Reasoning about the Future in Blockchain Databases

Sara Cohen, Adam Rosenthal, Aviv Zohar
The Rachel and Selim Benin School of Computer Science and Engineering, The Hebrew University of Jerusalem
sara@cs.huji.ac.il, adamrosenthal2@gmail.com, avivz@cs.huji.ac.il

***Abstract*—A key difference between using blockchains to store data and centrally controlled databases is that transactions are accepted to a blockchain via a consensus mechanism, and not by a controlling central party. Hence, once a user has issued a transaction, she cannot be certain if it will be accepted. Moreover, a yet unaccepted transaction cannot be retracted by the user, and may (or may not) be appended to the blockchain at any point in the future. This causes difficulties as the user may wish to formulate new transactions based on the knowledge of which previous transactions will be accepted. Yet this knowledge is inherently uncertain.**

We introduce a formal abstraction for blockchains as a data storage layer that underlies a database. The main issue that we tackle is the need to reason about possible worlds, due to the uncertainty in transaction appending. In particular, we consider the theoretical complexity of determining whether it is possible for a denial constraint to be contradicted, given the current state of the blockchain, pending transactions, and integrity constraints on blockchain data. We then present practical algorithms for this problem that work well in practice.

I. Introduction

Typically, a blockchain is an append-only data structure, which is essentially a serialized record of accepted entries. Each individual block is an ordered batch of such updates that were committed together. Updates are added through a consensus mechanism in which all participants continually construct new blocks and accept them into the chain by consensus mechanisms.

Example 1. *The Bitcoin network is comprised of a set of nodes that form a P2P network. The* mempool *of a node contains the set of yet unauthorized transactions, which are gossiped to all nodes as soon as they are issued. In this way, all participants in the system have knowledge of pending transactions.*[1] *Nodes in the network authorize Bitcoin transactions and include them in batches called blocks. Each block is a collection of transactions that transfer the bitcoin currency between addresses (i.e., public keys used to denote ownership).*

Each block additionally contains a cryptographic hash of a predecessor block, and are thus arranged in a chain. Block creation is known as mining, *and nodes that invest the computational effort to try and create blocks are* miners. *Miners choose transactions to include in a new block, while trying to maximize transaction fees. However, it is intractable to determine an optimal set of transactions to be included.*

[1]Due to network delays, some mempools can be temporarily less updated than others.

As blocks can be created concurrently, slightly different versions of the chain may be held by different nodes. Miners reach consensus on the blockchain and quickly distribute newly found blocks to one another. Thus, a block is typically not considered to be accepted into the blockchain until it is several (often six) layers deep.

It is not possible to determine, a priori, when and whether a transaction will be included in the blockchain, in *any* blockchain system available currently. Indeed, due to a variety of reasons, it is quite possible for transactions not to be accepted by the network, or to get delayed for extended periods. Even an old record that was created and propagated to other nodes by its original creator may be suddenly added to the blockchain much later on. This can result in unexpected, and in particular, undesirable, behavior.

The inherent uncertainty due to yet unaccepted transactions is great, as there are typically a large number of transactions in the queue. (This is in contrast to the uncommon and relatively small uncertainty arising from temporary forks in the blockchain.) Due to interdependencies between transactions, not all subsets of transactions can be accepted to the blockchain. Hence, understanding the future for a blockchain based system requires the ability to reason about *possible worlds*, which is the focus of this paper.

Our main contributions are as follows. We present an abstract *model* of databases using blockchains as a storage layer that is independent of the specific protocol and implementation of the blockchain. We study the problem of determining whether a blockchain database can reach an undesirable state and provide complete *complexity results*. We present novel *algorithms* to test, in a real system, whether a blockchain database can reach an undesirable state.

II. Related Work

Lately, much interest has arisen in using blockchains as a storage mechanism for relational databases [1], [2], [3], [4]. These works focus on the consensus and implementation layers. In our work, we take a higher level view of a blockchain as a storage layer, in which the implementation details of the mechanisms are abstracted away, and the fundamental querying problems are given full focus.

