Smart-Views: Decentralized OLAP View Management using Blockchains

Kostas Messanakis[1], Petros Demetrakopoulos[1], and Yannis Kotidis[1]

Athens University of Economics and Business
kostas.messanakis@gmail.com, petrosdem@gmail.com, kotidis@aueb.gr

Abstract. In this work we explore the use of a blockchain as an immutable ledger for storing historical facts in a decentralized Data Warehouse deployment. We also exploit the ledger for storing definitions of aggregate data cube computations over these data, in the form of smart contracts. These smart contracts implement smart views that encapsulate frequently requested aggregations. We propose a novel modular architecture in which computations defined by the smart contacts are passed to suitable data processing engines. On each node, a smart view cache is also utilized, in the form of an in-memory database, so that frequently requested aggregates can be delivered with small latency. Our techniques model and take into consideration existing interdependencies between the smart views to expedite their computation and maintenance. We also propose efficient algorithms for managing the content of the smart view cache. Our experiments demonstrate that the combination of the cache and the post-processing offered by the data engine reduce by orders of magnitude the overhead of reading data from the ledger, resulting in fast delivery of results to the user.

1 Introduction

The emergence of cloud storage in the past decade, has familiarized businesses with the benefits of outsourcing their installations. Recently, blockchain technology has been described as a potentially disrupting innovation that can utilize decentralized storage, in the form of one or multiple cryptographically secured ledgers, for managing large and sensitive data. Much like in a Data Warehouse, historical data becomes immutable, when placed in a blockchain, because a correct copy is verified and subsequently stored at multiple locations. Still, there are important differences between the two technologies. The Data Warehouse goes well beyond storing raw data, as its main functionality is to provide fast access to multiple interesting aggregations on user-defined dimensions of interest [6]. Blockchains are mainly suited to publishing transactions and later prove that those transactions were published. Many blockchain infrastructures utilize some form of "smart contracts", as pieces of code that are embedded in the ledger implementing binding agreements between parties that can, for instance, auto-execute when certain conditions are met. Building on smart contracts, in this work we envision a decentralized Data Warehouse implemented on top of

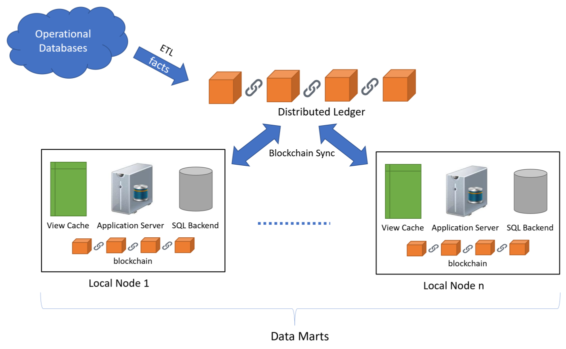


Fig. 1. Using Smart Views for building a decentralized Data Warehouse

a blockchain, where smart contracts are used as the means to describe data processing over the collected multidimensional data. In essence, these contracts define aggregate views over the data and we refer to them as smart views. Smart views can trigger computations enabling complex data analysis workflows.

Following the smart views paradigm, organizations can implement a decentralized data warehouse using the blockchain ledger for publishing its raw data (fact records) as is depicted in Figure 1. The smart view logic (that will be described in the next sections) feeds from this data and implements, on local nodes, smaller Data Marts [9] that are used for specific analytical purposes. For example, local node 1 in the figure can be utilized by data scientists working on sales data, while local node n by executives analyzing inventory data. Both user groups will need access to the distributed data warehouse repository. Each user group may have its own analytical queries focusing on specific aspects of the data. Their needs will be accommodated by their local node that (i) syncs data updates from the shared data warehouse ledger, (ii) manages local user inquires via the smart views API we provide and (iii) caches local results so that subsequent queries are expedited. Queries that are common among the groups will be shared at a semantic level, as their descriptions will be also stored in the distributed ledger.

Unlike recent proposals that aim to elevate blockchains into a data processing framework [1, 4], in this work we use these ledgers for their indented purpose; that is for storing in a decentralized manner raw transactions and snippets of code (in the form of smart contracts) that define requested aggregations over them. Smart views, much like traditional views in a database system, are virtual up to the moment when they are instantiated by a user request. On each node, we utilize a data processing engine (that can be a relational database or a big data platform) for doing the heavy lifting of computing their content. Additionally, an in-memory database acts as a cache, providing fast access to their derived data. This cache is used to serve multiple user requests on a node, orders of magnitude faster than what a blockchain could provide. In addition to engineering a solution that scales blockchains for processing Data Warehouse workloads, we also propose and compare algorithms that can manage effectively the smart views content cached in the in-memory database.

