Efficient Data and Indexing Structure for Blockchains in Enterprise Systems (Extended Abstract)

Christian Riegger Tobias Vinçon Ilia Petrov

Data Management Lab, Reutlingen University
{first}.{last}@Reutlingen-University.DE

October 10, 2018

Abstract

Blockchains yield to new workloads in database management systems and K/V-Stores. Distributed Ledger Technology (DLT) is a technique for managing transactions in 'trustless' distributed systems. Yet, clients of nodes in blockchain networks are backed by 'trustworthy' K/V-Stores, like LevelDB or RocksDB in *Ethereum*, which are based on Log-Structured Merge Trees (LSM-Trees). However, LSM-Trees do not fully match the properties of blockchains and enterprise workloads.

In this paper, we claim that Partitioned B-Trees (PBT) fit the properties of this DLT: uniformly distributed hash keys, immutability, consensus, invalid blocks, unspent and off-chain transactions, reorganization and data state / version ordering in a distributed log-structure. PBT can locate records of newly inserted key-value pairs, as well as data of unspent transactions, in separate partitions in main memory. Once several blocks acquire consensus, PBTs evict a whole partition, which becomes immutable, to secondary storage. This behavior minimizes write amplification and enables a beneficial sequential write pattern on modern hardware. Furthermore, DLT implicate some type of log-based versioning. PBTs can serve as MV-Store for data storage of logical blocks and indexing in multi-version concurrency control (MVCC) transaction processing.

1 Introduction

In academia and commercial applications blockchains get growing attention as a possible technology for Distributed Ledgers. The cryptocurrencies *Bitcoin*[1, 2], as avantgarde of blockchains, and *Ethereum*[3, 4], which is more powerful, due to trustworthy executable code in *smart contracts*, enable secure transaction processing in 'trustless' distributed peer-to-peer networks by proof of consensus.

We give an overview of a blockchain network (depicted Fig. 1). Blockchains constitute a backward linked list of blocks, containing *transactions*. A blockchain network consists of several nodes. A blockchain is entirely replicated to each node. On every node, a blockchain network client is implemented (e.g Bitcoin-Core[5], Geth[6], Parity[7]). Clients are responsible for any operation to interact with the network. K/V-Stores are the backbone of clients and manage blockchain (meta) data and indexes. Indexes are necessary for data retrieval in the log-based DLT, without scanning the whole blockchain. Businesses may include blockchain data in their enterprise systems, by maintaining a virtual node and avoiding slow messaging requests or ETL-processes (depicted Fig. 2). A virtual node is part of the blockchain network, hosted by the enterprise system.

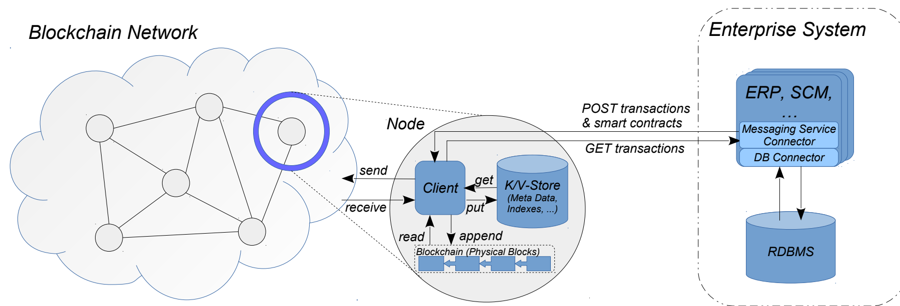

Figure 1: Blockchain Technology loosely coupled with Enterprise Systems: *Blockchain data is separated from indexes and meta data in K/V-Stores. Enterprise systems have to interact with the blockchain network by messaging services to a network client of a node. Current architecture, format conversions, etc. result in high latencies and stagnation in business processes.*

Accounts (human or machine) are able to negotiate the pricing for services in *smart contracts* and post these *unspent transactions* to the Distribted Ledger network. Miners in *Bitcoin* or validators in *Ethereum* (a special type of node) collect *transactions* and check for prerequisites (e.g. if the account's balance is sufficient) for accomplishing. Several *transactions* are collected and added to a block built up on hashes. The validator tries to append the created block at the end of the blockchain, containing a backward link to the latest block, its hash. New blocks are synchronized to every node in the blockchain network. If the block is approved by the network, the validator gets higher priority. There are several models for this process of consensus. Once a block was appended to the blockchain, it is impossible to modify covered *transactions* and data because the hash value of the block would change. Blocks may get rejected by the community and reorganizations are required.

Characteristics of Blockchain Data. Uniformly distributed hash values of blockchain data, e.g *accounts*, *transactions*, and *smart contracts*, are stored in

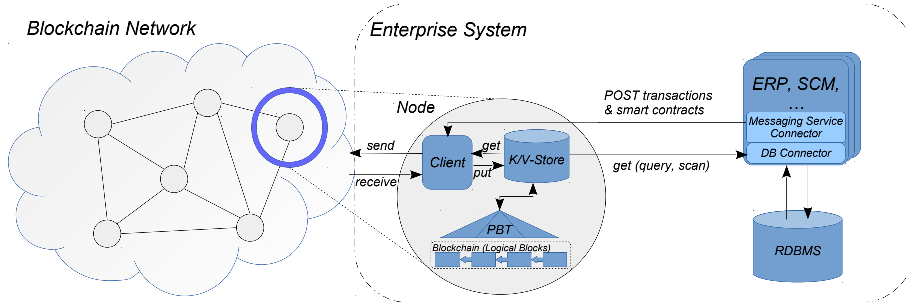

Figure 2: Blockchain Technology included in Enterprise Systems: *Including the K/V-Store in an enterprise system as a database reduce latencies, format conversions, etc. to a minimum. Therefore, a powerful and flexible data and indexing structure is required in K/V-Stores, that is able to store and manage all data of a blockchain in a "Logical Blocks" schema. It is not necessary to physically store blocks separated from indexing. Required data can be directly accessed by the enterprise system, if all required data is located in the K/V-Store. Decrypted information can be stored in authorized tablespaces, whereby range scans are supported*

blocks of a blockchain. Each *transaction* is analogous to a log-based version state in the Distributed Ledger. Newer data is stored in newer blocks. Blocks retaining consensus (based on the consensus model) as immutable data. Recently appended blocks are maybe rejected by the network and their data becomes invalid and / or reorganized. *Unspent transactions* are not immediately added to the blockchain, but have to be validated and processed first.

