Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SI: BEST PAPERS OF ICBC 2023

DELTA: A Modular, Transparent and Efficient Synchronization of DLTs and Databases

F. Javier Fernández-Bravo Peñuela[1] | Jordi Arjona Aroca[1] | Francesc D. Muñoz-Escoí[2] | Yuriy Yatsyk Gavrylyak[1] | Ismael Illán García[1] | José M. Bernabéu-Aubán[2]

[1]Distributed Systems, Instituto Tecnológico de Informática, València, Spain

[2]Departament of Computer Systems and Computation, Universitat Politècnica de València, València, Spain

Correspondence
Corresponding author F. Javier Fernández-Bravo.
Email: fjfernandez@iti.es

Funding Information
This research was supported by the i4Q project which has received funding from the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement No. 958205.

Abstract

Besides cryptocurrencies, DLTs may be also exploited in enterprise systems operated by a consortium of organizations. Their interaction takes usually place on a permissioned blockchain network that holds a set of data to be queried frequently. In this scope, the main problem of DLTs is their unsuitability for a fast service of complex queries on those data. In order to solve this issue, many proposals dump the ledger contents onto databases that, because of their own goals and design, are already optimized for the execution of those queries. Unfortunately, many of those proposals assume that the data to be queried consist in only a block or (cryptocurrency-related) transaction history. However, those organization consortiums commonly store other structured business-related information in the DLT, and there is an evident lack of support for querying that other kind of structured data.

To remedy those problems, DELTA synchronizes, with minimal overhead, the DLT state into a database, providing: (1) a modular architecture with event-based handling of DLT updates that supports different DLTs and databases, (2) a transparent management, since DLT end users do not need to learn or use any new API in order to handle that synchronization (i.e., those users still rely on the original interface provided by their chosen DLT), (3) the efficient execution of complex queries on those structured data. Thus, DELTA reduces query times up to five orders of magnitude, depending on the DLT and the database, compared to queries directed to the ledger nodes.

KEYWORDS
Blockchain, DLT, ledger, database

1 | INTRODUCTION

During the last decade, Distributed Ledger Technologies (e.g., Blockchain) have been regarded as disruptive technologies, mainly owing its popularity to its usage in cryptocurrencies such as Bitcoin[1] and Ethereum[2]. This disruptiveness generated, altogether, an enormous hype around the technology, enthusiasm about applying it to a variety of use cases and, inevitably, misconceptions and inappropriate applications. One of these inappropriate uses was employing blockchains as a storage technology and trying to exploit them as databases[3].

DLTs have several well known advantages, such as immutability, non-repudiation, transparency, high availability or helping to avoid intermediaries, among others[4]. However, most DLT implementations also hold some disadvantages, one of the most prominent of them being their unsuitability to efficiently perform complex queries on the shared data state stored by the nodes composing the network. Although DLTs are not conceived as databases, this does not vanish the fact that sometimes retrieving information from them becomes necessary[5]. Even in platforms that allow to perform queries on the state, e.g., Hyperledger Fabric (HLF), assets are usually serialized and stored in key-value databases, which causes the efficient execution of these queries to not be feasible in case the data possess a complex structure. This is troublesome, given the increasingly common

adoption of the Blockchain technology in enterprise-grade applications[6], whose state usually has such complex structure and demands those efficient and elaborate queries.

This increasing usage raises a second issue, the inherent complexity of DLTs. The interaction with DLT platforms is usually cumbersome for developers, especially when compared to the operation with traditional databases. Thus, we believe there is a need to offer DLT users and developers a tool that simplifies the interaction with the DLT, abstracting them from the underlying technology while, at the same time, it provides mechanisms for the efficient execution of complex queries.

On this line we present DELTA: a solution for transferring the data of a DLT to a local database and keeping their contents synchronized. With DELTA, we intend to ease the execution of read operations on the DLT. Once transactions are committed on the DLT, they will be updated in the database, making the result of changes on the ledger's data to be readable. This way, we offer a friendly and powerful interface for the efficient execution of potentially complex read operations, which is no other than that of the database without losing the virtues of DLTs, which remain as an immutable ledger that can be audited at any moment. Our main contributions are:

- A driver-based solution capable of synchronizing DLTs into databases enabling the efficient execution of complex queries on the data stored in the DLT. This synchronization is based on a purposely designed event specification and, similarly, fault tolerance mechanisms have been implemented for resilience.
- Drivers for two DLTs, HLF and Quorum, and two databases, MongoDB and PostgreSQL. Note that Quorum is an Ethereum-compatible permissioned blockchain. Thus, what is mentioned in regard to Quorum is easily portable to Ethereum and other Ethereum-based blockchains. The current MongoDB and PostgreSQL drivers show that DELTA may easily handle document-oriented and relational databases. Those drivers may be taken as a guide to develop similar drivers for other open-source document-oriented and relational databases.
- Base smart contracts implementing event emission for write operations, for HLF and Quorum. These base smart contracts are the single piece needed for ensuring that each DLT update will generate a specific event to be considered in our synchronization management. This reduces the adaptation of existing smart contracts to inheriting from that base contract, applying a few minor modifications to the original contract code. Thus, DELTA achieves transparency for DLT end users, since all existing smart contracts (or new smart contracts to be implemented) will keep their original operations interface (i.e., that of a non-synchronized DLT).

The rest of the paper is organized as follows. Section 2 presents a study of the state of the art in the subject of DLT synchronization into databases, identifying relevant aspects left uncovered by previously existing solutions. Section 3 provides an overview on DELTA, while Section 4 conducts an analysis of DELTA's high-level design, and describes its most remarkable features. Section 5 compares two DLT implementations –HLF & Quorum– synchronized on two databases –MongoDB & PostgreSQL– in terms of throughput and performance. Section 6 discusses the premises and limitations of the solution provided and, finally, Section 7 provides some conclusions about the research work carried out, the software solution developed and future work lines.

2 | RELATED WORK

Many publications[7,8,9,10,11,12,13,14,15,16] and tools[17,18,19,20] interested in querying DLTs focus on obtaining information about their blocks or their transaction history. Most of them synchronize the DLT contents with databases. To achieve this, they either use a data model that may be completely dependant on the chosen DLT or may propose a more abstract and general design that intends to be compatible with the most common technologies (e.g., Bitcoin and Ethereum in public networks, or HLF in permissioned ones). However, those proposals are unlikely to pay further attention to the management of the state data held by the ledger, which they usually store in raw text format within a single field that is bound to the corresponding transaction. This makes it easier to execute queries on the transaction history, but stands in the way of analyzing the shared state itself. Some examples of this approach are ether_sql[18] (synchronization of an Ethereum network over a relational database) and the integration of Azure's blockchain service with its CosmosDB[17] database.

This approach makes sense in the Bitcoin network, whose transaction logic is limited to the transfer of currency units between accounts. There, we find tools such as bitcoin-abe[19] and blockchainSQL[20]. In addition, datasets that contain a dump of the transaction history of various cryptocurrencies have been stored in Google's Big Query database[21], leveraging its support for

collections as the data type of a single column as well as the computing capabilities available through Google's own cloud infrastructure.

However, in DLTs with Turing-complete smart contract execution engines, e.g., Ethereum and HLF, the structure of the state data stored in the ledger may be complex. An increasingly common application of these networks consists in their integration as enterprise-grade systems, operated by a consortium, keeping a shared state that is updated by means of the transactional execution of smart contracts into which their business processes are codified[6]. Here, the DLT network is a priceless audit mechanism providing immutability and non-repudiation guarantees to the data emitted in transactions[4]. Even so, during the normal operation of these systems, what is usually required is not to query the transaction history, but the efficient execution of queries over the stored state[22].