2 Modular Node Architecture

In this section we describe the main components that are instantiated in each local node (Figure 1).

Distributed Ledger: This component utilizes blockchain technology to implement a permanent, indelible, and unalterable history of raw observations (facts). The blockchain ledger substitutes for what is commonly referred to as the "fact table" in a traditional Data Warehouse [9]. The inherent immutability provided by the blockchain provides data integrity, as each fact is timestamped and embedded into a "block" that is cryptographically secured by a hashing scheme that links to and incorporates the hash of the previous block. The ledger is also used to (i) store smart contracts that define and implement analytical operations over the raw data, in the form of smart views and (ii) encode proofs (in the form of hash codes) for materialized results of the aforementioned smart views that are maintained in the View Cache (discussed next). In our prototype implementation, we used the Ethereum Blockchain[1] for implementing the blockchain and the smart code functionality.

View Cache: This is a main-memory data store that preserves (locally at a node) recently computed results of smart views, triggered by the execution of the corresponding smart contract in the Application Server. The purpose of this component is to leverage the speed of modern in-memory databases in order to avoid costly look-ups on the blockchain during both query answering and when performing view updates. The integrity of the cache is ensured by the immutable smart contract code in the blockchain. Furthermore, the data are cryptographically signed and the resulting hash codes are stored in the distributed ledger. In our implementation this component is implemented as a Redis[2] data store.

SQL Backend: This can be any database backend offering SQL capabilities. Its purpose is to offload OLAP calculations defined in the smart contracts from the application server to a more suitable engine. This database is used for computing and updating the content of the smart views by consolidating data records extracted from the blockchain and, whenever available, from the View Cache. The results of these computations are passed back to the smart contract caller (via the Application Server) and, subsequently, are cryptographically signed and stored in the local View Cache for further reference. Thus, the backend is primarily used as an SQL computation engine. In our implementation, we use a MySQL relational database, however this component can be easily swapped, depending on the application needs, with more resource-friendly solutions (such as SQLite), or, for more demanding implementations with big data frameworks such as Apache Hive and Apache Spark that provide SQL APIs.

[1] https://ethereum.org/
[2] https://redis.io/

Application Server: This component orchestrates the whole process of defining, storing, reusing and updating the smart views. Smart views are defined globally via smart contracts and contain aggregate calculations over the raw data (facts) stored in the blockchain. Moreover, at the time of their definition, smart views are linked together by means of metadata descriptors forming a data cube lattice structure that captures interdependencies among them [7, 11]. These interdependencies are utilized by the Application Server in order to seek an efficient plan for computing the results of a smart view, when the corresponding smart contract API is called. This mechanism that will be described in more details in the forthcoming sections, utilizes cached results either from the same view or, from more detailed smart views that have been recently computed and their results are available in the local View Cache. The goal of these optimizations is to reduce the number of records that the application server retrieves from the blockchain, as this operation is typically orders of magnitude slower that reading from the in-memory View Cache. The corresponding calculations are passed to the SQL backend so that the smart contracts running in the application server do not need to perform data-heavy computations.

Our architecture is compromised of individual subsystems that are coordinated in their operation by the application server. Each of these subsystems operates according to its own specifications, as we do not assume some application specific or cross-layer functionality. As an example, the ledger can be easily replaced with alternative blockchains [2], other than Ethereum, with distributed blockchain-enabled databases such as [3, 8, 15, 16], or even with an outsourced verifiable database [5, 20]. The same is also true for the View Cache and the SQL Backend subsystems. This is because, only the schema of smart views is stored in the ledger and, thus, is shared across all participating nodes. We are not trying to synchronize their instances across the network as this would incur extreme overhead because of their size. Smart views contain projections of an OLAP Data Cube [6] and their combined size can be orders of magnitude larger than the raw data stored in the ledger [17]. Thus, each node maintains its local, independent copy in the cache for serving the requests it receives and there is no communication overhead among the nodes for managing their caches.