Although appending transactions to a blockchain perform well, there is a scalability bottleneck in blockchain networks, handling every transaction on-chain, as these transactions have to be broadcasted to the whole network. Low throughput[1] depends on these bottlenecks and consensus models (PoW, PoS, ...). Techniques like sharding[12] and off-chain transactions[11], based on *smart contracts* (ERC20 Tokens) can improve throughput and scale linear with the number of users in the network[10]. Moreover, private blockchains (like Hyperledger[13]) outperform public ones, due to lower complexity in consensus models [14]. Querying a blockchain is a much harder task with linear complexity, because in theory the whole blockchain has to be processed for data reconstruction, back to its initial block. For this and further purposes, clients of nodes in the blockchain network are backed by K/V-Stores. In *Ethereum* for instance, Geth applies

[1] *Bitcoin* can handle about 7 transactions per second (tps)[8] and *Ethereum* about 15 tps[9]. Payment systems like Paypal (115 tps in average) and VISA (peak at 47.000 tps)[8] allow an order of magnitude more tps, than blockchain based ones. Raiden[10] is in development which uses *Ethereum smart contracts* to increase throughput by off-chain transactions, promised to scale with number of users[11].

LevelDB[15] and Parity makes use of RocksDB[16]. Both K/V-Stores are built up on Log-Structured Merge Trees (LSM-Trees).

Although, LSM-Trees can handle high update rates well, their characteristics do not match the properties of blockchain data ordering. Mixed workloads, comprising payloads on the blockchain network and enterprise OLTP and OLAP workloads on virtual nodes in the network (as depicted in Fig. 2 and outlined in Section 6), will exceed performance of current K/V-Stores, if their data structures do not fully leverage characteristics of modern processing and storage technologies (outlined in Section 2) as well as properties of blockchain data.

In this paper, we claim that Partitioned B-Trees (PBT) [17] are able to overcome complexity of DLTs, handling mixed OLTP and OLAP workloads from enterprise systems in K/V-Stores, and leverage characteristics of modern hardware, due to their flexible partition management in one single index structure.

First, we give a short introduction in the characteristics of modern hardware technologies in Section 2 and outline the state of the art data structure LSM-Tree and the suggested data structure Partitioned B-Trees in Section 3. In Section 4, we compare both data structures in handling the complexities of blockchain data and how they perform as schema and indexing data structures in Section 5. Finally, we investigate workloads from enterprise systems on both data structures in Section 6.

2 Modern Hardware Technologies

Architectures and algorithms are optimized for characteristics of traditional hardware. Only two levels of the memory hierarchy were focused in database management system algorithms - main memory and disk[18]. There was a low relation in capacity and a huge access gap due to high latencies of disks (depicted in Fig. 3). Algorithms like the ubiquitous B^+-Tree[19] are optimized for this access gap.

Several trends in hardware technologies lead to rethinking of traditional architectures and algorithms. Developments in computing and storage technologies as well as increased main memory volumes require reflections of an optimal utilization of layers in the memory hierarchy and parallel computing concepts.

In contrast to traditional architectures, more levels in the memory hierarchy have to be considered and evaluated (depicted in Fig. 3). The number of cores within CPUs increase permanently, GPUs are used for computing in general purpose and field programmable gate arrays (FPGAs) allow processing enormous amounts of data in-situ. In general, these trends lead to higher levels of parallelism and improved computing performance. Minimizing idle times of processing units by optimizing data placement in modern memory hierarchies as well as leveraging characteristics of storage technologies should be focused in DBMS and K/V-Store algorithms and data structures.

Non-volatile semiconductors, like non-volatile memories (NVM) and solid state drives (SSD / Flash), distinguish from characteristics in several ways. There is an asymmetry in latencies of reading and writing data. Furthermore,