Proposals regarding the ability to launch efficient queries on the state data of a DLT can be split into two broad approaches. Within a first approach, we can find those proposals that build a distributed database assimilating the cryptographic mechanisms of a DLT, combining the security warranties provided by the DLT with the high efficiency to execute complex queries on the data stored within the database. This approach is followed by BigchainDB[23] (on top of MongoDB), Blockchain Relational Database[24] (on top of PostgreSQL) and ChainifyDB[25] (on top of PostgreSQL or MySQL). The second approach includes those proposals that dump a snapshot of the current state stored by the DLT network into a database through the connection to a node that hosts a replica of the ledger, keeping the database synchronized with the ledger state whenever new transactions are committed at the node. Proposals within this latter approach can themselves be classified attending to different aspects of their design.

(a) **The synchronization mechanisms.** A first step in the way to achieve DLT synchronization is to take a snapshot of (part of) its current contents on request. That approach might need a long time to collect such (possibly large) snapshot, but those read blocks –because of DLT immutability– do not need to be read again in subsequent requests that demand larger supersets of blocks. This synchronization mechanism needs to sequentially read a given range of DLT blocks, explicitly stated in the request, dumping them to an external database. That solution has been used in the system proposed by Shlomi Linoy *et al.*[13] where the read blocks are copied onto a Hadoop subsystem that facilitates parallel (indeed, MapReduce) querying. The main goal in that paper is to carefully analyze the transactions contained in a range of blocks of the Ethereum DLT. Although query time may be slightly worse than in the DLT the first time a given block is dumped, it becomes much shorter in subsequent queries once the requested blocks are in the Hadoop File System (HDFS).

Some other systems use a continuous synchronization approach. To this end, these systems, e.g. LedgerSync[26] and EtherQL[8], use the events emitted from the DLT node when committing a block of transactions as its single source of data for getting updates from the ledger state. This could become troublesome when missing an event's reception or in case the node already contains committed information that the database lacks at the moment of launching the tool and establishing the subscription.

Vent[27] and Ledgerdata Refiner[28] avoid that latter problem and, as a result, provide crash fault tolerance. They explicitly query for the last block committed, comparing it with the last block number stored in the database, retrieving those blocks that were not yet persisted. This approach corrects the deficiencies of the strategy outlined by LedgerSync. Ledgerdata Refiner relies only on periodic queries to every node every two seconds, while Vent mainly relies on the reception of events from the DLT, but it also includes mechanisms for detecting missing blocks, which will be retrieved straightaway from the node itself.

(b) **The complexity level in the structure of state data.** LedgerSync[26] (HL Sawtooth) and Vent[27] (HL Burrow) allow to synchronize the state of a peer node into a relational database, but they do not support nested data types in the definition of this state (stored as JSON objects by the DLT's internal key-value database), which implies that each document type would be stored into a different single table. This is an important limitation, since the DLT state is unlikely to stick to this restriction and, moreover, the true potential of relational databases regarding the modelling of the relationships between associated entity types is not being exploited.

Ledgerdata Refiner[28] (HLF) and ChainSQL[29,30] (Ripple) also synchronize the state data of a peer node into a relational database, allowing the structure of the state to be more complex than previous options and including nested elements. However, both of them display important disadvantages. Ledgerdata Refiner includes an interesting mechanism that performs an inference of the data schema, but, as the state is still stored as raw text, it makes the system inefficient for the execution of queries on it (even if they are allowed). ChainSQL's design proposal turns out to be too invasive, since every interaction with both the peer node and the database must be performed through their middleware's API. In addition, their smart contracts actively include information about how the state will be mapped into the relational database, which distorts the purpose of DLT transactions and would force to adapt all other systems that should interact with that DLT network.

(c) **The target database.** Relational databases are a common choice since they are by far the most established and feature efficient indexing mechanisms and query language (the SQL standard). Some systems that synchronize the state of a DLT into a relational database are Vent[27], Ledgerdata Refiner[28] and ChainSQL[29].

Although this approach makes it possible to provide efficient query interfaces, its main disadvantage is the fact that the state data structure must be known beforehand since it is needed to create the database tables. A particular exception is the case of Ledgerdata Refiner, but at the expense of having to parse the state in every single query (due to the aforementioned disadvantages in the state storage format), unless the query execution engine is combined with a text analysis tool.

Other proposals opt for document-oriented databases (observe these are not key-value databases such as LevelDB). BigchainDB[23] chooses MongoDB, while LedgerSync[26] chooses RethinkDB. This approach's main advantage is that it allows to deal with schema-free documents, which makes updating the database state from the DLT state to be straightforward, since there is no need to restructure the state data in the process. Its possible disadvantages are that queries could not be as efficient as in relational databases (when they involve some of the document's inner fields instead of the indexing keys). In addition, they do not provide a common language as SQL, thus forcing the user to commit to the specific query language of each database provider.

An interesting third alternative consists in mixing the two previous approaches; i.e., to be able to synchronize the DLT contents onto relational or document-oriented databases. That has been a goal first achieved in the library proposed by Massimo Bartoletti *et al.*[7], where MongoDB and MySQL have been used as the document-oriented and relational choices, respectively. Thus, customers may choose the database model to use in the resulting system, depending on their requirements.

As we may see, relational and document-oriented databases are the preferred target database alternatives. The performance evaluation presented in Bartoletti *et al.*[7] shows that queries and insert operations are usually faster in document-oriented databases than in the relational ones. We have also discussed that the relational model may introduce problems when the target database schema should be designed, since it is static and should be specified beforehand in the relational model, while document-oriented databases are schema-free. So, why do we need relational databases? The answer to that question heavily depends on the expected reliability and on the kind of queries to be applied onto the synchronized data. There are several important differences between both types of databases in this regard:

- Relational databases support ACID[31] transactions, while document-oriented databases do not (although MongoDB[32] may provide transactions that ensure the atomicity of a sequence of CRUD operations, it still misses the standard serializable isolation level guarantees of relational transactions). With ACID transactions we may choose the isolation level[33] to be used in each transaction and this rules how inter-transaction conflicts will be managed[34,35]. Relaxed isolation levels improve performance, but may introduce some consistency anomalies. On the other hand, strong isolation levels introduce performance penalties, but they avoid every possible consistency anomaly. Therefore, programmers may choose the adequate balance between consistency (or reliability) and performance in the relational model, while they are compelled to sacrifice consistency in the document-oriented model.
- Document-oriented databases denormalize the stored data, while relational databases use a normalized schema. Because of this, the former store in each document (i.e. a record or row in relational terms) all the information being needed in order to answer a given query and may provide as many collections (i.e. tables in the relational model) as kinds of queries to be managed. This may introduce a moderate to high degree of data redundancy that may compromise consistency and eliminates the need of managing specific inter-table join operations. Join operations are still possible in the relational model.
- Usually, "group by" clauses cannot be used in query languages for document-oriented databases, while they are supported in SQL (i.e., in relational databases). This fact, combined with "join"-operation management make SQL able to express complex queries that cannot be natively managed in document-oriented databases. However, that improved expressiveness does not come for free, since it may introduce performance penalties.
- In spite of what was found in the evaluation carried out by Bartoletti *et al.*[7], document-oriented databases are not always faster than relational ones. Indeed, Makris *et al.*[36] show that in the geospatial research area, where the data to be returned in a query usually belongs to a single table, relational databases (concretely, PostgreSQL) behave much better than document-oriented ones (concretely, MongoDB), being almost twice faster in 10 out of 11 types of query. On the other hand, Jung *et al.*[37] studied in 2015 a set of data that demands three tables in the relational model but only one collection in the document-oriented one. That scenario is one of the worst for the relational model, and even in that case query times were similar in both database types, but relational insertions and deletions required much longer intervals than in the

TABLE 1 Main interfaces required to implement a driver for a DLT or a database.

