Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database

Senthil Nathan[1], Chander Govindarajan[1], Adarsh Saraf[1],
Manish Sethi[2], and Praveen Jayachandran[1]

[1]IBM Research - India, [2]IBM Industry Platforms, USA
[1](snatara7, chandergovind, adasaraf, praveen.j)@in.ibm.com, [2]manish.sethi1@ibm.com

ABSTRACT

In this paper, we design and implement the first-ever decentralized replicated relational database with blockchain properties that we term *blockchain relational database*. We highlight several similarities between features provided by blockchain platforms and a replicated relational database, although they are conceptually different, primarily in their trust model. Motivated by this, we leverage the rich features, decades of research and optimization, and available tooling in relational databases to build a blockchain relational database. We consider a permissioned blockchain model of known, but mutually distrustful organizations each operating their own database instance that are replicas of one another. The replicas execute transactions independently and engage in decentralized consensus to determine the commit order for transactions. We design two approaches, the first where the commit order for transactions is agreed upon prior to executing them, and the second where transactions are executed without prior knowledge of the commit order while the ordering happens in parallel. We leverage serializable snapshot isolation (SSI) to guarantee that the replicas across nodes remain consistent and respect the ordering determined by consensus, and devise a new variant of SSI based on block height for the latter approach. We implement our system on PostgreSQL and present detailed performance experiments analyzing both approaches.

PVLDB Reference Format:
Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, Praveen Jayachandran. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *PVLDB*, 12(11): 1539-1552, 2019.
DOI: https://doi.org/10.14778/3342263.3342632

1. INTRODUCTION

Blockchain has gained immense popularity over recent years, with its application being actively explored in several industries. At its core, it is an immutable append-only log of cryptographically signed transactions, that is replicated and managed in a decentralized fashion through distributed consensus among a set of untrusted parties. Opinion on the technology has varied widely from being hailed as the biggest innovation since the Internet to being considered as another database in disguise. For the first time, we undertake the challenge of explaining blockchain technology from the perspective of concepts well known in databases, and highlight the similarities and differences between the two. Existing blockchain platforms [18, 52, 40, 15], in an attempt to build something radically new and transformative, rebuild a lot of features provided by databases, and treat it as just a store of information. We take a contrarian view in this paper. By leveraging the rich features and transactional processing capabilities already built into relational databases over decades of research and development, we demonstrate that we can build a *blockchain relational database* with all features provided by existing blockchain platforms and with better support for complex data types, constraints, triggers, complex queries, and provenance queries. Furthermore, our approach makes building blockchain applications as easy as building database applications; developers can leverage the rich tooling available for backup, analytics, and integration with existing enterprise systems. Applications that have a strong compliance and audit requirements and need for rich analytical queries such as in financial services that are already built atop relational databases, or ones that are reliant on rich provenance information such as in supply chains are likely to most benefit from the blockchain relational database proposed in this paper.

With a focus on enterprise blockchain applications, we target a permissioned setup of known but untrusted organizations each operating their own independent database nodes but connected together to form a blockchain network. These database nodes need to be replicas of each other but distrust one another. Replication of transaction logs and state among multiple database nodes is possible using master-slave [17, 26, 46, 51] and master-master [26, 49, 51] protocols. In these protocols, the master nodes need to be trusted; for any transaction, a single node executes it and propagates the final updates to other nodes either synchronously [39, 21, 45] or asynchronously [29, 43, 47] , which would not work in a blockchain network. Hence, transactions need to be independently executed on all nodes. Further, such an execution and subsequent commits across multiple untrusted nodes must result in the same serializable order to ensure consistency. We need a novel replication protocol with a notion of decentralized trust. Although fault-tolerant synchronous commit protocols [30, 50] exist, they cannot be used as they either require a centralized trusted controller

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342632

or multple rounds of communications per transaction which would be costly in a network with globally distributed nodes.

In this paper, we address this key challenge of ensuring that all the untrusted database nodes execute transactions independently and commit them in the same serializable order asynchronously. Towards achieving this, we make two key design choices. First, we modify the database to separate the concerns of concurrent transaction execution and decentralized ordering of blocks of transactions. We leverage ordering through consensus only to order blocks of transactions, and not for the serializable ordering of transactions within a single block. Second, independently at each node, we leverage serializable snapshot isolation (SSI) [24, 44] to execute transactions concurrently and then serially validate & commit each transaction in a block to obtain a serializable order that will be identical across all untrusted nodes.

We make the following contributions in this paper: (1) We devise two novel approaches of building a blockchain platform starting from a database, (a) where block ordering is performed before transaction execution, (b) where transaction execution happens parallelly without prior knowledge of block ordering. (2) We devise a new variant of serializable snapshot isolation [44] based on block height to permit transaction execution to happen parallelly with ordering. (3) We implement the system in PostgreSQL using only 4000 lines of C code and present the modified architecture.

The rest of this paper is organized as follows. We highlight the properties required in a blockchain platform, motivate how several of these are either partially or fully available in relational databases, and identify shortcomings that need to be addressed in §2. We discuss our design in §3 and describe two approaches to transaction processing leveraging serializable snapshot isolation. We provide an account of our prototype implementation on PostgreSQL in §4. In §5, we present a detailed performance study. We discuss related work in §6 and conclude in §7.

2. MOTIVATION

Table 1 presents a comparison of properties required by a blockchain platform and features available in relational databases. We argue that there is much to be gained by leveraging and enhancing the features of relational databases to build a permissioned blockchain platform, rather than building one from scratch. This approach helps us leverage the decades of research in relational databases, stable enterprise-ready code with rich features, the large community, libraries, tools and integrations with enterprise applications. Next, we compare blockchains and databases across eight features and the enhancements needed.

(1) Smart contract & transaction: Many blockchain platforms such as Hyperledger Fabric [18], Ethereum [52] and Corda [31] support smart contracts which are functions (written in platform specific languages like Go) that manage state on the blockchain ledger. Transactions are invocations of these functions. This is similar to stored procedures in relational databases written in PL/SQL, a feature complete procedural language that is powerful enough to match functionality provided by contracts in other platforms. However in a blockchain platform, independent execution of the contracts by different peers needs to be deterministic which may not be true for PL/SQL if utility functions such as `random` and `timestamp` are used. Hence, we need to constrain PL/SQL to ensure determinism.

(2) User authenticity & non-repudiability: Permissioned blockchain systems employ public key infrastructure for user management and ensuring authenticity. Users belong to organizations that are permissioned to participate in the network. Transactions are digitally signed by the invoker and recorded on the blockchain, making them non-repudiable. Relational databases have sophisticated user management capabilities including support for users, roles, groups, and various user authentication options. However, submitted and recorded transactions are not signed by the invoker making them repudiable.

(3) Confidentiality & access control: Some permissioned blockchains support the notion of confidentiality; smart contracts, transactions and data are only accessible by authorized users. In addition, access to invoke functions and to modify data is restricted to specific users. Relational databases have comprehensive support for access control on tables, rows, columns and PL/SQL procedures. Some relational databases even provide features like content-based access control, encryption and isolation of data. Advanced confidentiality features where only a subset of the nodes have access to a particular data element or smart contract can be implemented, although we don't consider that in this paper.

(4) Immutability of transactions and ledger state: The blockchain ledger is an append-only log of blocks containing sets of transactions that are chained together by adding the hash of each block to it's succeeding block. This makes it difficult for an attacker to tamper with a block and avoid detection. The ledger state can only be modified by invoking smart contract functions recorded as transactions on the blockchain and are immutable otherwise. Although a database logs all submitted transactions, executed queries, and user information, it isn't possible to detect changes to the log as there are no digital signatures.

(5) Consistent replicated ledger across distrustful nodes: All non-faulty peers in a blockchain must maintain a consistent replica of all transactions and the ledger state. This is ensured by all peers committing transactions in the same sequential order as agreed through consensus. Note that consensus is performed on blocks of transactions rather than on individual transactions to be more performant. Databases support replication of transaction logs and state among nodes using master-slave [17, 26, 46, 51] and master-master [26, 49, 51] protocols. As discussed earlier, in these protocols, master nodes are trusted. Hence, we need a replication protocol with a notion of decentralized trust, while ensuring that the replicas remain consistent.

(6) Serializability isolation, ACID: Blockchain transactions require serializable isolation, in which dirty read, non-repeatable read, phantom reads, and serialization anomalies are not possible. Transactions, when executed in parallel, must follow the same serializable order across all nodes. Further, transactions should be ACID [23] compliant. Serializable isolation can be achieved in databases by employing: (i) strict 2-phase locking [39], (ii) optimistic concurrency control [33], or (iii) SSI [24, 44]. While these techniques can be leveraged to a large extent, they need to be enhanced to follow the block order as determined through consensus.

