPBPT* indexing model. For this, it takes the state version v as input

Algorithm 4: PPBPT_search()

```
Input: $v$, be the state version to be searched with $PPBPT$
Output: true if version $v$ is found; false otherwise
Let $bPlus$ be the predefined $B$-plus tree generated with order $M$ and height $h$
Compute $T_s = \sum_{l=1}^{h} M^l$
Let $Partition = []$
Let $Root$ be the root head node
Compute the partition number of $B$-plus tree, $vp = \left\lfloor \frac{v}{T_s} \right\rfloor$
Compute the position, $v_{position} = v \bmod T_s$
Let $n$, be the number of consecutive state versions from $v$ that we want to search
Let $check$ be initialized to zero
Function Search($vp, v_{position}$):
    Let $cur_{root}$=$Root$
    while $check \leq n$ do
        if $vp == cur_{root}-> Partition$ then
            Let $SN$ be the sequence number in $bPlus_{vp}$ equal to $v_{position}$
            Let $prefix$, be the corresponding prefix value associated with $SN$
            $version = v_{position} - prefix$
            $vp = \left\lfloor \frac{version}{T_s} \right\rfloor$
            $v_{position} = version \bmod T_s$
            $check++$
            $Search(vp, v_{position})$
            $search_{output} = true$
        else
            $search_{output} = false$
    return $search_{output}$
End Function
```

and then compute the B-plus tree's partition number vp and position $v_{position}$ where v could be appeared (see lines 5-6). It then calls the function *search*() with these parameters to find the version v. It also sets the number n to specify the number of consecutive state versions the user would like to find based on state dimensions (e.g., reputation or balance). After that it checks whether vp exists in the root head (see line 12), and if so, it looks for the location of this version v in the B-plus tree, otherwise v does not exist in *PPBPT*. Next, it checks the corresponding prefix values that associated with this position to find the versions to proceed to next (see lines 15-17). To do so, it uses the recursive function *search*() (see line 19) and this process goes up to n times.

7. System Evaluation

We have implemented the proposed approaches based on Hyperledger Fabric, and evaluate its feasibility and performances accordingly in this section.

7.1. Experimental Setup

We implemented the proposed indexing models in Hyperledger fabric v2.0 with the help of chaincode. To set up the Hyperledger environment, we considered two organizations, one orderer, two endorsers, and two peers. Then, we created 16384 transactions for an account under a peer. As we mentioned, our proposed model can index multi-dimensional state values associated with blockchain accounts, therefore we considered that each transaction can modify any dimension of the state values associated with the account. For experiments, we considered two to sixteen dimensions of the state value. Therefore, we generated almost sixteen prefix fields to indicate which state values of the account was modified by blockchain transactions. For example, if a transaction modifies the account's balance and / or reputation of that account, assign 0 to the corresponding prefix field, and 1 to that prefix if its associated state

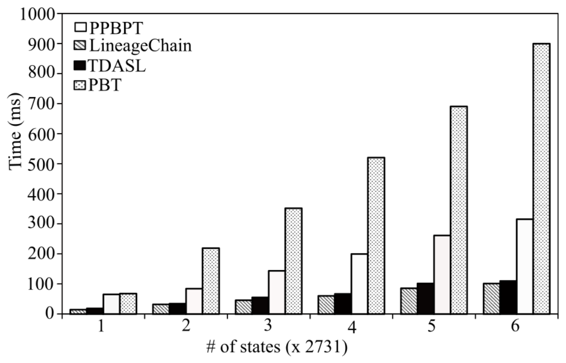


Figure 6: Construction times for *PPBPT*, *LineageChain*, *TDASL* and *PBT*

value is not modified. We constantly checked for updates and increment the prefix value by 1 if transactions did not continuously change a certain state value. To do so, we created a function *invoke* in a chaincode and generated 16384 state versions.

However, existing research [15] implemented their indexing approach for tracking state versions in Hyperledger Fabric by replacing Hyperledger's storage layer with their own implementation. We did not change anything of Hyperledger. However, in order to implement our indexing models, we first store all the state versions with the indexing approaches in chaincode. In particular, we have built three functions; one for *PPBPT*, another for *LineageChain*, and the next for *TDASL*, respectively. The chaincode uses these functions to build the aforementioned indexing approaches individually. Then we deployed the chaincode in Hyperledger Fabric. Moreover, we built an application to interact with the deployed chaincode and used these functions to query the state version.