DLT Driver interfaces		**Database Driver interfaces**	
`IDeltaClient`		`IDeltaDatabaseConnector`	
query	subscribeToBlockEvents	connect	createAsset
invokeSync	subscribeToCodeEvents	disconnect	updateAsset
invokeAsync	disconnectEventHub	storeBlock	upsertAsset
replayBlockEvents	retrieveAssetSchema	addTransactionsToBlock	deleteAsset
replayCodeEvents	instantiateSmartContract	getBlock	retrieveAsset
detectNewSmartContracts		getBlockFromTransaction	retrieveAllAssets
Other DLT interfaces		getLastBlock	getLastTransaction
SmartContract	BlockEvent	getAllBlocks	getAllTransactions
ContractEvent	Transaction	executeQuery	querySynchronizedSmartContracts

document-oriented data store. Because of this, in regard to query efficiency, users should choose one database type or the other depending on the database schema complexity and the types of query to be used on stored data.

At a glance, there is a clear shortcoming in the support of complex data states, as existing solutions directed to address this issue propose either a potentially inefficient approach for query execution[28] or a design that complicates the integration with other systems[29]. In an ideal world, new systems that synchronize and manage complex DLT data states should do so without altering the native DLT behavior, i.e., they should be fully transparent for regular DLT users, and should support data synchronization onto different kinds of target databases, making possible the election of the most appropriate one for each scenario, thus providing a comfortable API and functionality for data readers. Subsequent sections describe how DELTA fills those gaps.

3 | OVERVIEW

DELTA's purpose is to facilitate the interaction with DLTs for developers and users. This is achieved through the development of a software solution that enables the easy execution of read operations on a distributed ledger, supporting efficient queries on data states with a complex structure. In addition, this solution includes mechanisms acting as Simplified Payment Verification (SPV), which makes it feasible to match the data assets in the database to their related DLT transactions, thus verifying their authenticity.

DELTA specifies an implementation model that allows to transparently plug in different DLTs and databases in an interchangeable fashion. This is achieved by generalizing and abstracting the specification of the operations that DLTs and databases need to provide. Integrating a particular DLT (or database) will then require the development of a driver implementing the operations of this abstract specification. Some of the most relevant interfaces and methods to implement when designing a DLT or Database driver are listed in Table 1, with special emphasis in `IDeltaClient` and `IDeltaDatabaseConnector`.

The adoption of a component-oriented approach, generating a modular design depicted in Figure 1, enables the encapsulation of algorithms and common behavior patterns into well defined components, contributing to code reuse, ease of debugging and maintenance. These components are independent of the underlying technologies, operating on top of the abstract operation interfaces already mentioned, which are implemented by technology-specific drivers. Examples of components are a block explorer (which checks the network progress in terms of blocks and transactions, and either logs out this information or inserts it into a database), a smart contract resolver (notifies when a previously unidentified smart contract is detected to be submitting transactions to the DLT), and a code event subscriber (subscribes to events emitted from a smart contract and performs an action upon their reception). Applications are conceived as an assembly of high-level components (which can, in turn, integrate other lower-level components) and act as selector wrappers that link the components instantiated to the chosen technologies. An example of this would be an application that initializes the HLF and MongoDB connectors, instantiates the block explorer component, supplying it with the connectors, and launches the component.

Presently, DELTA comes with multiple driver implementations. In regard to DLTs, there are drivers for HLF and Quorum, both popular choices, well established in the area of permissioned DLTs. On the other hand, concerning databases, DELTA comes with implementations for MongoDB and PostgreSQL. MongoDB is also a widely used, well established, and popular document database. Since smart contracts usually store their state as JSON resources, the choice of a document-oriented database is a good match for a DLT-DB replication, avoiding the need to perform transformations on the data structure of an asset. DELTA's flexible

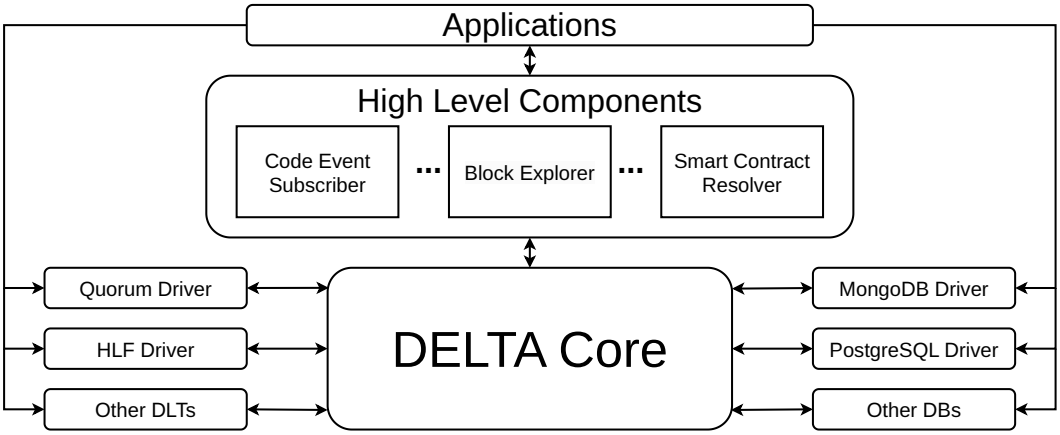


FIGURE 1 Interrelationship between the conceptual areas composing the synchronization tool.

TABLE 2 Support of synchronization features by the technologies analyzed, compared to DELTA

	Storage of complex assets	Efficient queries on complex assets	Crash fault recovery	Multiple DLT technologies	Multiple DB technologies
BigchainDB [23]	✓	✓	✓	✗*	✗
Blockchain Relational Database [24]	✓	✓	✓	✗*	✗
LedgerSync [26]	✗	✗	✗	✗	✗
Vent [27]	✗	✗	✓	✗	✗
Ledgerdata Refiner [28]	✓	✗	✓	✗	✗
ChainSQL [29]	✓	✓	✓	✗	✗
DELTA	✓	✓	✓	✓	✓

*Implements its own DLT, instead of coupling to a node in a network of a previously existing technology.

design allows integrating other database technologies, such as relational databases, and our PostgreSQL driver illustrates that possibility, making possible the use of ACID transactions, joins, and group-by clauses in the accesses to the stored data. In the case of structured (e.g., relational) databases, this expressivity comes at the cost of providing a definition of the smart contract's data model, obtainable via a public query method. This schema definition, retrieved from the smart contract before subscribing to its events, is used by the PostgreSQL driver to dynamically generate the code of the data access layer and, if needed, also instantiate the structure of the local database. Thus, when dealing with structured databases, the application's data access layer is generated and imported at runtime, based on the data model of the contracts to be synchronized. To this end, a transformation of the data embedded in the code events emitted from smart contracts, from JSON documents to a tabular structure, has been conveniently implemented for their storage in a relational database.