(7) Asynchronous transactions: As transaction processing and consensus may involve a non-trivial delay, clients submit transactions and leverage notification mechanisms to later learn the transaction status. Databases provide notification channels and triggers that can serve this purpose.

(8) Provenance: The append-only log of transactions in blockchains can be harnessed as a provenance store for sev-

Table 1: Similarities between blockchain properties and relational database features.

Blockchain Properties	Relational Database Features	Enhancement Needed
Smart contract	PL/SQL procedure	Deterministic execution
Authenticity, non-repudiability	User management with groups and roles	Crypto-based transaction authentication
Access control	Role & content-based ACL policies	None
Immutable transaction logs	Transaction logs	Digitally signed transactions
Consistent replicated ledger between untrusted nodes	Master-slave & master-master replication with trust on transaction ordering and update logs	Decentralized trust and transaction ordering determined by consensus
Serializable isolation level	Strict 2-phase locking, optimistic concurrency control, serializable snapshot isolation	Order must respect block ordering obtained through consensus
Async transaction & notification	LISTEN & NOTIFY commands	None
Provenance queries	Maintains all versions of a row	Enable query on historical records

eral use cases including supply chain tracking and financial compliance. However, most blockchain platforms today do not support complex queries on historic data or are not optimized for provenance queries. In certain relational databases which support Multi-Version Concurrency Control [20] such as snapshot isolation [19], all versions of a row are maintained though SQL queries cannot access old rows (which are purged periodically). For provenance queries, we need to disable purging and enable queries on the old rows.

3. DESIGN

We identify two approaches to achieve our goal of building a consistent replicated ledger across untrusted nodes starting from a relational database. The first approach, *order-then-execute*, orders all the transactions through a consensus service and then nodes execute them concurrently, whereas in the second approach, *execute-order-in-parallel*, transaction execution happens on nodes without prior knowledge of ordering while block ordering happens parallelly through a consensus service. Intuitively, while the first approach is simpler and requires fewer modifications to the relational database, the second approach has the potential to achieve better performance. We design and implement both approaches to study their trade-offs.

We describe the key components of our system in §3.1. We provide background on Serializable Snapshot Isolation in §3.2 and show later in §3.3 that SSI, if directly applied, does not guarantee serializability and consistency across nodes for the *execute-order-in-parallel* approach. Figure 1 juxtaposes the two proposed approaches and we describe them in detail in §3.3, including a novel SSI based on block height technique. We discuss security properties of the proposed approaches in §3.4. We describe a mechanism for peer node recovery in §3.5 and then discuss how a permissioned blockchain network can be bootstrapped in §3.6.

3.1 Key Components

We consider a permissioned network of organizations that are known to one another but may be mutually distrustful. The network is private to the participating organizations and a new organization must be permissioned to join the network. Each organization may include clients, database peer nodes and ordering service nodes, which we describe below, that together form the decentralized network.

Client: Each organization has an administrator responsible for onboarding client users onto the network. The administrator and each client have a digital certificate registered with all the database peers in the system, which they use to digitally sign and submit transactions on the network (we describe the network setup process in §3.6). This helps support authenticity, non-repudiability and access control properties. Clients may also listen on a notification channel to receive transaction status.

Database Peer Node: An organization may run one or more database nodes in the network. All communication to send and receive transactions and blocks happens via a secure communication protocol such as TLS. Each node also has a cryptographic identity (i.e., public key) and all communications are signed and authenticated cryptographically. Each database node maintains its own replica of the ledger as database files, independently executes smart contracts as stored procedures, and validates and commits blocks of transactions formed by the ordering service.

Ordering Service: Consensus is required to ensure that the untrusted database nodes agree on an ordering of blocks of transactions. To leverage the rich literature on consensus algorithms with different trust models, such as crash fault tolerant (e.g., Raft [41], Paxos [35], Zab [32]) and byzantine fault tolerant (e.g., PBFT [25], XFT [38], BFT-SMaRt [22]) consensus we make the ordering service in our system pluggable and agnostic to the database implementation. The ordering service consists of consensus or orderer nodes, each owned by a different organization. Each orderer node, similar to database nodes, have their own digital certificate or identity. The output of consensus yields a block of transactions, which is then atomically broadcast to all the database nodes. A block consists of (a) a sequence number, (b) a set of transactions, (c) metadata associated with the consensus protocol, (d) hash of the previous block, (e) hash of the current block, i.e., hash (a, b, c, d); and (f) digital signature on the hash of the current block by the orderer node.

3.2 Serializable Snapshot Isolation (SSI)

Strict 2-phase locking (S2PL), optimistic concurrency control (OCC), and SSI are approaches to achieve serializability. As SSI offers greater concurrency than S2PL and OCC, we have chosen SSI in our design. SSI achieves serializability using a modified snapshot isolation (SI) technique.

Snapshot Isolation and Anomalies. In SI, each transaction reads from a consistent snapshot of the database comprising of the last committed values that existed at the time the transaction started. Although SI prevents dirty read, non-repeatable read, and phantom read, it cannot guarantee serializability due to SI anomalies [19] which violates consistency (i.e., C in ACID), specifically integrity constraints (refer to [19, 24, 44] for examples). Hence, Cahill et. al. [24] proposed SSI to detect and resolve anomalies automatically.

Background on SI Anomalies. To detect SI anomalies, Adya et. al. [16] proposed multi-version serialization history graph. This graph contains a node per transaction, and an edge from transaction T_1 to transaction T_2, if T_1 must have preceded T_2 in the apparent serial order of execution. Three types of dependencies can create these edges:

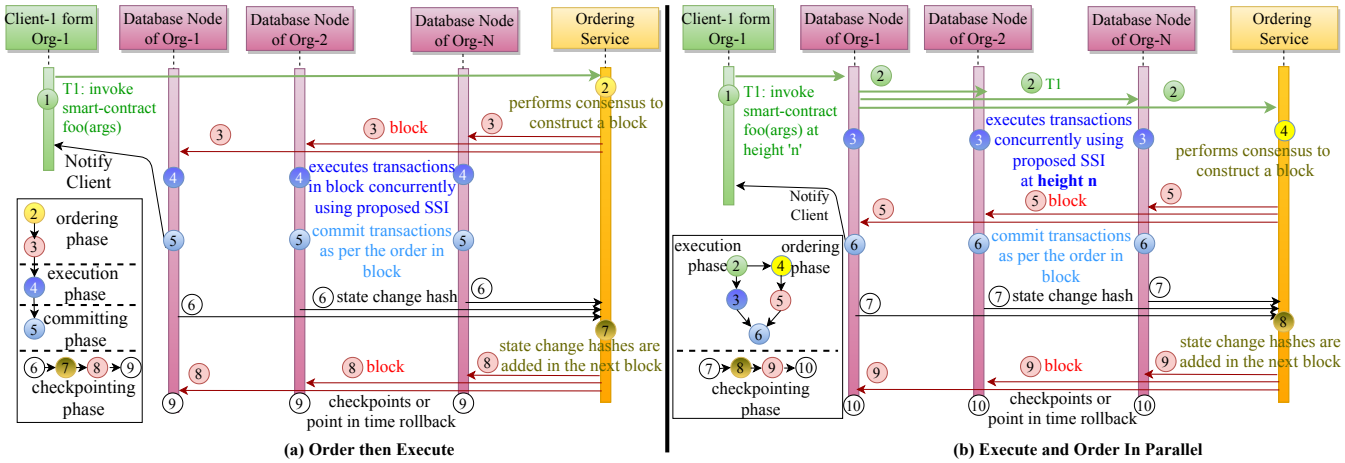

Figure 1: Proposed transaction flows—*order-then-execute* and *execute-and-order-in-parallel*

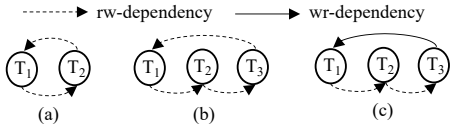

Figure 2: SI Anomalies between 2 and 3 transactions

- *rw-dependency*: if T_1 writes a version of some object, and T_2 reads the previous version of that object, then T_1 appears to have executed after T_2, because T_2 did not see the update. To represent this, an edge from T_2 to T_1 with a label rw needs to be added. As we will see, these rw-conflicts are central to SSI. Further, a dependency can also be caused by predicate reads.
- *wr-dependency*: if T_1 writes a version of an object, and T_2 reads that version, then T_1 appears to have executed before T_2 (an edge from T_1 to T_2 with a label wr).
- *ww-dependency*: if T_1 writes a version of some object, and T_2 replaces that version with the next version, then T_1 appears to have executed before T_2 (an edge from T_1 to T_2 with a label ww).