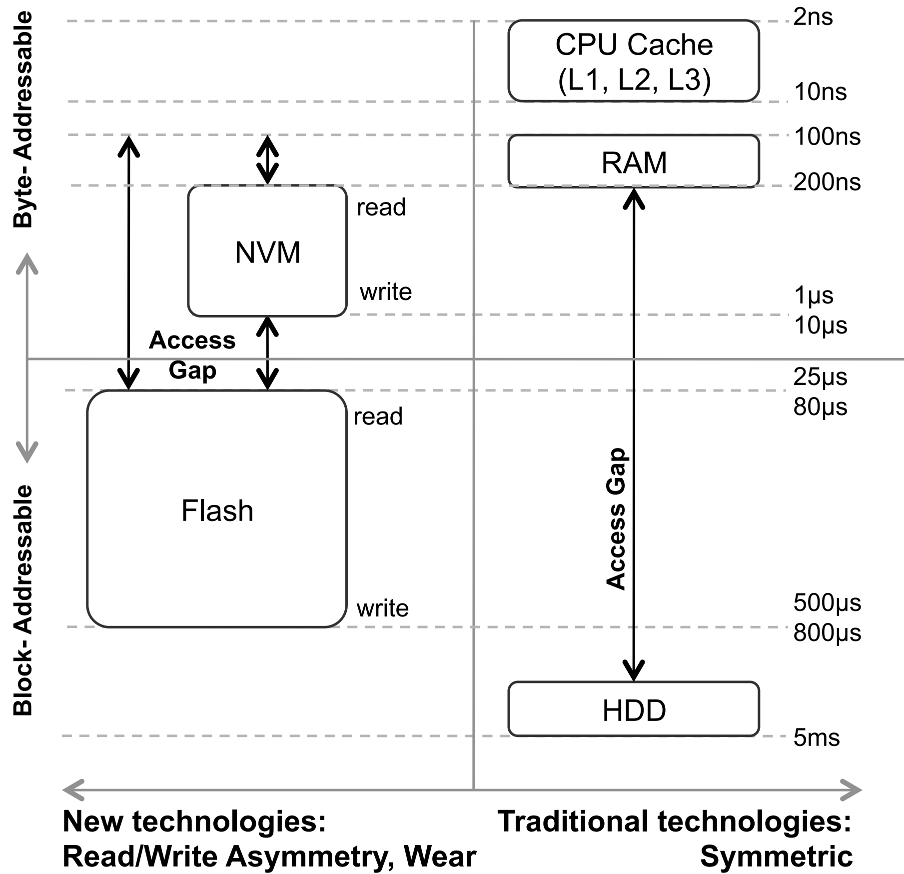

Figure 3: Complex Memory Hierarchy in modern Hardware

writes wear out storage cells and reduce the longevity of a storage medium. Like DRAM, NVMs are byte-addressable, but persist data even after a power loss. However, they suffer of higher I/O latencies. SSDs / Flash have shorter latencies than disks and a high internal parallelism. Sequential writes benefit latencies as well as data placement in blocks. Updates are undesirably stored out-of-place in new pages, and as a result, garbage collection has to be performed and write amplification increase.

Data structures need to leverage complex memory hierarchies and developments in processing technologies. Modifications can be performed well in main memory, due to short symmetric latencies for reads as well as writes. Data located in NVMs and SSDs should stay immutable, due to high asymmetric latencies in writes and wear. Furthermore, immutable data on secondary storage enable a good configuration for Near-data Processing (NDP) on FPGAs.

Result sets can be computed in-situ and propagated to main memory without considerations about concurrent updates in CPU caches or RAM.

3 Overview: Data Structures

In this section, we give a short overview of the data structures and algorithms of Log-Structured Merge Trees (LSM-Trees) [20] and Partitioned B-Trees (PBT)[17].

3.1 Log-Structured Merge Trees

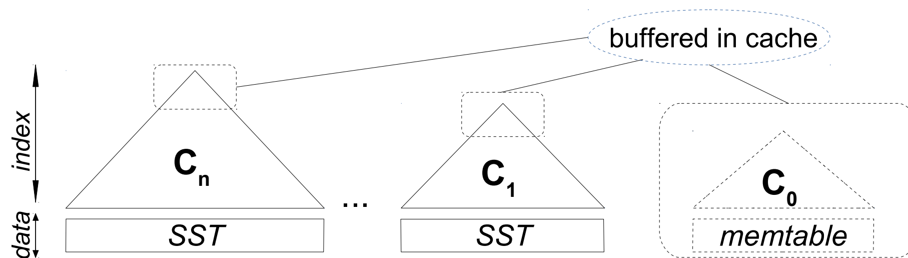

Figure 4: Structure of a LSM-Tree: *Several components are placed on different levels in memory hierarchy and maintain a separate index structure. C_0 memtable absorbs updates in main memory. Merges reduce number of components and enable garbage collection. Version ordering of blockchain data is intermingled as part of evolvement on current workload and write amplification increase, due to merges and rewrites of principally unchanged data.*

LSM-Trees[20] are optimized for high random update rates, as well as bulk loads, and achieve sequential write patterns to secondary storage media by presorting inserted and updated data in a fixed sized C_0 component, which is located in main memory. Components (C_1 to C_n) resident on secondary storage media. The data structure is depicted in Fig. 4. Evictions of C_0 component and merge operations between all components are performed well in out-of-place appends. LevelDB and RocksDB achieve this behavior by implementing immutable sorted string tables (SST) on secondary storage and mutable indexes and SST (memtable) in memory. SSTs are merged in a compaction procedure at specific points in time.[15, 16]

Read amplification increases with every new component, what result in inacceptable look-up and scan performance [21]. Each component maintains its own (tree-based) index structure. Components are sequentially processed from C_0 to C_n, until a matching record is found. Merges lower the effects of read amplification, but increase write amplification undesirably. In a write-heavy

workload[2] more components are created and have to be merged. The main memory C_0 component gets full, before merge operations terminated. It is flushed to secondary storage and reclaims space in main memory for further updates. The more components are created, the more already written data has to be merged for acceptable read performance. Secondary storage media is stressed by merge operations, of in principle unchanged, immutable and uniformly distributed hash-key data in a blockchain workload. Read amplification can be further reduced by bloom filters and reduction of components with intelligent merge scheduling in bLSM-Trees [21].

Scheduling merges of uniformly distributed hash-key values is difficult, because there is no mentionable skew in workloads. Keys always overlap and all data of components is processed and appended in a merged component. This behavior increases write amplification and does not match the version data ordering characteristic of blockchain data. Furthermore, *unspent transactions* and data of rejected blocks bring more problems to the LSM-Tree. Modifiable records may be evicted to secondary storage media or merged with further components. This records are updated by insertions of replacement records and garbage collected on merge operations.

Predefined component size thresholds lead to more existing components and separation of data as actually required. More data has to be moved and merged to unwind components. Furthermore, administrative effort is necessary to define these thresholds in a complex schema. With static thresholds in component sizes, it is impossible for the K/V-Store to react on switches in workload. Self-balancing component size thresholds would be more valuable.

In a short resume, LSM-Trees enhance throughput for random update-intensive workloads, due to a beneficial sequential write pattern on secondary storage media and modifications in main memory, at the cost of look-up performance. Merge operations in compaction procedures lead to undesirable high write amplification of actually unchanged data. Switches in workloads are not supported, due to inflexible component size thresholds.

3.2 Partitioned B-Trees

We implemented and extended the Partitioned B-Tree (PBT) [17] as an adaption of the traditional B$^+$-Tree, outlined in [22]. PBT makes use of most B$^+$-Tree algorithms, with few modifications. The essential difference is the introduction of an artificial leading search key column – the partition number. A record consists of a partition number, its search key columns and a record-id for indexing or a string of values for data storage. Every partition number uniquely identifies a single partition. This idea enables the PBT to maintain partitions within one single sorted tree structure and reusing existing algorithmic optimizations, buffering, etc. Partition numbers are transparent for higher database layers and each PBT maintains partitions independent from other PBTs. Partitions appear

[2] Using a blockchain for transaction management, e.g. in an IoT-scenario or payment system with off-chain transactions, implies an update-intensive workload on a nodes K/V-Store.

and vanish as simple as inserting or deleting records and can be reorganized and optimized on-line in merge steps, depending on workloads [17]. Partitioning data within a single B$^+$-Tree enables several possible applications (outlined in [17, 22]). In this paper, we focus on workloads in the outlined blockchain enterprise system scenario.

