Improving Interoperability between Relational and Blockchain-based Database Systems: A Middleware approach

Rafael Avilar Sá[1,2], **Leonardo O. Moreira**[1], **Javam C. Machado**[1,2]

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

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

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

Abstract. *Multi-model, federated and polystore architectures allow for querying data from different sources through a unified interface, providing interoperability for databases. However, support for blockchain-based databases remains scarce. MOON is a middleware designed to enable cross-model querying of data in relational and blockchain databases through standard SQL syntax. This paper aims to promote the interoperability of blockchain-based and relational database systems through a new approach, called Inter-MOON. Through experimentation, Inter-MOON was found to offer near-total support for SQL DML query syntax, be up to 10x faster than MOON, and show comparable performance to similar tools.*

1. Introduction

The blockchain, originally conceived as part of the Bitcoin electronic cash system [Nakamoto 2008], allows storing data without a trustworthy third party to create an immutable, irrefutable, and tamper-proof distributed linked list. However, blockchains are characteristically slow at writing operations [Zheng et al. 2018]. 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 relational databases offer great performance, they cannot easily produce the same level of security and integrity as blockchains. Therefore, they have different priorities and divergent data models, each presenting unique challenges.

Considering the divergent characteristics of data, and the many kinds of storage solutions which are now available, there is a need to make interoperability of heterogeneous data easier [Babcock et al. 2002, Stonebraker and Çetintemel 2018]. Federated databases, multistores, and polystores are all examples of this phenomenon. However, despite the growing market for blockchains [Gadekallu et al. 2022], attempts at providing support for them in federated systems are still barely seen.

The *approach to data **M**anagement on relati**O**nal database and bl**O**ckchai**N*** (MOON) [Marinho et al. 2020] is a tool meant to act as a singular entry-point for database queries by applications using both blockchain-based and relational databases. Queries are written in SQL syntax, analyzed and mapped by MOON's middleware, and then executed

using either the relational or blockchain-based environments. This approach simplifies development by eliminating the need for clients to use different query languages, frameworks, or libraries for each entity, following in the footsteps of federated databases and polystores.

Our new approach, called Inter-MOON, enhances interoperability between blockchain-based and relational databases. In this work, we present Inter-MOON, test and compare it against similar solutions, and point out its limitations and possible future works. In summary, our contributions are:

1. Exploration of interoperability of relational and blockchain databases.
2. Proposal and development of Inter-MOON, a modified version of MOON with a focus on interoperability and improved SQL grammar support, database support, query processing speed, blockchain index search speed, and blockchain asset definition.
3. Testing and comparison of MOON, Inter-MOON, and other available open-source polystore solutions.

2. Related Work

For multistores, MISO [LeFevre et al. 2014] focuses on the optimal materialization of data in heterogeneous big data environments, using both RDBMS and HDFS (Hadoop). Inter-MOON is not meant for big data workloads and uses virtual tables created on the RDBMS store to minimize processing on the blockchain component.

CloudMdsQL [Bondiombouy et al. 2016] is a cloud-based multistore system with a SQL-like language that enables querying of relational and NoSQL sources while taking advantage of each source's native functions. We apply native SQL instead, focusing on blockchain-based DBs rather than generalized NoSQL.

[Duggan et al. 2015] is one of the polystore founding works and introduces BigDAWG, which enables heterogeneous data retrieval through custom markup. It presents the concept of *islands of information* and utilizes a subset of each query language associated with a data model. The user must also choose where to materialize data, unlike in Inter-MOON. On the other hand, Polyphony-DB [Vogt et al. 2018] conceptualizes a self-adaptive system with data replication and partitioning. [Singhal et al. 2019] also presents the building blocks of Polystore++, which envisions a highly performance-oriented polystore solution.

This work utilizes a 'one-size-fits-all' approach to query languages using SQL, while polystores generally strive for mixed query languages. Moreover, none of these systems consider blockchain-based databases, which is a primary concern in this work. Finally, this work also showcases experimental results, while both Polyphony-DB and Polystore++ are vision papers. To the best of our knowledge, no known federations, multistores, polystores, or similar tools offer explicit support for blockchain-based solutions.

