electronics

MDPI

Article

Design and Implementation of Enabling SQL–Query Processing for Ethereum-Based Blockchain Systems

Jongbeen Han [1], Yunhyeong Seo [1], Sangjin Lee [2], Sunggon Kim [3] and Yongseok Son [2,*]

[1] Department of Computer Science and Engineering, Seoul National University, Seoul 08826, Republic of Korea; stompesi@snu.ac.kr (J.H.); linas@snu.ac.kr (Y.S.)
[2] Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, Republic of Korea
[3] Department of Computer Science and Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea; sunggonkim@seoultech.ac.kr
* Correspondence: sysganda@cau.ac.kr

Abstract: A blockchain is designed to establish consistent and reliable agreements in an untrusted and decentralized environment. In addition, the blockchain enables transaction processing and the creation of smart contracts. It empowers end users to execute contracts without any intermediate entities. However, there are some issues when it comes to retrieving information, such as the state and history of smart contracts and regular transactions in the blockchain. For example, in a smart contract, user-defined data structures can be used to recall the state of the smart contract for a range query, which can decrease the general performance. In addition, an external database can be required to retrieve regular transactions for range queries, which increases management costs. To achieve this, we propose a new scheme that enables SQL query operations to retrieve a smart contract and regular transaction information within the blockchain system. To achieve this, we combine an embedded relational database with an Ethereum-based blockchain system to provide the SQL query. It enables range queries on smart contracts without requiring user-defined data structures and decreases management costs for regular transactions without any external database. We implement the proposed blockchain system on *quorum*, which is an Ethereum-based blockchain system. Also, we evaluate the proposed system using a synthetic benchmark. The performance of retrieving smart contract data is improved by up to approximately 22×, with low memory usage compared with the existing system. Moreover, the proposed system demonstrates a similar search performance to the existing system, even when considering external databases in regular transactions.

Keywords: blockchain; SQL query; Ethereum; SQLite

Citation: Han, J.; Seo, Y.; Lee, S.; Kim, S.; Son, Y. Design and Implementation of Enabling SQL–Query Processing for Ethereum-Based Blockchain Systems. *Electronics* **2023**, *12*, 4317. https://doi.org/10.3390/electronics12204317

Academic Editor: Bahman Javadi

Received: 19 September 2023
Revised: 10 October 2023
Accepted: 16 October 2023
Published: 18 October 2023

1. Introduction

A blockchain is a distributed ledger and data management technology known for data integrity [1–3]. The blockchain has become famous for its ability to distribute data, such as state and transaction information, to each node, supporting the Byzantine fault tolerance (BFT), and enabling P2P cryptocurrency transfers and contracts without any intermediate entities. In particular, a smart contract, which is the only function that blockchain provides, can be consistently executed in a network of mutually distrusting nodes without a trusted third party. Blockchain is now widely used because of the benefits of these features. Applications include IoT applications, [4], energy markets [5], the domain name service using blockchain [6], etc.

There is a need for various blockchain data analytics to optimize the performance of blockchain systems and predict the flow of the cryptocurrency market. We focus on Ethereum [2,7], which is a popular permissionless blockchain that had more than 10 TB of data as of September 2022 [8]. Ethereum-based blockchain systems have some challenges in managing all the data, despite their many benefits. Ethereum stores these data in the form of a trie structure. States generated by transactions are stored in the state trie by Ethereum.

Also, transactions are stored in the transaction trie [7,9]. Nodes within the system verify whether they are valid or not via the state and transaction trie. Throughout this process, all the data, such as the state value of the smart contract, are stored in LevelDB.

LevelDB is a type of key–value store known for its great performance in sequential reads and writes. However, LevelDB (i.e., key–value database) does not support SQL query operations, such as range queries. Furthermore, LevelDB provides low retrieval performance when handling large amounts of data. Therefore, to handle these issues, existing systems use user-defined data structures for smart contracts and external relational databases for regular transactions to retrieve a range of data [8,10]. However, user-defined data structures in a smart contract can decrease the overall performance. Also, an external database can increase the management cost in the blockchain system.

Previous studies [8,11,12] have proposed new select query operations aimed at enhancing search performance in Ethereum-based blockchain systems. Etherscan [8] introduced the select query operation that retrieves blockchain information, such as transactions, tokens, addresses, and prices, using an external database system. Pratama et al. [11] have enabled users and developers to easily access blockchain data by adding three main query functions (retrieval query, aggregate query, and ranking query). The Graph [12] provides services for querying blockchain data by continually scanning all of the events that Ethereum contains. Our study aligns with these studies [8,11,12] in terms of investigating the search performance issues within a blockchain system. In contrast, we concentrate on retrieving a range of data from both smart contracts and regular transactions using an embedded relational database within a blockchain system to improve the search performance and reduce management costs.

To achieve a higher search performance, we combine an embedded relational database with an Ethereum-based blockchain system. It enables range queries for smart contracts and regular transactions without any external database or user-defined data structures. To achieve this, we propose two managers, i.e., register and query managers. The register manager manages smart contracts and regular transactions required by users for retrieving the range of data. The manager registers the smart contracts and regular transactions into the embedded relational database to perform `SELECT` query operations and eliminates the smart contracts and regular transactions that are no longer needed. The query manager classifies queries into different categories, such as single, range, and conditional queries, to search for states within smart contracts and regular transactions. By doing so, we enable each blockchain node to use each embedded relational database (i.e., SQLite [13]) instead of using a user-defined data structure and an external database within a blockchain system, as seen in previous studies [8,14]. We implemented the proposed scheme based on *quorum*, which is an Ethereum-based blockchain system, and evaluated the proposed system via a synthetic benchmark. The experimental results show that the proposed system can improve the search performance of smart contract data, with up to a 22× increase compared to the existing system, while maintaining low low memory usage. In addition, our system shows a similar search performance to the existing system that employs an external database for regular transactions.

