A modular approach to Hybrid Blockchain-based and Relational Database Architectures

Rafael Avilar Sá[1,2], Leonardo Oliveira Moreira (Coorientador)[1], Javam de Castro Machado (Orientador)[1,2]

[1]Laboratório de Sistemas e Banco de Dados (LSBD)
Departamento de Computação
Universidade Federal do Ceará (UFC) – Fortaleza, CE – Brasil

[2]Mestrado e Doutorado em Ciência da Computação (MDCC)
Departamento de Computação
Universidade Federal do Ceará (UFC) – Fortaleza, CE – Brasil

{rafael.sa,leonardo.moreira,javam.machado}@lsbd.ufc.br

Nível: Mestrado.
Ingresso: Maio, 2021.
Previsão de Término: Maio, 2024.
Etapas concluídas: Disciplinas, Definição do Problema, Revisão Bibliográfica Preliminar, Definição das Tecnologias de Implementação, Metodologia de Avaliação Preliminar
Defesa da Pré-Proposta: Outubro, 2022
Defesa da Proposta: Janeiro, 2023

Abstract. *In this new cloud-powered web landscape, applications may find it beneficial to focus on both efficient data storage and proper security enforcement. Blockchains provides immutable and irrefutable data, but can be quite slow, while relational databases show great efficiency. MOON is a tool designed to act as a bridge between application and databases to create a hybrid approach to database management. It enables the use of both a Relational Database and a Blockchain-based database. This paper analyzes and proposes improvements to MOON's architecture to increase compatibility, maintainability, and help advance research into the area of convergence between blockchain and traditional databases.*

Resumo. *Com a popularização da computação em nuvem, novas aplicações neste cenário podem considerar importante focar em ambos segurança e eficiência. Blockchains proporcionam imutabilidade e irrefutabilidade de dados, mas pode ser bem lenta, enquanto bancos de dados relacionais demonstram grande velocidade. MOON é uma ferramenta criada para atuar como uma ponte entre aplicações e bancos de dados, para criar uma abordagem híbrida ao gerenciamento de multiplos bancos de dados. Ela permite o uso em conjunto de tanto um banco de dados relacional como um baseado em blockchain. Esta pesquisa analisa e propõe melhorias para a arquitetura MOON em prol de aumentar compatibilidade, manutenbilidade e ajudar a avançar pesquisas na área de convergência entre blockchain e bancos de dados tradicionais.*

1. Introduction

The blockchain, originally conceived as part of the Bitcoin electronic cash system [Nakamoto 2008], allows the storing of data without a trustworthy third party, focusing on creating an immutable, irrefutable and tamper-proof distributed linked list. However, this powerful security comes at a steep performance cost. Blockchains are characteristically slow at writing operations [Zheng et al. 2018], especially when compared to highly efficient databases. This is, partly, done on purpose, due to the usage of computationally-intensive security algorithms such as *Proof-of-Work* (PoW) [Gervais et al. 2016]. On the other hand, while many traditional relational databases offer great performance, they cannot easily produce the same security and integrity properties that blockchains do. Therefore, one has to understand what kind of data they wish to store and utilize whatever architecture best suit their storage needs.

Considering how much and how many types of information modern applications collect and use, different types of data storage solutions are often implemented simultaneously. For example, web applications may use a simple key-value database such as *Redis* for caching purposes, and a relational database like *MySQL* to store more complex types of data. These hybrid architectures are becoming increasingly common, and alongside them, we can also see the increase in abstract, high-level approaches to data manipulation. For example, *Prisma* is an Object Relational Mapper (ORM) developed for the *Node.js* environment which abstracts the need to construct database queries, allowing developers to write a single model representing a database entity, which can be easily used with a number of different database engines, relational or otherwise.

The *approach to data Management on relatiOnal database and blOckchaiN* (MOON) [Marinho et al. 2020] was a project inspired by such tools to act as a singular entry-point for database queries by applications utilizing both Blockchains and Relational Databases. Developers write queries in the SQL (Structured Query Language) format, which are then translated by the MOON's middleware into an equivalent query for either a Blockchain or Relational Database. This hybrid approach to database management simplifies development by eliminating the need for developers to use different query languages, frameworks or libraries for each database, as well as know where each entity is located.