Our experiments were performed on a system running Ubuntu 20.04, which is equipped with Intel® Core™ i7-10700KF CPU @ 3.80GHz $\times$ 16 and 64GB of RAM.

7.2. Results and Discussion

We carried out several experiments to evaluate the performance of our proposed indexing approaches. The experiments were designed in order to consider five objectives in mind.

In the first experiment, we measured the construction times of the indexing approaches *LineageChain*, *TDASL*, *PPBPT*, and *PBT*. Essentially, this experiment compares the construction times of each model by changing the number of the state versions to observe the construction times required for each model.

In the second experiment, we computed the query time of our proposed approaches in order to compare the performance with the existing model *LineageChain* for single dimensional state version. Moreover, *TDASL* and *LineageChain* use the skip list and in skip list, the nodes' heights are not same. Some nodes appear at express list and some are in the baseline list. Therefore, we believe the nodes that appear at the top levels can be found faster than other nodes. On the other hand, *PPBPT* uses *B*-plus tree where all nodes exist at the same level. In order to investigate this, we divided this experiment into two parts: (i) measure the query time to find the particular state versions that appear at the top levels in *TDASL* and *LineageChain*; and (ii) measure the query time for the state versions that appear in the baseline of *TDASL* and *LineageChain*. It is important to note that no matter where these particular versions are in *PPBPT* as all nodes have the same height in *PPBPT*.

Third, we carried out an experiment to compute the storage needed by these indexing approaches to query single dimensional state version. To be more precise, we evaluated the storage cost of the indexing approaches by varying the version sizes.

In the fourth experiment, we conducted experiments for multi-dimensional state. With this aim, we considered two aspects: i) the query time of *PPBPT* and *TDASL* for searching multi-dimensional state version; ii) the update time for newly generated multi-dimensional state version in indexing models *PPBPT* and *TDASL*. The aim of this experiment is to evaluate the performance of the proposed models *PPBPT* and *TDASL* when the multi-dimensional state versions are frequently queried and updated in blockchain.

Finally, we carried out an experiment to compute the storage space required by these indexing approaches for multi-dimensional state version. More precisely, we evaluated the storage cost of the indexing approaches*TDASL* and

