Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2018) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

The 16th International Conference on Mobile Systems and Pervasive Computing (MobiSPC)
August 19-21, 2019, Halifax, Canada

Trusted relational databases with blockchain: design and optimization

Amin Beirami[a], Ying Zhu[b], Ken Pu[a]

[a] *Faculty of Science, UOIT, Oshawa, ON, Canada*
[b] *Faculty of Business and IT, UOIT, Oshawa, ON, Canada*

Abstract

With the emergence of large scale data collection from Internet of Things and mobile devices, the notion of trust is now an increasingly important aspect of the next generation of data processing systems. We propose a blockchain enabled relational storage system that supports immutable transactions and temporal snapshots. By embedding blockchains in relational tables, the database stores trust related information in a tamper proof fashion, making the data provenance provably verifiable. To support large query workloads, we further propose an optimization algorithm that determines the best temporal snapshots to materialize in order to minimize the total time cost of answering a given query workload. Experimental evaluation shows that our materialized snapshot approach improves the performance of large analytical query workloads by as much as 50 times.

Keywords: trusted database; blockchain; query answering; optimization

1. Introduction

In this paper, we study the problem of implementing a trusted relational database system to support immutable and auditable transactions, and efficient processing of large numbers of analytical queries.

1.1. Motivation

Consider a relational database used to store purchase orders of various customers and suppliers. Over time, customers and suppliers may update the database: insert new orders, make modifications to previous ones, or delete

* Corresponding author. Tel.: +1-905-721-8668
E-mail address: ken.pu@uoit.ca

existing ones. Trust is an exceedingly important aspect of electronic data management. The system must maintain the provenance of all database transactions in scenarios of dispute. For example, a customer C_1 may claim that an order has been cancelled *before* an agreed deadline, but the supplier S_2 may dispute the time of cancellation. There are many such scenarios in practice. When such disagreements over the history of the database arise, we would like to have an audit process for which the finding is *indisputable* and hence *trusted* by all parties.

We would like to design a trusted relational database system built on existing and mature technology (such as modern standard RDBMS[1]) to store the transactions in such a way that the entire history of the database is tamper proof. Namely, while we may not be able to prevent unwanted modifications to the database (e.g., malicious root users or accidental hardware failures), *any* modification to the data and its history will be revealed by an audit process. In other words, we can *trust* the state of the database by verifying that the database is intact and free of tamper.

Equally important to trust, the database must also support efficient evaluation of analytical queries. We envision that many users may submit a large number of queries, which we will refer to as the *query workload*, to the database. Each query examines the state of the database at a query specific timestamp. Each query may be a simple select query, or one that contains complex data transformations and aggregations. So another functional requirement of the trusted database is to support query workloads on a large scale.

1.2. Problem Definition

Let D be a relational database. We are interested in the evolution of the database, hence the database is parameterized by *versions*. A database state at some version is given by $D(t)$ where t is a timestamp. For simplicity, we will assume $t \in \mathbb{N}^+$. Let U be the users. A transaction is a change applied to the database (addition, deletion and modification of tuples in D), loosely denoted by ΔD. We also denote the updated database after transaction ΔD_t submitted by user $u \in U$ as: $D(t+1) = D(t) + (\Delta D_t, u)$

When the user is understood, we may write $D(t+1) = D(t) + \Delta D_t$

Assuming that the database starts with an empty initial state, $\emptyset$, at $t = 0$, we define the timeline of the database up to some time t as the series $\mathrm{TL}(t)$ given by:

$$\mathrm{TL}(t) = \begin{bmatrix} \emptyset \\ \text{null} \\ 0 \end{bmatrix}, \begin{bmatrix} D_1 \\ u_1 \\ 1 \end{bmatrix}, \begin{bmatrix} D_2 \\ u_2 \\ 2 \end{bmatrix} \dots \begin{bmatrix} D_{t-1} \\ u_{t-1} \\ t-1 \end{bmatrix}.$$

Namely, D_i is the i-th transaction submitted by user u_i at time $t = i$.