If a cycle is present in the graph, then the execution does not correspond to any serial order, i.e., a SI anomaly has caused a serializability violation. Otherwise, the serial order can be determined using a topological sort.

In SI, a *wr-dependency* or *ww-dependency* from T_1 to T_2 denotes that T_1 must have committed before T_2 began. In practice, writes acquire an exclusive lock to prevent *ww-dependency* between two concurrent transactions. Hence, a *wr-dependency* also cannot occur between two concurrent transactions. In contrast, only *rw-dependency* occurring between concurrent tra-nsactions is relevant when studying serialization anomalies.

Adya et. al. [16] observed that every cycle in a serialization graph contains at least two *rw-dependency* edges. Fekete et. al. [27] subsequently found that two such edges must be adjacent. Figure 2(a) shows the only possible cycle with two transactions, and Figure 2(b) and (c) show the two possible cycles with three transactions. If any of the transactions is aborted, a serializable order could be achieved.

SSI - Resolving Anomalies. SSI automatically detects and resolves anomalies. As tracking rw & wr dependencies, and subsequently detecting cycles is costly, SSI applies heuristics that are anomaly-safe, but could result in false positives. They are:

(1) abort immediately: Cahill et. al. [24] used two flags per transaction T: (a) *inConflict*—set when there is a *rw-dependency* from a concurrent transaction to T; (b) *outConflict*—set when there is a *rw-dependency* from T to a concurrent transaction. As soon as both of these flags are set for T, which could lead to an SI anomaly, T is aborted.

(2) abort during commit: Ports et. al. [44] maintained two lists per transaction T: (a) *inConflictList*—maintains a list of concurrent transactions from which there is a *rw-dependency* to T. (b) *outConflictList*—maintains a list of concurrent transactions to which T has a *rw-dependency*. The transactions present in inConflictList of T are called `nearConflicts`. The transactions present in the inConflictList of each nearConflict are called `farConflicts`. For e.g., in Figure 2(b), for T_2, T_1 is a nearConflict and T_3 is a farConflict. Recall that *rw-dependency* occurs only between concurrently executing transactions (such as in Figures 2(a) and (b)). For each pair of nearConflict and farConflict, if both transactions are not yet committed, then this heuristic aborts the nearConflict so that an immediate retry can succeed. In contrast, a *wr-dependency* only occurs between a committed and a just commenced transaction (Figure 2(c)). In this case, only if T_3 has committed first, its nearConflict T_2 is aborted. Otherwise, no transactions are aborted. In other words, while the heuristic does not track *wr-dependency*, it accounts for its possibility and aborts a transaction whose outConflict has committed.

3.3 Proposed Approaches

In this subsection, we describe the transaction flows for (1) Order then Execute; and (2) Execute and Order in Parallel as shown in Figure 1. In both flows, transactions are committed asynchronously across nodes.

3.3.1 *Order then Execute*

A transaction submitted by a client in the *order-then-execute* approach comprises of (a) a unique identifier, (b) the *username* of the client, (c) the PL/SQL procedure execution command with the name of the procedure and arguments, and (d) a digital signature on the `hash(a, b, c)` using the client's private key. The transaction flow consists of four pipelined phases: ordering, execution, committing, and checkpointing, wherein a submitted transaction needs to first be ordered, then executed, and finally committed before recording a checkpoint.

(1) Ordering phase: Clients submit transactions directly to any one of the ordering service nodes. On a periodic timeout, the ordering service nodes execute a consensus protocol to construct a block of transactions and delivers the block to the database nodes. In Figure 1(a), steps 2 and 3 denote this phase.

(2) Execution phase: On receiving a block of transactions, each database node verifies whether the received block is in sequence and sent by the ordering service. On successful verification, the node appends the block to a block store maintained in the local file system. In parallel, the node retrieves unprocessed blocks one at a time, in the order of their block sequence number, from the block store and performs the following four operations:

1. The database node assigns a thread per transaction (provided that the identifier present in a transaction's field is unique) to authenticate and execute it. In an append-only *ledger table*, it records each transaction in a block. This *ledger table* is used for recovery as explained in §3.5 and also for provenance queries as demonstrated in §4.2.
2. Each thread retrieves the public key associated with the *username* in the transaction, to verify the user's digital signature. On successful authentication, the *username* is set for the thread's session which is needed for access control. We leverage the database's access control capabilities without modification.
3. Each thread executes the PL/SQL procedure with the passed arguments as per the transaction. To ensure that on all nodes the transactions are executed on the same committed state, all valid transactions in a block are executed concurrently using the *abort during commit* SSI variant. This helps in ensuring that the set of transactions marked to be committed are the same and that they follow the same serializable order across all nodes.
4. Once a thread completes execution of a transaction, the transaction would be ready to either commit or abort (as per the procedure's execution logic), but waits without proceeding. This is because it could result in a different commit order in different nodes, if the execution completion order is different (which could result in different aborts by SSI in each node).

Only after committing a block of transactions, the execution phase will process the next block. In Figure 1(a), step 4 denotes the execution phase.

(3) Committing phase: To ensure that the commit order is the same on all nodes, the order in which the transactions get committed is the order in which the transactions appear in the block. Only when all valid transactions are executed and ready to be either committed or aborted, the node serially notifies one thread at a time to proceed further. Every transaction applies the *abort during commit* approach to determine whether to commit, and only then does the next transaction enter the commit stage. While it is possible to apply SSI for all transactions at once to obtain a serializable order, we defer such optimizations for simplicity. The node records the transaction status in the *ledger table* and emits an event via a notification channel to inform the client. In Figure 1(a), step 5 denotes the committing phase.

There is one additional challenge. In SSI, *ww-dependency* is handled using an exclusive lock. For example, if T_1 and T_2 are competing to write an object, only one transaction can acquire a lock (say T_1) and the other (say T_2) must wait for the lock to be released which can happen only after the commit/abort of T_1. However, in our system, to ensure consistency across all nodes, the committing phase cannot start unless all transactions complete execution. So, we cannot use locks for *ww-dependency*. However, as the ordering between transactions is determined by consensus and fixed across all nodes, we leverage this to permit both transactions to write to different copies of the object, but subsequently commit only the first as determined by the ordering. We show how such an implementation is possible in §4.

It is noteworthy that certain existing blockchain platforms such as Ethereum also follow an *order-then-execute* approach. However, they execute transactions sequentially in the order they appear in the block to ensure serializability and consistency across nodes, but this affects performance. In contrast, leveraging SSI to execute transactions concurrently and then sequentially issuing committing as per ordering from consensus, enables us to optimize performance.

(4) Checkpointing phase: Once all transactions in a block are processed, each node computes the hash of the write set, which is the union of all changes made to the database by the block, and submits it to the ordering service as a proof of execution and commit. When a node receives subsequent blocks, it would receive the hash of write set computed by other nodes. The hash computed by all non-faulty nodes would be the same and the node then proceeds to record a checkpoint (note that this is different from the Write-Ahead Logging checkpointing). It is not necessary to record a checkpoint every block. Instead, the hash of write sets can be computed for a preconfigured number of blocks and then sent to the ordering service. In Figure 1(a), steps 6, 7, and 8 denote the checkpointing phase.

3.3.2 Execute and Order in Parallel

A transaction submitted by a client in the *execute-order-in-parallel* approach comprises of (a) the *username* of the client, (b) the PL/SQL procedure execution command with the name of the procedure and arguments, (c) a block number, (d) a unique identifier which is computed as `hash(a, b, c)`, and (e) a digital signature on the `hash(a, b, c, d)` using the client's private key. The transaction flow consists of four phases: execution, ordering, committing, and checkpointing phase. A submitted transaction is executed by the database nodes and in parallel ordered by the ordering nodes and placed in a block. This is followed by the commit and checkpoint phases. We describe each phase in detail below.