However, MOON currently presents several restrictions. One such restriction is that the use case provided in the original MOON article supports only *BigchainDB* and *PostgreSQL*. This limits MOON's scope, as well as its utility and interoperability. This paper's primary goal is to propose improvements to MOON's current architecture, focusing on enhancing compatibility and maintainability. The secondary goal of this work is to help advance research into the area of convergence between traditional Database Management Systems (DBMS) and the more recent blockchain-based databases (such as *BigchainDB*), by observing the MOON middleware implementation, exploring its structure, analyzing how it could be enhanced, and proposing improvements. This paper presents the initial results of ongoing research.

2. Background

MOON was initially pitched as a hybrid approach to data management, for use in applications that could benefit from the properties offered by both blockchain and relational databases, such as those in the *e-health* software field [Marinho et al. 2020]. Compared to heterogeneous database systems, which aim to integrate divergent DBMS via a single interface, there are some unique challenges in attempting the same with a DBMS and blockchain. Blockchains are, by design, append-only, consequently they do not show the same properties that databases do, like support for *DELETE* or *UPDATE* operations. Some blockchains, however, do possess more database-like qualities. BigchainDB, for example, permits queries to be written similarly to MongoDB [Bigchain and Gmb 2018].

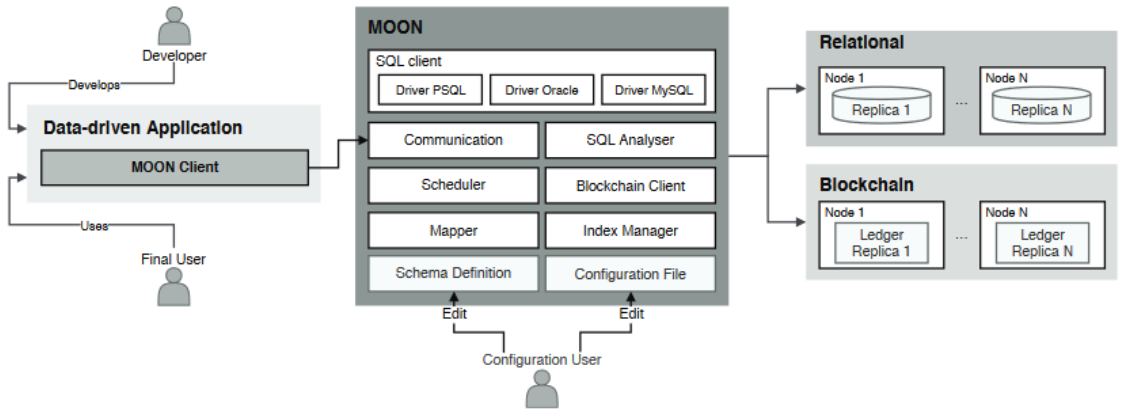


Figure 1. Overview of the MOON architecture [Marinho et al. 2020].

In this context, to achieve integration, MOON's architecture was designed as two separate main components: the MOON Client and the MOON middleware. In short, the client receives SQL requests from applications and users, forwarding them to the middleware. The middleware analyzes those requests, identifies the type of request and involved entities, processes the request and then sends it to the correct persistence model (Blockchain-based or Relational). Upon receiving a response, the middleware fowards it back to the client. To handle all this, the middleware was divided into a variety of modules, each performing a different role. For example, the Scheduler, which queues requests, and the SQL Analyzer, which translates requests into the specified persistence model. A python program was later developed to evaluate the proposed architecture's response time and correctness.

3. Our Approach

In this section, we will present techniques which could serve to improve the MOON middleware with two new features: Compatibility and Maintainability. We define compatibility as the capacity for two systems to work together without having to be modified to do so. Compatible systems use the same formats, can be used together or interchangeably. Specifically, we will focus on compatibility on the database type subsystem. For this work, we also define maintainability as the ability of a software system to support changes, be adaptable to the environment, and having the quality of quick repairability in case of failure or downtime.

The original MOON design (Figure 1), already shows these features to some degree: by separating the middleware into several modules, we can increase maintainability. On the other hand, the SQL Client promotes both compatibility and maintainability by encapsulating drivers from several Relational Database Management Systems (RDBMS), such as *Oracle*, *MySQL* and *PostgreSQL* [Marinho et al. 2020].