Again, supporting any other DLT or database does not imply any modification in the components or applications developed for DELTA. It just requires the development of a driver implementing the corresponding operation interface (mainly, `IDeltaClient` or `IDeltaDatabaseConnector`) for that technology. In the case of adopting a new DLT, developing DELTA-conformant base smart contracts is recommended, providing an implementation of the event emission mechanisms described in Section 4, that can be inherited and extended by user smart contracts.

Table 2 compares DELTA with some of the approaches that we presented in Section 2 in terms of their coverage of five different aspects of synchronization, namely: storage of complex assets, efficient queries on complex assets, crash fault recovery, support to multiple DLTs, and support to multiple databases. This aims to display a qualitative measure of the completeness of such approaches and highlight their strong and weak points. DELTA fulfills these aspects, overcoming the shortcomings of previous systems.

4 | HIGH LEVEL DESIGN

In this Section we describe the inner details of DELTA. First, we present the synchronization strategy, that uses event emission from the DLT along with event reception and processing by the client application. Next, we describe the custom smart contracts that broadcast these events, highlighting how they extend the data modification primitives of basic smart contracts. Thereafter, we introduce some advanced techniques for synchronization, such as the crash fault tolerance mechanisms, which enable the application to recover from crashes, and the dynamic subscription model, which gets the application automatically subscribed to events from new smart contracts as soon as they start committing transactions into the ledger. Finally, the mechanisms implemented for storing the complete value history of an asset are outlined.

4.1 | Event synchronization strategy

The synchronization of the ledger contents into a database is based on the broadcast of events from the DLT network nodes when applying committed transactions. DELTA operates as a client application which connects to one of the nodes in the network and keeps listening to events broadcast from that node. These events can be block or code events. The former are automatically broadcast when the node receives a new transaction block that has been approved by the consensus algorithm. The latter originate from event emission instructions included in the smart contract code by its developer, and are broadcast as the node applies in its ledger replica the transaction comprising these events. Code events are a part of the transaction, since they are generated from the smart contract execution, along with data read and write operations. The subscription to block and code events is enabled by calls to the `subscribeToBlockEvents` and `subscribeToCodeEvents` methods, respectively, in the `IDeltaClient` interface.

Block events include data such as block number, block hash and the identifiers of the transactions contained in the block. DELTA holds a subscription to new block events, in order to keep a global persistent registry of the blocks received and the transactions contained within. This feature, in addition to allowing to check the network's progress in terms of transaction blocks committed, works as a reverse cache to determine the block each transaction belongs to. This facilitates the query of individual transactions directly on the DLT nodes and becomes very useful for performing accountability tasks.

The emission of code events is defined in smart contracts by the inclusion of specific instructions, so that these events are broadcast (bound to the smart contract and transaction identifiers) when the node applies the corresponding transaction. DELTA creates a listener for each smart contract the application gets subscribed to, managing the subscription to each smart contract in an independent execution thread.

These code events are used in DELTA to synchronize the ledger state into the database. For this, a specification of the format to be followed by code events has been defined, according to CRUD operation semantics. Every time a smart contract executes a data modification operation over the ledger state, it emits an event into which such an operation is encoded. When the client application receives the event, DELTA processes it according to the code event specification and applies the corresponding data modification operation on the database (`createAsset`, `updateAsset`, or `deleteAsset`, as defined by the `IDeltaDatabaseConnector` interface). This way, the database contents are kept in sync with the ledger as long as code events are received, processed, and applied, meeting the rules of eventual data consistency.

4.2 | Smart contracts

DELTA requires that, every time a data modification operation is executed in the smart contract code, an event that encodes the operation according to the DELTA's event specification format is emitted. In addition, synchronizing a DLT into a structured database requires the inclusion of a definition of the smart contract's data model, that will be mapped onto the local database for the creation of its schema and the transformation of data assets into (and from) a tabular representation.

Nevertheless, it would not be acceptable to compel smart contract developers who wish to benefit from DELTA's advantages to get familiar with the specification and explicitly build and emit those events. In order to avoid this issue, a base smart contract is provided for both HLF (TypeScript) and Quorum (Solidity), supplying the protected (in terms of function visibility) implementation of a CRUD interface. Each of this interface's data modification operations receives a couple of parameters sticking to key-value semantics, from which the operation is executed on the smart contract's persistent state and the corresponding event is built and emitted, matching DELTA's event specification. This enables smart contract developers to make their own

contracts extend from this parent contract by means of inheritance mechanisms and get DELTA's benefits, just calling operations provided by the base smart contract instead of native data modification primitives. Of course, smart contract developers can always avoid inheritance from the base smart contract provided and manually implement data modification operations along with event construction and emission.

DELTA's event specification establishes that events must contain a (serialized) JSON document with a fixed structure, shown in Listing 1 as a TypeScript interface. The `asset` field contains the value bound to the key provided in the smart contract persistent state. Although this field can be an instance of any data type, it will usually be a nested JSON document.

LISTING 1 DELTA event structure

```
interface DeltaEvent {
    // Event specification version number
    delta: string,
    // [CREATE, UPDATE, DELETE]
    operation: DeltaOperation,
    data: {
        // Key
        key: string,
        // Value. Optional, only for CREATE|UPDATE
        asset: any
    }}
```

4.3 | Advanced synchronization mechanisms

A key aspect of DELTA's design is the ability of the synchronization tool to recover from failures, as well as the type of failures the application can recover from. DELTA's design and implementation have been conceived in such a way that it keeps a registry of the smart contracts it is syncing and the index of the last block synchronized for each one of them, which works as a set of counters. This information is stored in the local database and can be retrieved by calling the `querySynchronizedSmartContracts` and `getLastBlock` methods in the `IDeltaDatabaseConnector` interface. Therefore, upon events like an application reboot or a temporary connection loss, the application can resume the synchronization process at the point it was before the interruption happened, receiving every event emitted from the last block synchronized for each smart contract. The mechanisms designed for this ensure that no transaction blocks are missed and that all transactions are applied to the corresponding smart contracts, according to the order determined by the network's consensus algorithm.

Moreover, after a shutdown period during which the network kept committing transaction blocks, these fault tolerance mechanisms allow the application to restart the previous subscription, request every transaction block that could have been missed, process it and get up to date with the ledger most recent status. An edge case is that of having lost the whole local database (or launching the synchronization process over a fresh database and a DLT network that has already been running for a while); in this scenario the mechanisms aforementioned will perform a complete recovery, making the database reach the ledger's current status by getting and applying sequentially every transaction committed since the network originated.

Another topic that is also addressed by DELTA is how to determine which smart contracts, of all those deployed in the DLT network, must be synchronized into the database. Since it is actually a decision that depends mostly on the user's context and purpose, DELTA provides two modes of subscription to smart contracts. The first option is to allow the user to provide the identifiers of the smart contracts to be synchronized, denoted as fixed subscription model. The second option is to enable dynamic smart contract resolution, by means of the `detectNewSmartContracts` method in the `IDeltaClient` interface. This second mechanism starts a new subscription every time a new transaction originates in a smart contract for which no active subscriptions exist. In this case, every smart contract in the DLT network will eventually get synchronized without the user being required to specify any of their identifiers. These two subscription modes are not mutually exclusive, since the user can set a fixed subscription to some smart contracts and, in addition, enable dynamic resolution in order to get subscribed to those smart contracts that, while being previously unregistered, commit transactions since the point in time when the synchronization was started or resumed.