(1) Execution Phase: Clients submit transactions directly to one of the database nodes. When a node receives a transaction, it assigns a thread to authenticate, forward and execute the transaction. On successful authentication (same as in the *order-then-execute* approach), the transaction is forwarded to other database nodes and the ordering service in the background. The four steps described in §3.3.1 for the execution phase applies for the *execute-order-in-parallel* approach as well, except for the default SSI and the logic to handle duplicate transaction identifiers. This is shown in steps 1-3 in Figure 1(b). Unlike the *order-then-execute* approach where execution of transactions happen *after* ordering and on the committed state from the previous block, in the *execute-order-in-parallel* approach we endeavor to execute and order in parallel. To ensure that the transaction is executing on the same committed data on all nodes, the submitted transaction includes the block height on which it should be executed (the client can obtain this from the peer it is connected with). We propose SSI based on block height as depicted in Figure 3, as the default SSI cannot ensure

Table 2: Abort rule for our proposed SSI variants when Transaction T is committing.

nearConflict	farConflict	to commit first among conflicts	Abort
same block			
✔	✔	nearConflict	farConflict
✔	✔	farConflict	nearConflict
✔	✗	nearConflict	farConflict
✗	✔	farConflict	nearConflict
✗	✗	-	
✗	none	-	

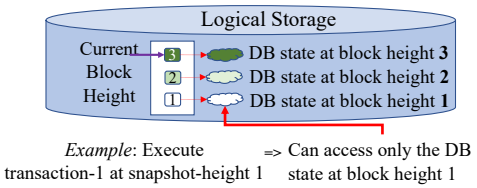

Figure 3: Logical Database Storage to Enable Snapshot Isolation Based on Block Height.

that transactions are executed on the same committed state on all nodes as each node can be at a different block height depending on its processing speed.

SSI Based on Block Height: In this isolation level, each row of a table contains a `creator` block number and `deleter` block number which denote the block that created and deleted this row, respectively (irrespective of the transaction within the block that created it). Note that, `creator` and `deleter` block number are part of the implementation for the *order-then-execute* approach as well, but are used only for provenance queries and are not required for SSI. During a transaction execution, based on the block number specified, the transaction can view the database state comprising of all commits up to this block height as shown in Figure 3. We refer to this specified block number as the transaction's `snapshot-height`. In other words, a transaction sees all rows with `creator` lesser than or equal to its `snapshot-height`. For all such rows, the `deleter` should be either empty or greater than the `snapshot-height`. For this SSI to work, we need to maintain all versions of a row and every update should be a flagging of the old row and an insertion of the new row. Every delete should be a marking of `deleter`. This facilitates the transaction to be executed on the same committed data on all nodes.

However, the current block (`current-block`) processed for commit by a node might be either lower or higher than the specified `snapshot-height` of a transaction. If the `current-block` is lower, the transaction would start executing once the node completes processing all blocks and transactions up to the specified `snapshot-height`. If the `current-block` is higher, the transaction would be executed immediately, but the serializability requirement could be violated because of a phantom or stale data read, which needs to be detected and handled. For example, assume that a transaction is updating all rows in a table which satisfy a given predicate mentioned in a `WHERE clause`. There is a possibility that a row that satisfies the predicate was committed by a block with a number which is greater than the specified `snapshot-height` and lesser than or equal to `current-block`. In this scenario, a phantom read [19] has occurred that violated the serializability. Similarly, a transaction can read a row from a snapshot as of `snapshot-height`, but if that row was either updated or deleted by a subsequent block it would result in a stale read for this transaction.

In order to ensure serializability, the proposed SSI approach detects such phantom reads and stale data reads to abort the corresponding transaction. To detect and abort such a transaction, the proposed approach applies row visibility logic on the committed state: (1) when a row with `creator` greater than the specified `snapshot-height` satisfies the given predicate, abort the transaction provided that the `deleter` is empty (handles phantom read); (2) when a row with `creator` lesser than `snapshot-height` satisfies the given predicate, abort the transaction provided that the `deleter` is non-empty and greater than `snapshot-height` (handled stale read). Further, concurrent transactions which are going to be committed during or after `current-block` can also create a phantom read or a stale read problem. Such cases are tracked by our proposed SSI as described in the committing phase.

(2) Ordering Phase: Database nodes submit transactions to the ordering service unlike the *order-then-execute* approach. Note that the transactions are being executed in the database nodes while they are being ordered by the ordering service. The rest of the logic is same as explained in §3.3.1 for the *order-then-execute* approach. In Figure 1(b), the steps 4 and 5 denote the ordering phase.

(3) Committing Phase: Like in the *order-then-execute* approach, an important pre-condition for entering commit phase is that all transactions in a block must have completed its execution and waiting to proceed with commit/abort. However, there are two key differences compared to the commit phase of *order-then-execute*. First, after receiving a block, if all transactions are not running (this occurs if the node has not received communication of a transaction from another peer due to maliciousness or network delay), the committer starts executing all missing transactions and waits for their completion before proceeding to the committing phase. Second, unlike the previous approach, it is possible for concurrent transactions to be executing at different snapshot heights (as specified by the respective clients). Further, transactions that are concurrent on one node, may not be concurrent on another, but we need the set of transactions that are decided to be committed to be the same on all nodes. As a result, we don't support blind updates such as `UPDATE table SET column = value;` which might result in a lock for *ww-dependency* only on a subset of nodes. Note, ww-dependencies are handled in the same way as described in §3.3.1. Further, instead of employing the *abort during commit* variant of SSI, we propose a *block-aware abort during commit* variant of SSI as described next.

Proposed SSI variant—block-aware abort during commit. Table 2 presents the abort rules for the proposed SSI variant. In addition to `nearConflict` and `farConflict`, our variant considers two additional parameters: (1) whether the `nearConflict` and `farConflict` are in the same block; (2) if they are in the same block, which among them is earlier as per the ordering. When either or both the conflicts are in the same block as transaction T, it is straightforward to abort the one that comes later in the ordering, and is deterministic on all nodes.

The tricky case is when neither of the conflicts are in the same block. In this case, we abort the *nearConflict* transaction. Note, the nearConflict is not in the same block but executes concurrently with T—this means that the `snapshot-height` specified for both could be lesser than or equal to the current block height at the node. With no synchroniza-

tion on transaction executions between nodes, it is possible for an SI anomaly to occur only at a subset of nodes. To ensure consistency between nodes, we need to ensure that the same set of transactions are aborted on all nodes. Let us consider possible scenarios in other nodes (say T_2 is the nearConflict transaction): (1) If T_2 is concurrent with T and an anomaly structure is detected, then T_2 is aborted as per our heuristic; (2) If T commits before T_2 starts execution, then T_2 being a nearConflict for T read a stale value and would be aborted as discussed earlier; (3) If T_2 is concurrent with T, but T is committed first, then this is a case of a *rw-dependency* where the outConflict has already committed leading to an anomaly structure (similar to Figure 2(c)) and T_2 will be aborted in this case as well. Hence, we must abort the nearConflict irrespective of the presence of a farConflict, whenever the nearConflict is not in the same block.

(4) Checkpointing Phase is same as explained in §3.3.1.

The unique identifier used for a transaction must be the hash of (a) the *username* of the client, (b) the PL/SQL procedure execution command with the name of the procedure and arguments, and (c) a block number specified for the SSI by the client. The reason is that if two different transactions are submitted to two different nodes with the same unique identifier, whichever transaction executes first on a given node is the one that would succeed, whereas the other would fail due to the duplicate identifier. As this can result in an inconsistent state across nodes, the unique identifier is composed of the three fields in the transaction to ensure that no two different transactions have the same identifier.

3.4 Discussion on Security Properties

(1) Submission of invalid transactions. A malicious client can potentially submit a large number of invalid transactions (e.g., ones that will eventually fail due to stale reads, try to perform operations they do not have access for) in an attempt to launch a denial of service attack. This can be thwarted in one of two ways. First, similar to permissionless blockchain networks, a transaction fee could be levied for each transaction using a native currency (it is possible to implement a native currency in our system if desired). Second, by monitoring clients and their behavior on the network, it is possible to exclude them from participation. We leave such enhancements for future work as it does not affect the core design of our system.

(2) Obscuration of valid transactions. In *order-execute* approach, when a malicious orderer node receives a transaction from the user, it might not include the transaction in a block. Similarly, in *execute-order-in-parallel* approach, when a malicious database node receives a transaction from the user, it might not forward the transaction to other database and orderer nodes. In both scenarios, at the client side, the transaction request would timeout. The client can then submit the same transaction to some other orderer or database node depending on the approach. Note that even if the client side timeout was a false alarm (i.e., the transaction is forwarded, included in a block and executed), the resubmission of the same transaction does not affect the data consistency as all nodes check for the unique identifier included in the transaction before execution.

In *execute-order-in-parallel* approach, if the database node forwards the transaction to an orderer node but not to other database nodes, eventually the request would be added into a block and delivered to all database nodes. The default committing phase described in §3.3.2 would take care of executing missing transactions while processing the block.