The original work [Marinho et al. 2020] introduces MOON, while this work introduces Inter-MOON, an extension of MOON focused on improving interoperability between relational and blockchain-based DBs through middleware using a subset of SQL syntax. While both works share a similar core architecture, some key differences exist. MOON does not support DELETE operations or multi-valued INSERT, and only offers

limited support for subqueries and aggregations, while our approach fully supports them. We also optimize blockchain asset retrieval by returning all necessary assets in a single trip, while MOON uses a simple mechanism to obtain each asset one trip at a time. Finally, the original work shows a driver system for RDB and a SQL Analyzer module, but offers very few or no implementation details, unlike this work. More details in regard to architectural modifications are present in each subsection of Section 3.

3. The Inter-MOON Approach

The Inter-MOON (Figure 1) approach is composed of three major parts: the middleware, the SQL DB, and the Blockchain DB. The middleware is further divided into several modules with separate functions. In summary, the Communicator accepts and forwards queries to the Scheduler, which enqueues requests, keeping blockchain-related operations in proper order. The SQL Analyzer and Mapper extract information from queries, which are then executed using the Blockchain or SQL Client as needed. One departure from MOON is the removal of the MOON Client module. In the original work, it is a bridge present in the application used to send the SQL and the DB, and blockchain credentials over to MOON. In Inter-MOON, the Communicator module directly receives SQL, and environmental configuration files hosted by the middleware are used to store credentials. This change is meant to eliminate the need to send sensitive information through the network alongside every request.

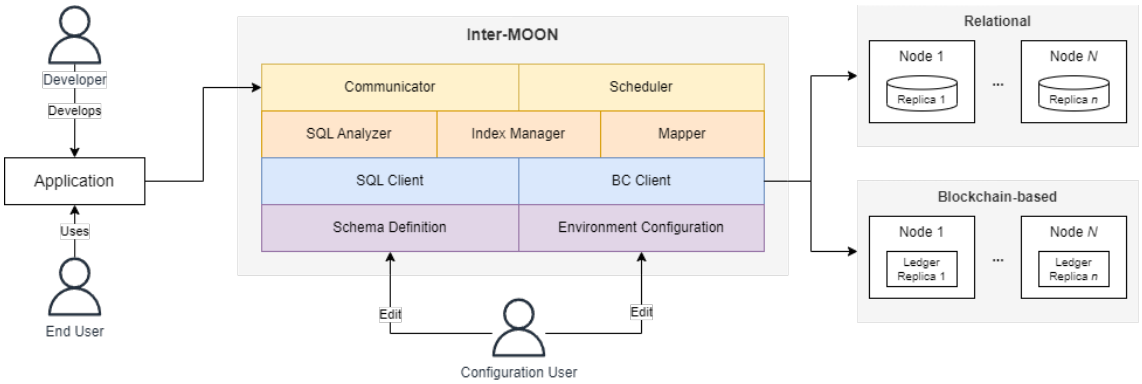

Figure 1. Overview of the Inter-MOON architecture.

Inter-MOON is not a federation but a middleware that enables cross-querying blockchain-based and relational entities through SQL. Entities are kept separate in their respective data models, and there is no data replication. Like in polystores, the autonomy of each DB is preserved, and the granularity of each store is left untouched. For example, BigchainDB is document-based, as it uses a local MongoDB instance to save transaction data.

In literature, the concept of interoperability is frequently divided into layers [Hasselbring 2000]. In this work, we adopt a broad definition of interoperability, referring to it as the overall ability of a system to comprehend and engage with others. We consider two layers: (L1) Interoperability between Inter-MOON and its clients at the application layer and (L2) Interoperability among the storage engines at the middleware layer. Additionally, we identify three qualities that compose interoperability in L2:

- **Support** - The middleware's capability to communicate with data storage engines.

- **Generality** - The middleware's capacity to understand and accurately map queries to their correct engine.
- **Efficiency** - The middleware's efficiency in finding and joining data in each storage engine.

In the Inter-MOON architecture (Figure 1), the Communicator demonstrates interoperability in (L1), allowing Inter-MOON to receive and answer requests from clients. For (L2), the SQL and Blockchain clients allow the middleware to interact with storage engines (Support), while the Mapper, Schema, SQL Analyzer, and Index Manager modules work in tandem to extract information, join data and map SQL requests to the blockchain data-model (Generality and Efficiency).

This paper focuses on interoperability in L2. Inter-MOON improves interoperability in this context by increasing the number of supported DBs (Section 3.1), the number of supported SQL keywords and operations (Section 3.2), and optimizing data retrieval (Section 3.3). The proposed approach is generally applicable as long as both the relational DB driver follows the Python DB-API interface and the blockchain-based DB has a local instance of MongoDB or MongoDB-like database for storage and querying.