3 Smart Views

3.1 Preliminaries and Definitions

Smart views provide access to consolidated information from the detailed data (fact records) stored in the blockchain. At a semantic level, smart views are inspired from aggregate OLAP views [10]. However, their implementation differs in that their content is computed both from data available in the blockchain and the View Cache. OLAP promotes interactive access to vast collections of records available in a Data Warehouse via the use of a carefully selected set of "dimensions" of interest. These dimensions are derived from attributes available in the raw records and enable analysts to render their data in exponentially different points of view. For example, in a sales data warehouse, customer, product and store-location may be the dimensions of interest. Additionally, one or sometimes multiple hierarchies may be defined over each dimension. For example products may be further categorized by their type, function, or maker.

The Data Cube was introduced in [6] in order to formalize all possible aggregations in a Data Warehouse. An aggregate OLAP view contains a subset of the records in the Data Cube depending on the selected dimensions and additional filters present in the view. Given a selection of aggregate views that are of interest to the analyst, the Data Cube framework permits us to denote their interdependencies, in a manner that guarantees correct summarisability during roll-up or drill-down operations [11].

In our exposition, in order to simplify the notation used, we will assume that the data warehouse implements a set of dimensions $D = \{d_1, d_2, \ldots d_n\}$ and a single measure m. The extension to examples with hierarchies and multiple measures is straightforward and omitted from our running examples. The equivalent of a Data Warehouse centralized fact table is, in our decentralized framework, the blockchain, where all detailed records are stored. Thus, we can think of the blockchain data as the equivalent of a fact table with schema $DW(d_1, d_2, \ldots d_n, m)$. This basic schema can be easily extended to cover more complex application scenarios including fact constellations [9].

A smart view V is defined by projecting the fact table records on a subset $D_V \subseteq D$ of the available dimensions and computing a function F_V on their measure values. For this discussion, we assume that F_V is one of the commonly used distributive or algebraic functions such as $MAX()$,$MIN()$, $SUM()$, $COUNT()$, $AVG()$, $STDEV()$, $top_k()$, etc. Optionally, the view may contain a set of conjunctive predicates P_V on the dimensions in set $D - D_V$ that can be used, for instance, in order to perform a *slice* operation [6]. We restrict our exposition to conjunctions of simple atomic predicates of the form 'd_i OP $scalar$', where $d_i \in D$, $OP \in \{<, >, =, \leq, \geq, \neq\}$ and scalar∈domain(d_i). Thus, a smart view can be described as a triplet $V = \{D_V, F_V, P_V\}$.

As an example, assume a Data Warehouse schema with three dimensions $D = \{customer, product, store\}$ and one measure *amount*. A smart view may compute the total sales per customer and product for a specific store S. Then, $D_V = \{customer, product\}$, F_V=$SUM()$ and P_V=$\{'store = S'\}$. An equivalent SQL query for the same view can be stated as "SELECT customer, product, SUM(amount) FROM DW WHERE store=S GROUP BY customer, product", assuming relation DW stores the fact table records. We refer to this expression as the *view query* of smart view V.

Given two views $V_i = \{D_{V_i}, F_{V_i}, P_{V_i}\}$ and $V_j = \{D_{V_j}, F_{V_j}, P_{V_j}\}$, we say that view V_i is more detailed than view V_j ($V_j \preceq V_i$) iff the following conditions hold

1. All dimensions in view V_j are also present in view V_i, i.e. $D_{V_j} \subseteq D_{V_i}$.
2. F_i and F_j are the same aggregation function.
3. Database query $Q_j(P_{V_j})$ is contained in query $Q_i(P_{V_i})$, for every state of the Data Warehouse fact table DW. Query $Q_i(P_{V_i})$ is expressed in relational algebra as: $\pi_{D_{V_i}}(\sigma_{P_{V_i}}(DW))$, and similarly for $Q_j(P_{V_j})$.

As an example, for $V_1 = (\{customer\}, SUM(), \{'store = S'\})$ and $V_2 = (\{customer, product\}, SUM(), \emptyset)$, it is easy to verify that $V_1 \preceq V_2$. This relationship essentially permits us to compute the result of smart view V_1 from a previously computed result of V_2 [11]. This relationship will be key to our smart view materialization and update policies.