(3) Withholding of a block or transaction commit. In both approaches, when a malicious database node receives a block, it might skip committing the block or a transaction in that block. In such a scenario, the hash computed during the checkpointing phase would differ from other nodes, and it would become evident during the checkpointing process that the malicious node did not commit the block correctly. As other organizations can detect such maliciousness, there is no incentive for the organization to engage in such an activity. Since database nodes validate and commit blocks independently of one another, a malicious database node cannot hamper the liveness of the network.

(4) Byzantine ordering nodes. A malicious ordering node could send an incorrect block to database nodes connected to it. If an organization's database node trusts the ordering node it operates, but the ordering node is malicious, then that database node would also become faulty. This scenario would then be similar to the previous scenario of the database node being malicious. If a database node does not trust the ordering node, it should obtain blocks from $2f+1$ ordering nodes (assuming the consensus algorithm tolerates f failures or malicious behavior).

(5) Tampering of queries. In both approaches, the user can submit a read query to a single database node to fetch the stored data. A malicious node can tamper with the query and return incorrect results to the user. The following are two ways to detect such malicious behavior. The user can submit the query to multiple database nodes and verify whether the results are the same. Otherwise, any of the existing query authentication [36, 37, 42, 53, 54] methods can be used to verify the integrity of query results. Further, when a stored data is tampered with, it would eventually be identified through the checkpointing phase.

(6) Tampering of blockchain. Each database node stores all blocks in a separate store called the *block store* as described in §3.3. If any malicious database node tampers its block store, it will not affect the other replicas maintained at other organizations' node. In order to tamper the block store and not be detected, the database node would need the private cryptographic key of the orderer node as well as the client who submitted the transaction to forge their signatures. Even if the malicious node achieves this, if the majority of the nodes are non-malicious, honest organizations could prove that the malicious organization has tampered with the block store by comparing the replicated chain when a conflict occurs. Only if 51% of the nodes are malicious, a blockchain can be successfully tampered.

3.5 Recovery After a Failure

A blockchain network is required to tolerate node failures and we need to ensure that a failed node recovers to a consistent state when it restarts. In both approaches, the block processing stage has the following two common operations per block: (1) atomically store all transactions information in the *ledger table* along with the block number; (2) atomically store all transactions' status (i.e., commit/abort) in the *ledger table*. Note, only after all transactions get written to write-ahead-logging and the default transaction logs, the step 2 gets executed. A node can fail during any of the above two stages. When the node restarts, it executes the following operations to recover the database:

1. Retrieves the last processed block number from the *ledger table* and checks whether all transactions have a status.

Note, as we store all transactions' status atomically, either all transactions must have a status or none.

2. If a status is found in the *ledger table* for transactions, it means that the block was committed successfully and no recovery is needed. Note, if a node fails after committing a transaction, during restart, the default recovery mechanism that uses write-ahead logging (WAL) would take care of disk consistency.
3. If a status is not found in the *ledger table* for transactions, a node might have (a) either failed after committing all those transactions but before updating the *ledger table* with the status, or (b) failed before committing all or some those transactions. If a transaction was successfully committed, the relational database would have recorded the status on the default transaction logs. The node first checks the status of each transaction by reading the default transaction logs. If a status is present for all transactions, the node would update the *ledger table* with the status (i.e., case (a)).
4. If a status is not present in the transaction log for all or some transactions (i.e., case (b)), the node needs to rollback all other committed transactions in the same block. Then the node can start re-executing all transactions as described in §3.3, commit them, and record the status in the *ledger table*. The rollback of committed transactions is required as we need to execute all transactions in a block parallelly using SSI at the same time to get a consistent result with other nodes as well.

The handling of blocks missed out during failure is as simple as fetching, processing and commiting them one by one. We observe that it is possible to simplify the recovery process by doing a single atomic commit of a a block of transactions but we have omitted describing this in the interest of space.

3.6 Network Bootstrapping

To bootstrap a permissioned network, first, we need to bring up the database and orderer nodes each with their respective admin users and secure communication via TLS between them. We need to then create users & roles for each organization, and deploy PL/SQL procedures as smart contracts.

Setting up database nodes and ordering service. At network startup, each organization shares domain names of database and orderer nodes, TLS certificates, and credentials of admin users (i.e., public keys) with all other organizations. Each organization then starts its database and orderer nodes with the above information.

Support for blockchain and non-blockchain schema. Both blockchain (replicated) and non-blockchain (private) schemas are supported, and it is possible to invoke read-only queries on a node that span both schemas.

Setting up of PL/SQL procedures. To facilitate deployment of smart contracts, each node exposes the following four system smart contracts created during setup. These contracts can only be invoked by organization admins, are considered blockchain transactions and follow the transaction flow described earlier. (1) `create_deployTx()` creates, replaces or drops a smart contract. An entry is added to the deployment table, but does not execute yet. Access control policies need to be embedded within a smart contract itself; (2) `submit_deployTx()` executes the SQL statement present in the deployment table after verifying that an admin from each organization has approved the deployment transaction. If not all organizations have approved, the invocation returns an error; (3) `approve_deployTx()` approves a deployment transaction by adding a provided digital signature of the organization's admin; (4) `reject_deployTx()` rejects a deployment transaction by adding a provided digital signature of the organization's admin and a reason.

4. IMPLEMENTATION

PostgreSQL [13] is the first open source database to implement the *abort during commit* SSI variant [44]. Further, it maintains all rows even after an update or delete and is highly modular and extensible. For these reasons, we chose to modify PostgreSQL to build a blockchain relational database, rather than implementing one from scratch. The modifications only amounted to adding around 4000 lines of C code to implement both approaches. We present relevant background on PostgreSQL in §4.1. Then, in §4.2 and §4.3, we describe the components we added and modified in PostgreSQL respectively. In §4.4, we present implementation details of a crash fault-tolerant ordering service, and describe the transaction flows in §4.5.

4.1 PostgreSQL Background

PostgreSQL supports three isolation levels—read committed, repeatable read (i.e., SI), and serializable (i.e., SI with detection and mitigation of anomalies). A snapshot comprises of a set of transaction IDs, which were committed as of the start of this transaction, whose effects are visible. Each row has two additional elements in the header, namely *xmin* and *xmax*, which are the IDs of the transactions that created and deleted the row, respectively. Note, every update to a row is a delete followed by an insert (both in the table and index). Deleted rows are flagged by setting xmax instead of being actually deleted. In other words, PostgreSQL maintains all versions of a row unlike other implementations such as Oracle that update rows in-place and keep a rollback log. This is ideal for our goal of building a blockchain that maintains all versions of data. A snapshot checks *xmin* and *xmax* to see which of these transactions' ID are included in the snapshot to determine row visibility.

Clients connect to the PostgreSQL server, `postgres`, using an application interface such as libpq [12]. A backend process [11] is then assigned to each client connection to execute queries. PostgreSQL supports background workers [10] (additional processes other than backend) to perform other activities such as logical replication and statistics collection. Note, the `postgres` server, backends, and background workers coordinate and share data through shared memory regions. Both the shared memory data structures and background workers are easily extensible.

4.2 New Components Introduced

Communication Middleware & Block Processor. We introduced two new background workers: (1) communication middleware to communicate with other nodes and orderer, to transfer and receive transactions/blocks. The received blocks are stored in an append-only file named *pg-Blockstore*. Further, the middleware is also responsible for starting a transaction using libpq in the *order-then-execute* flow, and for starting any missing transactions in the *execute-order-in-parallel* flow; (2) block processor to process a block. It executes the commit phase as described in §3.3.

Shared Memory Data Structures. We introduced the following four data structures in the shared memory (the last two are used only for the *execute-order-in-parallel* flow).

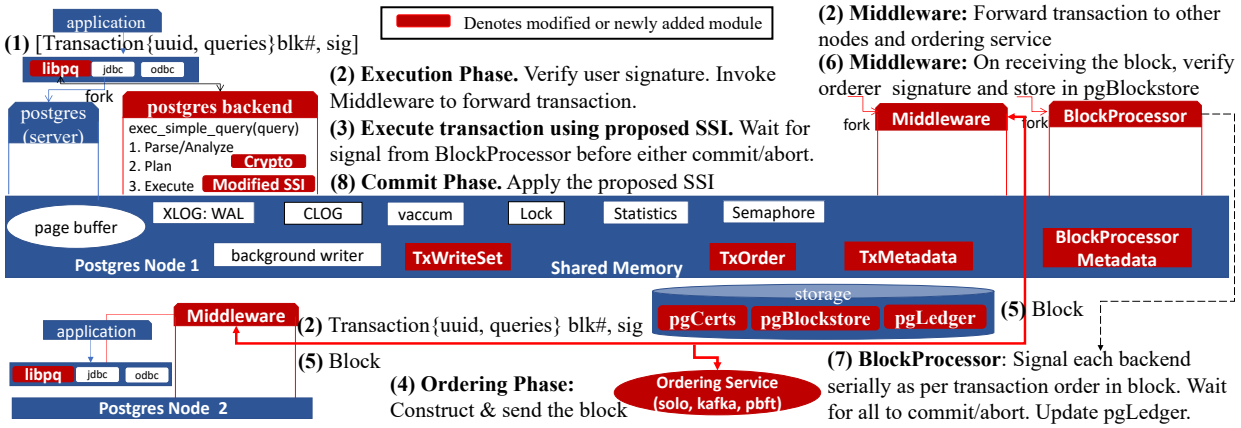

Figure 4: Transaction Flow in PostgreSQL for *execute-order-in-parallel*

Table 3: Example provenance queries to audit user transactions.