3.1. Support

To improve support, we must increase the number of data storage engines supported by the middleware and the quality of the offered support. For relational DBs, our approach is reminiscent of the *Django* [Holovaty and Kaplan-Moss 2009] and *Laravel* [Stauffer 2019] designs for multiple database engine support. In short, it takes the form of a generic **D**ata**B**ase **D**river (DBD) object, which contains an adapter implementing database access functions (Figure 2). The generic DBD structure is analogous to the popular decorator design pattern for software architecture, while the drivers are to the adapter pattern.

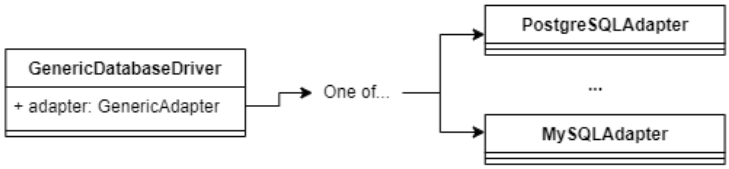

Figure 2. Simple rendition of the generic database driver.

The Inter-MOON middleware was developed using Python. Python's *DB-API*, a standard protocol for designing database access libraries, greatly optimizes the development of the generic DBD. Listing 1 shows a basic pseudo-code implementation. Connection settings can be obtained from the environmental configuration, as per Figure 1. This structure promotes maintainability, decoupling, and database support, provided adapters are developed following Python's *DB-API* specification.

```
1 import psycopg2
2
3 class GenericDatabaseDriver:
4     def __init__(self, adapter):
5         self.adapter = adapter
```

```
6
7       def connect(self, *args, **kwargs):
8           return self.adapter.connect(*args, **kwargs)
9
10  driver = GenericDatabaseDriver(psycopg2)
11  with driver.connect("config.cfg") as conn:
12      conn.execute("SELECT * FROM users;")
```

Listing 1. The Generic Database Driver basic structure. It holds an adapter object which represents the driver of a database engine.

Increasing support for blockchain-based DBs is beyond the scope of this work, but a similar approach could be used. However, the lack of a unified data model for blockchain is a current research issue [Meyer and dos Santos Mello 2022, Yuan and Wang 2018], and introduces a considerable challenge in the creation of a base interface for blockchain DB access.

3.2. Generality

Towards generality, our goal is to understand and accurately execute as many SQL DML commands as possible to promote interoperability with blockchain on a querying level. According to the *ISO/IEC 9075-1* specification, SQL-data statements can be used to perform queries and insert, update, and delete information [Melton 2016]. Consequently, the Inter-MOON middleware must be able to correctly map SQL-data statements to appropriate blockchain operations. These queries must be written using standard SQL syntax and only contain one SQL statement per request. DDL commands are not supported by default.

First, the SQL Analyzer module extracts and processes information from received queries. MOON used a simple lazy search algorithm to find the first matching M_s token (eg. *"SELECT"*) or character literal (eg. brackets, comma) of index i given a query string Q_s where $M_s = Q_s(i)$ or $M_s \subset Q_s$. A match was made when $M_s \neq \emptyset$. This was used, for example, to find the kind of operation being requested (SELECT, INSERT, UPDATE), entities (table names), attributes, or the presence of conditionals.

Inter-MOON extends that behavior by applying a smart SQL parsing mechanism, allowing it to observe statements as groups of tokens. Each group contains keywords, identifiers, functions, or conditionals, all assigned based on the token's semantic meaning inside of its group. This helps prevent ambiguity and allows easier handling of nested subqueries. For example, in the statement depicted in Figure 3, we can find conditionals by searching for any groups beginning with a WHERE clause, or extract the table name by looking for the first *Identifier* after a FROM keyword that is not a subquery.

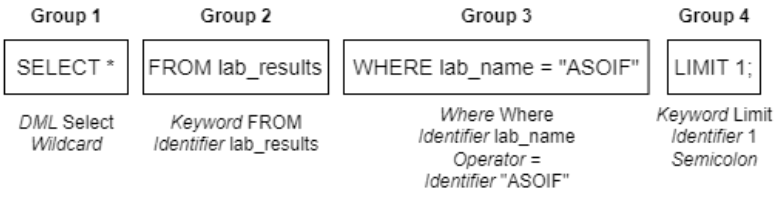

Figure 3. Inter-MOON SQL analyzer token grouping example.