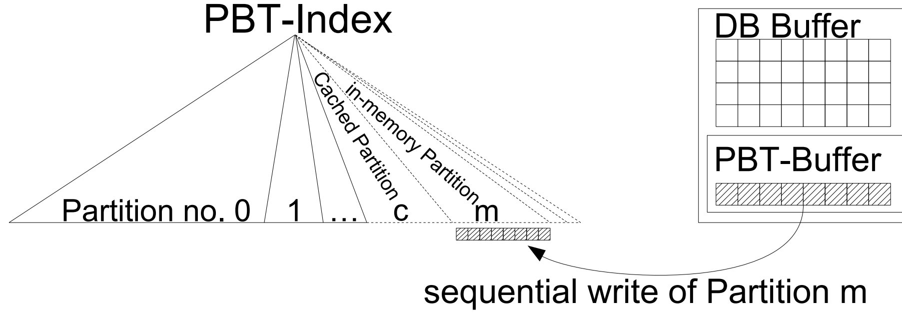

Figure 5: Structure of a Partitioned B-Tree: *Each partition is part of a common index structure. Partitions are located on different levels in memory hierarchy. Updates are absorbed in main memory by a mutable partition. Further complexity can be handled by additional active partitions. Partitions on secondary storage are immutable and cannot be modified from current workload. Version ordering of blockchain data buoys, but can be reorganized for query performance in internal transactions, as well as Cached Partitions can be created from result sets of frequently queried data and reduce read amplification.*

Traditional B$^+$-Trees tend to have the disadvantage of a high write amplification[22], especially for uniformly distributed hash-key values without mentionable skew, like in a blockchain workload. PBTs write any modification of records exactly once at eviction time of the respective partition, except for subsequently performed reorganization or garbage collection operations. This is realized by forcing sequential writes of all leaf pages of an entire partition. The operation is illustrated in Fig. 5. Leaf nodes of a mutable partition are stored in a separate area of the database buffer – the PBT-Buffer. The PBT-Buffer is shared for all PBTs in a database for balancing partition sizes based on workloads. Records can be inserted, updated and deleted only in mutable partitions in the PBT-Buffer. Once the PBT-Buffer gets full, it selects a victim partition, which becomes immutable, and writes it sequentially to secondary storage. A simple well sized bloom filter is created as part of the eviction process.[22]

The entries of a PBT-Buffer resident partition are also part of the higher levels of the PBT index. Hence, they are traversed as part of any look-up operation first.

It is fair to say, that all modifications are cached and leaf pages have a near optimal write amplification. Inner nodes in the PBT are passed to the regular cache logic. In fact, that modifications are applied to its latest mutable

partitions, as well as look-ups are also directed to high numbered partitions first, these inner nodes are cached as well and evictions are very rare for most replacement policies. As a result, write amplification of inner nodes is low and traversals are fast.

Updates of already written data in previous partitions require an out-of-place invalidation. Inserting a Replacement-Record in the mutable partition is sufficient. Deleting a record in an immutable partition requires an insertion of an Anti-Record in the new partition [17]. Therefore, Replacement-Records can be viewed as a combination of an Anti-Record and a newly inserted Index-Record, if its search key changes. In the search algorithm, Anti-Records matching to the search criteria can be collected and invalidated Index-Records can be excluded from the result set. This is possible, because the PBT starts probing the latest partition and ends up in the oldest partition, except the search algorithm can break up earlier [22]. In case of unique hash keys like in blockchain data, the algorithm can break on the first matching Index- or Replacement-Record.

Probing every partition leads to a higher read amplification by a factor of the partitions count in comparison to a traditional B^+-Tree. There is a certain impact on performance, because well sized bloom filters protect immutable partitions from unnecessary traversals. Newly inserted data is well buffered in main memory and data can be reorganized [22]. Cached Partitions can be built up on frequently queried data and located in main memory. Records in Cached Partitions are of the Replacement-Record type. As a result, frequently queried data can be found without processing further partitions. In case of switches in the workload, Cached Partitions can be easily cropped from the tree structure without data loss. If it is valuable to keep Cached Partitions, e.g. for OLAP, it is possible to evict them to secondary storage. If the key range of a Cached Partition is large enough one or more partitions (which are in the key range of the Cached Partition) can be cropped from the tree structure and occupied space can be freed. This is one possible approach for garbage collection in PBT. Furthermore, modern storage media support fast reads and FPGAs enable in-situ data processing.

Additional mutable partitions can be maintained for a PBT in main memory. These partitions can handle further complexity in workloads from blockchain data, such as bulk loads (data from recently appended blocks) and *unspent transactions* (mutable data, which is not yet appended to a block), in respect to characteristics of modern hardware technologies as outlined in Section 4. In a bulk load, data can be absorbed by a separate partition with low effects on current workload and merged in-memory on commit time. If the transaction fails, the entire partition can be cropped from the tree structure in one single step, because data is not intermingled with data from other transactions.

When maintaining further mutable partitions, it is possible for a PBT to serve as a version store in multi version concurrency control (MVCC). Therefore, a timestamp, either for validation or invalidation, based on the record type, can be maintained for every record. Consequently, Multi-Version PBTs (MVPBTs) are able to perform index-only visibility checks [17, 23].

PBT seems to be a flexible data structure, that matches characteristics

of modern hardware technologies, as well as mixed workloads in a blockchain enterprise system scenario. Flexible partition management (including partition sizes, which are based on current workload and resources; separation of mutable and immutable data in different in-memory partitions; out-of-place updates in mutable partitions; etc.) minimize write amplification and enable a beneficial sequential write pattern on secondary storage media. Cached Partitions, simple bloom filters and asymmetric fast reads from secondary storage media guarantee acceptable look-up and scan performance. Garbage collection is performed, if required, decoupled from current workload.