Audit Scenarios	Queries
Get all invoice details from table *invoices* which were updated by a supplier S between blocks *100* and *300*	`SELECT` invoices.* `FROM` invoices, pgLedger `WHERE` pgLedger.blockNumber `BETWEEN` 100 `AND` 300 `AND` pgLedger.user = S `AND` invoices.xmax = pgLedger.txid `AND` invoices.deleter `IS NULL`;
Get all historical details of an invoice from table `invoices` whose *invoiceID* (primary key) is k and was updated by either supplier S or manufacturer M in the last 24 hours	`SELECT` invoices.* `FROM` invoices, pgLedger `WHERE` invoiceID = k `AND` pgLedger.user `IN` (S, M) `AND` pgLedger.commitTime > `now()` - `interval` '24 hours' `AND` invoices.xmax = pgLedger.txid `AND` invoices.deleter `IS NULL`;

(1) `TxMetadata` enables communication and synchronization between block processor and backends executing the transaction (as needed in the commit phase). The block processor uses this data structure to check whether all transactions have completed its execution and to signal each backend to proceed further. Each entry consists of the global transaction identifier (as present in the transaction), transaction ID assigned locally by the node, process id of the backend, a semaphore to denote whether the transaction has completed its execution and waiting for a signal to proceed further, and a final status (i.e., commit/abort).

(2) `BlockProcessorMetadata` helps in signaling block processor from the backend once it commits/aborts a transaction. Additionally, this data structure holds the last committed block number, current block number (so that the backend can use it to set the `creator` & `deleter` block number in rows), and a semaphore to enable signaling from middleware to block processor.

(3) `TxWriteSet` holds the write-set of each transaction so that after SSI validation and before writing to WAL, backend can store the `creator` or `deleter` in each row. For an update, it stores both the old and new row pointer. For delete, it stores only the old row pointer. Further, `TxWriteSet` is used to compute the hash of the write-set for each block.

(4) `TxOrder` helps backend to apply our proposed *block-aware abort during commit* variant of SSI. It stores the global transaction identifier of each transaction in the block as per the commit order to aid in finding whether a `nearConflict` or `farConflict` is present in the same block.

Blockchain Related Catalog Tables. We introduced two system catalog tables, *pgLedger* and *pgCerts*. The former is the *ledger table* described in §3.5, and stores information about each transaction such as the global identifier, local transaction ID, the query, the submitting client and commit/abort status. It is used for recovery and for supporting provenance queries. The latter stores the cryptographic credentials of the blockchain users.

Provenance Query. We introduced a special type of read only query called *provenance*. This query type can see all committed rows present in tables irrespective of whether it is inactive (i.e., marked with *xmax*) or active. As it can access all historical content, it enables support for very complex analytical and audit queries with the help of *pgLedger*. Table 3 presents two examples of provenance queries.

4.3 Components Modified

Application Interface and Deterministic PL/SQL Procedures. We have enhanced the default application interface, i.e., libpq, with additional APIs to submit blockchain transactions & provenance queries, and fetch the latest block height at the node. To make the PL/SQL procedure deterministic, we have restricted the usage of date/time library, random functions from the mathematics library, sequence manipulation functions, and system information functions. Further, `SELECT` statements must specify `ORDER BY primary_key` when using `LIMIT` or `FETCH`. Additionally, it cannot use row headers such as *xmin*, *xmax* in `WHERE clause`.

SSI Based on Block Height. We have added two fields for each row: (i) `creator` block number, (ii) `deleter` block number. During commit, these two fields are filled for entries in the `TxWriteSet` depending on whether an entry is an update, insert, or delete. SI applies a row visibility logic using *xmin* and *xmax* to identify whether a row should be seen by a transaction (as described in §4.1). We enhance the row visibility logic to have additional conditions using the row's creator and deleter block number and the `snapshot-height` of the transaction (as described in §3.3.2). This is in addition to the default row visibility logic that helps avoid seeing rows updated/deleted by concurrent transactions. During predicate reads, the default visibility logic in PostgreSQL traverses rows as per the index entries that satisfies the given predicate or traverses the whole table when an index is missing. For our approach to work (mainly to avoid phantom or stale read described in §3.3.2), all read access

is enforced to happen via index entries only which satisfies a given predicate clause. Otherwise, there is a high possibility of transactions getting aborted due to false positive phantom or stale read. Hence, if an index is not available for the given predicate clause, nodes abort the transaction. Further, `SELECT * FROM table_name;` is not allowed from PL/SQL procedures as it always traverses the whole table. It is possible to implement our approach without the need for an index, but for simplicity we defer such optimizations. Note, any kind of individual `SELECT` statements without any writes on the blockchain schema will not be affected by SSI as the transaction would be marked as read-only and would be executed on one node only.

SSI Block-Aware Abort During Commit and ww-dependency. For the *execute-order-in-parallel* approach, we have modified the abort rule to follow our proposed rules in Table 2. The modified SSI utilizes the `TxOrder` data structure in the shared memory. For ww-conflicts, we allow writes to the same object by different transactions (as updates are anyway not in-place in PostgreSQL) by maintaining an array of *xmax* values comprising of transaction IDs of all competing transactions in the row being updated. During commit, for each old row entry in the `TxWriteSet`, the backend (corresponding to the transaction that is committing now) checks *xmax* values and marks all other transactions for abort as only one transaction can write to the row to avoid lost update. Finally, it retains only its own transaction ID in the *xmax* field.

4.4 Ordering Service

As described in §3.1, any consensus algorithm can be leveraged. We support two alternate ordering services: an Apache Kafka [1] and ZooKeeper [2] based crash fault tolerant ordering, and a BFT-SMaRt [22] based byzantine fault tolerant ordering. Clients/peers connect to independent orderer nodes to submit transactions and receive created blocks. When using Kafka-based consensus, orderer nodes publish all received transactions to a Kafka cluster which then delivers the transactions in a FIFO order. Likewise, when using BFT consensus, they submit the transactions to a BFT-SMaRt cluster which creates a total ordering of the received transactions. For creating a block of transactions, we use two parameters: *block size*, the maximum number of transactions in a block, and *block timeout*, the maximum time since the first transaction to appear in a block was received. Each orderer node publishes a *time-to-cut* message to the Kafka cluster or sends it to the BFT-SMaRt cluster (depending on which ordering is used) when its timer expires. The first time-to-cut message is considered to cut a block and all other duplicates are ignored. Once a block is cut, orderer nodes append their signatures to the block, persist the block in filesystem and then send it to connected peers.

4.5 Transaction Flow

Figure 4 depicts the new and modified components described in the previous sections and outlines the *execute-order-in-parallel* transaction flow. As the *order-then-execute* flow is simpler, we omit presenting the details for it in the interest of brevity. The application leverages the blockchain interfaces in libpq to fetch the latest block height and submit the transaction to the PostgreSQL backend. After verifying the client signature leveraging *pgCerts*, the transaction is forwarded to other PostgreSQL nodes and to the ordering service using the communication middleware (the client can also submit the transaction to all peers, rather than one peer forwarding to all other peers). The backend updates `TxMetadata`, executes the transaction leveraging the SSI variant based on *block-aware abort during commit*, and collects the write-set into `TxWriteSet`. It then sets *ready-to-proceed* status in `TxMetadata` and waits for a signal from the block processor.

On receiving a block from the ordering service, the middleware verifies the orderer signature and stores the block in *pgBlockstore*. The block processor retrieves each unprocessed block from the *pgBlockstore* one at a time and adds all transactions to *pgLedger*. It confirms from `TxMetadata` that all transactions have completed execution and then serially signals each backend as per the order in the block to proceed further. On receiving this signal, the backend validates the transaction based on the *block-aware abort during commit* logic as explained in §3.3.2, sets either *commit/abort* status in `TxMetadata`, and signals the block processor. Once all transactions have been processed, the block processor updates the *pgLedger* with the status for each transaction. We are yet to implement the checkpoint flow described in §3.