As for query mapping and execution, at its core, Inter-MOON behaves similarly to MOON with some key changes. Blockchain entities are virtualized in session-available temporary tables in the RDB for read operations, and the blockchain API driver is activated for blockchain write operations. To map a SQL request, both approaches use a Schema Manager module to manage blockchain entity schema in a similar fashion to a relational data model, keeping track of its attributes.

However, unlike MOON, Inter-MOON allows blockchain entities to have a mutable schema. While blockchain should offer immutability, it is only concerning stored information, not the structure that any piece of information should have. Hence, the aforementioned virtual tables are built using both the attributes present in the queried assets and the schema. When an asset of a given blockchain entity contains an attribute not found in its schema, the attribute is ignored. When the opposite is true, the value of the missing attribute is set to NULL. This helps promote further interoperability by bringing the blockchain schema applied by MOON closer to the relational data model.

Upon receiving a query, Inter-MOON expects it to fall into one of the following scenarios: (1) SELECT, INSERT, UPDATE, or DELETE with only relational entities, (2) SELECT, INSERT, UPDATE, or DELETE with only blockchain entities and (3) SELECT with both blockchain and relational entities. For (1), Inter-MOON simply forwards the query to the RDB and sends back the response. For (2), there are separate approaches depending on the type of SQL statement, explained further below. The approach for (3) is similar to the one used in (2) for SELECT.

MOON optimized the execution of SELECT statements without WHERE by skipping blockchain entity virtualization and simply retrieving and returning every asset. This had the side effect of ignoring many SQL tokens which could be used even without WHERE. Therefore, for SELECT in (2) and (3), Inter-MOON supports all common SQL tokens applicable to SELECT statements by always virtualizing needed blockchain entities in the RDB. This includes joins, aggregations, and subqueries. The query is then executed and results are returned as tuples. To see how Inter-MOON handles optimization, seek Subsection 3.3. It is still possible for Inter-MOON to fail to run a properly formatted query if the query information extraction step fails.

For INSERT, while MOON only supports simple INSERTs, multi-valued INSERTs are also available in Inter-MOON. However, INSERT statements using a subquery are not supported in either. We consider INSERT to be categorically equivalent to APPEND, with multi-valued equivalent to several APPENDs. Attributes are extracted from SQL and used as the asset data (Figure 4). Data is stored in its given type, defined in the schema, to preserve integrity. Assets are stored in a JSON-like structure through transactions inside blocks.

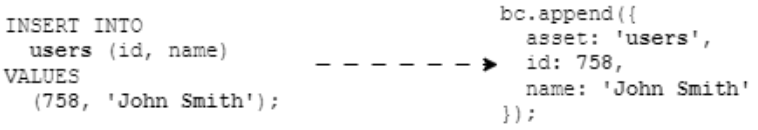

Figure 4. Rendition of the mapping mechanism for INSERT operations.

Blockchains have inherent limitations when performing DELETE or UPDATE op-

erations due to their append-only nature. In both MOON and Inter-MOON, for UPDATE operations, a new transaction is created with updated data and a pointer to the old asset. As for DELETE, MOON offers no support. Inter-MOON tackles DELETE by implementing a soft-delete mechanism in which the asset index is removed from the blockchain index table of the Index Manager module, preventing retrieval through Inter-MOON and offering a quasi-DELETE functionality. This approach ensures that blockchain consensus and immutability are maintained.

While these approaches preserve consensus mechanisms, scalability becomes a concern when dealing with a large number of assets. The scalability issue is an ongoing topic of research in the field of blockchains [Zhou et al. 2020]. Some studies aim to explore mutability in blockchains, which would further align them with relational systems [Politou et al. 2019]. However, developing a new blockchain or blockchain-based technology is beyond the scope of this work.

3.3. Efficiency

The Index Manager module tracks the index and primary id of each blockchain asset by storing them in tables in the RDB. When a query with blockchain entities is received, the middleware consults these index tables to obtain the hash of all blockchain assets of said entities for virtualization. In the original work, MOON retrieved each asset one at a time. Consequently, this process grew slower as the number of indexes increased, following a non-linear growth curve. See Algorithm 1 for an overview.

Algorithm 1 index searching algorithm in MOON.