In previous work [15], we focused on serving SQL query operations for smart contracts to enhance the performance of retrieving smart contract information. In this article, we extend our scheme to provide SQL query operations for regular transactions.

Our contributions to this work are as follows:

- We analyze the performance of existing blockchain systems with select operations.
- We propose a scheme that enables SQL query processing to decrease the management overhead and increase search performance on a blockchain.
- We show that the proposed system improves the search performance of smart contract data.
- We show that our scheme reduces the management costs for regular transactions without any external database.

The rest of this article is organized as follows: Section 2 discusses the background and motivation. Section 3 presents the design and implementation of our proposed system. Section 4 shows the evaluation results. Section 5 discusses the varied applications of our model, acknowledging limitations, and addressing future improvements. Section 6 discusses the related work. Section 7 concludes this article.

2. Background and Motivation

2.1. Regular Transactions in Ethereum-Based Blockchain Systems

Regular transactions in the Ethereum-based blockchain system are cryptographically signed instructions from accounts. An account will initiate a transaction to update the state of the account in the Ethereum network. Thus, a regular transaction involves transferring Ethereum-based assets from one wallet address to another wallet address. For example, if Bob sends Alice 1 ETH, Bob's account must be debited and Alice's must be credited. This state-changing action takes place within a transaction [16]. The submitted regular transaction includes the following information: recipient, signature, the amount of ETH to transfer from the sender to the recipient, data, gas limit, the maximum amount of gas, etc.

2.2. Smart Contracts in Ethereum-Based Blockchain Systems

Due to its popularity and greater generality compared to other blockchain systems, this paper focuses on an Ethereum-based blockchain system [17]. A smart contract is a important program within the Ethereum blockchain system; it was initially proposed by Vitalik Buterin [2]. Smart contracts are carried out correctly on the blockchain system due to the consensus protocol [18]. In addition, a smart contract can encode any set of rules represented in its programming language (e.g., solidity). A smart contract, for example, may make transfers and apply various types of business logic, including financial instruments, insurance, real estate, and medical applications, among other things, on the blockchain.

2.3. Key–Value Store

An Ethereum blockchain system maintains three tries (i.e., world state trie, transaction trie, and transaction receipt trie) [7,19]. The Ethereum blockchain system's world state trie includes the user and smart contract states; the transaction trie records transactions that alter states of users and smart contracts; and the transaction receipt trie keeps the results of completed transactions. These three tries are stored in a key–value database that is designed for storing, retrieving, and managing data (i.e., LevelDB [20]). The key–value store performs well in sequential reads and writes and offers quick read and write operations for each key. However, the key–value store can be slow when performing operations on multiple keys because it lacks range query operations without the relational data model.

2.4. Motivation

By using smart contracts, users can implement a wide range of business logic, such as distributed applications (DApps), on the blockchain.

Smart contracts frequently require the ability to obtain data in a range (such as purchase history and sales history). For example, OpenSea, famous for its NFT market, shows that 121 million people visit the site every month [21]. However, as shown in Figure 1, there are only 974,220 transactions per month. In other words, the frequency of checking details in each wallet is higher than the frequency of transactions. However, a key–value store in the blockchain is challenging since it does not provide range query processing. In this instance, a smart contract can often retrieve the range of data in one of the following two ways:

- Using a user-defined data structure. One method for retrieving the range and conditional data of a smart contract involves a user-defined data structure, which can be used in the smart contract. For example, when a user requests states within a range, the blockchain system retrieves the requested states, saves the results in the data

structure (e.g., an array or map), and gives it to the user. Even though this method can provide a range of data, it can decrease the overall performance. The reason is that it requires the Ethereum virtual machine (EVM) to be loaded, and it runs the smart contract by reading states from the database one by one instead of using a range query.

- Using an external relational database. To retrieve the range and conditional data of smart contracts and regular transactions, we can use an external database within a blockchain system. To serve transaction history, DApps usually collect all transactional data occurring on the blockchain, with a specific focus on transactions taking place through the DApp's smart contract. After that, the collected transaction event history is stored in an external database. Finally, when a user queries a transaction on the DApp, the DApp then retrieves the relevant data from the external database and provides the results to the user. The external database enables range query processing, resulting in higher search performance. However, this method can require additional management in the blockchain system. For instance, we should manually carry out many tasks when a blockchain node connects to an external database system (e.g., setting database APIs, constructing a database, and making tables). However, in our proposal, each node within a blockchain system has an embedded database system to automatically carry out these processes. By doing this, the user may quickly connect to the database system and leverage the embedded database system's range query processing.

In regard to services provided after moving to an external database in this way, there is a possibility that the Oracle problem may occur in the situation in which data are transferred [22]. In other words, the Oracle problem refers to the inability of confirming the correctness of the data collected by oracles, which can lead to malfunctions and deliberate tampering. And more than one provider is needed to provide these services.