3.2 On-demand Materialization of Smart Views

In our framework, a smart view is defined via a smart contract stored in the blockchain. The smart view remains virtual, up to the time when its content is first requested by a user, by calling a *materialize*() API function published by the contract. For each smart view V, we maintain a list of more detailed views V_i ($V \preceq V_i$) also defined for the same data warehouse schema via the smart contract. By definition, this list also contains view V. A view V_i from this list, whose result $cached(V_i)$ is stored in the View Cache can be used for computing the instance of V, by rewriting the view query to use this result, instead of the fact table DW. Nevertheless, we should also provision for fact table records that have been appended to the blockchain after the computation of the cached instance of V_i. Let t_i be the timestamp of the computation of the cached result of V_i (this timestamp is stored by the smart contract in the blockchain). Let $deltas(t_i)$ denote the records that have appeared in the blockchain after timestamp t_i. In order to be able to retrieve these records efficiently, each fact table record also maintains a timestamp of its creation.

After retrieving the cached, stale copy of view V_i and the deltas, the contents of the view are computed by executing the view query over the union of the data in $cached(V_i)$ and $deltas(t_i)$. This functionality is provided by the SQL backend. The result of the smart view is returned to the caller of function *materialize*() and is also sent to the View Cache.

3.3 Cost-based Smart View Materialization

The process described in the previous subsection may be used to implement function *materialize*() considering any available smart view V_i: $V \preceq V_i$. Depending on the definition of view V_i, and the number of results in set $deltas(V_i)$ the execution times may differ substantially. Let $size_{cached(i)}$ denote the size of the cached result of view V_i in the View Cache and $size_{deltas(i)}$ the size of the deltas, respectively. Running the smart view query over the union of the cached results and the deltas by the SQL backend involves the execution of an aggregate SQL statement over these data. Such aggregations are typically computed either by hashing or sorting [14]. Both require linear I/Os over their input (e.g. two-phase sort). Thus, we can estimate the cost at the backend as $w_{sql} \times (size_{cached(i)} + size_{deltas(i)})$, for some weight parameter w_{sql} that reflects the speed of the SQL backend.

Retrieval of records in $deltas(t_i)$ is performed by the application server that is also running a blockchain node. We estimate this cost as $w_{ledger} \times size_{deltas(i)}$.

In total, the cost of using smart view V_i in order to materialize view V is estimated as

$$cost(V_i, V) = w_{sql} \times (size_{cached(i)} + size_{deltas(i)}) + w_{ledger} \times size_{deltas(i)}$$

or, equivalently

$$cost(V_i, V) = w_{sql} \times [(1 + \frac{w_{ledger}}{w_{sql}}) \times size_{deltas(i)} + size_{cached(i)}]$$

Constants w_{sql}, w_{ledger} denote the cost of post-aggregating cached results with delta records and retrieving data from the blockchain, respectively. In most implementations we expect $w_{ledger} >> w_{sql}$. Thus, for the purpose of ranking the views V_i and selecting the top candidate for materializing view V the cost formula can be simplified as

$$cost(V_i, V) = \alpha \times size_{deltas(i)} + size_{cached(i)} \tag{1}$$

for some constant $\alpha >> 1$. The formula suggests that stale views in the cache are less likely to be used for future materializations of smart views. This is because, as new fact table records are stored in the ledger, the cost of reusing a stale view increases, as $size_{deltas(i)}$ is increased.

3.4 View Cache Maintenance

As new smart views are materialized and stored in the cache, we need to provision for the case that the View Cache exceeds the available storage space. Recall that in our implementation the cache is maintained by an in-memory database. A straightforward approach that uses standard policies such as LRU or LFU for evicting results in case of a full cache would be very inefficient because the smart views have different sizes and recomputation costs. We also need to take into consideration the interdependencies between them. These interdependencies are captured by Equation 1 that depicts the cost of materializing view V from another view V_i in the cache.

Let $\mathcal{V}$ denote the set of views that have been requested earlier by the application. This set is easy to maintain in the Application Server. We will use this set in order to estimate the cost of evicting a cached result. For each view $V \in \mathcal{V}$ we also maintain the frequency f_V of calling function $materialize(V)$. For the calculations that follow, we first remove from set $\mathcal{V}$ those smart views that can not be computed using results in the View Cache. These views do not affect the calculations we perform for updating the cache.