Require: Set $I = \{i_1, i_2, i_3, ..., i_n\}$ of blockchain indexes, given $|I| > 0$.
Ensure: Set $A = \{a_1, a_2, a_3, ..., a_n\}$ of blockchain assets.

```
A ← ∅
n ← 0
N ← |I|
while n < N do
    i ← I(n)
    B ← GetAssetByIndex(i)
    A ← A ∪ B
    n ← n + 1
end while
```

$GetAssetByIndex(i)$ represents a request sent to the blockchain network to fetch an asset of index i. As the number of requests increase, the network overhead present in each request accumulate and response times grow. In a single network trip, there are 3 instances of present latency, L_{req}, L_{in} and L_{res} for the request, in-network, and response latency respectively, totaling $L_{sum} = L_{req} + L_{in} + L_{res}$ for the total network latency produced by every usage of $GetAssetByIndex(i)$.

Inter-MOON optimizes L_{sum} by executing one trip only for each query, retrieving all assets with indexes in I using a $GetAssetsByIndexSet(I)$ function and outputting $A \leftarrow GetAssetsByIndexSet(I)$. However, this approach requires more computational power from the network and the Inter-MOON middleware host to process and return all necessary assets in one trip. Further optimization can be achieved by using batch-loading

to retrieve an X number of indexes per trip, maintaining lower latency while reducing processing load. If a set or range of primary keys is specified in the query, another possible avenue for optimization is to only retrieve the assets pertaining to those identifiers, rather than all assets of each involved entity.

4. Experiments

A prototype of the Inter-MOON middleware was developed using Python 3.6.9. Three experiments were prepared in order to test and compare Inter-MOON against MOON and the BigDAWG polystore.

4.1. Comparing the performance of MOON & Inter-MOON

The goal of the first experiment was to compare the performance of MOON and Inter-MOON. Response speed was chosen as the metric, calculated using the full round-trip time taken from the moment the client sends the request to when it receives a response. The data consisted of a synthetic dataset generated using the *Mimesis*[1] library. The *Patients* entity was stored on the RDB and *Lab Results*, on the blockchain DB (Figure 5). The testing environment was comprised of a series of Ubuntu 18.04.6 VMs running on a local network (Figure 6). Table 1 describes each VM in detail. VM-1 contained instances of both MOON and Inter-MOON, only one of which was running at any time. Postgres 10.23 was used for the SQL database in VM-2, while BigchainDB 2.2 was used for the blockchain nodes in the network. Lastly, a machine running Ubuntu 22.04, 4 GB of RAM, and an Intel i5-4300 2.60 GHz CPU was used to simulate the client.

Lab Results (100 rows)	
`varchar`	uid
`integer`	patient_id
`varchar`	content_base64
`date`	datetime
`varchar`	lab_name
`integer`	lab_site
`integer`	expired

Patients (100 rows)	
`integer`	id
`varchar`	name
`varchar`	email
`varchar`	phone
`date`	birth_date

Figure 5. Entity schema used for the first experiment.

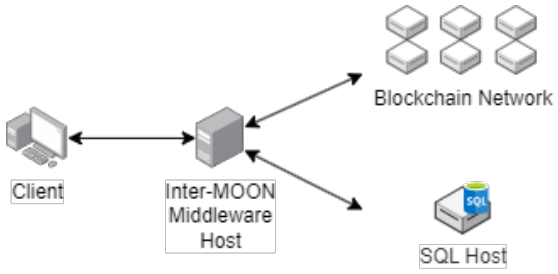

Figure 6. Testing environment.

A set of four queries (Table 2) was executed on MOON first, and then on Inter-MOON. The modified algorithm of Inter-MOON was expected to provide significantly improved response speeds in queries involving many entities while maintaining similar speeds in other kinds.

[1] `https://mimesis.name/en/master/`. Accessed: May 29, 2023

Name	Role	RAM	Disk Read & Write Speed
VM-1	Middleware host	4 GB	7.5 GB/sec & 0.8 GB/sec
VM-2	SQL database host	2 GB	6 GB/sec & 0.8 GB/sec
VM-3 . . . VM-8	Blockchain network nodes	1 GB/each	5 GB/sec & 0.4 GB/sec

Table 1. Summary of the virtual machines used in the first experiment.

Query	SQL
Q1	INSERT INTO lab_results (<...columns>) VALUES (<...values>);
Q2	SELECT * FROM lab_results;
Q3	SELECT * FROM lab_results JOIN patients ON lab_results.patient_id = patients.id;
Q4	UPDATE lab_results SET expired = 1 WHERE uid = <uid>;

Table 2. Set of queries used in the first experiment.