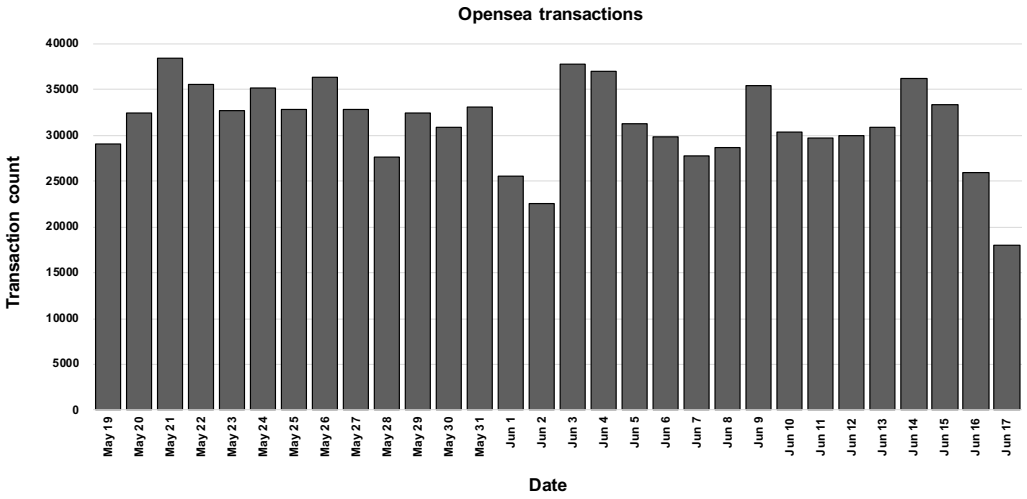

Figure 1. OpenSea transaction statistics, May–June 2023.

There are multiple challenges associated with providing services that rely on blockchain data being transferred to external databases. Firstly, it is critically important for the service provider to maintain data consistency between the blockchain and the external database. Each new transaction that occurs on the blockchain requires immediate updates to the external database, which can introduce significant technical complexities. Secondly, the use of external databases, which are inherently centralized systems, introduces significant security concerns. It means that there is a possibility that the Oracle problem [22] may occur in the situation in which data are transferred. Lastly, the fundamental design of blockchain technology promotes data that are public and accessible to all. However, this principle could be compromised when using an external database. In such a case, the company operating the database assumes full ownership, which could limit data accessibility. This poses a potential contradiction to the inherent decentralization ethos of blockchain

technology. To solve this problem, we would like to introduce the integration of SQL within the blockchain to facilitate efficient range queries. This integration not only preserves the authenticity and integrity of data but also obviates the necessity for external databases, mitigating the Oracle problem and associated management complexities.

3. Design and Implementation

To improve the performance of search operations and reduce management costs, our goal is to enable to retrieval of the range of data and conditional data in the blockchain system. To achieve this, we combine an embedded database system into the blockchain system that allows SQL query operations for data ranges within the blockchain. This eliminates the need for constructing and managing an external database and an additional user-defined data structure.

Figure 2 shows the existing system (with an external database) system and the proposed scheme with an embedded database system integrated into the blockchain system. The key difference is whether the database is integrated with the application or operates solely outside the application [23]. As shown in Figure 2a, to maintain an external database, it has to operate as an additional application, which incurs additional overhead. In contrast, as shown in Figure 2b, the proposed embedded database is integrated as part of an application and stores the data of the users without additional management of a separate database server [24]. Also, when using the external database for retrieving regular transaction information, the blockchain system has to rely on a third entity (e.g., a DBMS server). Meanwhile, when using the embedded database, the blockchain system does not need to rely on a third entity.

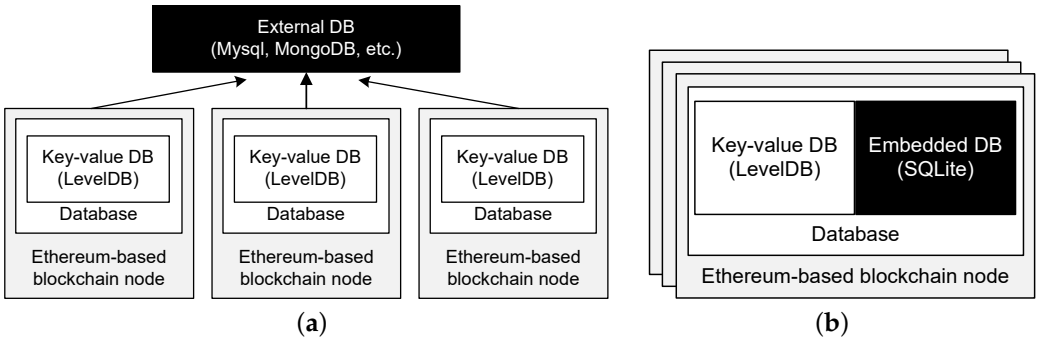

Figure 2. Classification of the database on an Ethereum-based blockchain node. (**a**) External database system; (**b**) embedded database system.

As an embedded database provides these benefits, we selected the embedded database and integrated it into the blockchain node. Therefore, through the embedded database, our scheme can enable a decentralized architecture of the blockchain system and allow SQL query operations for users to retrieve range and conditional data. For the embedded database, we utilize SQLite, which is a popular relational database system and is widely used as an embedded database for existing systems. SQLite is an open-source and lightweight database management system (DBMS). It manages data into a single file and operates in an embedded manner without maintaining a separate database server. So, via SQLite, we integrate a database system into a client application [13].

In addition to the embedded database, we also maintain the user data in an existing key–value database to utilize the advantage of the key–value database. For example, when a key–value database finds a block or a transaction and performs the validation of the transaction or block, it provides better performance in searching. Thus, by maintaining the existing key–value database, our system can provide the same level of consistency in existing blockchain systems.

3.1. Design

Figure 3 shows the system overview (architecture) of the existing and proposed systems. In the existing system, as presented in Figure 3a, when the application starts and transmits the service interface layer to the blockchain system, it obtains the transaction from the application layer. Then, it is sent from the service interface layer to the transaction layer. After that, the transaction layer classifies if the transaction type is a smart contract transaction or a regular transaction. If the type of received transaction is a smart contract, the smart contract transaction is processed on the Ethereum virtual machine (EVM) by the smart contract manager. After processing the transaction, it is then validated by the smart contract manager. If the transaction is valid, it is stored in `mempool` as a pending transaction.