It is important to note that the mechanisms used for dynamic smart contract resolution entail an overload in computational and temporal terms, since, for every transaction block received, they need to perform an explicit query to the node in order to retrieve the whole block contents, analyze each transaction to find out the smart contract having submitted it, and verify whether an active subscription to that smart contract already exists or a new subscription must be launched. We measure this overhead, jointly with the time required to synchronize a DLT and a database, in Section 5. Another potential disadvantage of indiscriminately making

use of this smart contract resolution mechanism, without a proper assessment, is that the whole network ledger status will be synchronized, instead of the state of just a specific set of smart contracts. This fact could lead to store in the local database a higher than desired volume of data, in case the DLT network holds a high amount of deployed active smart contracts.

4.4 | Asset value history

DLT's underlying data structures enable to traverse the transaction chain backwards and check the different values held by the data assets and variables of a smart contract along its history, binding each value assignment to a specific transaction. Therefore, keeping that ability could be useful, for traceability and auditing purposes, when storing the ledger contents into a database, since it would enable users to efficiently retrieve the chronological sequence of values held by a smart contract's assets. DELTA's synchronization can be configured to store every value of an asset, instead of overwriting (or logically deleting) it into the local database upon successive assignments to the same identifier. Thus, every past value of an asset may be queried, instead of only keeping the current one.

Even if the key of a record is still bound to its current value, previous values of an asset can be accessed, sorted in ascending chronological order, and also retrieving the hash/identifier of the transaction that submitted each one of those values. When launching a query based on filtering criteria specified by the user, an additional field can be checked to specify whether the search results should include the previous values of documents (in the default case, only the most recent value is obtained). Queries can also be parameterized to retrieve the smart contract's status in a given transaction or block height, this is, to query over a snapshot in a specific point of history.

Although the persistence of the historic values of assets has been implemented on such a way that it does not impact the performance of regular queries, this mode certainly requires larger storage capabilities, since new values are appended instead of just being overwritten. Hence, it can be conditionally enabled when launching DELTA. It must be noted that, in case of restarting the application, the synchronization process resumes at the point it stopped before, dictating how new data values will be stored in the database onwards.

5 | EVALUATION

In this section we assess DELTA in order to evaluate different aspects of its performance. First, we measure the overhead and time required to synchronize a DLT with a database in the worst case scenario, i.e., having to synchronize all the assets in the ledger. Secondly, we estimate the improvements in terms of query efficiency that can be achieved with a solution of this kind. To do so, we evaluate DELTA in two different scenarios: executing key-value and complex queries, with three different types of assets and with both HLF and Quorum as DLTs.

5.1 | Experimental setup

All experiments have been executed on a DLT network of seven nodes, for both HLF and Quorum, selecting Raft as consensus algorithm. Each node is placed in a different virtual machine, running Ubuntu 20.04.2 and comprising 4 vCPUs of 2GHz each, 8GB RAM and 50GB HDD. The client application executing the tests is a Node v12.22.10 process running in another virtual machine with equal specifications and deployed in the same VLAN as the DLT nodes. Regarding specific versions of the technologies being evaluated, we have used HLF 2.4.2 and Quorum 20.10.0 as DLTs, while we have used MongoDB 4.4 and PostgreSQL 12.4 as databases.

For both DLTs, HLF and Quorum, we used three different types of JSON documents with increasingly complex structure and size: small (0.3KB, no nesting), medium (2KB, 2 nesting levels) and large (18KB, 4 nesting levels). Both factors, document size and the complexity of its structure, became relevant when executing queries on them. When synchronizing data over PostgreSQL, every record within a single asset is inserted in the context of the same transaction (nested writes), having selected the database isolation level as "read committed".

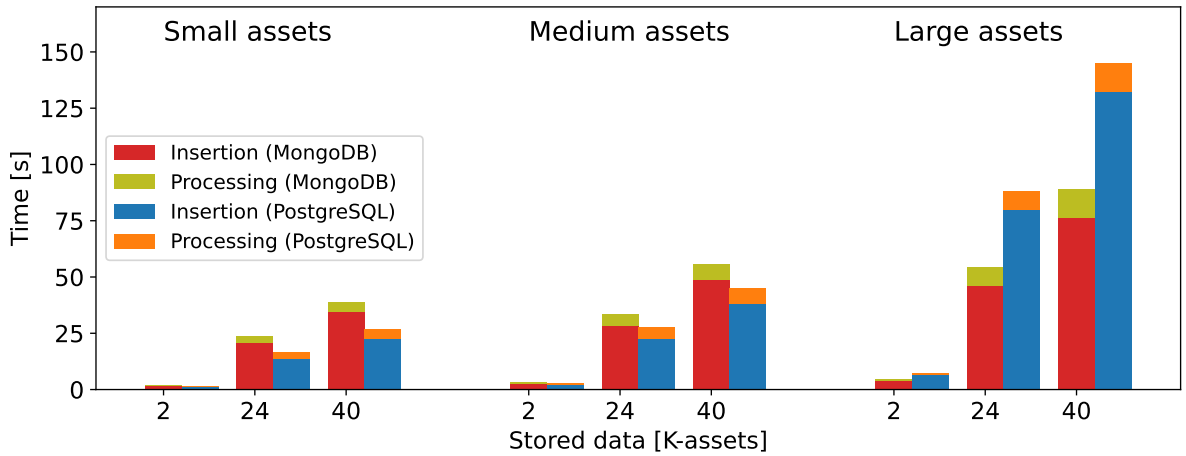


FIGURE 2 Latency in DELTA's asset synchronization per volume of Quorum documents stored, differentiating between event processing and database insertion.

5.2 | DELTA's overhead and synchronization time

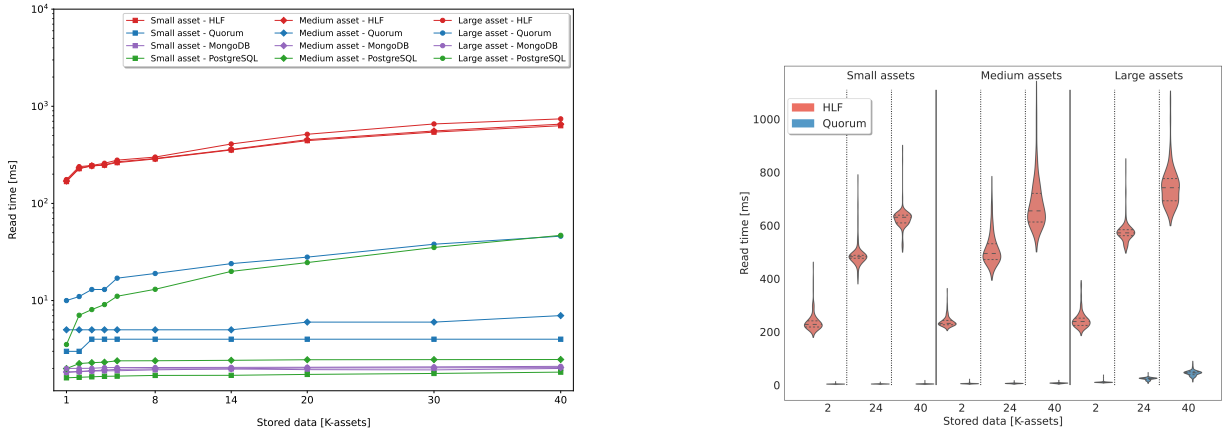
Section 4 mentioned that our solution introduces an overhead when it is updating a database to the contents of the DLT, as DELTA must process and validate all events received from a DLT node, and prepare them to get inserted in the database. Similarly, we also mentioned that DELTA can recover from errors such as shutdown periods or loss of the database, although this introduces a delay due to the re-synchronization. We measured both aspects assuming the worst case scenario, i.e., losing the database and having to synchronize it completely.