Results (Figure 7) show that Inter-MOON was generally much faster. In Q1, the results were in the same ballpark. In Q2 and Q3, they were about 10 times higher. In Q4, there was an improvement of about 5.5 times, instead. UPDATE-type transactions, which is the case for Q4, are more computationally expensive and latency-inducing, as they involve several trips to both database systems in order to read, update and write the updated information.

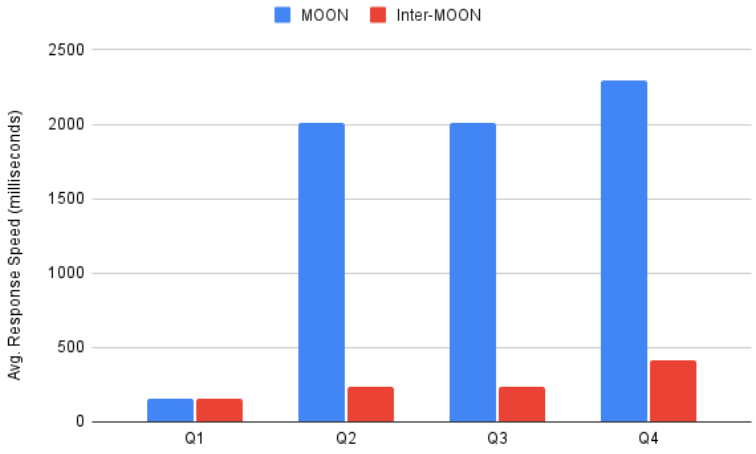

Figure 7. Graphical comparison of the Avg. Response Speed of 100 query executions between MOON and Inter-MOON.

4.2. SQL Syntax Support

In the second experiment, the goal was to evaluate Inter-MOON in regard to SQL syntax support in read operations. The TPC-H[2] decision support benchmark was used for this experiment, as it is an industry-tested standard with a wide variety of queries that showcase critical business needs. We also compared Inter-MOON against BigDAWG, to show how a similar tool fares in this regard. Both systems were populated with a 1 GB scale factor workload of the benchmark data and then 20 of its 22 queries were executed using a simple Python script. Execution results were compared against the expected

[2]https://www.tpc.org/tpch/. Accessed: May 29, 2023.

output, given by the benchmark. Q17 and Q20 were ignored due to having a long runtime. The metric was simply the factor of successful queries over the total number tested: $SupportScore(S) = Q_{success}/20$.

Inter-MOON was capable of running 18 queries, attaining a score of 0.9 while BigDAWG successfully ran 6, scoring 0.3 (Table 3). Both systems failed to run Q15, which created and then utilized a view. Inter-MOON additionally failed to run Q22, due to its usage of the *substring()* SQL function. BigDAWG demonstrated problems executing queries containing a mix of nested subqueries, aggregation, and sorting. Results indicate that BigDAWG supports only a small subset of SQL, while Inter-MOON could understand much of the standard syntax.

System	Successful queries	S Score
Inter-MOON	18	0.9
BigDAWG	6	0.3

Table 3. Support score for Inter-MOON and BigDAWG.

4.3. Cross-model Query Performance

The last experiment evaluated Inter-MOON's cross-model querying performance. Once again, the BigDAWG polystore was chosen for comparison. For the environment, the Docker[3] container setup provided by the BigDAWG project[4] was used, alongside a custom-built setup for Inter-MOON, with a container each for Inter-MOON, PostgreSQL 9.6, and BigchainDB 2.2. Both container setups were initialized in the same machine used for the client in the first experiment (See Subsection 4.1), although only one would be up at any time. A supermarket sales history dataset[5] was inserted into both BigchainDB (Inter-MOON) and Accumulo (BigDAWG). Synthetic data representing the customer of each sale was generated and inserted into the SQL databases. JMeter 5.5[6] running on a separate machine was used to monitor and execute the test plan, consisting of executing a simple JOIN query (Table 4) using data from both data models, with 1000 threads and a ramp-up time of 1000 seconds, simulating a constant stream of transactions. The evaluated metrics were average latency, standard deviation, and failure count.

```
SELECT c_name, s_unit_price
FROM customers c JOIN sales s ON c.c_id = s.c_id
ORDER BY s.s_unit_price DESC LIMIT 10;
```

Table 4. Query used in the third experiment.