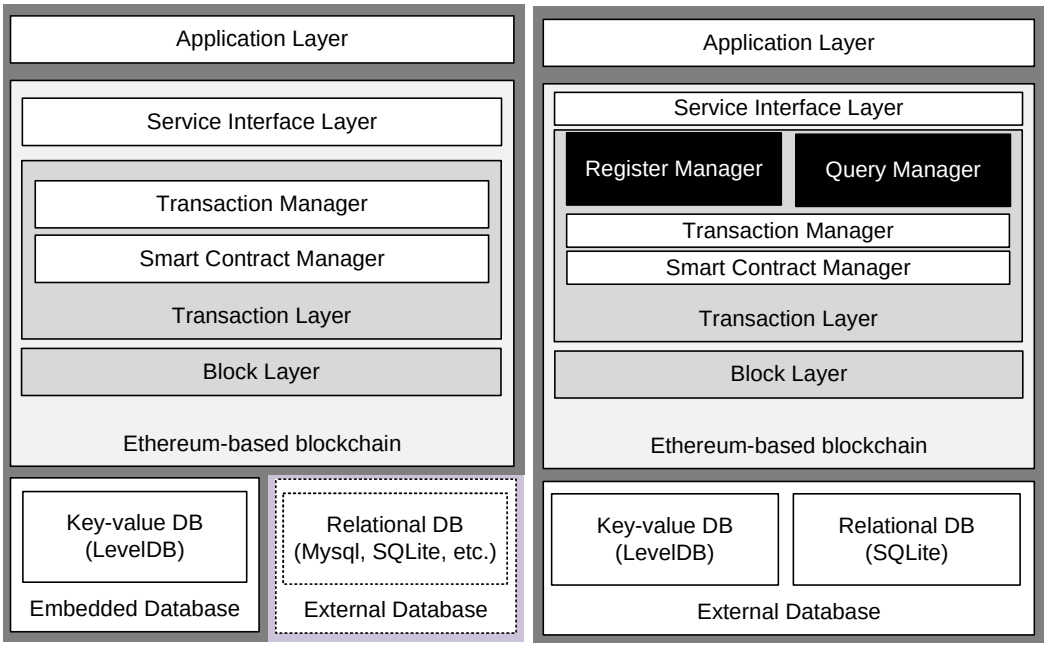

Figure 3. System overview. (**a**) Existing system; (**b**) proposed system.

On the other hand, if a transaction is a regular transaction, the transaction manager verifies the regular transaction by checking the sender's balance recorded in the transaction. If the transaction is verified as a valid transaction, the transaction manager writes the transaction as a pending transaction in a transaction pool (`mempool`). Thus, the transaction pool will maintain only the valid and pending transactions.

When a block has to be generated, the block layer generates a new block, which includes the pending transactions from the transaction pool. The pending transactions include both types of transactions (regular and smart contracts) in the `mempool`. To generate a new block, the pending transactions that will be included in the new block are performed and the states of the transactions are updated based on the results of executing the transactions. Finally, the block and its transactions are written to the storage using a key–value database (i.e., LevelDB). Because the blockchain constructs a database as a key–value database, it is impossible to search for conditions or ranges in the existing blockchain. Therefore, a separate database has been established for retrieving the range and conditions of blockchain data (e.g., smart contracts and regular transactions).

In the proposed system, as shown in Figure 3b, we propose an additional register manager and a query manager, which are used to modify the block layer. The managers handle both smart contract transactions and regular transactions in the proposed scheme. The register manager registers the smart contracts and wallet addresses for regular transactions.

These registered transactions are required to read the range of data from the application and service interface layers. Then, similar to the existing system, the transaction manager and the smart contract manager perform the transactions, validate the transaction results, and store the transactions in `mempool`. When a new block needs to be generated, the block layer generates a block using pending transactions from `mempool`, which can be of smart contracts and regular transactions. However, in addition to generating the block, our proposed block layer identifies if the pending transaction is part of the smart contracts or wallet addresses that are previously registered by the register manager. If that is the case, the block layer additionally stores the associated transaction to the embedded relational database (e.g., SQLite). By storing these transactions in the relational database, the query manager can perform SQL queries when a user requests the retrieval of range and conditional data from smart contracts and regular transactions.

3.1.1. Register Manager

To reduce the overhead of managing transactions in the embedded database, we only register the smart contracts and wallet addresses that are requested rather than all the transactions. This allows our scheme to track the transactions that are stored in the relational database (i.e., SQLite), which are also associated with the registered smart contract and wallet address.

To enable this, we propose the register manager in the proposed scheme as shown in Figure 3b. In terms of smart contracts, we support APIs such as `registerContract-Address`, which registers a smart contract requested in the register manager.In addition, when the smart contract register request is received by the register manager with the `registerContractAddress` API call, the address of the requested smart contract is stored in SQLite. Then, the address is used to identify each smart contract. Also, in terms of regular transactions, we create APIs in the register manager, such as `registerWalletAddress`, which registers the wallet address. Similar to a smart contract address, the wallet address is stored in SQLite, which is used to identify each wallet.

After the registration, as shown in Figure 3b, the results of transactions from the registered smart contract and wallet address are stored in SQLite by the block layer. Through this registration process, the result of the transaction can be retrieved using an embedded database, enabling the range and conditions during the retrieval. We will further explain the process of storing the results of smart contracts and regular transactions in SQLite in Section 3.1.3. Meanwhile, if the result of the transaction is no longer needed by the user, the corresponding smart contract and wallet address can be removed using API calls, such as `removeContractAddress` and `removeWalletAddress`. After the execution of APIs, the smart contract and wallet address are no longer tracked, reducing management overhead.