We found that the overhead depends on the DLT we are synchronizing with and its event model. For HLF, events are emitted as JSON documents and their processing overhead is negligible (for instance, event processing only demanded 179ms out of 76230ms for database insertions in the worst case, when 40000 large assets had been handled). This, however, was not the case for Quorum, where the overhead was more significant, as we show in Figure 2. This figure stacks event processing time and database insertion time when synchronizing data from a Quorum node for different combinations of number of assets and asset size. The reason for this –perceptible– overhead is that, in Quorum, events must be reconstructed, requiring the explicit decoding of complex data types, which is a time consuming process. However, our aggregated transaction dispatching rate in this (worst) case scenario is still much higher than when emitting transactions to any of the two DLTs tested, according to the experiments run. For instance, as we can observe in Figure 2, we could sync 40000 large assets in less than 90 seconds (in MongoDB) or 150 seconds (in PostgreSQL), i.e., less than 2.25ms or 3.75ms per asset, while inserting them in the Quorum DLT took around 50 minutes.

In regard to the comparison between databases, PostgreSQL needs shorter times than MongoDB when the synchronization manages either small or medium assets. However, with a synchronization consisting of large assets, PostgreSQL demands longer times than MongoDB, and this difference increases as the number of assets to insert grows. Since large assets, because of their structure nesting level, need to perform insertions on additional tables in the resulting schema, that behavior seems to depend on the amount of tables to be used and the resulting data size.

5.3 | Key-value queries

In this first read-related scenario, we evaluate the cost of performing key-value queries, i.e., retrieving an asset from its identifier or indexing key. Figure 3(a) displays the median times for the execution of 200 random key-value queries, repeated as the ledger/database size increases, for the four technologies compared (HLF, Quorum, MongoDB and PostgreSQL) and for each of the three types of document presented. MongoDB times keep almost constant for these key-value queries, regardless of the number of documents stored or their volume and its performance is, by large, superior to that of Quorum or HLF. PostgreSQL times are clearly shorter than those of MongoDB for small assets, since there is no nesting in their structure and a simple table may hold all their data. However, when medium and large assets are considered, their involved nested structure requires additional



(a) Median times for key-value queries.

(b) Query time dispersion in HLF and Quorum.

FIGURE 3 (a) MongoDB times remain almost constant for key-value queries, regardless of the number of documents stored or their volume. Quorum is between two and five times slower than MongoDB, except for large assets, where its performance decays with the ledger size, being one order of magnitude worse than MongoDB. HLF, although not much affected by the asset size, can become almost three orders of magnitude slower than MongoDB as the ledger grows. (b) While dispersion in Quorum is limited, in HLF it is quite relevant, specially as the number of assets stored increases.

tables in PostgreSQL and that fact introduces noticeable penalties in the query times. Even in those cases, PostgreSQL demands shorter times than Quorum. Quorum is between two and five times slower than MongoDB, except for large assets, where its performance decays with the ledger size, being one order of magnitude worse than MongoDB in this experiment. HLF, although not much affected by the asset size, can become almost three orders of magnitude slower than MongoDB as the ledger grows.

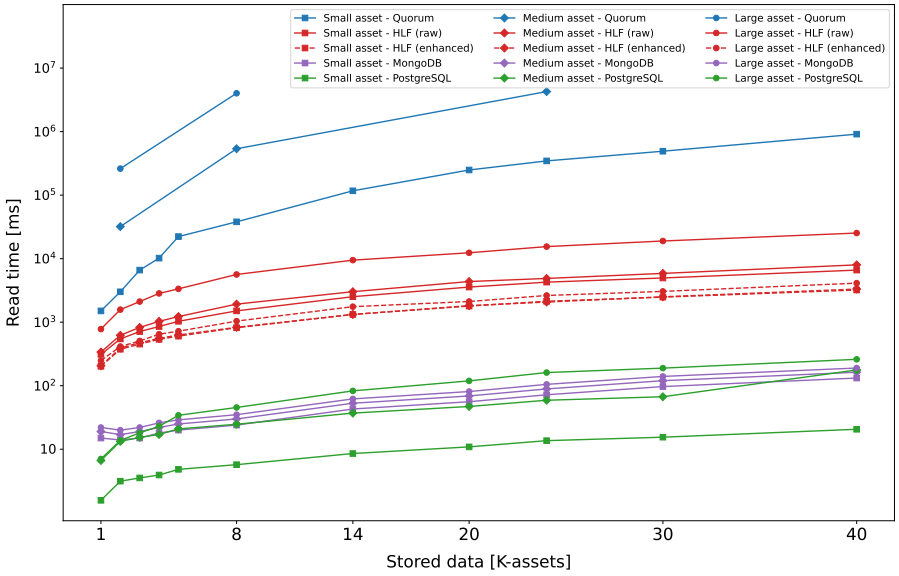
The adjacent plot in Figure 3(b) shows the dispersion in the recurring execution of these queries, for HLF and Quorum, grouped by the document type and the ledger size. While dispersion in Quorum is limited, in HLF it is quite relevant, specially as the number of assets stored increases. We did not include the dispersion for MongoDB as it was negligible.

Both latency and dispersion are higher in HLF than in Quorum. Key-value asset retrieval in HLF requires querying the underlying CouchDB from the smart contract. Quorum implements this operation by directly extracting the corresponding value from a Solidity mapping (a data structure describing a hash-map), although the smart contract's state is still stored in an underlying LevelDB that would need to be queried anyway. The results observed may imply either that the submission of read-only calls to smart contracts is faster in Quorum than in HLF, that Quorum's internal data structures are better than HLF's for key-value asset retrieval, that LevelDB is faster and more timely consistent than CouchDB for such operations, or that the database connector used by Quorum is noticeably more performant than the one used by HLF.

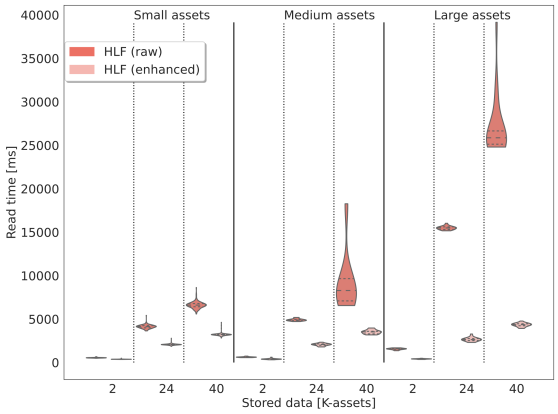
5.4 | Complex queries

In the second read-related scenario, we evaluate the cost of performing complex queries. By complex queries we refer to the retrieval of all assets matching specific filtering criteria, an intricate expression composed of multiple custom conditions. HLF splits into raw and enhanced queries. The latter take advantage from the capabilities of HLF's smart contracts to execute queries via the API of CouchDB when it is the database supporting the ledger, while the former are implemented by linearly traversing all assets stored in the smart contract's data state, applying the filtering criteria to them and keeping those that match the comparison. Since Solidity does not provide facilities for querying, we implemented a query mechanism on Quorum's smart contracts that imitates HLF's raw queries.