4 Blockchain Data Storage

Clients of nodes in *Ethereum* are backed by K/V-Stores, like LevelDB or RocksDB, which are based on LSM-Trees for storage. As outlined in Section 3.1, LSM-Trees are not optimal for storing blockchain data, due to high write amplification of immutable data on merge operations. Partitioned B-Trees (PBT) leverage characteristics of blockchain data and modern hardware technologies. This enables PBT to store logical blocks of the blockchain as well as provide further indexes for query optimization from enterprise workloads (this configuration is depicted in Fig. 2).

In the following we compare the effect of LSM-trees and PBT with respect to different blockchain properties / features such as: storing uniformly distributed hash-keys, immutability and consensus, invalid blocks and reorganization, unspent and off-chain transactions, log-structured data version and state ordering, distributed operations and node synchronization.

Uniformly distributed hash-keys. Data in blockchains is identified by unique hash-keys. These hashes exhibit uniform distribution in a range of keys. B^+-Trees are predestined for querying and scanning data with logarithmic complexity. Unfortunately, modifications, like inserts or updates, of uniformly distributed keys result in poor performance, due to high write amplification on eviction of modified leaf pages and structure modifications. Hash-based structures guarantee faster look-ups for point queries, but do not support scans and result in a similar undesirable random write pattern on secondary storage. LSM-Trees overcome this problem by presorting keys of the current load in-memory and writing out components. Complexity in query and scan data is amplified by the number of components, thus merges have to be performed. *PBTs* handle modifications almost like LSM-Trees, due to its in-memory PBT-Buffer, that absorbs current updates. Bloom filters reduce complexity in querying data in *PBT*, a technique that is also applied for LSM-Trees in RocksDB. The major benefit of *PBT* is the single index structure for all partitions, because capacity and height of the tree are in a logarithmic relation. Several small components in a LSM-Tree require more index nodes than one single component. For large data sets, the tree height of each component is similar to the absolute height of a *PBT*. Index nodes are commonly processed and buffered in a single index structure.

As a result, traversal operations in a LSM-Tree are more expensive than in a *PBT*. Furthermore, recently inserted data is guaranteed to be located in a higher numbered partition, that is queried first. Due to this fact, skewed reads of unique hash-keys can be answered in-memory. If the partition of a frequently requested record is evicted to secondary storage, a Replacement-Record is inserted in a Cached Partition in main memory. LSM-Trees maybe evict their comprising component or merge it with further components on secondary storage.

Immutability and Consensus of Blockchain. Every appended block in a blockchain is immutable. Blocks can accomplish consensus, based on several models. As a result, data of confirmed blocks in a K/V-Store require a similar behavior for resource-gentle storage. LSM-Trees manage data in components, which are merged and already persisted data have to be rewritten to secondary storage media, whereby write amplification undesirably increase. Evicted partitions in *PBTs* and most of its index nodes remain immutable on secondary storage, similar to confirmed blocks (and comprised data) in a blockchain. For query optimization, partitions can be reorganized and merged, like components in a LSM-Tree. These operations can be decoupled from current workload, e.g. in times of low occupancy. Partition sizes flexibly depend on current workload and storage resources in contrast to fixed thresholds of LSM-Trees. Flexible partition sizes and merge operations decoupled from current workload reduce write amplification. Furthermore, in *PBT*, frequently queried data and key ranges can be applied to a main memory partition for fast look-ups. Cached Partitions contain replacement records for this data. As a consequence, further partitions do not have to be traversed and queries can be answered in-memory. In case of workload switches, Cached Partitions can easily be cropped from the tree-structure, if it is not valuable to keep it in main memory. Moreover, if *intelligent storage* is used, large parts of data processing, e.g. in case of OLAP queries, can be performed in-situ using Near-data Processing (NDP) techniques, to avoid expensive data transfers and increase performance. Outlined techniques minimize required merge operations and write amplification for *PBT*.

Invalid Blocks and Reorganization. Recently appended blocks in a blockchain may get rejected by the network, e.g. in case of forking, not feasible transactions, etc., and force reorganizations. In a K/V-Store, data from such blocks is to be marked as invalid and eventually get pruned. Forks, however, can span several blocks and therefore the amount of invalid data can be high. [24] suggest to index only data of consensual blocks in a linked data index for performance improvement, because of enormous effort in handling these issues. This strategy imposes further complexity, while sequentially scanning the newest blocks in a blockchain. LSM-Trees may have evicted a component, which handles data of these blocks, or even worse, already merged it with further components. Removing this data requires invalidation via tombstone records in the C_0 component and merges with further components. *PBTs* can handle this complexity by maintaining more in-memory partitions, containing data of

recently appended blocks. Once a block is appended to a blockchain, its data is applied to an empty partition in a bulk load operation. If a block gets rejected by the network, a partition, which is handling its data, can easily be cropped from the structure. Data in partitions of consensual blocks can be moved in the regular in-memory partition in a further merge operation.

Unspent Transactions in the Network. *Unspent transactions* are coming up next for processing in the network, but are not yet part of the blockchain. These *transactions* are relevant for miners / validators as well as provider of enterprise services for forecasting and acceleration of pre-processes (as outlined in Section 6). *Unspent transactions* may get appended to a new block, stay long time unprocessed, or even never get processed. K/V-Stores should be able to handle *unspent transactions*. LSM-Trees include data from *unspent transactions* in the main store or require a separate data structure. The location as well as the processing time is unknown, therefore in meantime, data can change or get invalidated. LSM-Trees may evict or merge data of *unspent transactions* and require to insert replacement or tombstone records and further merges. *PBTs*, on the other hand, collect *unspent transactions* in a separate partition in-memory and migrate them to further in-memory partitions at processing time.