We wish to design a system **DB** to manage the timeline TL with the following properties:

- **Timeline retrieval**: for any time $t > 0$, the timeline $\mathrm{TL}(t)$ can be completely retrieved from **DB**.
- **Snapshot retrieval**: for any time $t > 0$, the state of the database D can be reconstructed from **DB**.
- **Verification**: There exists a function verify : **DB** $\mapsto$ {true, false} that performs the verification of the system such that any *tampered* version of the system **DB**$' \neq$ **DB** will be fail the verification. We make no assumption regarding access control to the storage, so the source of tampering may include root level access to **DB**. Thus, we limit our problem to *detection*, not prevention of tampering.
- **Query Workload**: let q be some query defined on a snapshot $D(t_q)$. We call t_q the query timestamp. The system **DB** can efficiently evaluate a large number of such queries with different query timestamps. The collection of queries is called the *query workload*, written $Q = \{q_1, q_2, \dots, q_N\}$.

2. Related Work

Trustworthiness has received attention in recent years in various areas including real-time distributed systems [9], secret sharing schemes in the cloud [6], and utilizing log files for forensic analysis [11].

Our work focuses on trusted transactions in an environment where users cannot be trusted, not even users with root access to the underlying database. The scenario of privileged malicious users has been discussed by several

[1] We use Postgresql for our implementation.

researchers [3, 14, 13]. The approaches used have been network based and data inspection based detection mechanisms for database tampering. In contrast, we rely on embedded blockchains, and do not require deep access to the system level layer (such as filesystem activity or network traffic inspection).

There have also been work on database level forensic inspection [7, 8] using triggers and related database features. When running at scale, triggers can be prohibitively expensive. Our approach uses efficient ad hoc inspection of the blockchain structures to determine the integrity of the database.

Query answering using materialized views is a well established field in database [5, 12, 10, 1]. The focus has been on views that are defined by general SQL queries. In our work, the snapshots are generated by queries in a very specific form, and thus, we are able to specialize the materialized view selection problem to obtain a more efficient algorithm for finding the optimal solution.

3. Design of trusted relational tables

Based on the problem definition given in Section 1.2, we describe the design of the trusted database management system. In our design, we embed blockchains in the relational tables to support immutable transactions and verification.

3.1. Users, keys and digital signatures

Let U be a collection of *users*. Each instance in U can be a user in the conventional sense, a mobile device or an Internet of Things (IoT). Without loss of generality, we call all such instances *users*. A user $u \in U$ is uniquely characterized by a pair of public and private keys: public(u), private(u). The key pairs are typically generated using the RSA scheme [2] using large number of bits (e.g. 4096) to avoid key collisions even when there are billions of instances in U.

Given arbitrary data x, the encoding function is given as $\text{enc} : (x, \text{private}(u)) \mapsto \hat{x}$ where $\hat{x}$ is a lossless encoding of x, commonly known as the ciphertext. The ciphertext can be decoded using the encoding function again, but with the public key:

$$\text{enc}(\hat{x}, \text{public}(u)) = x$$

To prove authorship of some data x by user u, the following digital signing scheme is used:

1. Compute the hash of x by a fixed hash function: $h(x)$.
2. Compute the ciphertext of $h(x)$: $\text{sig}(x) = \text{enc}(h(x), \text{private}(u))$.
3. Publish the data, the signature and the public key: $\langle x, \text{sig}(x), \text{public}(u)\rangle$. The published data does not leak the private key of the user.

To verify the authenticity of the authorship from the published data:

1. Decode the signature: $y = \text{enc}(\text{sig}(x), \text{public}(u))$.
2. Verify that the decoded signature is the hash value of x using the fixed hash function h: $h(x) \stackrel{?}{=} y$

3.2. Blockchain in relational tables

We propose a scheme to augment each relational table in a database D with additional attributes to store the transaction timestamp, a new table signature, a previous table signature, and the user public key. We also need an extra bit flag to indicate if the transaction is a deletion. In practice, the deletion bit flag can be padded to the end of the transaction signature, but for readability, we assume yet another boolean attribute added to the table.