3.1.2. Query Manager

The query manager handles an SQL query, which can be used to retrieve smart contracts and regular transactions by using the embedded relational database system (i.e., SQLite). To enable this, we support APIs such as `getData`, which enables the retrieval of the smart contract and its data. After receiving the retrieval request through `getData`, the query manager proceeds to fetch data from both the smart contract and regular transaction stored in SQLite. Also, to prevent the unauthorized modification of data stored in the database, only the `SELECT` query is executed. Other queries that modify the data (i.e., `INSERT`, `UPDATE`, and `DELETE`) are filtered and ignored to prevent modifying the data from outside. Thus, with the proposed database, the query manager can handle range or conditional query operations by calling `getData`. If there are unregistered smart contracts or wallet addresses, the query is ignored and the query manager does not perform SQL operations.

3.1.3. Block Layer

In the proposed system, the modified block layer stores blocks—with smart contracts and regular transactions—to the original key–value database (i.e., LevelDB) and the pro-

posed relational database (i.e., SQLite). This is to support original blockchain operations through a key–value database while supporting range and conditional retrievals through the database. To achieve this, the proposed block layer performs a two-level check operation. Initially, the block layer determines the type of each transaction, distinguishing between a smart contract transaction and a regular transaction. For smart contract transactions, the block layer checks its association with the already registered smart contracts—a task managed by the register manager.

On the other hand, if the transaction is a regular transaction, the block layer checks the association with the wallet address that was registered previously by the register manager. If the transaction has been registered through the register manager, the block layer proceeds to store the transaction in both LevelDB and SQLite. However, if it has not been registered, the transaction data are exclusively stored in LevelDB, as the key–value database is responsible for preserving the integrity of the Ethereum-based blockchain functionality, while the embedded relational database is utilized for retrieving the range of data within the smart contract. In regard to the remove request via `removeContractAddress` or `removeWalletAddress` from the register manager, it deletes all data associated with the smart contract or wallet address from SQLite.

3.2. Implementation

As shown in Figure 4, we modified the *quorum* to implement our scheme. We chose *quorum* as it is a widely used Ethereum-based distributed ledger protocol and supports the privacy of transactions and contracts. In addition, it supports new consensus algorithms, such as raft and Istanbul BFT, for private blockchains. To modify the applications, we utilize the web3.py python library. web3.py and web3.js are libraries that enable interactions with Ethereum-based blockchain nodes, whether they are local or remote, by making JSON-RPC calls and utilizing either an HTTP or IPC (inter-process communication) connection. Furthermore, we employ an SQLite3 library based on the Go programming language (golang). This SQLite3 library functions as a driver that conforms to the integrated *database/sql* interface within `golang`.

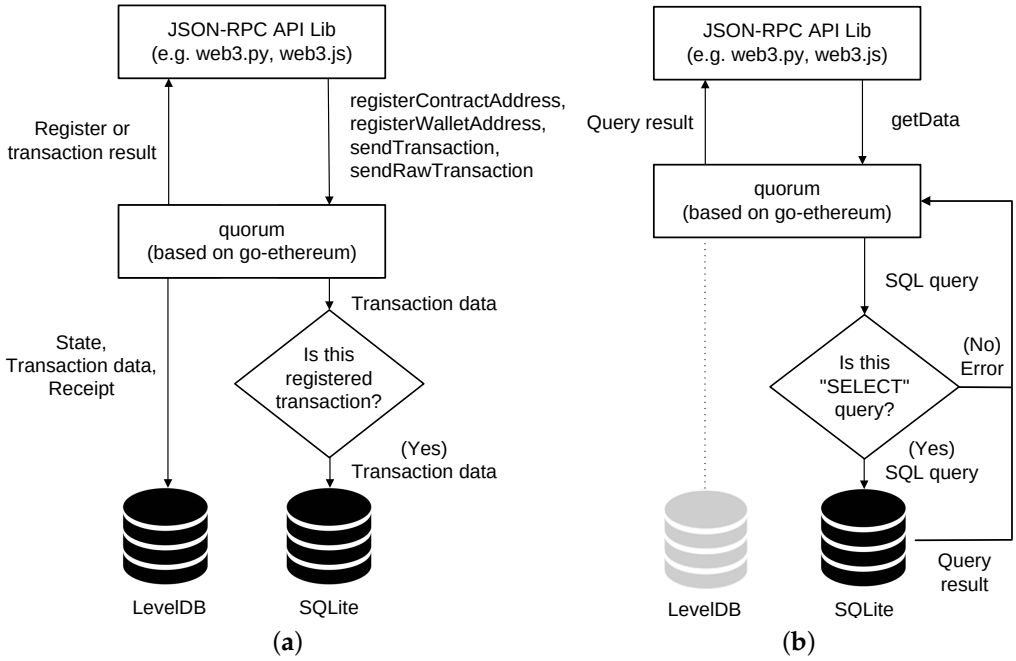

Figure 4. Process overview. (**a**) Register & Store process; (**b**) Query process.

As shown in Figure 4a, to retrieve range and conditional data in smart contracts and regular transactions, a user initiates the registration of a smart contract address by utilizing `registerContractAddress`. Upon receiving this request, the register manager proceeds to

establish a dedicated table within SQLite that aligns with the smart contract's structure for storing transaction outcomes. Once the table creation is successfully completed, the register manager proceeds to record the smart contract address within the SQLite database. In addition, to retrieve range and conditional data in a regular transaction, a user requests the registration of a wallet address via `registerWalletAddress`. At that time, unlike a smart contract, the register manager stores the wallet address at the pre-defined table in SQLite without creating a table.