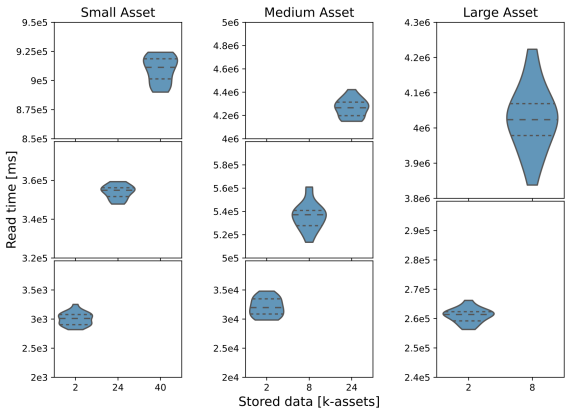
Figure 4(a) displays the median times for the execution of 20 complex queries, repeated as the ledger/database size increases, for the four technologies compared and each of the three types of document contemplated. For these queries, MongoDB latency increases with the database size independently of the documents' complexity and volume and, as expected, outperforms both



(a) Median times for complex queries.



(b) Query time dispersion in HLF.



(c) Query time dispersion in Quorum.

FIGURE 4 (a) MongoDB complex query latency increases with the database size independently of the assets' complexity. HLF queries follow a similar trend, although times are between one and two orders of magnitude worse. Quorum query times are substantially higher and performance decays faster, being 5 orders of magnitude slower than MongoDB. PostgreSQL times are comparable to those of MongoDB in medium and large assets, but they are one order of magnitude shorter in small assets. (b) Dispersion in HLF complex queries remained controlled but became relevant for the 40K assets sample and raw queries, which did not cope well with nested documents. (c) On the other hand, dispersion in Quorum is severely affected by the asset complexity.

HLF and Quorum. In MongoDB, the differences among the three different asset sizes are minimal, since for a given asset all its attributes are held in the same document. PostgreSQL, on the other hand, shows significant query time differences depending on the asset complexity, since storing a large asset requires multiple records. Thus, PostgreSQL times are between 4 and 6 times shorter than those of MongoDB for small assets, 30% shorter –on average– for medium assets (except when 40000 assets are handled, where PostgreSQL times are slightly longer than those of MongoDB: 177ms vs 163ms), and much more dependent on the amount of queried elements when large assets are considered, where PostgreSQL generates shorter times when fewer

than 5000 assets are queried, but generating progressively longer times when more than 5000 assets are handled. In the end, for querying 40000 assets, PostgreSQL takes 260ms while MongoDB only takes 190ms. HLF queries follow a trend similar to that of MongoDB, although times are between one (enhanced queries) and two (raw queries) orders of magnitude worse than in MongoDB. Quorum query times are substantially higher and its performance decays faster, resulting in query times that were between 2 and 5 orders of magnitude higher than for MongoDB or PostgreSQL. Observe that we could only run queries for up to 8000 and 24000 assets for the large and medium asset types, respectively. Beyond these volumes, the Geth processes running on the Quorum nodes started throwing timeouts and aborting the running query. If we had been able to execute queries for larger volumes of assets, this difference would have gone beyond 5 orders of magnitude.

Figure 4(b) shows the dispersion in HLF queries, grouped by asset type and for different amounts of assets in the ledger, and differentiating between raw and enhanced queries. Dispersion here remained controlled but became relevant for the 40K assets sample. This difference is due to the higher efficiency of CouchDB, that reduced the impact of the asset size on the enhanced queries. The nesting in the large asset did affect the raw queries severely. Figure 4(c) shows the dispersion in Quorum queries, which, on the other hand, is severely affected by the asset complexity, i.e., the nesting and volume. As in the key-value case, dispersion in MongoDB queries was negligible and it is not displayed.

As opposed to key-value queries, complex queries are far slower and their latencies are more sparse in Quorum than in HLF. While HLF's queries (both raw and enhanced) could be successfully executed for this experiment, Quorum queries did not even complete for documents more complex than the small asset. The brute-force approach of complex queries in Quorum greatly penalizes its performance, especially when compared to the enhanced queries of HLF on top of CouchDB. However, HLF's implementation of raw queries is actually similar to Quorum's, but the former are way faster than the latter; even if they are one order of magnitude slower than HLF's enhanced queries, and their dispersion is also greater when the asset complexity and the document size increase. The significant performance difference between raw queries on HLF and Quorum could be just a consequence of how the operations they make use of (looping, data selection, retrieval and decoding, and value comparison) are internally built in the smart contract execution runtime of each platform. A specific profiling process would be required to assess this reasoning.

6 | DISCUSSION

Our solution intends to mitigate DLT's limitations related to data retrieval, by adding a synchronized local database. It must be noted that, nevertheless, restrictions of the underlying DLT related to transaction throughput and the deterioration of its performance as the ledger size grows still apply. Therefore, our solution does not enable the adoption of DLTs on inappropriate use cases, where such restrictions are not admissible.

DELTA's synchronization is based on the subscription to events emitted from smart contracts, since it is provided by most current mainstream DLTs. Relying upon event emission implies that the integration of a DLT whose smart contracts do not feature this mechanism would require a rework in DELTA's core, in order to enable an alternative procedure for synchronization.

Smart contracts also require to define the encoding and emission of events matching DELTA's specification whenever an operation involving persistent data modification is executed. DELTA provides base contracts including implementations of these operations for every technology currently supported. However, in case the DLT does not support contract upgrade, smart contracts would still require to be DELTA-conformant at the moment they are first deployed. Unlike schema-free databases (MongoDB), synchronization on a structured (e.g., relational) database requires providing information about the smart contracts' data model, which will translate to the destination database schema. Since DELTA currently supports PostgreSQL, this is performed by embedding information about the data model in the contract's code and enabling its retrieval via an additional read-only public method. The data model definition is used by the PostgreSQL driver to dynamically generate and import the data access layer code, as well as instantiate the database structure if needed, before synchronizing the events emitted by the smart contract.

Finally, the local database itself does not hold any guarantees related to data integrity. Its purpose is to boost queries, but the DLT must still be checked in case of data auditing. To this end, every asset stored in the database contains the identifier of the corresponding transaction, so that they can be traced back to the DLT to verify their validity.

7 | CONCLUSION

DELTA provides a tool to synchronize the state of a DLT on a local database with negligible overhead, thus enabling the operation with smart contracts whose data states possess a complex structure. It facilitates the execution of complex queries on that shared state, kept by the nodes hosting replicas of the ledger. Query times decrease up to five orders of magnitude, depending on the DLT, compared to queries directed to the ledger nodes.

This solution is an improvement upon previously existing approaches, providing additional useful features. Examples of such innovative features are the ability to recover and resume the synchronization process upon crashes or network interruptions, as well as the dynamic identification of smart contracts deployed on the network, enabling the automatic synchronization of their contents. Its architecture is based on a modular approach, agnostic to specific technology implementation details, thus supporting multiple DLTs and databases and easing the adoption of new technologies on both scopes. The current release of DELTA includes the implementation of drivers for Hyperledger Fabric and Quorum as source DLTs, and for MongoDB and PostgreSQL as target databases.

Since DELTA enables DLTs to be queried by using database languages and semantics, potential future research lines include complementing this effort with the ability to write on DLTs (actually, submit transactions) by means of similar mechanisms, thus further simplifying adoption of DLTs, supplying them with a complete database-like interaction model that, moreover, supports the efficient execution of complex custom queries.