For each table T in the database D with attributes attr(T), we assume that at least one attribute, *id*, is the row id[2]. The table T is then augmented with the additional attributes: (t, sig, sig$'$, pubkey, del?), each is as described above. The new table with the augmented attributes is written as $\hat{T}$. Note, we keep track of two signatures: sig and sig$'$ corresponding to the signatures of the table *after* $T(t)$ and *before* $T(t-1)$ the transaction respectively. This is necessary to ensure the blockchain integrity [4, 15].

Transaction updates: Suppose a user $u \in U$ wants to commit a transaction ΔD. For each table T involved in the transaction, let ΔT be the rows affected (added, modified, or removed). For each tuple $x \in \Delta T$, we insert into $\hat{T}$ the augmented tuple $\langle x, i, s, s', \text{public}(u), \text{del?}(x)\rangle$ where i is the current transaction timestamp, s, s' the signatures as described below, public(u) the public key of the transaction author, and finally del?(x) is the boolean flag indicating if x is removed by in the transaction.

The signatures are given as:

- $s' = x[\text{sig}'] : x[t] = i - 1$. This is uniquely determined as the signature of the previous transaction at timestamp $i-1$.
- $s = \text{enc}(\text{hash}(\Delta T + s'), \text{private}(u))$. This is the digital signature of the transaction data *and* the previous signature.

Verification: Each table-level transaction ΔT is uniquely identified by its timestamp t. It is also uniquely identified by its signature sig_t. The verification of the transaction is given as:

$$\text{verify}(\Delta T) = \text{enc}(\text{sig}_t, \text{public}(u)) \stackrel{?}{=} \text{hash}(\Delta T + \text{sig}_{t-1})$$

This allows us to verify all the tables $\hat{T}$ in the temporal database. Any tampering will be detected by the verification function.

4. Query answering with optimal snapshots

In this section we will discuss the evaluation of query workloads with the database given in Section 3. Each table $\hat{T}$ is a timeline of transactions, digitally signed and verifiable. Recall in Section 1.2, we need to process a large number of queries at different query timestamps.

To support arbitrary query q, we need to compute the snapshots of all the tables $\hat{T}$ that are needed by q at t_q. This means we need to perform a group-by operation using the row id (from the original table T), with each group aggregated to the row with the latest transaction timestamp. Then we remove all the rows with their deletion boolean flags set to true. This can be expressed in SQL using windowing functions as shown in Figure 1.

Finally, the query q is evaluated as: $q(\text{snapshot}(\hat{T}_1, t_q), \text{snapshot}(\hat{T}_2, t_q), \dots)$.

For the remainder of this section, we present a family of optimization techniques to efficiently process query workload $Q = \{q_1, q_2, \dots, q_N\}$ by optimally materialize a fixed number of snapshots.

4.1. Materialized single snapshot

If we have a snapshot $S = \text{snapshot}(T, t_s)$, and a query q with query timestamp $t_q \ /t_s$, we can evaluate q by constructing snapshot(T, t_q) based on S. This involves applying all the transactions in the interval of $[t_s, t_q]$ (commit transactions) or $[t_q, t_s]$ (rollback transactions). Since each transaction has its unique timestamp, the number of transactions in the interval is given by $|t_q - t_s|$.

[2] Popular RDBMS such as MySQL and Postgresql have built `row_id` to generate row ids.

```
snapshot(T̂, t_q):
  WITH V AS (
    SELECT id, {last_value(x) : x ∈ attr(T)} OVER W
    FROM T̂
    WHERE T̂.t ≤ t_q
    WINDOW W AS PARTITION BY id ORDER BY T̂.t
  )   SELECT id, {x : x ∈ attr(T)}
  FROM V
  WHERE NOT V.del?
```

Fig. 1: Aggregation query to generate the snapshot of a time at a given timestamp.

Suppose we only have a single materialized snapshot s, and a query workload Q. When every query $q \in Q$ utilizes s to compute snapshot(S, t_q), we have the following cost model:

$$\text{cost}(Q|s) = \sum_{q \in Q} |t_q - t_s|$$

This leads to the first optimization problem.