Conceptually, a database using blockchains as a storage layer consists of a current state (i.e., a set of relations, stored in the blockchain), integrity constraints that must hold in every state, and a set of append-only transactions that have been

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00206

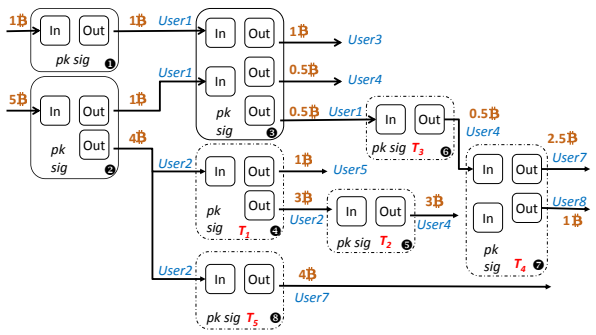


Fig. 1: Structure of some Bitcoin transactions.

issued by users, but have not been incorporated (yet) into the blockchain. Thus, a blockchain database is a succinct representation of a large number of possible worlds.

Previous work considered inconsistent databases [5], [6], [7], [8], [9], which represent many possible worlds in a succinct manner. The results in this paper are somewhat in this spirit. However, the setting is significantly different, as are the problems studied, and previous results cannot be used.

III. Background

We ground our results by using the running example of a Bitcoin system. However, the problems considered in this paper are interesting over many different types of blockchain usage scenarios.

A transaction in Bitcoin, which uses the UTxO model, is a transfer of funds from inputs to outputs (many-to-many). Bitcoin constraints require, e.g., that the inputs to a transaction must be outputs of previously accepted transactions and a given output can be used as an input to a single new transaction. Transactions that share even a single input are considered conflicting and cannot be accepted into the blockchain together. Transactions that spend the outputs of previous ones are dependent upon these parent transactions and cannot be included without first including the parent transactions.

Figure 1 shows an example of Bitcoin transactions, along with their inputs and outputs. For now, the dotted lines around some of the transactions can be ignored. Due to the underlying constraints it is not possible to include both T_1 and T_5 simultaneously in the blockchain (as they expend the same output), and T_2 can be included only if T_1 is previously included (as T_2 expends an output of T_1).

Transactions in Bitcoin may be delayed due to fee fluctuation. In this case, the user would still want to issue the transaction, perhaps with an increased fee the second time around. Alas, once two transaction messages are broadcast to the network, it is quite possible that both the new and the old message will be included in the blockchain. Once signed, a transaction can be rebroadcast to the network by anyone, and is often stored by nodes, with the hope of later inclusion in the blockchain. Unlike a typical database system, there is no method to retract an issued transaction. The unfortunate consequence of two such transactions making it to the blockchain is that the user would then pay twice the intended amount. This problem is not hypothetical, but was used to attack exchanges in the past [10].

The design of blockchain systems does include a remedy for this situation: two transactions can be made to conflict in such a way as to rule out their co-existence in the blockchain. Transactions contradicting the underlying constraints of the system are not propagated and are immediately discarded. For example, to avoid double-paying, users can issue a new transaction that conflicts with a previous transaction. Thus, it will be impossible for both transactions to co-exist in the blockchain. Reasoning about questions more complex than whether double-paying may occur, is much more difficult.

Example 2. *Bitcoin exchanges keep user funds in two wallets: a strongly secured, but less accessible, "cold wallet", and a less secure, more accessible "hot wallet". Access to the cold wallet often requires human intervention, as secret keys are distributed in multiple secure locations. The exchange cannot cover the full amount of user's deposits from its hot wallet, as it prefers to keep most funds in the more secure cold wallet.*

During its normal operation, the exchange receives deposit and withdrawal requests and these are served exclusively from its hot wallet. An important question for an exchange is: given the currently issued transactions, is it possible for the balance of the hot wallet to go below (or far above) a given threshold range. Too few funds in the hot wallet will prevent the exchange from fulfilling transaction requests, while overly large hot wallets may not be covered by insurance. The only way to correctly answer this question is by reasoning about possible worlds. In particular, a simple summing up of deposit and withdrawal requests will not yield the correct answer, due both to dependencies between transactions, and to the uncertainty of transaction acceptance.

IV. Formal Framework

A *relation* $R(A_1, \ldots, A_n)$ is associated with a set of ground tuples of arity n. Given a set of relations $\mathcal{R}$, an *insert transaction* (or simply, *transaction*, for short) for $\mathcal{R}$ is a set T of ground tuples for (some of) the relations in $\mathcal{R}$. Intuitively, T is a set of tuples that we would like to insert into the relations of $\mathcal{R}$. We consider only insert transactions, as blockchain databases are append-only databases.

We consider three types of integrity constraints, namely, *key constraints* (key), *functional dependencies* (fd) and *inclusion dependencies* (ind). Satisfaction of a set of integrity constraints $\mathcal{I}$ by a set of relations $\mathcal{R}$ is defined in the standard fashion, and is denoted $\mathcal{R} \models \mathcal{I}$.