After registering the smart contract, the smart contract and regular transactions initiated by `sendTransaction` and `sendRawTransaction` from a user are received and processed via *quorum*. When the block and its transactions are stored by the block layer, the block layer checks whether each transaction in a block should be stored in SQLite for the range query, according to their registration via `checkIsTrackedContract`. If the block layer stores transaction data in SQLite, the block layer stores the transaction data in both SQLite and LevelDB; otherwise, the block layer only stores the transaction data in LevelDB. For example, in smart contract transactions, the block layer stores the transaction data of the smart contract performed through EVM in SQLite via `insertValue(smart_contract_address, params)`. In regular transactions, the block layer stores regular transaction data in SQLite via the `transfer (source address, destination address, amount)`.

Figure 4b shows the query processing procedure in the proposed system. When retrieving a range of data in the smart contract, JSON-RPC, such as `getData(SQL_select_query)`, is utilized. In that case, as presented in the figure, only the `SELECT` query is performed in SQLite, and other SQL queries, such as `INSERT`, `UPDATE`, and `DELETE`, are eliminated via a regular expression. If the SQL query syntax is incorrect or if there are any other issues with the queries, the query manager returns an error message. Otherwise, the query manager provides the data results corresponding to the syntax.

3.3. Usage

Figure 5 shows how to retrieve regular transactions in the blockchain. To retrieve the transaction information over a specific period, a user can initiate a query using user A's wallet address and the period. After that, the blockchain returns the result of the query request. The result is the transaction information related to user A in the period. For example, user A sends 0.1 ETH to user C, user A receives 0.2 ETH from user B, and user A sends 0.1 ETH to user D. In addition, user A can retrieve transactions with a specific user. To achieve this, user A enters their own address and another user's (user B) address. After that, the blockchain returns the result of the query request. The result is the transaction information related to user A and user B. For example, user A sends 0.1 ETH to user B, user A sends 0.2 ETH to user B, and user A sends 0.1 ETH to user B. In this case, the query manager checks whether the query types are related to regular transactions. After that, the query manager looks up the regular transaction table in SQLite, checks the search conditions, and responds to the query.

On the other hand, in an existing system (with an external database), it is necessary to build a separate database by synchronizing data information from the blockchain. The reason is that the database used in the blockchain is a key–value store; therefore, it is hard to retrieve conditions or ranges of data. So, it receives blockchain data at the time the block is generated and stores the data in a separate database, such as MySQL. Afterward, the method of retrieving general transactions is similar to the proposed system, i.e., one initiates a query to a separate service using an external database and then receives the results accordingly.

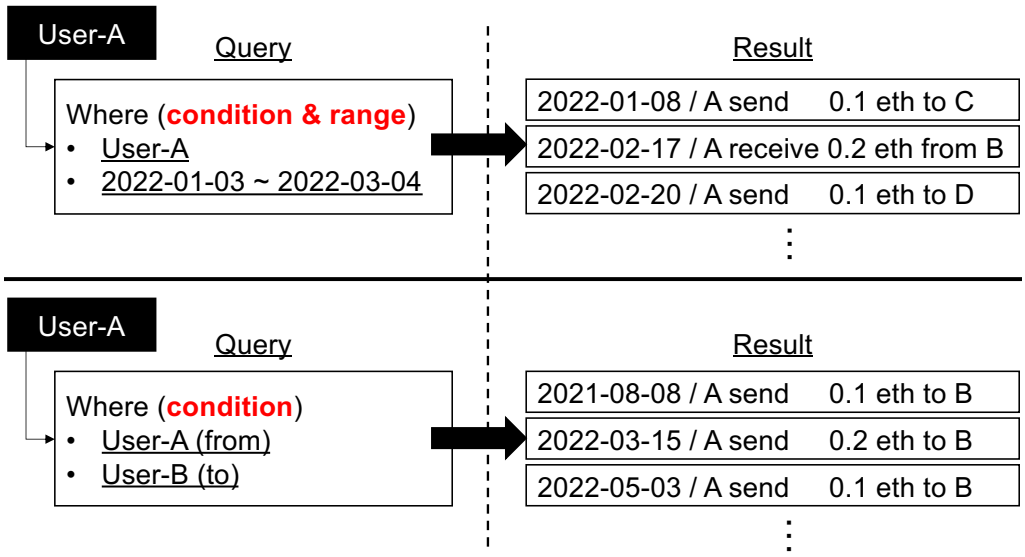

Figure 5. Retrieval of regular transactions in the blockchain.

4. Evaluation

4.1. Experimental Setup

To evaluate the proposed scheme, we utilize five machines, each with a 32-core CPU. Each has two Intel Xeon E5-2683 processors, 64 GiB DRAM, and operates on the Ubuntu 16.04.5 LTS distribution with Linux kernel 4.4.0. In terms of language, we utilize *golang* 1.10.7, *python* 3.7, and *jmeter* [25]. In terms of benchmark, we use a synthetic benchmark. The smart contract scenario in the synthetic benchmark represents an energy usage storage application where a user records electric energy consumption every 15 min, with the data storage spanning a total duration of one year.

We devised a smart contract for the evaluation. It consists of a variable and an array of user-defined data structures for each user. The variable stores a timestamp that records the most recent update made by a user, while the array holds the actual energy consumption data. The range to be retrieved in the smart contract is calculated as follows:

$$\begin{aligned} startIndex &= MNE - (LSTS - STS)/c - 1 \\ endIndex &= MNE - (LSTS - ETS)/c - 1 \\ return\ &array[startIndex : endIndex] \end{aligned} \tag{1}$$

`MNE` represents the maximum number of entities in the smart contract during one year, which is 35,040 in our evaluation scenario. `STS` and `ETS` represent the start and end timestamps provided by a user, specifying the range of data to be retrieved, respectively. `c` is a constant that denotes the duration of the storage cycle in seconds. We set `c` as 900 s to convert 15 min to seconds. Using this smart contract, we evaluate the existing and proposed systems in terms of `INSERT` and `SELECT` performances.