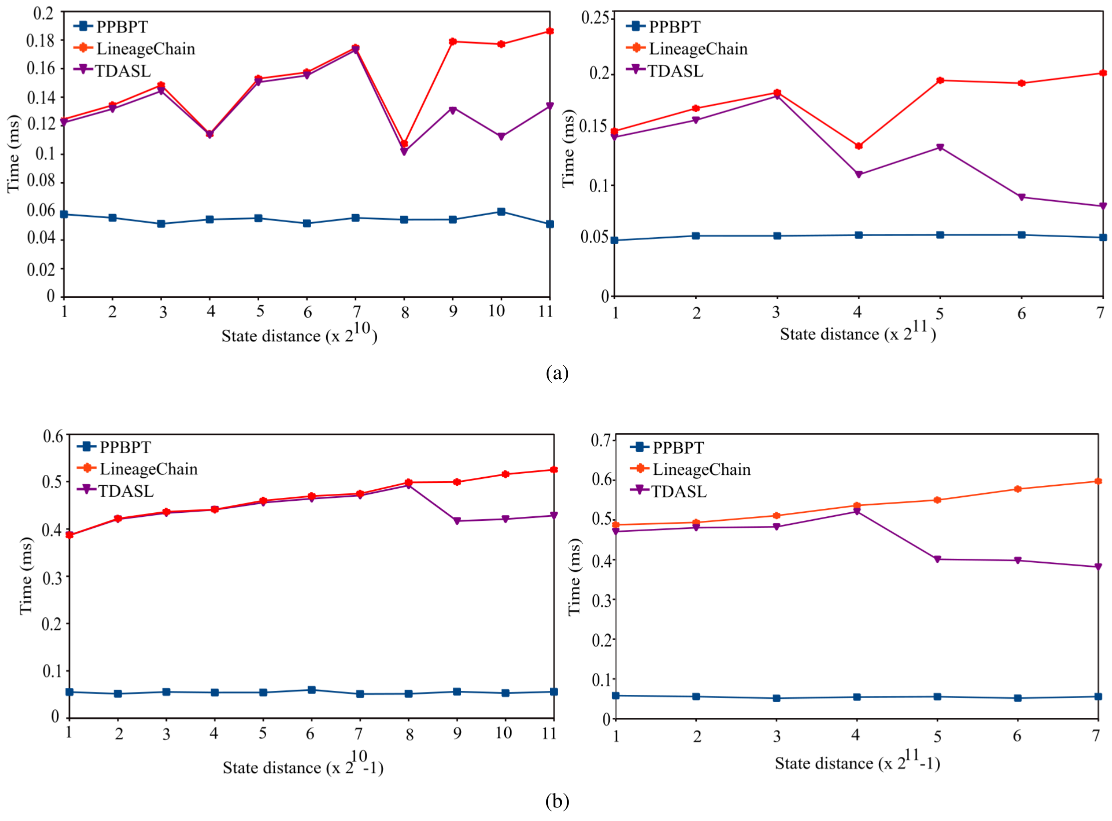


Figure 7: Latency of state version query obtained with *PPBPT*, *LineageChain* and *TDASL* : (a) increasing state distance 2^{10} and 2^{11} (b) increasing state distance $2^{10} - 1$ and $2^{11} - 1$

PPBPT by varying the size of the multi-dimensional state versions.

7.2.1. Construction Time

This experiment focused on showing the construction times for *LineageChain*, *TDASL*, *PPBPT* and *PBT* models by successively taking into account different sizes of blockchain states, as shown in Figure 6. In particular, as we stated that we generated 16384 state versions. We ran this experiment six times by increasing the size of the state versions by 2731 (approximately) shown on x-axis and the time is plotted on y-axis in Figure 6. It can been seen that *LineageChain* requires less time in order to build its index than other approaches. In addition, as several studies have shown, traditional *B*-plus tree requires more construction time because blockchain data is generated more often [14, 21]. It is worth noting that *PBT* can reduce this time [14]. It can be seen from the Figure 6 that the time required for *PPBPT* is significantly less than *PBT*. However, it takes more time than *LineageChain* and *TDASL*.

7.2.2. Single dimensional blockchain state version

In this section, we conducted experiments by considering the single dimension of blockchain state and compute the query time and storage overhead of *LineageChain*, *TDASL* and *PPBPT* in order to compare their performances.

Latency of state versions query. In this experiment, we investigated to query state versions from two aspects. First, we focused on the state versions that appeared at higher levels of *TDASL* and *LineageChain* except at the base level. To do so, we considered the state distances 2^{10} and 2^{11} respectively and increase these distances accordingly in order to compute the query times of the indexing models. Figure 7a shows the state distance in x-axis and query times in y-axis. It can been seen From Figure 7a that *LineageChain* needs more time to query the old state versions, whereas, *TDASL* provides the better performance than *LineageChain* when looking for old state versions. In the case of continuing to track newer versions, its performance is going to equivalent to the *LineachChain*, as shown in Figure 7a. This is because the query distance from the search point becomes equal in the two methods. We also compared

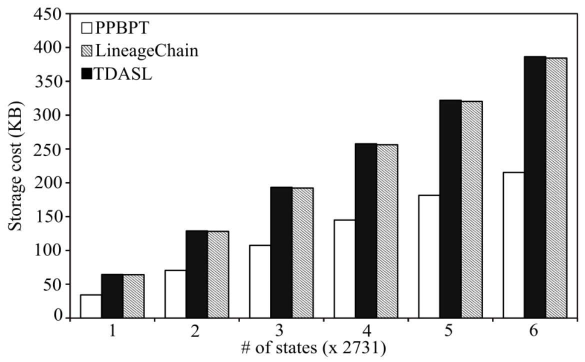


Figure 8: Storage space required for *PPBPT*, *LineageChain*, and *TDASL*

the performance of *PPBPT* with *LineageChain* and *TDASL*. Figure 7a shows that *PPBPT* requires almost the same amount of query time for all versions. Moreover, in Figure 7a, it can been observed that *PPBPT* provides the less query time than *TDASL* and *LineageChain*.