Both Inter-MOON and BigDAWG demonstrated difficulty in reaching higher throughput. Inter-MOON (Figure 9) provided an overall consistent latency of 800-890 ms, with zero errors. BigDAWG (Figure 10) provided lower average latency, but much higher deviation as well as a total of 45% failure rate, slightly below half of all queries.

[3]`https://www.docker.com/`. Accessed: May 29, 2023.

[4]`https://github.com/bigdawg-istc/bigdawg`. Accessed: May 29, 2023.

[5]`https://www.kaggle.com/datasets/aungpyaeap/supermarket-sales`. Accessed: May 29, 2023.

[6]`https://jmeter.apache.org/`. Accessed: May 29, 2023.

Customers (1000 rows)	
integer	c_id
varchar	c_name
varchar	c_email
varchar	c_gender
varchar	c_phone
date	c_birth_date
varchar	c_type

Sales (1000 rows)	
integer	s_id
decimal	s_unit_price
integer	c_id
...	

Figure 8. Entity schema used for the third experiment. Only the relevant attributes from the *Sales* entity are being shown here.

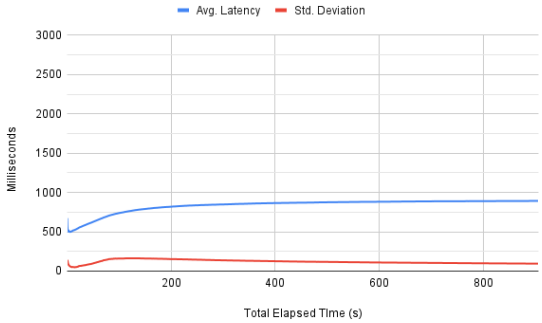

Figure 9. Inter-MOON Avg. Latency and Std. Deviation over time.

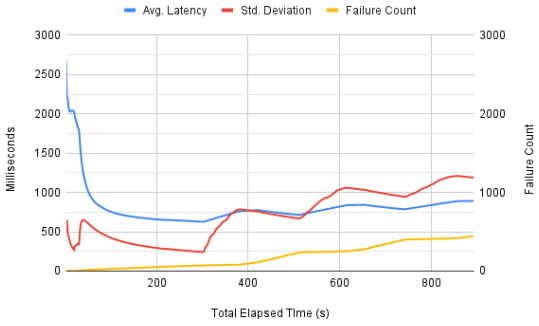

Figure 10. BigDAWG Avg. Latency, Std. Deviation and Failure Count over time.

As the flood continues, BigDAWG accumulates failures (Yellow line) in cascade, which heavily impacts latency. After some additional testing with lower ramp-up time, both tools show much worse performance, with BigDAWG increasing the failure rate even more and Inter-MOON showing exponentially higher latency. This indicates the existence of concurrency issues when obtaining data from separate data models simultaneously. However, more testing needs to be done to confirm this issue.

5. Conclusion and Future Work

In this work, we detailed our approach to providing interoperability of relational databases and blockchains by developing Inter-MOON, an extension of MOON. Inter-MOON provided average response times of 5 to 10 times faster than MOON in most tested queries, which represent common SQL DML operations. Along with increased performance, DELETE operations, as well as nested queries and aggregations, are now fully supported. Data integrity is enhanced, alongside database support with the generic database driver. Finally, while we cannot claim Inter-MOON offered better performance than BigDAWG, it showed fewer errors and surpassed it in regards to SQL syntax support.

For future works, a heavier workload of parallel processing and throughput tests, and tests with higher node counts, could provide more insight into Inter-MOON's performance and bottlenecks. Support for DDL statements could elevate Inter-MOON to a fully-featured data manipulation tool, and allowing user-defined SQL-style transactions (eg. COMMIT TRANSACTION and ROLLBACK) and stored procedures, possibly via smart contracts, could also prove fruitful. There is still room for performance optimiza-