However, the existing SQL Client description is still somewhat vague in regards to implementation, and the test case originally showcased only supports PostgreSQL. Therefore, by developing the SQL Client, we would be making improvements to both compatibility and maintainability. To this endeavor, we have drafted two options: (1) by developing a custom database accessor that supports multiple database engines via adapters or (2) by using ODBC Drivers.

3.1. Using custom interfaces

This approach is the closest in theory to the concept of the SQL Client as originally described, and is reminiscent on the Django [Holovaty and Kaplan-Moss 2009] and Laravel [Stauffer 2019] architectures for multiple database support. Essentially, it takes the form of a group of entities that abstracts library function calls. In practice, this version of the SQL Client contains adapters that abstract generic SQL database engine methods, like *connect* and *execute sql*. Upon being instantiated, the SQL Client reads a configuration singleton to know which library to use, and then instantiates the appropriate adapter. When receiving function calls, it forwards those calls to the adapter, which itself calls the library. See Figure 2 for an initial diagram of this structure.

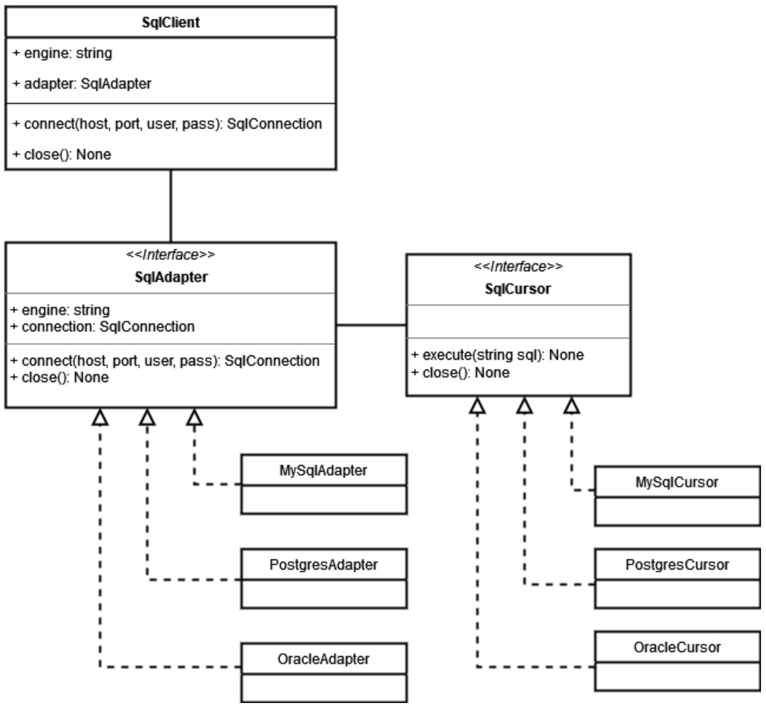


Figure 2. Initial proposal of the new SQL Client architecture

While this approach can be considered inconvenient from a developer perspective, as it requires the development of multiple adapters, as well as several third-party libraries, it could prove fruitful in regards to performance. Many well-supported and efficient libraries for database manipulation have been developed, for a variety of languages. It's also not uncommon to see standardized interfaces for accessing databases within languages, such as PDO (PHP Data Objects) for PHP or the Python Database API Specification format, which can minimize the potential structural overhead. However, there are still some foreseeable drawbacks. There could still be some lack of compatibility between the library function implementations and our own custom adapter structure, and for each DBMS supported, more dependencies are required.

3.2. Using ODBC Drivers

ODBC drivers [Geiger 1995] use the Open Database Connectivity (ODBC) interface by Microsoft that allows applications to access data in DBMS using SQL as a standard. As a protocol, ODBC facilitates interoperability, allowing a single application to access many different DBMS by consuming their ODBC drivers. There are already a plethora of existing drivers, even some available for NoSQL databases like MongoDB. It is also possible to develop custom ones by implementing the ODBC protocol.