Furthermore, we investigated the performance of *PPBPT*, *LineageChain*, and *TDASL* when querying state versions that appear in the baseline of the skip list. To do this, we calculated the query times of the indexing models by considering the state distances $2^{10} - 1$ and $2^{11} - 1$ respectively and increasing the distances accordingly. Figure 7b shows that *LineageChain*'s query time is increasing gradually from new versions to the old versions. Moreover, from Figures 7b and 7a, it can been seen that the query time of *LineageChain* is longer for the versions appearing at the baseline compared to the versions appearing at the top levels of the skip list. Similarly, it can be seen from Figures 7a and 7b that *TDASL* needs more time to query the versions that exist at the baseline of the skip list. However, it can also be observed that *TDASL* performs better than *LineageChain* when querying the older versions, whereas it takes almost the same amount of time to query the newer versions as compared to *LineageChain*. In addition, it is worth noting that Figure 7b shows *PPBPT* provides the shortest query time for searching any versions than *LineageChain* and *TDASL*. Moreover, it can be observed from Figures 7a and 7b that *PPBPT* needs almost same amount of time to query any state versions. In other words, *PPBPT* provides the same query time for the worst, average, and best cases that we have not seen with *TDASL* and *LineageChain*.

Storage Overhead. In this experiment, we investigated the storage overhead of *PPBPT*, *LineageChain*, and *TDASL*. For this, we considered to design these indexing approaches with increasing number of state versions. In particular, we considered to increase the state versions 2731 for each time and examined the overhead of *PPBPT*, *LineageChain*, and *TDASL* respectively.

Figure 8 shows the result of comparison among *PPBPT*, *LineageChain*, and *TDASL*. It can been observed that *PPBPT*'s overhead is smaller than *LineageChain* and *TDASL*. Moreover, Figure 8 also demonstrates that *TDASL* requires more space than *LineageChain* since it uses an additional layer in the top of the skip list.

7.2.3. Multidimensional blockchain state version

In this experiment, we considered that a blockchain state has multiple dimensions, and if any transaction changes any dimension of the state, the state of the blockchain will be updated accordingly. Therefore, examining update and query times of the proposed indexing models for multi-dimensional state is an important aspect. Moreover, we investigated the storage cost of the indexing models for multi-dimensional blockchain state. For this, we did not consider *LineageChain* as it can not support to search multi-dimensional state value. In the following, we explain these experiments accordingly.

Query latency. We conducted this experiment to check the performance of *TDASL* and *PPBPT* in terms of query latency for multi-dimensional state versions. For this experiment, we considered various dimensions (e.g., 2-D, 4-D, etc.) of the blockchain states with various query distances (e.g., 30, 60, 90, etc.) for computing the query latency. More precisely, we run the query from latest state to find the states by varying query distances, as shown in Figure 9. From Figure 9, it can be seen that *PPBPT*'s query latency is lower than *TDASL* to query the multi-dimensional blockchain state.