Cost-based View Eviction Given that multiple views from the cache may be used for materializing the result of another view V, we need to quantify the benefits of keeping a result in the View Cache. Let $costMat(V)$ denote the minimum cost of materializing view V and $\mathcal{VC}$ denote the set of all views presently stored in the View Cache. Then,

$$costMat(V, \mathcal{VC}) = min[\min_{V_i \in \mathcal{VC}: V \preceq V_i} cost(V_i, V), costBC)]$$

where *costBC* is the cost of computing V directly from the blockchain data. This cost is the same for all views based on the cost model described in the previous subsection, as it implies fetching all records from the ledger. The *displacement cost* $dispCost(V_i)$ of cached result V_i is defined as the cumulative increase in the materialization cost of all smart views $V \preceq V_i$, in case view V_i is evicted. Thus,

$$dispCost(V_i) = \sum_{V \in \mathcal{V}: V \preceq V_i} f_V \times (costMat(V, \mathcal{VC}) - costMat(V, \mathcal{VC} - V_i))$$

For all views, we amortize their displacement costs by dividing this number with their size. In this way, we take into consideration the potentially large differences between the space that these results occupy in the cache. Thus, we estimate the amortized (per-space) displacement cost of result V_i as

$$amortizedDispCost(V_i) = \frac{dispCost(V_i)}{size_{cached(i)}}$$

Our proposed cost-based view eviction policy (COST) lists the views in the cache in increasing order of their amortized displacement cost. The eviction process then discards views from the cache by scanning this list until enough space is generated for storing the new result.

Eviction based on Cube Distance In order to capture the temporal locality exhibited between successive calls to materialize views in the Data Cube lattice, we propose an alternative eviction policy (DIST) based on the data cube distance metric. The lattice contains an edge (V_i, V_j) iff $V_i \preceq V_j$ and $|D(V_j)| = |D(V_i)|+1$. The Data Cube distance (*distCube*) in the lattice between two views is defined as the length of the shortest path between the views. As an example, $distCube(V_{product,store}, V_{product,customer})$=2. For a newly computed view V, we order the views in the cache in decreasing order based on their distance from V and discard views until enough space is generated as a result of these evictions.

4 Experiments

In this section we provide experimental results from a proof-of-concept implementation of our proposed system. Ganache was used for setting up a personal Ethereum Blockchain and implementing the Smart Views API using Solidity contracts. We used MySQL as a relational backend and Redis for implementing and View Cache. The application server functionality was written in Node JS. Within the code we made several optimizations so that heavy data processing operations are delegated to the SQL database for efficiency and scalability. All experiments were contacted in a Intel Core i7-5500U 2.4Ghz laptop with 8GB memory and a 640GB HDD.

We used synthetically generated data using d=5 dimensions with cardinality 1,000 integer values each. Each fact table record contained 5 keys (one for each dimension) and one randomly generated real value for the measure. In order

Operation	Time (sec)
Blockchain Reads	86178.5
View Cache Reads	8.1
SQL Backend Time	354.2
Application Server Time	972.5

Table 1. Query Execution Times for W_{mix} (1000 queries)

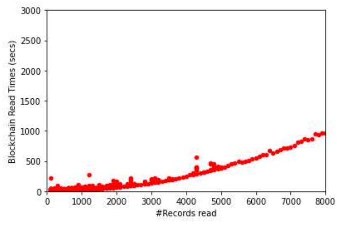
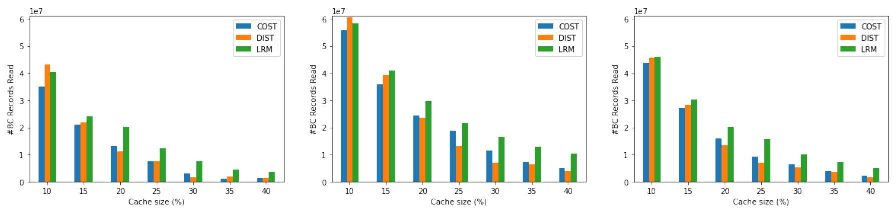


Fig. 2. Blockchain Read Times

to evaluate the performance of the View Cache we generated three workloads consisting of 1,000 queries (views) each. In the first, denoted as W_{olap}, we started from a random view in the lattice and each subsequent query was a roll-up or drill-down from the previous one. The second workload W_{random} contained randomly selected views from the lattice (with no temporal locality). For the last workload, W_{mix}, a subsequent query was a roll-up or a drill-down from the previous query or was a new randomly generated one. In order to test the View Cache we implemented three eviction policies. The first two, cost based (denoted as COST in the graphs), and distance-based (denoted as DIST in the graphs) are discussed in subsection 3.4. We also implemented Least Recently Modified (LRM), a simple policy that selects as a victim the cached view that that has not been modified for the longest time in the cache. The intuition is that older results in the cache require fetching more deltas from the blockchain (Equation 1), when used for materializing a new smart view request.