We formally define the notion of a *blockchain database*. A *blockchain database* $\mathcal{D}$ is a triple $(\mathcal{R}, \mathcal{I}, \mathcal{T})$, where

- $\mathcal{R}$ is a set of relations, called the *current state,*
- $\mathcal{I}$ is a set of integrity constraints, such that $\mathcal{R} \models \mathcal{I}$,
- $\mathcal{T} = \{T_1, \ldots, T_k\}$ is a finite set of transactions for $\mathcal{R}$, each of which is a set of tuples.

The transactions in $\mathcal{T}$ may be appended to the database. However, it is possible that they will not be appended, e.g., because they are not mutually consistent with $\mathcal{I}$, due to network problems, or simply due to lack of financial incentive.[2]

We define the *can-append relationship* $\mathcal{R} \rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$ for $\mathcal{D}$, as follows. We write $\mathcal{R} \rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$ if $\mathcal{R}' = \mathcal{R}$ or if there is some $T \in \mathcal{T}$ such that $\mathcal{R}' = \mathcal{R} \cup T$ and $\mathcal{R}' \models \mathcal{I}$. We denote the transitive closure of this relationship as $\mathcal{R} \Rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$. Intuitively, this relationship defines a new instance of relations that can be derived by incrementally adding transactions from $\mathcal{T}$, while preserving all integrity constraints. We say that $\mathcal{R}'$ is a *possible world* for $\mathcal{D}$ if $\mathcal{R} \Rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$, and denote the set of all possible worlds of $\mathcal{D}$ by $Poss(\mathcal{D})$. Possible worlds can be efficiently recognized, in PTIME.

Over time a blockchain database evolves by the addition of transactions to the current state. However, at any given point in time, it is not possible to know for certain which pending transactions in $\mathcal{T}$ will be permanently added to $\mathcal{R}$. A user may wish to ensure that in all possible worlds, specific undesirable things will not happen, by answering the following question: *Given a Boolean query q, does q evaluate to false over all possible worlds of $\mathcal{D}$?* We call such a Boolean query, which we desire to remain unsatisfied, a *denial constraint*. Formally, a denial constraint q is satisfied by a blockchain database $\mathcal{D}$, denoted $\mathcal{D} \models \neg q$, if, for all $\mathcal{R}' \in Poss(\mathcal{D})$, it holds that $q(\mathcal{R}') =$ false. For example, the following simplified denial constraint requires that all bitcoins from Alice are given to trusted individuals (as only then, will it always return false):

$$q() \leftarrow \mathsf{Input}(\text{'AlcPK'}, ntx), \mathsf{Output}(ntx, pk), \neg\mathsf{Trusted}(pk)$$

Being able to determine whether a denial constraint is satisfied by a blockchain database is a fundamental issue, that allows users to interact with the database with some degree of clarity as to the possible downsides of their transactions. Hence, we formally define and study this problem. Let $\mathcal{D}$ be a blockchain database and q be a denial constraint. The *denial constraint satisfaction problem* is to determine whether $\mathcal{D} \models \neg q$. We use the measure of data complexity (i.e., we assume the denial constraint is of constant size, while the size of the database is unbounded), as otherwise, query evaluation over a standard database is already intractable.

The complexity of the denial constraint satisfaction problem varies greatly, depending on the language of the denial constraints (conjunctive/aggregate), as well as the types of integrity constraints (among key, fd, ind) allowed. Table I summarizes the main complexity results.

V. Algorithms

From a practical standpoint, the complexity results of the previous section are of questionable usefulness. First, only limited cases are tractable. Indeed, denial constraint satisfaction is intractable for all query classes if databases include both key constraints and inclusion dependencies. This result is not satisfying, as *(1)* it is common to have both key constraints and inclusion dependencies and *(2)* notwithstanding the theoretical results, it is possible that in practice such worst case scenarios will not arise. Second, for the tractable cases, our results provide a polynomial time algorithm. However, this algorithm is of time that is order of the database with an exponent in the size of the query. Since the query size is constant, this gives a theoretically polynomial time algorithm. However, in practice, such an algorithm is quite expensive.

[2]In the real world, transactions are also associated with a fee that is accrued when it is added to the current state.