In regular transactions, the regular transaction scenario of the benchmark involves sending and receiving cryptocurrency between users. As it is a basic function of the blockchain, we exploit the internal function without writing a separate contract. In addition, to compare the existing system (with an external database), we built the blockchain explorer, which stores synchronized data of blocks and transactions from the blockchain with the external database. As the external database, we use MySQL since MySQL is typically used in blockchain as an external database. In our test, we use JMeter v5.4.1 to make requests to a database with 50 threads. The client's environment from which the requests are sent is a MacBook Pro 2019 with a 2.3 GHz 8-core Intel Core i9, 32 GB 2667 MHz DDR4. We run each experiment with 10 measurements and report the average.

4.2. Performance Results

4.2.1. SELECT Performance

Figure 6a presents the `SELECT` performances of existing and proposed systems in smart contracts. For experimental parameters, we set the number of threads as 1 and the number of entities as 10,000, 20,000, 30,000, and 35,040. Thus, the performance results according to the number of entities are shown. As shown in the figure, the proposed system improves the performance by up to 16.9×, 16.5×, 15.8×, and 15.7×, compared with the existing system, where the number of entities is 10,000, 20,000, 30,000, and 35,040, respectively. The proposed system provides range query operations, while the existing system has to retrieve the data one by one without range queries. Therefore, it shows better performance than the existing system. The execution times of the existing and proposed systems increase as the number of entities increases. This is because, as the number of entities grows, both the data retrieval time and the data volume increase. Moreover, these findings indicate that the execution time of the current system escalates quickly, while the execution time of the proposed system experiences a more gradual increase as the number of entities increases.

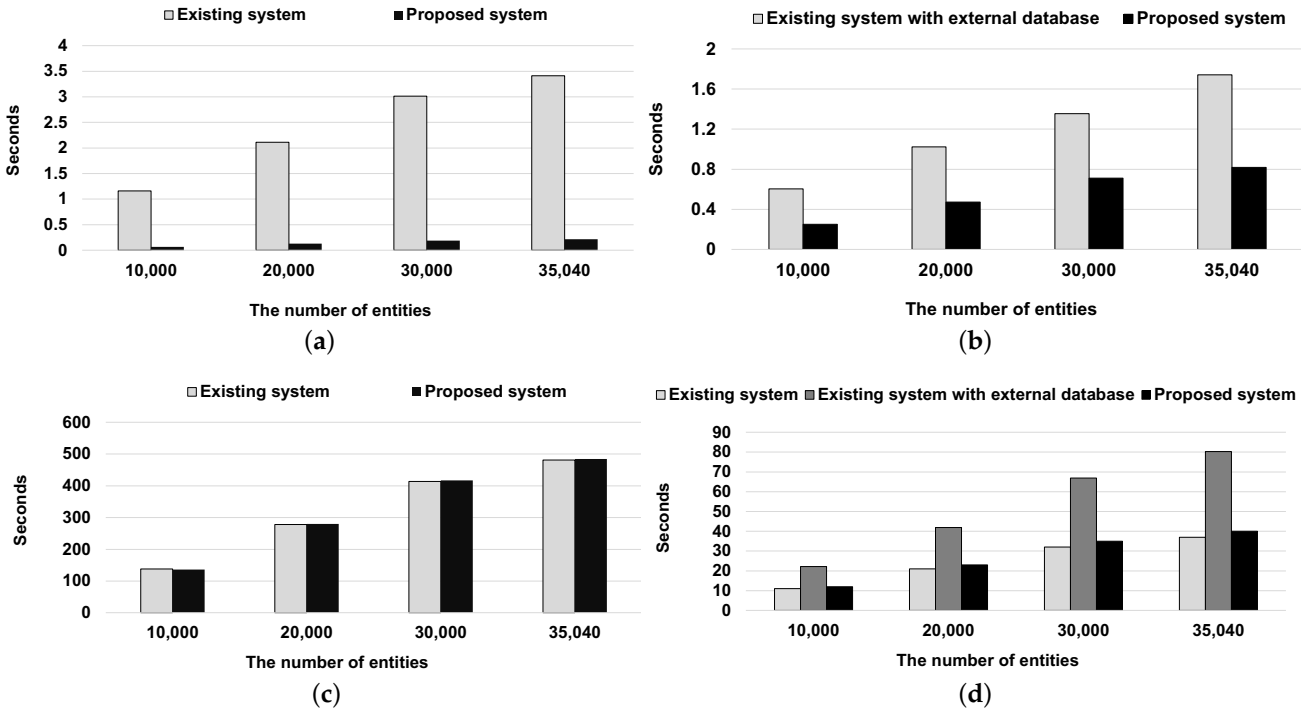

Figure 6. Performance results with different entity numbers. (**a**) Execution times of select operations in smart contracts; (**b**) execution times of select operations in regular transactions; (**c**) throughput of insert operations in smart contracts; (**d**) throughput of insert operations in regular transactions.

Figure 6b presents the `SELECT` performance comparison between the existing system with the external database and the proposed system in regular transactions. For experimental parameters, like a smart contract experiment, we set the number of threads to 1 and the number of entities as 10,000, 20,000, 30,000, and 35,040. As shown in the figure, the proposed system improves the performance by up to 2.40×, 2.16×, 1.90×, and 2.12× compared to the existing system (with an external database), where the number of entities is 10,000, 20,000, 30,000, and 35,040, respectively. This is because our proposed scheme combines SQLite for regular transactions, which is faster and simpler than the existing system with MySQL.