The intuition behind the architecture discussed in Section 2 is that we can overcome the shortcomings of the blockchain technology by using (i) an SQL processing engine to off-load heavy computations and (ii) an in-memory database to provide fast access to frequently computed aggregations. In the fist experiment we put our intuition into test using a small dataset consisting of 10,000 facts and progressively add transactions in batches of 100 records for 1000 epochs. Thus, the final blockchain contained 110,000 records. In each epoch we execute one query from workload W_{mix}. The size of the cache for this experiment was set to 40% of the cumulative size of all views in W_{mix}. The eviction policy was COST.

In Table 1 we report statistics of the time spent for processing the query workload. Retrieving data from the ledger amounts for most (98.5%) of the time spent for serving the user with the results of the requested view. In comparison, the overhead of the application server that includes communication with the different components, execution of smart contracts, collection and formatting of the results into a json file served to the caller of function *materialize*() amounts for 1.1% of the time. SQL processing, which includes sending data and queries to the MySQL server and retrieving the results, amounts for 0.4% of the time spent on average for each query. Finally, the time-wait for fetching results from the View Cache is practically negligible, as a result of using an in-memory database for



(a) W_{olap} (b) W_{random} (c) W_{mix}

Fig. 3. Number of records fetched from the blockchain.

this task. Figure 2 plots the blockchain read times (in secs) for different queries in W_{mix} over the number of records read ($deltas(i)$). Clearly, there is a strong correlation between the blockchain read times and the size of the delta records fetched for each query, as was discussed in subsection 3.3. These results verify our assumption that, by far, the largest overhead in materializing smart views is attributed to fetching records from the blockchain. Based on these results, in what follows we move on to larger datasets and concentrate on the performance of the different cache eviction policies in reducing the number of records read from the blockchain.

For the next experiment we used a larger dataset that initially contained 100,000 records. We run the experiment for 1,000 epochs. During each epoch we inserted 100 new records and executed a query from the respective workload. Thus, at end of the run, the blockchain stored 200,000 fact table records. In Figure 3 we plot the number of blockchain records read for the three different eviction policies and the three workloads used, as we varied the View Cache size from 10% up to 40% of each workload result size in increments of 5%. As expected, the W_{olap} workload resulted is much fewer reads from the blockchain, as successive queries target nearby views in the Data Cube lattice due to the roll-up and drill-down operations. On the other hand, W_{random} contains queries with no temporal locality, resulting in fewer hits in the cache. Nevertheless, the benefits of the cache are significant, even for this random workload. Compared to not using the cache, even the smaller cache size (10%) with the COST policy reduces the total number of records fetched from the blockchain from 150,050,000 down to 55,631,700, i.e. a reduction of 63%. The largest cache tested (40%) reduced this number by almost 97%. The results for workload W_{mix} are in between, as expected. Comparing the three eviction policies, we observe that the COST policy works better in constrained settings (smaller cache sizes), as it avoids flashing from the cache, smart view results that help materialize many other views in the workload. The DIST policy is often better for intermediate cache sizes, while LRM performs worst, as it does not take into consideration the interdependencies between the cached view results.

5 Related Work

Existing studies [3] show that blockchain systems are not well suited for large scale data processing workloads. This has been our motivation for decoupling in the proposed architecture the definition and properties of the smart views (stored in the ledger) from their processing and maintenance that are handled by appropriate data management modules. Our architecture differs from existing solutions that either add a database layer with querying capabilities on top of a blockchain [1, 4] or, extend databases and, in some cases NoSQL, systems with blockchain support [8, 15, 16]. A nice overview of these systems is provided in [13]. There are also proposals that aim to build verifiable database schemas [5, 20] that can be shared among mutually untrusted parties. These proposals retain the familiar SQL interface for accessing shared data, but this is achieved over a verification layer that utilizes blockchain primitives.