Definition 1 (Optimal single snapshot placement problem)**.** *Given a query workload, What is is an optimal snapshot placement* s^* *where* $s^* = \operatorname{argmin}_s \text{cost}(Q|s)$*?*

One can show that the single snapshot placement can be solved by placing the snapshot at the median of the query timestamps.

Theorem 1 (Optimal single snapshot placement)**.**

$$t_s^* = \text{median}(t_q : q \in Q)$$

4.2. Materializing multiple snapshots

Naturally, we expect to improve the performance with multiple snapshots. This takes us to the multiple snapshot problem. Let $S = \{s_1, s_2, \ldots, s_m\}$, and Q the query workload. Each query will be evaluated using the *nearest* snapshot. So, the cost becomes:

$$\text{cost}(Q|S) = \sum_{q \in Q} \min\{|t_q - t_s| : s \in S\}$$

Definition 2 (Optimial m-snapshots placement problem)**.** *Let* $S^* = \{s_1, s_2, \ldots, s_m\}$ *be m snapshots. What are the placements so that* cost$(Q|S)$ *is minimized? We will denote* S^* *as* opt(Q, m)*.*

We first observe that the m-snapshot problem enjoys a property that we call the *optimality of subproblems*.

Theorem 2 (Optimality of subproblems)**.** *Let $S^* = \text{opt}(Q, m)$. Then S^* partitions Q by the nearest snapshot relation. The prefix of S^* is also an optimal snapshot placement of the prefix of the partition of Q.*

This leads to a recursive solution to the m-snapshot placement problem shown in Figure 2.

Base case:
$\text{opt}(Q, 1) = \{\text{median}(Q)\}$

Induction:
let $i^* = \underset{i \in [1,n]}{\text{argmin}}\ \text{cost}(\text{opt}(Q[1:i], m-1))$
let $n = |Q|$
$\text{opt}(Q, m) = \text{opt}(Q[1:i^*]) \cup \{\text{median}(Q[i^*+1:n])$

Fig. 2: Recursive solution to the m-snapshot placement problem.

It can be shown that the recursion in Figure 2 can be implemented as dynamic programming with complexity $O(mn^2)$ where m is the number of snapshots, and n the number of queries.

4.3. Approximate snapshot placement using clustering

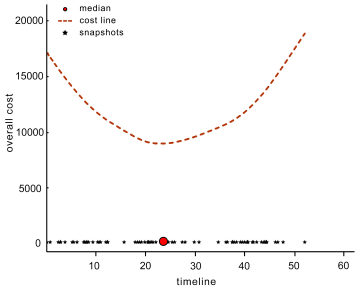
While the algorithm in Figure 2 computes the optimal snapshot placement in polynomial time, its complexity is still too high if the query load is large (thousands) with many materialized snapshots (hundreds).

We make the observation that the optimal m-snapshot placement is based on the medians of subsets of queries. This is also true for the case of $m = 1$ as given in Theorem 1. This leads us to believe that the cluster structure in the timestamps of the query workload is a strong determining factor in snapshot placement. Thus, we propose to *approximate* the optimal snapshot locations by the *mean* centroids of m-clustering of $\{t_q : q \in Q\}$ as shown in Figure 3.

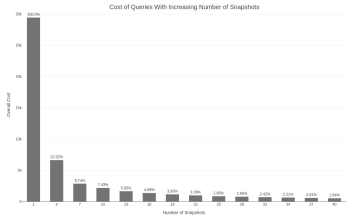
approx(Q, m):
$\mathbf{C}$ = k-mean-clustering$(\{t_q : q \in Q\}, m)$
return $\{\text{mean}(C) : C \in \mathbf{C}\}$

Fig. 3: Approximate snapshot placements using clustering

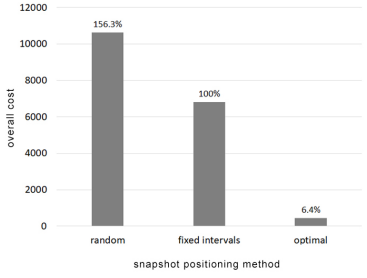
This is encouraging since k-mean-clustering has a complexity of $O(mn)$. In fact, we can also adjust the runtime by controlling the number of iterations over the query timestamps. By reducing the number of iterations we can further speed up the snapshot placement computation.