Off-Chain Transactions via Smart Contracts. *Ethereum* provides the possibility to conclude *smart contracts*. These functions can be used to accomplish off-chain transactions, e.g. exchanging tokens in Raiden[10]. Off-chain transactions do not require consensus by the common blockchain network and scale with the number of users, which accomplished such types of *smart contracts* in-chain. Enterprises may include this workloads in the K/V-Store of their virtual node. LSM-Trees are able to absorb high update rates of such workloads by the C_0 in-memory component. *PBTs* have a mutable partition in main memory, which absorbs high update rates, like a C_0 component in LSM-Trees. *PBTs* use a common index structure for all partitions (as outlined in Section 3.2). As a result, queries and scans benefit from common processing and buffering of index nodes (as already described in Paragraph *Uniformly distributed hash-keys*). As a result, *PBT* handle high update rates similar to LSM-Trees, but benefit from cheaper traversal operations.

Log-Structured Data Version Ordering in the Blockchain. Over time, blockchains grow from an initial block up to a backward linked list. Every new block contains a backwards link to a predecessor block (its hash value). As a result, old and maybe "cold" data states / versions reside in old blocks near to the initial block in the ledger. New and "hot" data states / versions are located at the “hot” end of the list. Data structures in a K/V-Store should adopt this characteristic. LSM-Trees typically lose this property as they evolve over time, because of the regular merge operations. Data is migrated from in-memory C_0 component to further components (C_1 to C_n) on secondary storage devices. As a result, "cold" and "hot" data are intermingled in large persistent components on

secondary storage. *PBTs* are able to merge intrinsically immutable partitions for better read performance, yet it is also possible to keep data ordering in line with the blockchain ordering (as depicted in Fig. 6). Recently inserted data is located in newer partitions, which are queried first. These partitions can be located in main memory as well as in fast non-volatile memories, and can be aged out to slower, but cheaper, storage devices over time. If older data is frequently queried, it is possible to locate it in a Cached Partition in-memory with replacement records, so requests on older partitions on slower storage devices are rare. For analytical processing of data, FPGAs can process immutable partitions near its storage location and result sets can be propagated to CPU and merged in one single step.

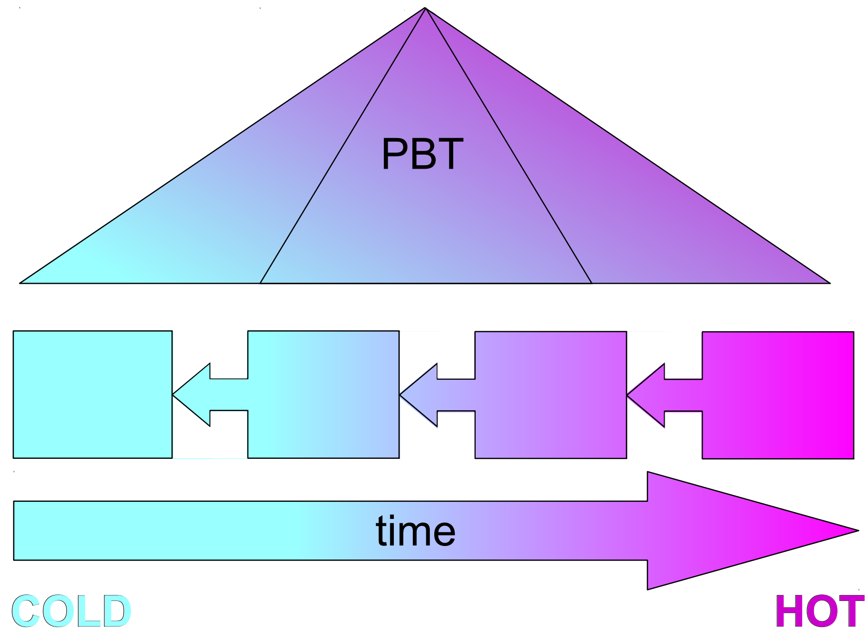

Figure 6: Partitioned B-Trees and Blockchains Data Version Ordering is in line.

Distributed Operations. In a distributed ledger, blocks are created and processed from different nodes and synchronized over the entire network. This is possible because of the immutability of a block. Synchronizing K/V-Store operations is not that easy. It is cheaper to process small operations (e.g. insertions in a SST) on every node in main memory, rather than share effort and synchronize it with further K/V-Stores. *PBTs* perform current workload also on every node. Reorganization activities enable distributed processing, because of the immutability of partitions. For query optimization a low occupancy node can

be instructed to merge several partitions and broadcast leaf nodes of one merged partition over the network. This work could also be shared based on key spans. Furthermore, nodes can build index pages within the partition and assimilate the subtree. Afterwards, merged partitions can be cropped from the tree-structure concurrently with low effects on current workload. Partition management is performed on every node based on system and hardware configurations, so partition sizes and data ordering may differ. Correctness is guaranteed, because only immutable partitions are merged and records in high numbered partitions replace data of low numbered ones. If a partition of a further node is not fully merged, it would not be cropped from the tree structure and data remains entirely consistent. With versioning information about block hashes of the immutable blockchain, fully merged partitions can be easily identified.

Node Synchronization. Physical blocks can be replicated to further nodes with low effort. The hash identifier of the latest block is required and all successor blocks can be sequentially read and synchronized. When storing blocks of a blockchain logically in the K/V-Store, replication to further nodes becomes a problem. New nodes are added to the blockchain network or long-serving nodes come up after a crash or version update. Blocks have to be replicated to this nodes. The data structure, which contains logical blocks, should consider their version ordering for fast synchronization. LSM-Trees intermingled blocks version order, because of merge operations. Blocks are stored by hash value, rather than processing time. Replications result in random read I/O along all components for restoring block after block in a sequential order. *PBTs* do not have to process every specific block. When a node comes up, it requests for the actual chain state by broadcasting its latest block hash. A further node can search for the partition, that contains this block. The node submit all data in this and successor partitions to the upcoming node in one step. Synchronization units change from blocks up to partitions. If blocks are already indexed in the upcoming node, the blocks in the synchronized partition replaces this hashes. Data remains consistent and can be reorganized and garbage collected in a further merge step. The upcoming node is faster on the current state, compared to a sequential block after block reconstruction, even if data placement is not optimal. Moreover, while synchronizing partitions of predecessor blocks, current workload can be processed in a separate partition.