Query Type		Integrity Constraints	Complexity
conjunctive		$\{key, fd\}$, $\{ind\}$	PTIME
conjunctive		$\{key, ind\}$	CoNP-c
max	$>$	$\{key, fd\}$, $\{ind\}$	PTIME
	$>$	$\{key, ind\}$	CoNP-C
	$<, =$	$\{key, fd\}$	PTIME
	$<, =$	$\{ind\}$	CoNP-C
cnt, cntd, sum	$<$	$\{key, fd\}$	PTIME
	$<$	$\{ind\}$	CoNP-C
	$>, =$	$\{key\}$, $\{ind\}$	CoNP-C

TABLE I: Summary of complexity results for the denial constraint satisfaction problem.

In this section we present algorithms for the denial constraint satisfaction problem in the presence of functional dependencies (including key constraints) and inclusion dependencies, when the denial constraint is *monotonic*, an important common case. This algorithm can be improved for denial constraints that are *connected*.

A Boolean query q is *monotonic* if, for all sets of relations $\mathcal{R}$ and $\mathcal{R}'$, such that $\mathcal{R} \subseteq \mathcal{R}'$, if $q(\mathcal{R}) =$ true then also $q(\mathcal{R}') =$ true . Intuitively, for monotonic denial constraints it is sufficient to consider possible worlds that are maximal. Therefore, we create a graph $G^{fd}_{\mathcal{T}}$ with transactions as nodes, and edges between transactions that are mutually consistent with respect to the functional dependencies. Maximal possible worlds correspond to maximal cliques.

The algorithm NaiveDCSat in Figure 2 can be used to determine satisfaction of a monotonic denial constraint. Observe that once we restrict ourselves to sets of transactions $\mathcal{T}' \subseteq \mathcal{T}$ that are maximal cliques in $G^{fd}_{\mathcal{T}}$, for each such clique there is a single maximal possible world over $(\mathcal{R}, \mathcal{I}, \mathcal{T}')$. This maximal possible world can be generated using the algorithm GetMaximal. Thus, the algorithm NaiveDCSat simply iterates over all maximal possible worlds, and checks whether the denial constraint q returns true over at least one such world.

For queries that are *connected* we can improve upon the algorithm NaiveDCSat. Formally, a query is *connected* if it is conjunctive (i.e., not an aggregate query) and the *Gaifman graph* of the query is connected. In this case, we divide up the sets of pending transactions $\mathcal{T}$ into disjoint subsets that can be considered independently. The graph $G^{q,ind}_{\mathcal{T}}$ is over the set of nodes $\mathcal{T}$ and contains an edge between two transactions if they may be needed together in order to form a satisfying possible world. (Details omitted due to lack of space.) We also further improve our algorithm by considering constants appearing in the query. Such constants can be used to filter

Algorithm NAIVEDCSAT($\mathcal{R}, \mathcal{I}, \mathcal{T}, q$)
1. **for** each maximal clique $\mathcal{T}'$ in $G^{fd}_{\mathcal{T}}$
2. **do** $\mathcal{R}' \leftarrow$ GETMAXIMAL($\mathcal{R}, \mathcal{I}, \mathcal{T}'$)
3. **if** $q(\mathcal{R}')$ = true
4. **then return** false
5. **return** true

Algorithm GETMAXIMAL($\mathcal{R}, \mathcal{I}, \mathcal{T}$)
1. $\mathcal{R}' \leftarrow \mathcal{R}$
2. $size \leftarrow 0$
3. **while** $\mathcal{T} \neq \emptyset$ and $size \neq |\mathcal{T}|$
4. **do** $size \leftarrow |\mathcal{T}|$
5. **for** each $T \in \mathcal{T}$
6. **do if** $\mathcal{R}' \cup T \models \mathcal{I}$
7. **then** $\mathcal{T} \leftarrow \mathcal{T} \setminus T$
8. $\mathcal{R}' \leftarrow \mathcal{R}' \cup T$
9. **return** $\mathcal{R}'$

Fig. 2: Algorithm for monotonic denial constraints.