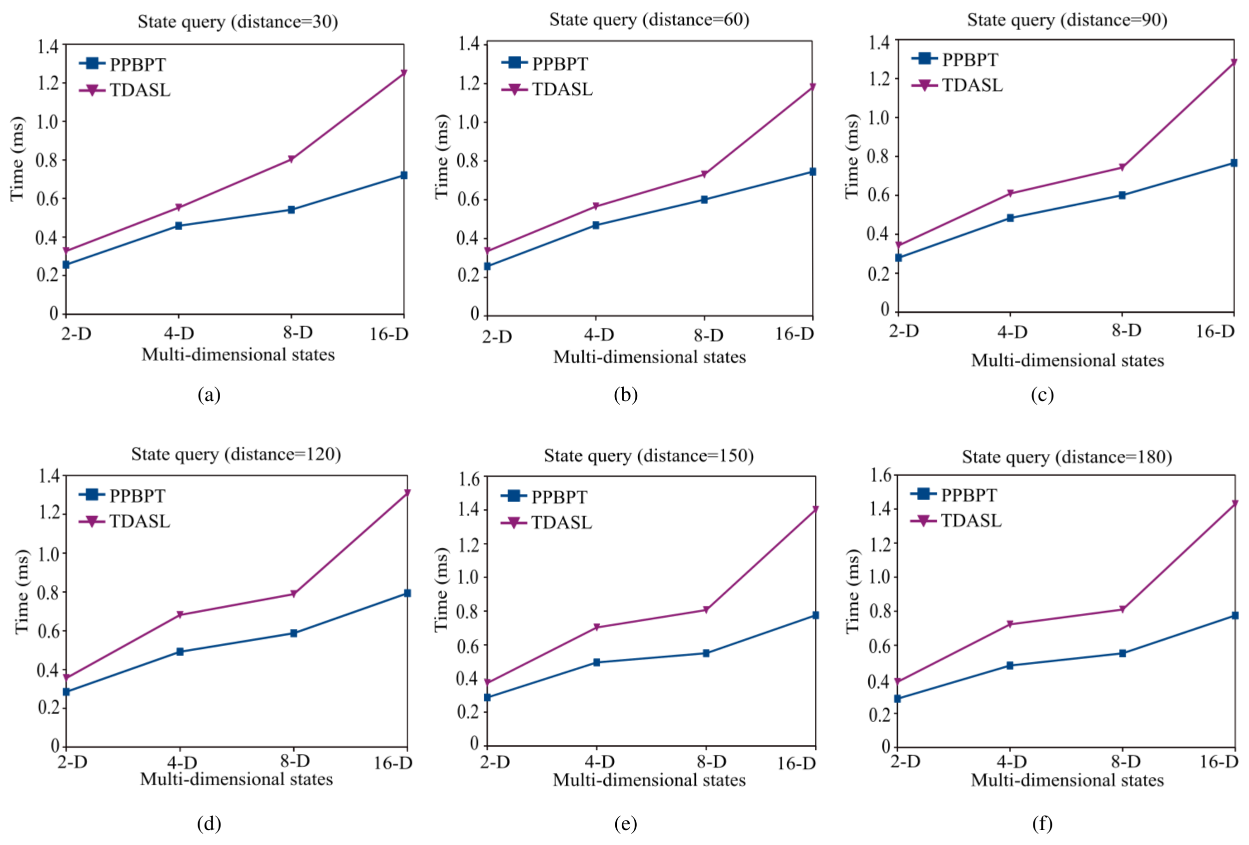


Figure 9: Latency of the multi-dimensional state query with increasing state version query distances

Update time. This experiment investigated the time required by *TDASL* and *PPBPT* to update their indexes with the multi-dimensional blockchain state. It can be observed from Figure 10 that *PPBPT* requires more time than *TDASL* to update its index with the multi-dimensional state.

Therefore, it can be concluded from Figures 9 and 10 that *PPBPT* is more useful for searching purpose, but it is slightly worse in case of updating its index than *TDASL* when data is generated very frequently. In addition, it can also be concluded that *TDASL* is perfect for dynamically generated data as it can update its index in fast way but it needs more time to search the state versions than *PPBPT*.

Storage Overhead. We conducted this experiment to compute the storage cost for *TDASL* and *PPBPT* considering various dimensions of the blockchain state. Figure 11 shows the result. To perform this experiment, we have increased the number of states on the x-axis, and computed the storage for two dimension to sixteen dimension of the blockchain state. The results show that when we considered more dimensions for the blockchain state, each model requires more space. However, it can also be observed that *PPBPT* needs less storage cost for all dimensions compared to that of *TDASL*.

8. Conclusion

In this paper, we have presented two indexing models for blockchain historical data, namely $TDASL$ and $PPBPT$. Our proposed approaches can find the state version much faster than *LineageChain* [15, 16]. Moreover, these proposed models can handle the multi-dimensional historical data in order to efficiently query state versions. Experimental results show that *PPBPT* is more efficient than *LineagChain* and $TDASL$ in terms of query and storage perspectives, but it needs more time than $TDASL$ to build and update its index. Consequently, *PPBPT* is perfectly suitable in the blockchain environments where query performance is essential and data generation frequency is low. On the other hand, $TDASL$ is good to use where more dynamic data is generated as it can update its index quickly. In addition, the

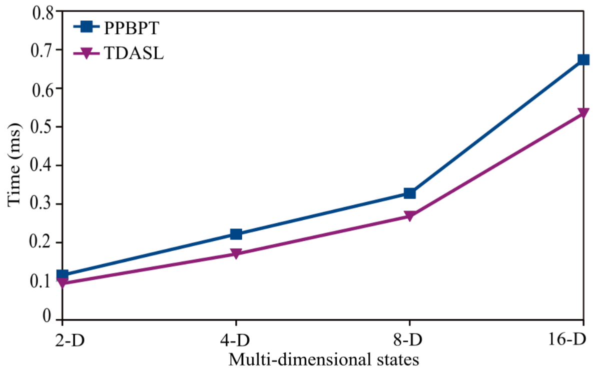


Figure 10: Update times required for multi-dimensional states with *PPBPT* and *TDASL*