5 Blockchain Schema and Indexing

Blockchains consist not only of *transactions*, but also of *blocks*, *accounts*, *smart contracts*, and possibly of tokenized off-chain data, which are related to each other (e.g. the user of account A sends B in transactions T credits for a service defined in contract C). This linked data implies a schema and further indexing for processing and analysis. We list some use cases:

Schema Data. Miners / Validators require to validate *transactions*, before adding them to a block, otherwise blocks may get rejected by the network and processing effort is wasted. Coins / gas or tokens in *smart contracts* of *accounts* need to be verified to accomplish *transactions*. Such transactions in a K/V-Store require to meet serializable ACID properties. Current balances of *accounts* need to be known. Multi-Version Concurrency Control (MVCC) can improve performance in this constellation. LSM-Trees are not able to handle multi version data natively. *MV-PBTs* are able to maintain multi version data and perform visibility checks, if one timestamp is added to each data record for validation or invalidation time (based on record type) and new tuple versions are added in higher numbered partitions[17, 23]. Therefore, several mutable partitions are maintained in main memory to absorb version ordering. Accounts balance can be determined without blocking concurrent transactions in serializable snapshot isolation.

Schema Indexing. Data in blockchains is described by a schema. *Transactions* send credits from one *account A* to another *account B*. But what are the transactions of account *A*? LSM-Trees and *PBTs* are able to index multi column records and answer partial key look-ups, but their column ordering has to match the search predicates. Additional indexes in the K/V-Store can solve this problem and improve search performance[24]. With *PBT* several indexes can be created, based on query requirements. *PBTs* are expected to have a similar update performance like LSM-Trees, because modifications are performed in a mutable in-memory partition, with benefits due to less required index nodes and flexible reorganization. Furthermore, *PBTs* are able to operate as version store. With bloom filter-protected immutable partitions, near B^+-Tree-like query performance with LSM-Tree-like update throughput enables a fully indexed data schema in traditional concurrency control protocols as well as in MVCC. Complexity in *smart contracts* may increase and is integrated in enterprise architectures, therefore fast analytics are required.

6 Enterprise Workload

In a business landscape, public blockchains offer new markets to customers. *Smart contracts* enable deployment of business logics on the blockchain, but *transactions* and further data have to be integrated in evolved enterprise systems, such as enterprise resource planning (ERP), customer relationship management (CRM), supply chain management (SCM), product life cycle management (PLM) and business intelligence reporting (BI). These enterprise systems may are connected via private blockchains in a "trustworthy" environment[25], such as Hyperledger[26], which also applies LevelDB as K/V-Store. Data gathering from markets of blockchain can be solved by *ETL* processes (Extraction, Transformation, Load), but it is desirable to integrate and subsume blockchain technologies into larger and evolved systems.

In both scenarios, a messaging service based integration (as depicted in Fig.

1) is not suitable for large data sets, due to high latencies in complex look-ups and type conversions. The enterprise system sends a query command to the blockchain network node client. The client can check indexes in the K/V-Store, if the query search predicates match. If the queried data is located in the K/V-Store, it can be returned in a string to the enterprise system, otherwise the data location is returned to the client. Afterwards, the client looks-up the physical block outside of the K/V-Store. Required data in this block is read and converted to a string, which is returned to the enterprise system. This process is repeated for every record tuple.

Storing blocks logically in the K/V-Store (as depicted in Fig. 2) enables fast look-ups. The K/V-Store can be directly queried by the enterprise system. Values can be identified by keys in data and indexing structures. Result sets can be calculated and returned to the enterprise system. For this workloads, data and indexing structures have to leverage characteristics of blockchain data and modern hardware technologies.

ETL processes would not stress blockchains K/V-Stores on queries from enterprise systems. Data can be extracted once to an enterprise system, while post-processing can be handled on its database. In this case, extractions scan *PBTs* high numbered partitions, containing new and not yet extracted data. Already extracted data located in low numbered partitions has not to be processed. LSM-Trees may intermingled already loaded and new data, due to merge operations, and all components have to be processed. *ETL* processes result in a delay on load phases and data has to be loaded to every enterprise system. Management come to decisions based on may obsolete data. Transaction processing and resource planning in the enterprise systems (e.g. sales processes of real goods or not in the blockchain integrated services by *smart contracts*) stagnate between load phases. Fundamental blockchain characteristics, such as *unspent transactions* (outlined in Section 4) are lost in enterprise systems. Furthermore, K/V-Stores on blockchain nodes are principally able to handle workloads from enterprise systems, especially if schema and indexing is provided as outlined in Section 5.

Enterprise systems may integrate blockchains by connecting to their nodes K/V-Stores as depicted in Fig. 2. *Transactions* immediately become visible to enterprise systems, without any delay for processing transactions, resource planning and data gathering for management decisions.

Starting a business process from a blockchain *transaction* from *smart contracts* may include several enterprise systems, because of its complexity and evolved system landscape. Pricing of services or material goods may differ for customers or requests, based on turnover or market demarcation. As a pre-process, a *smart contract* can request the special pricing from the CRM or the actual pricing of a service from the suppliers SCM to negotiate the price in *unspent* or *off-chain transactions.* Furthermore, required resources can be planned in an ERP before the *transaction* was appended in a consensual block in the blockchain. *PBTs* are more flexible than LSM-Trees in data placement. Due to uniformly distributed hash-keys in the blockchain, LSM-Trees are not able to plan merges efficiently. Components with data of *unspent transactions* are evicted to secondary storage

or merged with further components. *PBTs* can handle business complexity by partitions located in main memory and accelerate processes.

While OLTP workloads are processed on the system, on-line analytical processing (OLAP) is performed by business management. Queries are often performed on equal search predicates and few variations in data ranges. Cached Partitions in *PBT* can be built up on frequently queried data and stay in main memory for fast look-ups and scans. Cached Partitions are created out of a result set of an analytical scan. In fact, records buffered in Cached Partitions replace records in lower numbered partitions, queries and scans require only to process this and in-memory partitions, which absorb the current workload. If Cached Partitions exceed the main memory, they can be evicted to a secondary storage and FPGAs might be able to efficiently calculate result sets with NDP techniques. Partitions keep track of record counts, so intermediate counts can be calculated without processing every record, if the search key range is in the query predicates range. For OLAP, tracking intermediate results can be applied to sums or averages of payloads with only a little effort. Solely invalidated records in higher numbered partitions have to be excluded from intermediate results. LSM-Trees require to process every matching record in every component.