(a) Answering a single query using the nearest snapshots

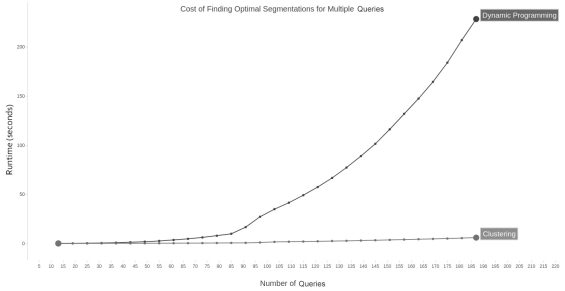


(b) Reduction in cost with increasing number of snapshots

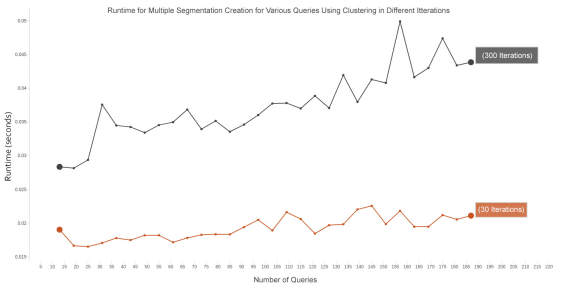


(c) Different snapshot selection algorithms

Fig. 4: The effects of query workload answering using materialized snapshots

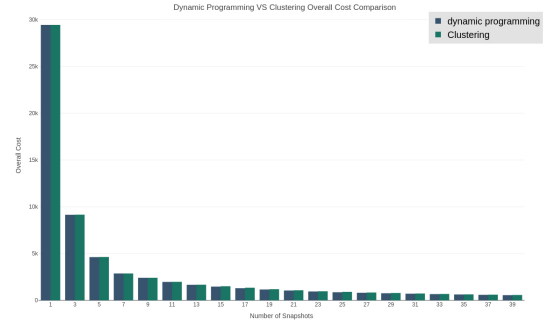


(a) Optimal vs heuristic runtime

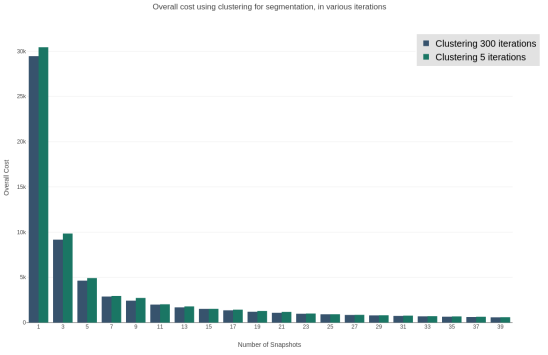


(b) Heuristc with 300 iterations and 30 iterations

Fig. 5: Runtime comparision of different snapshot placements



(a) Exact optimal cost v.s. heuristic approximation using clustering



(b) Approximation quality of 300 iterations vs 5 iterations

Fig. 6: Quality of the approximate snapshot placements using clustering

5. Evaluation and experiments

We have conducted a number of experiments to evaluate the effectiveness of the proposed algorithms.

A synthetic query workload is generated against a database with over 1 million tuples. Figure 4 shows the benefit of having optimally placed snapshots. Figure 4a shows the effect of having just a single snapshot. One can see that the cost is minimal when the materialized snapshot is at the median of the query workload. Figure 4c shows the benefit of having more snapshots optimally materialized. We see that with 40 materialized snapshots, the cost is reduced to less than 2% when compared to just one materialized snapshot. Figure 4b shows the relative performance of different snapshot placement strategies. We see that random placements and fixed interval placements are significant worse than the optimal placement strategy, with the cost being over 15 times worse.

We have compared the runtime performance of the optimal snapshot placement using dynamic programming and clustering based heuristics. Figure 5a compares the runtime performance of dynamic programming and clustering with 300 iterations. One can see that the clustering heuristics scales extremely well with increasing query workload size. To further speedup the snapshot placement computation, we can reduce the number of iterations further as shown in Figure 5b.