5. EVALUATION

In this section, we study the performance and cost of both our design approaches. We measure performance in terms of *throughput* and *latency*. *Throughput* is defined as the average number of unique transactions committed per second in the blockchain network, and *latency* is defined as the average time taken to commit a transaction, measured from the time a client submits it. A transaction is said to be committed in the blockchain network when majority of the nodes commit. The cost is measured in terms of resource utilization such as CPU, memory, disk and network bandwidth utilization. We study the effect on performance and cost by varying several system parameters, namely (a) block size, (b) transaction arrival rate, (c) smart contract query complexity, (d) deployment model (local/wide area network), and (e) the number of database nodes (network size).

We use two smart contracts (1) `simple contract`—inserts values into a table; (2) `complex contract`—performs complex joins between two tables to execute aggregate operations and writes the result to a third table. Note that while `simple contract` can be implemented in most blockchain platforms today, `complex contract` is impossible to implement efficiently. We consider two deployment models, the first where all organizations host their database nodes on a common public cloud data center (LAN), and second where nodes are hosted independently leading to a multi/hybrid-cloud setup (WAN). In our experiments, the WAN setup involved four data centers spread across four continents. These models help us study the communication overheads of the two approaches. In both deployments, database and orderer nodes were hosted on virtual machines each with 32 vCPUs of Intel Xeon E5-2683 v3 @ 2.00GHz and 64 GB of memory. In the multi-cloud deployment, the network bandwidth between nodes was between 50 and 60 Mbps, whereas, it is 5 Gbps in the single cloud deployment.

In addition to *throughput* and *latency*, we also measure the following seven micro metrics (all as averages) to gain a deeper understanding of system behavior: (1) block receive rate (`brr`)—the number of blocks received per second at the middleware from orderer; (2) block processing rate (`bpr`)—the number of blocks processed and committed per second at the block processor; (3) block processing time (`bpt`

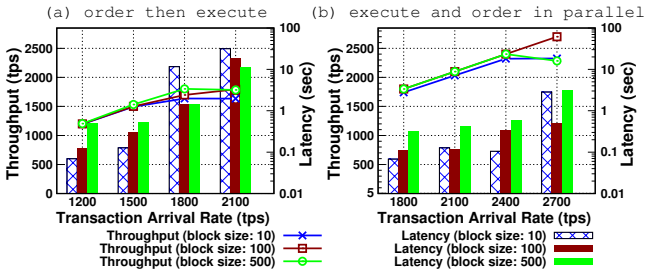

Figure 5: Performance with `simple contract`

Table 4: Order then execute: micro metrics for an arrival rate of 2100 tps.

bs	brr	bpr	bpt	bet	bct	tet	su
10	209.7	163.5	6	5	1	0.2	98.1%
100	20.9	17.9	55.4	47	8.3	0.2	99.1%
500	4.2	3.5	285.4	245	44.3	0.4	99.7%

in *ms*)—the time taken to process and commit a block; (4) block execution time (`bet` in *ms*)—the time taken to start all transactions in a block till they suspend for commit/abort; (5) transaction execution time (`tet` in *ms*)—the time taken by the backend to execute a transaction until it suspends for commit/abort; (6) block commit time (`bct` in *ms*)—the time taken to perform the serial commit of all transactions in a block and can be measured as `bpt` − `bet`; (7) missing transactions (`mt`)—the number of transactions missing per second while processing blocks at the block processor (relevant for *execute-order-in-parallel*). We define two additional terms: (1) the *peak throughput* as the maximum achieved throughput for a given smart contract type and block size across all possible arrival rates; (2) the *system utilization* (`su`) as the fraction of time the block processor is busy (`bpb`) expressed as a percentage. `bpb` can be estimated as the average number of blocks processed in 1s (`bpr`) multiplied by the time taken to process each block (`bpt`). When a peak throughput is achieved, our system utilization should be close to 100%.

Unless mentioned otherwise, our experiments use a single data center with three organizations, each running a database and a kafka-based orderer node. At the orderer, the block timeout was set to 1s and the block size was varied. We used *pgTune* [9] to configure PostgreSQL and *max_connection* (i.e., number of backends) was set to 2600. The arrival rate was load balanced among nodes.

5.1 Block Size and Arrival Rate

Order then Execute. Figure 5(a) plots the *throughput* and *latency* achieved in *order-then-execute* approach using the `simple contract`. Table 4 presents the micro metrics for an arrival rate of 2100 transactions per second (*tps*). With an increase in transaction arrival rate, the throughput increased linearly as expected till it flattened out at around 1800 *tps*, which was the peak throughput (system utilization close to 100% as shown in Table 4). When the arrival rate was close to or above the peak throughput, the latency increased significantly from an order of 100s of milliseconds to 10s of seconds. For an arrival rate lower than the peak throughput, with an increase in the block size, the latency increased. The reason is that with an increase in block size, it took longer for the orderer to fill a block with transactions leading to an increased block creation time, and hence, a larger waiting time for each transaction at the orderer. For an arrival rate greater than the peak throughput, with an increase in block size, the latency decreased. This is because there was no waiting for transactions to arrive and form a block, and more transactions were executed in parallel. This is also observable as the block processing time (`bpt`) of a block of size n was always lesser than the sum of `bpt` of m blocks each of size $\frac{n}{m}$. For the same reason, with an increase in the block size, the throughput also increased.

Execute and Order in Parallel. Figure 5(b) plots the same for *execute-and-order-in-parallel* approach. Table 5 presents the micro metrics for an arrival rate of 2400 *tps*. The maximum throughput achieved was 2700 *tps*, i.e., 1.5× higher than what was achieved with *order-then-execute*. Note that the system utilization (`su`) was only 90%. When the arrival rate was greater than 2700 tps, the throughput started to degrade and `su` never reached 100%. We believe the reason to be a large number of active backends that resulted in a lot of contention on the shared memory datastructures (further study is reserved for future work). Though the block processing time (`bpt`) and the block execution time (`bet`) were observed to be lesser with *execute-and-order-in-parallel* as compared to *order-then-execute*, the block commit time (`bct`) was observed to be higher. This could again be because of a large number of active backends.

Comparison With Ethereum's Order then Execute. Blockchain platforms such as Ethereum also adopt an *order-then-execute* approach, but execute transactions serially once the block is formed. To emulate this behavior, we made our block processor also execute and commit transactions one at a time. This resulted in a peak throughput of 800 *tps* (for a block size of 100, although the block size does not matter when we execute serially). This is only about 40% of the throughput achieved with our approach, which supports parallel execution of transactions leveraging SSI.

Table 5: Execute and order in parallel: micro metrics for an arrival rate of 2400 tps.

bs	brr	bpr	bpt	bet	bct	tet	mt	su
10	232.26	232.26	3.86	2.05	1.81	0.58	479	89%
100	24.00	24.00	35.26	18.57	16.69	3.08	519	84%
500	4.83	4.83	149.64	50.83	98.81	6.27	230	72%

5.2 Smart Contract Complexity

Order then Execute. Figure 6(a) plots the peak throughput and micro metrics observed with `complex contract` for *order-then-execute*. With an increase in block size, the throughput increased and reached a peak of 400 *tps*. This was less than 25% of what was observed with `simple contract`, primarily because of transaction execution time (`tet`) increasing by about 160× as compared to `simple contract`. The block processing time (`bpt`) and the block execution time (`bet`) also increased. The CPU and memory utilization for `simple contract` was 10% and 15 GB respectively, compared to 30% and 15GB for `complex contract`.