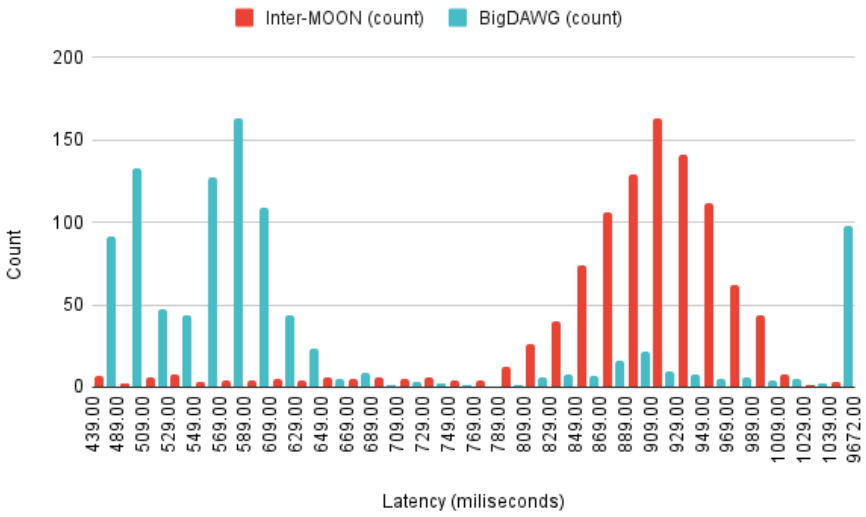

Figure 11. Histogram comparison of query latency from Inter-MOON and BigDAWG. This graph considers a bucket size of 20 and a 5% outlier rate.

tions, as mentioned briefly in Section 3.3. Concurrency and scalability issues, also discussed briefly in this work, would greatly increase interoperability if solved, but will also prove a laborious task, as blockchains themselves share similar issues.

References

[Babcock et al. 2002] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, page 1–16, New York, NY, USA. Association for Computing Machinery.

[Bondiombouy et al. 2016] Bondiombouy, C., Kolev, B., Levchenko, O., and Valduriez, P. (2016). Multistore big data integration with cloudmdsql. *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII: Special Issue on Database-and Expert-Systems Applications*, pages 48–74.

[Duggan et al. 2015] Duggan, J., Elmore, A. J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., and Zdonik, S. (2015). The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16.

[Gadekallu et al. 2022] Gadekallu, T. R., Huynh-The, T., Wang, W., Yenduri, G., Ranaweera, P., Pham, Q.-V., da Costa, D. B., and Liyanage, M. (2022). Blockchain for the metaverse: A review. *arXiv preprint arXiv:2203.09738*.

[Gervais et al. 2016] 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. In *2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 3–16, New York, NY, USA. Association for Computing Machinery.

[Hasselbring 2000] Hasselbring, W. (2000). Information system integration. *Communications of the ACM*, 43(6):32–38.

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

[LeFevre et al. 2014] LeFevre, J., Sankaranarayanan, J., Hacigumus, H., Tatemura, J., Polyzotis, N., and Carey, M. J. (2014). Miso: Souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1591–1602, New York, NY, USA. Association for Computing Machinery.

[Marinho et al. 2020] 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.

[Melton 2016] Melton, J. (2016). Iso/iec 9075-1 information technology-database languages-sql-part 1: Framework (sql/framework). *ISO/IEC*, 2016(E):9075–1.

[Meyer and dos Santos Mello 2022] Meyer, J. V. and dos Santos Mello, R. (2022). An analysis of data modelling for blockchain. In *Information Integration and Web Intelligence: 24th International Conference, iiWAS 2022, Virtual Event, November 28–30, 2022, Proceedings*, pages 31–44. Springer.

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

[Politou et al. 2019] Politou, E., Casino, F., Alepis, E., and Patsakis, C. (2019). Blockchain mutability: Challenges and proposed solutions. *IEEE Transactions on Emerging Topics in Computing*, 9(4):1972–1986.

[Singhal et al. 2019] Singhal, R., Zhang, N., Nardi, L., Shahbaz, M., and Olukotun, K. (2019). Polystore++: accelerated polystore system for heterogeneous workloads. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1641–1651. IEEE.

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

[Stonebraker and Çetintemel 2018] Stonebraker, M. and Çetintemel, U. (2018). *"One Size Fits All": An Idea Whose Time Has Come and Gone*, page 441–462. Association for Computing Machinery and Morgan & Claypool.

[Vogt et al. 2018] Vogt, M., Stiemer, A., and Schuldt, H. (2018). Polypheny-db: towards a distributed and self-adaptive polystore. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3364–3373. IEEE.

[Yuan and Wang 2018] Yuan, Y. and Wang, F.-Y. (2018). Blockchain and cryptocurrencies: Model, techniques, and applications. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(9):1421–1428.

[Zheng et al. 2018] 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.

[Zhou et al. 2020] Zhou, Q., Huang, H., Zheng, Z., and Bian, J. (2020). Solutions to scalability of blockchain: A survey. *Ieee Access*, 8:16440–16455.