Mixed workloads require flexible data structures, which are able to leverage characteristics of modern hardware technologies. OLTP workloads from blockchain network transaction processing require high update rates and acceptable look-up performance. OLAP workloads need a data organization for fast look-ups and scans. An optimal data structure brings data placement in complex memory hierarchies in balance for concurrent types of workloads. Straight transaction processing of blockchain data is well performed in traditional concurrency control protocols, because *transactions* of consensual blocks never change. Schema and indexing structures in a K/V-Store can benefit from MVCC in serializable snapshot isolation levels, because transactions modify data many times while querying data. Providing a snapshot for OLAP enables high concurrent update rates from OLTP[27]. Matured DBMS for mixed workloads, such as SAP HANA[28], implement MVCC protocols. LSM-Trees are neither designed for mixed workloads nor support MVCC natively (outlined in Sections 3.1, 5). *PBTs* allow high update rates for transaction processing, and flexible data placement and partition management for current mixed workloads. Cached Partitions are created as result of query requirements. Simple but powerful filters are calculated as part of the eviction process. Both speed up further look-ups and scans. Immutable partitions on secondary storage enable NDP with low complexity. By applying a transaction timestamp to records, *MV-PBT* can serve as version store in MVCC. *PBTs* data structure leverage requirements from blockchain and enterprise workloads and modern hardware technologies.

7 Conclusion

Blockchains need to be integrated in enterprise systems. K/V-Stores of maintained virtual nodes in the blockchain network can principally serve as a database

and can be subsumed by enterprise systems. Therefore, powerful data structures are required to handle the complexity of the blockchain characteristics, must fulfil the requirements of enterprise systems, and have to exploit the characteristics of modern hardware technologies. Partitioned B-Trees (PBTs) are able to leverage complex memory hierarchies, query and scan requirements from enterprise systems as well as characteristics of blockchain data, due to their partition management within a tree structure. Partitions can appear and vanish as simple as inserting or deleting records. Moreover, they absorb high update rates, bulk loads, additional complexity of non-consensual blocks, and unspent transactions in main memory. Evicted partitions retain immutable data efficiently on different storage media in complex memory hierarchies and enable Near-data Processing (NDP) with intelligent storage. Synchronizations with further nodes can be performed on partition level, not only in slow sequential block appends. Filter techniques enable a near B^+-Tree-like query performance. This efficient design enables further schema and indexing for query requirements of enterprise systems. State of the art data structures, like LSM-Trees, do not fully meet these requirements.

Acknowledgement

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

References

[1] Bitcoin.com | Bitcoin News and Technology Source, 2018. Accessed: 2018-03-01.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[3] Ethereum Project, 2018. Accessed: 2018-03-01.

[4] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07), 2017. Accessed: 2018-02-03.

[5] bitcoin/bitcoin: Bitcoin Core integration/staging tree, 2018. Accessed: 2018-03-01.

[6] ethereum/go-ethereum: Official Go implementation of the Ethereum protocol, 2018. Accessed: 2018-03-01.

[7] paritytech/parity: Fast, light, robust Ethereum implementation., 2018. Accessed: 2018-03-01.

[8] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoff s in blockchain protocols, Oct 2015. Accessed: 2018-02-17.

[9] Mattias Scherer. *Performance and Scalability of Blockchain Networks and Smart Contracts*. PhD thesis, 2017.

[10] Raiden Network - Fast, cheap, scalable token transfers for Ethereum, 2018. Accessed: 2018-03-01.

[11] Jacob Eberhardt and Stefan Tai. On or off the blockchain? insights on off-chaining computation and data. In *European Conference on Service-Oriented and Cloud Computing*, pages 3–15. Springer, 2017.

[12] Zhijie Ren and Zekeriya Erkin. A scale-out blockchain for value transfer with spontaneous sharding. arXiv:1801.02531, 2018. Accessed: 2018-01-17.

[13] Hyperledger Project, 2018. Accessed: 2018-03-01.

[14] Elyes Ben Hamida, Kei Leo Brousmiche, Hugo Levard, and Eric Thea. Blockchain for Enterprise: Overview, Opportunities and Challenges. In *The Thirteenth International Conference on Wireless and Mobile Communications (ICWMC 2017)*, Nice, France, July 2017.

[15] google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values., 2018. Accessed: 2018-03-01.

[16] facebook/rocksdb: A library that provides an embeddable, persistent key-value store for fast storage., 2018. Accessed: 2018-03-01.

[17] Goetz Graefe. Sorting and indexing with partitioned b-trees. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.

[18] Goetz Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[19] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981.

[20] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[21] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In K. SelÃ§uk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *SIGMOD Conference*, pages 217–228. ACM, 2012.

[22] Christian Riegger, Tobias Vinçon, and Ilia Petrov. Write-optimized indexing with partitioned b-trees. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '17, pages 296–300, New York, NY, USA, 2017. ACM.

[23] Christian Riegger, Tobias Vinçon, and Ilia Petrov. Multi-version indexing and modern hardware technologies: A survey of present indexing approaches. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '17, pages 266–275, New York, NY, USA, 2017. ACM.

[24] Allan Third and John Domingue. Linked data indexing of distributed ledgers. In *Proceedings of the 26th International Conference on World Wide Web Companion*, WWW '17 Companion, pages 1431–1436, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[25] Vincenzo Morabito. *Blockchain and Enterprise Systems*, pages 125–142. Springer International Publishing, Cham, 2017.

[26] Christian Cachin. Architecture of the hyperledger blockchain fabric, 2016. Accessed: 2018-02-10.

[27] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. Efficient transaction processing for hyrise in mixed workload environments. In *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014.*, pages 16–29, 2014.

[28] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. High-performance transaction processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.