Execute and Order in Parallel. Figure 6(b) plots the same for *execute-order-in-parallel*. Both the block processing time (`bpt`) and the block execution time (`bet`) were lower than the one observed in *order-then-execute*. This is because, by the time the block reaches the node, all transactions were either executing or already completed execution. The peak throughput achieved was more than twice that of *order-then-execute*. Unlike *order-then-execute* approach, this approach permits concurrent execution of a larger number of transactions unrestricted by the block size (in *order-then-execute*, the maximum number of concurrently executing transactions is capped by the block size). This manifested as a significantly larger increase to the transaction

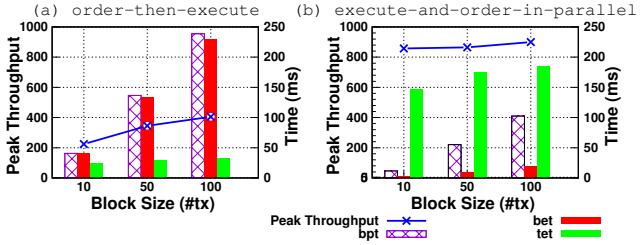

Figure 6: Performance with `complex contract`

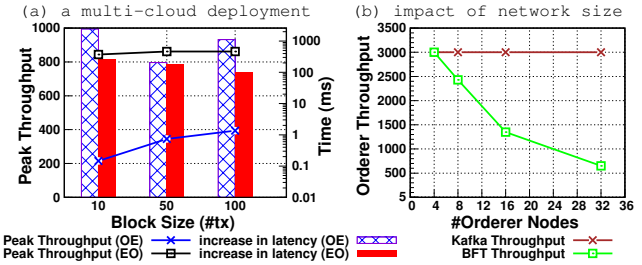

Figure 7: Performance (a) with `complex contract` in a multi-cloud deployment; (b) with different network sizes.

execution time `tet` compared to *order-then-execute*. For the same reason, the CPU utilization increased to 100%, i.e., 3.3× higher than what was observed for *order-then-execute*.

In another smart-contract, we used `SUM` function, `GROUP BY`, `ORDER BY`, and `LIMITS` instead of `JOIN` to benchmark the performance. For a block size of 100, the peak throughput achieved for *order-then-execute* and *execute-order-in-parallel* was 1.75× and 1.6× higher respectively than that of `JOIN`.

5.3 Deployment Model and Network Size

Deployment Model. Figure 7(a) plots the peak throughput achieved with both approaches in a multi-cloud deployment. As compared to a single cloud deployment (refer to Figure 6), only the latency increased on average by 100 *ms* due to the WAN network but the throughput reminded the same for most part except a 4% reduction in the peak throughput when the block size was 100. As the size of a block with 500 transactions was only about 100 KB (≈196 bytes per transaction), the lower network bandwidth did not have a significant impact on the performance.

Network Size Figure 7(b) plots the throughput achieved with kafka and bft based ordering service while varying the number of orderer nodes and fixing the transaction arrival rate to 3000 tps. With an increase in the number of orderer nodes, we observed no impact on the performance of kafka-based ordering service but the performance achieved with bft-based ordering service reduced from 3000 tps to 650 tps as expected due to the communication overhead. With an increase in the number of database nodes alone, the overall system throughput did not get affected but limited by the peak throughput of the ordering service.

6. RELATED WORK

Bitcoin [40] and **Ethereum** [52] adopt an order-execute model, where transactions are first ordered in blocks through consensus (such as proof-of-work) and each node serially validates and commits transactions locally. In our *order-execute* approach, we leverage SSI to execute transactions concurrently. Further, such platforms only support a simple key-value data model.

Hyperledger Fabric [18] adopts an *execute-then-order* approach, where transactions are first executed and endorsed by multiple nodes, then ordered by consensus, followed by serial validation and commit. In contrast, in our *execute-order-in-parallel* approach transaction execution and ordering happen parallelly. Fabric only supports goleveldb [5] and couchdb [3] as the underlying database, with support for composite keys and range queries. In contrast, we support the full spectrum of SQL queries in a smart contract with the exception of libraries that could introduce non-determinism and blind updates. Performance studies on Fabric [18, 48] have shown throughputs of 3000 tps with goleveldb for a `simple` smart contract and 700 tps with couchdb.

Hyperledger Composer [6] is a set of collaboration tools for building blockchain business networks utilizing Hyperledger Fabric. It allows use of basic SQL queries with limited syntax [7] which are internally converted to CouchDB JSON selector queries [4].

Corda [31] refers to their platform as a decentralized database. Transactions are executed by one node at a time (not parallelly executed by all nodes) and the results are shared with other nodes that have a need to know. There is no notion of a blockchain in Corda. However, optionally, a notary could order transactions and validate them for double spending. State objects can define a relational mapping, and an object serialization framework is used to store the states in an H2 embedded relational database. This permits querying the database using SQL and enables rich queries (such as joins) with an organization's private non-blockchain data. However, it does not enable rich query support within the smart contract itself.

Veritas [28] proposes shared verifiable tables using a set of Redis key-value stores each owned by an organization. Only the verifiability property, i.e., immutable logs, is supported. For ensuring consistency across replicas, it uses a centralized trusted timestamp server to order transactions. Further, a transaction is executed only on one of the nodes, and each node periodically ships logs of multiple read-write sets to other nodes via a Kafka-based broadcast service. Nodes vote on transactions to resolve any conflicts.

BigchainDB [15] employs Tendermint consensus [34, 14] over a set of independent MongoDB [8] instances, each owned by a different organization. It supports immutability and decentralization. While the overall goals of BigchainDB are similar to ours, there are fundamental architectural differences. It supports only user tokens/assets similar to Bitcoin and has no support for smart contracts. Transactions are serially executed post ordering, similar to Ethereum.

7. CONCLUSION

In this paper, we presented the design of a *blockchain relational database*, a decentralized database with replicas managed by different organizations that do not trust one another. The key challenge we addressed is in ensuring that all untrusted replicas commit transactions in the same serializable order that respects the block ordering determined by consensus. We proposed two design approaches that leveraged and modified SSI to achieve this, and devised a new variant of SSI based on block height. Leveraging features already available in databases enables us to better support complex data types, schemas, complex queries and provenance queries that are not provided by blockchain platforms today. We implemented the system on PostgreSQL and presented detailed performance results.

8. REFERENCES

[1] Apache kafka. https://kafka.apache.org.

[2] Apache zookeeper. http://zookeeper.apache.org.

[3] Couchdb. http://couchdb.apache.org/.

[4] Couchdb selector query. https://docs.couchdb.org/en/2.2.0/api/database/find.html.

[5] goleveldb. https://github.com/syndtr/goleveldb.

[6] Hyperledger composer. https://www.hyperledger.org/projects/composer.

[7] Hyperledger composer query language. https://hyperledger.github.io/composer/v0.19/reference/query-language.

[8] Mongodb. https://www.mongodb.com/.

[9] pgtune. https://github.com/gregs1104/pgtune.

[10] Postgresql background worker processes. https://www.postgresql.org/docs/10/bgworker.html.

[11] Postgresql frontend/backend protocol. https://www.postgresql.org/docs/10/protocol.html.

[12] Postgresql libpq - c library. https://www.postgresql.org/docs/10/libpq.html.

[13] Postgresql v10. https://www.postgresql.org/.

[14] Tendermint. https://tendermint.com/.

[15] Bigchaindb: The blockchain database. https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf. 2018.

[16] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 67–78, Feb 2000.

[17] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[18] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[19] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[20] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[22] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, June 2014.

[23] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012.

[24] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[25] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[26] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 739–752, New York, NY, USA, 2008. ACM.

[27] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[28] J. Gehrke, L. Allen, P. Antonopoulos, A. Arasu, J. Hammer, J. Hunter, R. Kaushik, D. Kossmann, R. Ramamurthy, S. T. V. Setty, J. Szymaszek, A. van Renen, J. Lee, and R. Venkatesan. Veritas: Shared verifiable databases and tables in the cloud. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[29] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[30] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006.

[31] M. Hearn. Corda 2016. https://www.corda.net/content/corda-technical-whitepaper.pdf.

[32] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[33] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[34] J. Kwon. Tendermint: Consensus without mining. 2014.

[35] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[36] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 121–132, New York, NY, USA, 2006. ACM.

[37] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation

queries. *ACM Trans. Inf. Syst. Secur.*, 13(4):32:1–32:35, Dec. 2010.

[38] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[39] D. A. Menasce, G. J. Popek, and R. R. Muntz. A locking protocol for resource coordination in distributed databases. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, SIGMOD '78, pages 2–2, New York, NY, USA, 1978. ACM.

[40] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[41] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[42] H. H. Pang and K. . Tan. Authenticating query results in edge computing. In *Proceedings. 20th International Conference on Data Engineering*, pages 560–571, April 2004.

[43] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[44] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PVLDB*, 5(12):1850–1861, 2012.

[45] S. H. Son. Replicated data management in distributed database systems. *SIGMOD Rec.*, 17(4):62–69, Nov. 1988.

[46] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, SE-5(3):188–194, May 1979.

[47] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[48] P. Thakkar, S. Nathan, and B. Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276, Sep. 2018.

[49] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[50] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 59–72, New York, NY, USA, 2007. ACM.

[51] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474, April 2000.

[52] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[53] C. Xu, J. Xu, H. Hu, and M. H. Au. When query authentication meets fine-grained access control: A zero-knowledge approach. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 147–162, New York, NY, USA, 2018. ACM.

[54] Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1480–1491, New York, NY, USA, 2015. ACM.