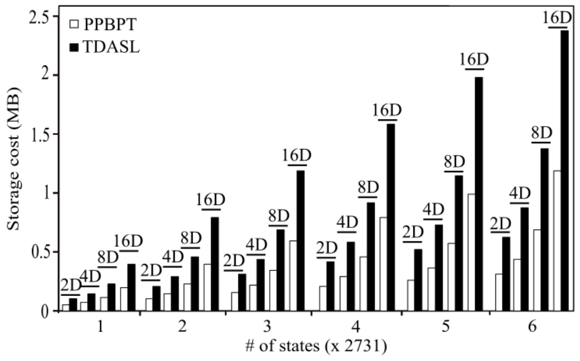


Figure 11: Storage cost required for multi-dimensional states with *PPBPT* and *TDASL*

results also show that $PPBPT$ does not have the worst, average, and best case, so it provides almost the same query times for all state versions, whereas, $TDASL$ and $LineageChain$ provide different query times for the state versions that appear in the baseline and higher levels of the skip list.

Although, $PPBPT$ has a significantly lower construction cost than PBT, but still it has a higher construction cost than $TDASL$ and $LineageChain$. Therefore, in the future, we will try to extend this work in order to reduce the reconstruction and update costs of $PPBPT$. We will also focus on user privacy preferences to query historical blockchain data.

References

[1] P. Du, Y. Liu, Y. Li, H. Yin, and L. Zhang. Etherh: A hybrid index to support blockchain data query. In *ACM Turing Award Celebration Conference-China (ACM TURC 2021)*, pages 72–76, 2021.

[2] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb: A shared database on blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019.

[3] M. El-Hindi, M. Heyden, C. Binnig, R. Ramamurthy, A. Arasu, and D. Kossmann. Blockchaindb-towards a shared database on blockchains. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1905–1908, 2019.

[4] J. Gehrke, L. Allen, P. Antonopoulos, A. Arasu, J. Hammer, J. Hunter, R. Kaushik, D. Kossmann, R. Ramamurthy, S. T. Setty, et al. Veritas: Shared verifiable databases and tables in the cloud. In *CIDR*, 2019.

[5] G. Graefe. Sorting and indexing with partitioned b-trees. In *CIDR*, volume 3, pages 5–8, 2003.

[6] H. Gupta, S. Hans, S. Mehta, and P. Jayachandran. On building efficient temporal indexes on hyperledger fabric. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 294–301. IEEE, 2018.

[7] Z. Liu and Z. Li. A blockchain-based framework of cross-border e-commerce supply chain. *International Journal of Information Management*, 52:102059, 2020.

[8] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[9] M. Mettler. Blockchain technology in healthcare: The revolution starts here. In *2016 IEEE 18th international conference on e-health networking, applications and services (Healthcom)*, pages 1–3. IEEE, 2016.

[10] B. K. Mohanta, S. S. Panda, and D. Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4. IEEE, 2018.

[11] S. Nakamoto and A. Bitcoin. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 4, 2008.

[12] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *arXiv preprint arXiv:1903.01919*, 2019.

[13] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[14] C. Riegger, T. Vinçon, and I. Petrov. Efficient data and indexing structure for blockchains in enterprise systems. In *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, pages 173–182, 2018.

[15] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment*, 12(9):975–988, 2019.

[16] P. Ruan, T. T. A. Dinh, Q. Lin, M. Zhang, G. Chen, and B. C. Ooi. Lineagechain: a fine-grained, secure and efficient data provenance system for blockchains. *The VLDB Journal*, 30(1):3–24, 2021.

[17] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen. Blockchain technology and its relationships to sustainable supply chain management. *International Journal of Production Research*, 57(7):2117–2135, 2019.

[18] F. M. Schuhknecht, A. Sharma, J. Dittrich, and D. Agrawal. Chainifydb: How to blockchainify any data management system. *arXiv preprint arXiv:1912.04820*, 2019.

[19] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, W. Fu, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *arXiv preprint arXiv:1802.04949*, 2018.

[20] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[21] H. XiaoJu, G. XueQing, H. ZhiGang, Z. LiMei, and G. Kun. Ebtree: A b-plus tree based index for ethereum blockchain data. In *Proceedings of the 2020 Asia Service Sciences and Software Engineering Conference*, pages 83–90, 2020.

[22] C. Xu, C. Zhang, and J. Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*, pages 141–158, 2019.

[23] C. Yue, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, S. Wang, and X. Xiao. Analysis of indexing structures for immutable data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 925–935, 2020.