REFERENCES

1. Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. *Bitcoin.* 2008.
2. Wood G, others . Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper.* 2014;151(2014):1–32.
3. Wüst K, Gervais A. Do you need a blockchain?. In: Crypto Valley Conference on Blockchain Technology (CVCBT). IEEE. 2018:45–54.
4. Yaga D, Mell P, Roby N, Scarfone K. Blockchain technology overview. *arXiv preprint arXiv:1906.11078.* 2019.
5. Reyna A, Martín C, Chen J, Soler E, Díaz M. On blockchain and its integration with IoT. Challenges and opportunities. *Future Generation Computer Systems.* 2018;88:173–190.
6. Davies S. Time for trust: The trillion-dollar reasons to rethink blockchain. https://pwc.blogs.com/technology-insights/2020/10/trust-transparency-efficiency-the-trillion-dollar-reasons-to-rethink-blockchain.html; . Last accessed on: 25.11.2022.
7. Bartoletti M, Lande S, Pompianu L, Bracciali A. A general framework for blockchain analytics. In: 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL@Middleware). ACM 2017; Las Vegas, NV, USA:7:1–7:6
8. Li Y, Zheng K, Yan Y, Liu Q, Zhou X. EtherQL: A Query Layer for Blockchain System. In: 22nd International Conference on Database Systems for Advanced Applications (DASFAA). Springer 2017; Suzhou, China:556–567
9. Bragagnolo S, Rocha H, Denker M, Ducasse S. Ethereum Query Language. In: 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB@ICSE). ACM 2018; Gothenburg, Sweden:1–8
10. Pratama FA, Mutijarsa K. Query Support for Data Processing and Analysis on Ethereum Blockchain. In: International Symposium on Electronics and Smart Devices (ISESD). 2018; Bandung, Indonesia:1-5
11. Brinckman E, Kuehlkamp A, Nabrzyski J, Taylor IJ. Techniques and Applications for Crawling, Ingesting and Analyzing Blockchain Data. In: International Conference on Information and Communication Technology Convergence (ICTC). IEEE 2019; Jeju Island, South Korea:717–722
12. Han J, Kim H, Eom H, Coignard J, Wu K, Son Y. Enabling SQL-Query Processing for Ethereum-based Blockchain Systems. In: 9th International Conference on Web Intelligence, Mining and Semantics, (WIMS). ACM 2019; Seoul, Republic of Korea:9:1–9:7
13. Linoy S, Mahdikhani H, Ray S, Lu R, Stakhanova N, Ghorbani AA. Scalable Privacy-Preserving Query Processing over Ethereum Blockchain. In: IEEE International Conference on Blockchain (ICBC). IEEE 2019; Atlanta, GA, USA:398–404
14. Huang X, Gong X, Huang Z, Zhao L, Gao K. EBTree: A B-plus Tree Based Index for Ethereum Blockchain Data. In: Asia Service Sciences and Software Engineering Conference (ASSE). ACM 2020; Nagoya, Japan:83–90
15. Özdayi MS, Kantarcioglu M, Malin BA. Leveraging Blockchain for Immutable Logging and Querying Across Multiple Sites. *BMC Medical Genomics.* 2020;13(82).
16. Thabet NA, Abdelbaki N. Efficient Quering Blockchain Applications. In: 3rd Conference on Novel Intelligent and Leading Emerging Sciences (NILES). IEEE 2021:365–369
17. Altimore P, Martinez J. Use Blockchain Data Manager to send data to Azure CosmosDB. https://learn.microsoft.com/en-us/sql/relational-databases/security/ledger/ledger-overview; . Last accessed on: 25.11.2022.
18. Chiplunkar A. ether_sql. https://ether-sql.readthedocs.io/en/latest/guides/syncing-blockchain.html; . Last accessed on: 25.11.2022.
19. Abe developers . bitcoin-abe. https://github.com/bitcoin-abe/bitcoin-abe; . Last accessed on: 25.11.2022.
20. Sphere 10 Software . blockchainSQL. https://blockchainsql.io/; . Last accessed on: 25.11.2022.
21. BigQuery: Cloud Enterprise Data Warehouse. https://cloud.google.com/bigquery/docs/introduction; . Last accessed on: 25.11.2022.
22. Bragagnolo S, Rocha H, Denker M, Ducasse S. SmartInspect: Solidity smart contract inspector. In: International Workshop on Blockchain-Oriented Software Engineering (IWBOSE). IEEE. 2018:9–18.
23. McConaghy T, Marques R, Müller A, et al. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB.* 2016.
24. Nathan S, Govindarajan C, Saraf A, Sethi M, Jayachandran P. Blockchain meets database: Design and implementation of a blockchain relational database. *arXiv preprint arXiv:1903.01919.* 2019.
25. Schuhknecht FM, Sharma A, Dittrich J, Agrawal D. chainifyDB: How to get rid of your Blockchain and use your DBMS instead. In: 11th Conference on Innovative Data Systems Research (CIDR). 2021.
26. Skcript . Ledger Sync. https://github.com/skcript/sawtooth-ledger-sync; . Last accessed on: 25.11.2022.

27. Hyperledger Burrow . Vent: Hyperledger Burrow's SQL mapping layer. https://cloud.google.com/bigquery/docs/introduction; . Last accessed on: 25.11.2022.
28. Zhou E, Sun H, Pi B, Sun J, Yamashita K, Nomura Y. Ledgerdata Refiner: a powerful ledger data query platform for Hyperledger Fabric. In: 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE. 2019:433–440.
29. PeerSafe . ChainSQL: A Blockchain Database Application Platform. https://github.com/ChainSQL/chainsqld; . Last accessed on: 25.11.2022.
30. Muzammal M, Qu Q, Nasrulin B. Renovating blockchain with distributed databases: An open source system. *Future Gener. Comput. Syst.*. 2019;90:105–117. doi: 10.1016/j.future.2018.07.042
31. Gray J. Notes on Data Base Operating Systems. In: Operating Systems, An Advanced Course, Lecture Notes in Computer Science, 60. Springer 1978:393–481
32. MongoDB . MongoDB Manual: Transactions. https://www.mongodb.com/docs/manual/core/transactions/; . Last accessed on: 11.08.2023.
33. Berenson H, Bernstein PA, Gray J, Melton J, O'Neil EJ, O'Neil PE. A Critique of ANSI SQL Isolation Levels. In: SIGMOD International Conference on Management of Data. ACM Press 1995; San Jose, California, USA:1–10.
34. Bernstein AJ, Lewis PM, Lu S. Semantic Conditions for Correctness at Different Isolation Levels. In: 16th International Conference on Data Engineering (ICDE). IEEE Computer Society 2000; San Diego, California, USA:57–66
35. Lu S, Bernstein AJ, Lewis PM. Correct Execution of Transactions at Different Isolation Levels. *IEEE Trans. Knowl. Data Eng.*. 2004;16(9):1070–1081. doi: 10.1109/TKDE.2004.34
36. Makris A, Tserpes K, Spiliopoulos G, Zissis D, Anagnostopoulos D. MongoDB Vs PostgreSQL: A comparative study on performance aspects. *GeoInformatica.* 2021;25(2):243–268. doi: 10.1007/s10707-020-00407-w
37. Jung MG, Youn SA, Bae J, Choi YL. A study on data input and output performance comparison of MongoDB and PostgreSQL in the big data environment. In: 8th International Conference on Database Theory and Application (DTA). IEEE 2015; Jeju, South Korea:14–17.