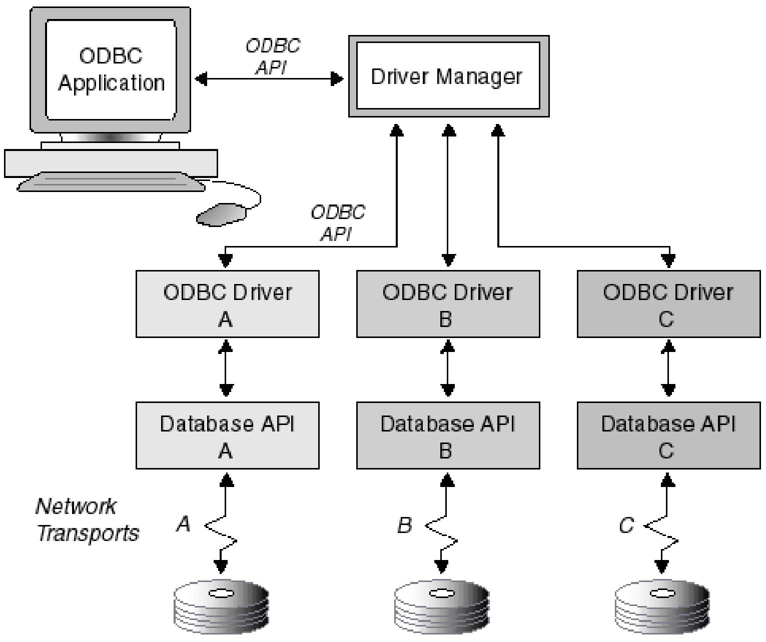
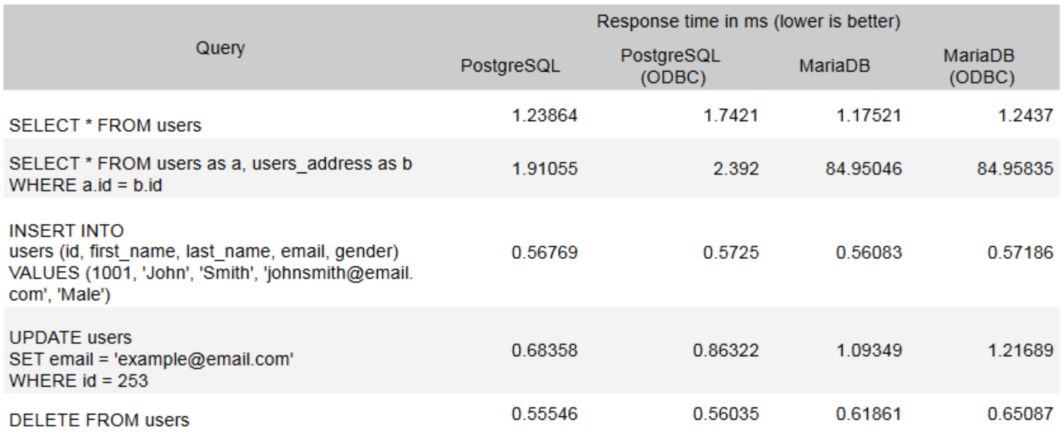


Figure 3. ODBC driver architecture [Oracle 2016]

By incorporating an ODBC bridge into MOON, it could achieve a great degree of database compatibility. Compared to the first proposal, this architecture is much easier to implement and supports just as many, if not more, database types (any database that implements an ODBC connector). It also shows greater benefits to maintainability and code structure, by using a single entity, the ODBC bridge, to make all database calls. However, it is also comparatively slower, as proper libraries are often built with performance in mind. Additionally, ODBC drivers are not only often Windows-only, they are also to be installed on the host machine, and then consumed via the ODBC bridge. Consequently, they are not contained within MOON itself, creating an environmental prerequisite on the middleware host.

3.3. Preliminary test results

Tests were conducted on a Windows 11 OS machine, sporting 16 GBs of DDR4 RAM and a 3.6 GHz Intel i3-10100F CPU, using early prototypes of each approach, developed with the Python scripting language. PostgreSQL and MariaDB were chosen for testing, due to the stark differences between both RDBMS. Two sample tables were generated, each with 1000 rows and a primary key. Table 1 contains *ID*, *first name*, *last name*, *email* and *gender*. Table 2 contains simply *ID* and *address*. Queries were made based on common database operations and executed first using the adapter prototype, then the ODBC prototype for comparison for each RDBMS. Each value below represents the average response time in ms (milliseconds) for 500 executions.