QLDB [1] from amazon is a ledger database with SQL capabilities. Since its primary purpose is to store a transaction log, it can be exploited in our architecture for storing the data warehouse fact records in an append only manner. However, unlike the Ethereum blockchain we utilize in our implementation, OLDB lacks decentralization support. BlockchainDB [4] utilizes blockchains as the native storage layer and implements an additional database layer with partitioning and sharding capabilities on top of it via the use of shared tables. Shared tables are database storage abstractions that differ from smart views is that they provide the means to define a database schema over the blockchain and are suitable of OLTP-type of transactions, while smart views provide aggregated information over the data warehouse records. Still, our work can benefit from the work of [4] by using shared tables for defining and manipulating the raw fact table records via a database-like interface.

Our work on smart views is motivated by previous work on view selection in Data Warehouses [10–12, 18] that demonstrated how materialized views can significantly enhance the performance of OLAP workloads. However, while past work has considered using materialized views within the context of a standalone system, in the proposed architecture, these views are defined and deployed in a decentralized manner via smart contracts. Unlike traditional Data Warehouses, the raw data required for deploying the smart views is stored in the ledger, while processing and management of their derived aggregate results happens off-the-chain. As a result, in this work we proposed a new cost model, tailored for our modular architecture and implemented new cache policies based on this model.

6 Conclusions

In this work we proposed a modular architecture and algorithms for defining, (re)using and maintaining materialized aggregate smart views in a decentralized manner using existing blockchain technology. Our experimental results, based on a proof-of-concept prototype we built have demonstrated the effectiveness of smart views in reducing significantly the overhead of fetching stored data from the distributed ledger. For the future, we plan (i) on benchmarking our architecture using a larger scale deployment and, (ii) considering extensions of smart views to support more complex OLAP analytics [19].

References

1. AWS. Amazon Quantum Ledger Database (QLDB).
2. M. Dabbagh, K. R. Choo, A. Beheshti, M. Tahir, and N. S. Safa. A survey of empirical performance evaluation of permissioned blockchain platforms: Challenges and opportunities. *Comput. Secur.*, 100:102078, 2021.
3. T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K. Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *SIGMOD*. ACM, 2017.
4. M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb - A shared database on blockchains. *VLDB*, 2019.
5. 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, Asilomar, CA, USA, January 2019*.
6. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *ICDE*, pages 152–159, 1996.
7. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD Conference*, pages 205–216, 1996.
8. S. Helmer, M. Roggia, N. E. Ioini, and C. Pahl. Ethernitydb - integrating database functionality into a blockchain. In *New Trends in Databases and Information Systems - ADBIS Short Papers and Workshops*, 2018.
9. W. H. Inmon. *Building the Data Warehouse.* QED Information Sciences, Inc., Wellesley, MA, USA, 1992.
10. Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *SIGMOD*, 1998.
11. Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *SIGMOD*, pages 371–382, 1999.
12. Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *ACM Trans. Database Syst.*, 26(4):388–423, 2001.
13. M. Raikwar, D. Gligoroski, and G. Velinov. Trends in development of databases and blockchain. In *SDS*, pages 177–182. IEEE, 2020.
14. R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.
15. M. S. Sahoo and P. K. Baruah. Hbasechaindb - A scalable blockchain framework on hadoop ecosystem. In *Supercomputing Frontiers, Singapore, March 26-29*, 2018.
16. F. M. Schuhknecht, A. Sharma, J. Dittrich, and D. Agrawal. chainifydb: How to get rid of your blockchain and use your DBMS instead. In *CIDR*, 2021.
17. Y. Sismanis and N. Roussopoulos. The complexity of fully materialized coalesced cubes. In *VLDB*, pages 540–551. Morgan Kaufmann, 2004.
18. D. Theodoratos, S. Ligoudistianos, and T. K. Sellis. View selection for designing the global data warehouse. *Data Knowl. Eng.*, 39(3):219–240, 2001.
19. P. Vassiliadis, P. Marcel, and S. Rizzi. Beyond roll-up's and drill-down's: An intentional analytics model to reinvent OLAP. *Inf. Syst.*, 85:68–91, 2019.
20. M. Zhang, Z. Xie, C. Yue, and Z. Zhong. Spitz: A verifiable database system. *Proc. VLDB Endow.*, 13(12):3449–3460, 2020.