The heuristic approach is highly effective in obtaining near optimal snapshot placements. Figure 6a compares the costs of optimal placements and approximate placements using 300 iterations. The increase in the cost is less than 2%. Even with just 5 iterations, as shown in Figure 6b, the resulting approximation is very close to the exact optimal placement.

6. Conclusion

We have presented the problems and solutions in building a trusted data management system with the following properties:

- The system supports transactional data processing with a large number of users and devices.
- Data authentication can be verified by the embedded blockchains that store trust related information associated with all transactions.

The database supports analytical queries at different query timestamps. To achieve high efficiency, we propose to optimally materialize snapshots at various timestamps in order to minimize the time cost in query evaluation. The problem is formulated as an optimization problem which can be solved exactly using dynamic programming. To reduce the optimization overhead, we constructed a clustering based heuristic algorithm. The experimental evaluation shows that materialize snapshots greatly improve query throughput (by a factor of 50), and the heuristic approximation of snapshot placement is very close (within 2%) to the exact optimal placements.

Future work. There are a number of directions we plan to further this research program.

- Our approach is currently limited to the relational model. We plan to generalize it to other data models including semi-structured and graph data.
- We focused on computing (near) optimal snapshot placements for static query workload. We plan to explore adaptive snapshots placements to handle dynamic query workload in an online fashion.
- We envision that the distributed blockchain consensus protocol can be adopted to manage trust for distributed databases to support automatic repair on tampered database instances.

References

[1] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. Clustering-based materialized view selection in data warehouses. In *East European Conference on Advances in Databases and Information Systems*, pages 81–95. Springer, 2006.
[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
[3] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *2009 the 18th conference on USENIX security symposium*, pages 317–334. USENIX Association, 2009.
[4] Vikram Dhillon, David Metcalf, and Max Hooper. *Blockchain Enabled Applications*. Apress, Berkeley, CA, 2017.
[5] Jiang Du, Renée J Miller, Boris Glavic, and Wei Tan. Deepsea: Progressive workload-aware partitioning of materialized views in scalable data analytics. In *EDBT*, pages 198–209, 2017.
[6] Rahul Dutta and Annappa B. Privacy and trust in cloud database using threshold-based secret sharing. In *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 800–805, 2013.
[7] Daniel Fabbri, Ravi Ramamurthy, and Raghav Kaushik. Select triggers for data auditing. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1141–1152, 2013.
[8] Werner K. Hauger and Martin S. Olivier. The role of triggers in database forensics. In *2014 Information Security for South Africa*, pages 1–7, 2014.
[9] Grace Khayat and Hoda Maalouf. Trust in real-time distributed database systems. In *2017 8th International Conference on Information Technology (ICIT)*, pages 572–579, 2017.
[10] Amit Shukla, Prasad Deshpande, Jeffrey F Naughton, et al. Materialized view selection for multidimensional datasets. In *VLDB*, volume 98, pages 488–499, 1998.
[11] Arunesh Sinha, Limin Jia, Paul England, and Jacob R. Lorch. Continuous tamper-proof logging using tpm 2.0. In *International Conference on Trust and Trustworthy Computing*, pages 19–36. Springer, 2014.
[12] Mohammad Karim Sohrabi and Vahid Ghods. Materialized view selection for a data warehouse using frequent itemset mining. *Jcp*, 11(2):140–148, 2016.
[13] James Wagner, Alexander Rasin, Boris Glavic, Karen Heart, Jacob Furst, Lucas Bressan, and Jonathan Grier. Carving database storage to detect and trace security breaches. In *Proceedings of the Seventeenth Annual DFRWS USA*, pages s127–s136. Elsevier, 2017.
[14] James Wagner, Alexander Rasin, Tanu Malik, Karen Heart, Jacob D. Furst, and Jonathan Grier. Detecting database file tampering through page carving. In *2018 21st International Conference on Extending Database Technology*, pages 121–132, 2018.
[15] Shangping Wang, Yinglong Zhang, and Yaling Zhang. A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems. *IEEE Access*, 6:38437–38450, 2018.