Algorithm OPTDCSAT($\mathcal{R}, \mathcal{I}, \mathcal{T}, q$)
1. **for** each connected component $\mathcal{T}'$ in $G^{q,ind}_{\mathcal{T}}$
2. **do if** COVERS($\mathcal{R}, \mathcal{T}', q$)
3. **then for** each maximal clique $\mathcal{T}''$ in $G^{fd}_{\mathcal{T}'}$
4. **do** $\mathcal{R}' \leftarrow$ GETMAXIMAL($\mathcal{R}, \mathcal{I}, \mathcal{T}''$)
5. **if** $q(\mathcal{R}')$ = true
6. **then return** false
7. **return** true

Fig. 3: Algorithm for connected monotonic denial constraints

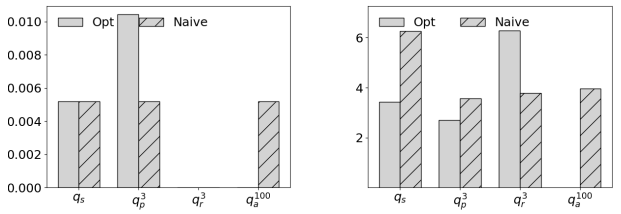
out the set of possible worlds that must be considered. The resulting algorithm OPTDCSAT appears in Figure 3.

VI. EXPERIMENTATION

In our experimentation, we studied the scalability of the system with respect to the data size, the query type and size, the number of pending transactions and the number of contradictions in the pending transactions. We demonstrate the scalability results with one example experiment.

We ran a Bitcoin node to get real Bitcoin data. In this experiment we used the first 200,000 Bitcoin blocks, consisting of over 7 million transactions. We used the subsequent 30 blocks, containing almost 4,000 transactions, as the pending transactions. We included 20 functional dependency contradictions. We considered four types of denial constraints q_s (simple equality condition), q_p^3 (a path of length three), q_r^3 (star query of size three) and q_a^{100} (a sum aggregation query). In Figure 4a and 4b we chose constants such that the denial constraints are satisfied and unsatisfied, respectively.

For the satisfied denial constraint, all runs complete in just a few milliseconds. In fact, some of the runtimes were so short that they are not visible in the graph. For unsatisfied denial constraints, runtime takes up to six seconds. For unsatisfied denial constraints usually the runtime of OPTDCSAT is significantly lower than that of NAIVEDCSAT, as the former considers much smaller possible worlds. Sometimes, however,



(a) Different Query Types (Satisfied Constraints)

(b) Different Query Types (Unsatisfied Constraints)

Fig. 4: Execution time.

the trend in runtime reverses, as algorithm NAIVEDCSAT may consider fewer possible worlds before it finds a satisfying assignment for underlying query (and thus, concluding that the denial constraint is not satisfied) when the worlds are larger. This is precisely what happens for q_r^3. The same phenomenon occurs in other experiments.

As is apparent from the experimentation, our algorithms are very fast for satisfied denial constraints, running at sub-second speeds. Typically, our algorithms will be run when a user wishes to issue a transaction, and wants to determine that no bad outcome can occur. The user hypothetically adds her transaction and runs the algorithm. When the transaction can be safely issued (i.e., the denial constraint is satisfied), the user will usually be notified of this quickly. For denial constraints that are unsatisfied, the user will often need to wait longer. However, this should be acceptable as in such cases the user indeed should avoid issuing her transaction.

ACKNOWLEDGMENT

The authors were partially supported by the Israel Science Foundation (Grants 879/16 and 1504/17), and by The Federmann Cyber Security Center in conjunction with the Israel national cyber directorate.

REFERENCES

[1] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran, "Blockchain meets database: Design and implementation of a blockchain relational database," *PVLDB*, 2019.
[2] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "Blockchaindb - A shared database on blockchains," *PVLDB*, 2019.
[3] S. Maiyya, V. Zakhary, D. Agrawal, and A. El Abbadi, "Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains," *PVLDB*, 2018.
[4] Y. Zhu, Z. Zhang, C. Jin, A. Zhou, and Y. Yan, "Sebdb: Semantics empowered blockchain database," in *ICDE*, 2019.
[5] M. Arenas, L. E. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases," in *SIGMOD*, 1999.
[6] S. Staworko, J. Chomicki, and J. Marcinkowski, "Prioritized repairing and consistent query answering in relational databases," *Ann. Math. Artif. Intell.*, vol. 64, no. 2-3, pp. 209–246, 2012.
[7] L. E. Bertossi, "Consistent query answering in databases," *SIGMOD Record*, vol. 35, no. 2, pp. 68–76, 2006.
[8] J. Chomicki, J. Marcinkowski, and S. Staworko, "Computing consistent query answers using conflict hypergraphs," in *CIKM*, 2004.
[9] M. C. Marileo and L. E. Bertossi, "The consistency extractor system: Answer set programs for consistent query answering in databases," *Data Knowl. Eng.*, vol. 69, no. 6, pp. 545–572, 2010.
[10] C. Decker and R. Wattenhofer, "Bitcoin transaction malleability and mtgox," in *ESORICS*, 2014.