Query	Response time in ms (lower is better)			
	PostgreSQL	PostgreSQL (ODBC)	MariaDB	MariaDB (ODBC)
SELECT * FROM users	1.23864	1.7421	1.17521	1.2437
SELECT * FROM users as a, users_address as b WHERE a.id = b.id	1.91055	2.392	84.95046	84.95835
INSERT INTO users (id, first_name, last_name, email, gender) VALUES (1001, 'John', 'Smith', 'johnsmith@email.com', 'Male')	0.56769	0.5725	0.56083	0.57186
UPDATE users SET email = 'example@email.com' WHERE id = 253	0.68358	0.86322	1.09349	1.21689
DELETE FROM users	0.55546	0.56035	0.61861	0.65087

Figure 4. Initial test results

Initial testing (Figure 4) indicates queries made using ODBC are indeed comparably slower than queries made using proper library accessors, although speed varies greatly depending on query structure and database environment, with some showing nearly the same response time.

4. Related Work

This research builds upon the foundation laid out by [Marinho et al. 2020]. [Nathan et al. 2019] adds blockchain properties, such as smart contracts, immutable transaction logs and a replicated ledger, directly into the PostgreSQL relational database. On the other hand, [Muzammal et al. 2019] introduces ChainSQL, a new blockchain structure with database properties that executes commands received via SQL or JSON format. Both works either present a new blockchain with database properties or database with blockchain properties, while ours attempts to better integrate preexisting blockchain-based databases and relational databases via middleware, allowing them to be used together.

Considering the architectural overhauls detailed in this article, [Brandani 1998] presents an SQL interface and ODBC driver for the Amos II system in order to simultaneously access multiple different database systems using the ODBC protocol, while we use preexisting drivers and a custom adapter structure. [Mason and Lawrence 2005] uses the Unity Java Database Connectivity (JDBC) driver to implement a system allowing for scalable integration of different data sources with minimal overhead. Our work utilizes ODBC, instead, as JDBC is only available for Java applications.

5. Conclusion

In this paper, we've shown MOON, a hybrid approach to database management for blockchain and relational systems, its current architectural foundation, and described some initial results in our ongoing research to add new features to it, namely compatibility and maintainability. In the future, we mean to continue this research, perform more in-depth testing of the architecture, and fully implement an updated version of the SQL Client. One idea is to develop two prototypes, one for each approach, and then compare their performance over real data. Lastly, another worthwhile experiment would be to test whether the usage of *C++*, *C#*, or another highly performing language, rather than Python, has any significant impact on operational performance.

References

Bigchain, D. and Gmb, H. (2018). Bigchaindb 2.0: The blockchain database. white paper.

Brandani, S. (1998). Multi-database Access from Amos II using ODBC. *Linköping Electronic Articles in Computer and Information Science*, 3(19).

Geiger, K. (1995). *Inside ODBC*. Microsoft Press, USA.

Gervais, A., Karame, G. O., Wüst, K., Glykantzis, V., Ritzdorf, H., and Capkun, S. (2016). On the security and performance of proof of work blockchains. CCS '16, page 3–16, New York, NY, USA. Association for Computing Machinery.

Holovaty, A. and Kaplan-Moss, J. (2009). *The definitive guide to Django: Web development done right*. Apress.

Marinho, S. C., Costa Filho, J. S., Moreira, L. O., and Machado, J. C. (2020). Using a hybrid approach to data management in relational database and blockchain: A case study on the e-health domain. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 114–121. IEEE.

Mason, T. and Lawrence, R. (2005). Dynamic database integration in a jdbc driver. In *ICEIS (1)*, pages 326–333.

Muzammal, M., Qu, Q., and Nasrulin, B. (2019). Renovating blockchain with distributed databases: An open source system. *Future generation computer systems*, 90:105–117.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260.

Nathan, S., Govindarajan, C., Saraf, A., Sethi, M., and Jayachandran, P. (2019). Blockchain meets database: Design and implementation of a blockchain relational database. *arXiv preprint arXiv:1903.01919*.

Oracle (2016). *Oracle Database Development Guide*. Oracle.

Stauffer, M. (2019). *Laravel: Up & running: A framework for building modern PHP apps*. O'Reilly Media.

Zheng, Z., Xie, S., Dai, H.-N., Chen, X., and Wang, H. (2018). Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375.