4.2.2. Insert Performance

Figure 6c presents the `INSERT` performances of existing and proposed systems. The experimental parameters used in the `INSERT` evaluation are the same as those used in the `SELECT` evaluation. The execution time of the proposed system increases by up to 1.013×, 0.994×, 0.992×, and 0.993× compared with the existing system, where the number of entities is 10,000, 20,000, 30,000, and 35,040, respectively. This result shows a minor overhead. In terms of throughput, the proposed system provides 73.3, 71.4, 71.9, and 72.3 entities/s, and the existing system provides 72.3, 71.8, 72.4, and 72.7 entities/s when the number of entities is 10,000, 20,000, 30,000, and 35,040, respectively. These results demonstrate that the throughput of the `INSERT` operations of the proposed system is almost the same as that of the existing system, although we support additional embedded relational databases (i.e., SQLite) for fast retrieval.

Figure 6d presents the `INSERT` performance of the existing system with the external database and the proposed system. The experimental parameters used in the `INSERT` evaluation are the same as those used in the `SELECT` evaluation. The proposed system increases the execution time by up to 1.090×, 1.095×, 1.093×, and 1.081× compared with the existing system, where the number of entities is 10,000, 20,000, 30,000, and 35,040, respectively. This is because the proposed system stores additional data at the SQLite in the blockchain. Meanwhile, the proposed system decreases the execution time by up to 1.84×, 1.82×, 1.91×, and 2.00× compared with the existing system (with an external database), where the number of entities is 10,000, 20,000, 30,000, and 35,040, respectively. This is because the existing system (with an external database) stores additional data at the external database outside the blockchain. The external database requires additional synchronizing operations with the blockchain. However, the proposed system stores the data in the embedded database (SQLite) inside the blockchain without synchronizing the operations. Therefore, the result demonstrates that the proposed scheme shows a better performance than the existing system with an external database.

Note that all the experimental results by regular transactions in Figure 6d show better performance than those by the smart contract in Figure 6c. Because the smart contract transaction should be executed by the smart contract function by the Ethereum virtual machine (EVM), it takes a longer time than a regular transaction.

4.3. Impact on the Number of Threads

Figure 7a,b present the performance in `SELECT` operations with different thread numbers. As shown in the figure, in all systems, the execution time increases as the number of threads increases. In terms of the smart contract, the proposed system improves the performance by up to 21.1×, 22×, 18.7×, 17.4×, 18.4×, and 15× compared with the existing system, where the number of threads is 1, 2, 4, 8, 16, and 32, respectively. In particular, in the existing system, the execution time experiences a sharp increase when the number of threads exceeds 16. Additionally, the execution time of the existing system is noticeably higher, reaching approximately 8 seconds more than that of the proposed system.

In terms of regular transactions, the proposed system improves the performance by up to 2.12×, 2.31×, 4.02×, 5.46×, 4.87×, and 5.34× compared with the existing system (with an external database), where the number of threads is 1, 2, 4, 8, 16, and 32, respectively. In particular, in the existing system (with an external database), the execution time increases rapidly when the number of threads is beyond 16. The execution time of the existing system with the external database is higher by up to about 21 s compared with that of the proposed system.

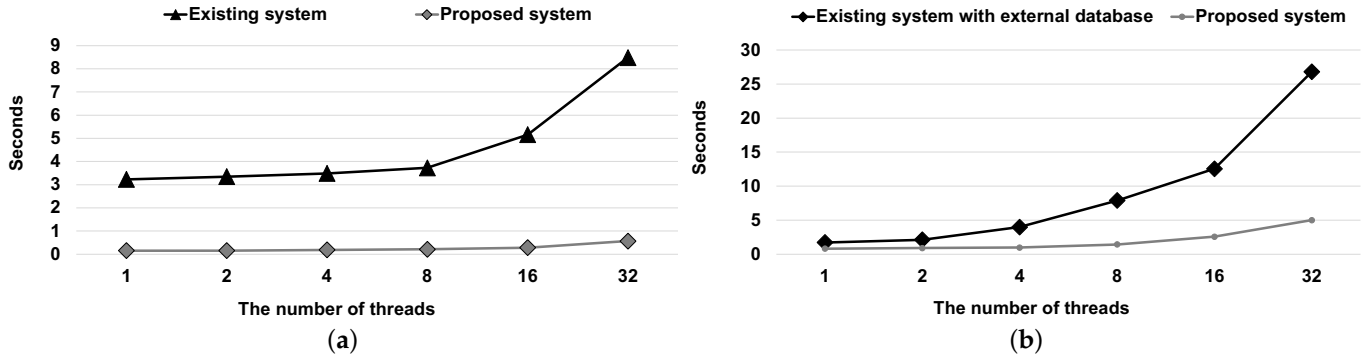

Figure 7. Execution times of select operations with different numbers of threads. (**a**) Execution times of select operations in smart contracts; (**b**) execution times of insert operations in regular transactions.

4.4. Measuring Resource Usage

To measure resource usage for one node when performing `SELECT` operations, we set the number of entities as 35,040 (one year) and measure the resource usage from the program start to the termination. Also, we set the 35,040 regular transactions to be the same as the smart contract. Figures 8 and 9 show the CPU and memory usage according to the number of threads and the number of entities in the smart contract and regular transaction. As shown in Figure 8a,b, the CPU and memory usage is almost the same when the number of entities increases and the number of threads is only one. This is because the number of entities used in our evaluation does not significantly affect memory usage.

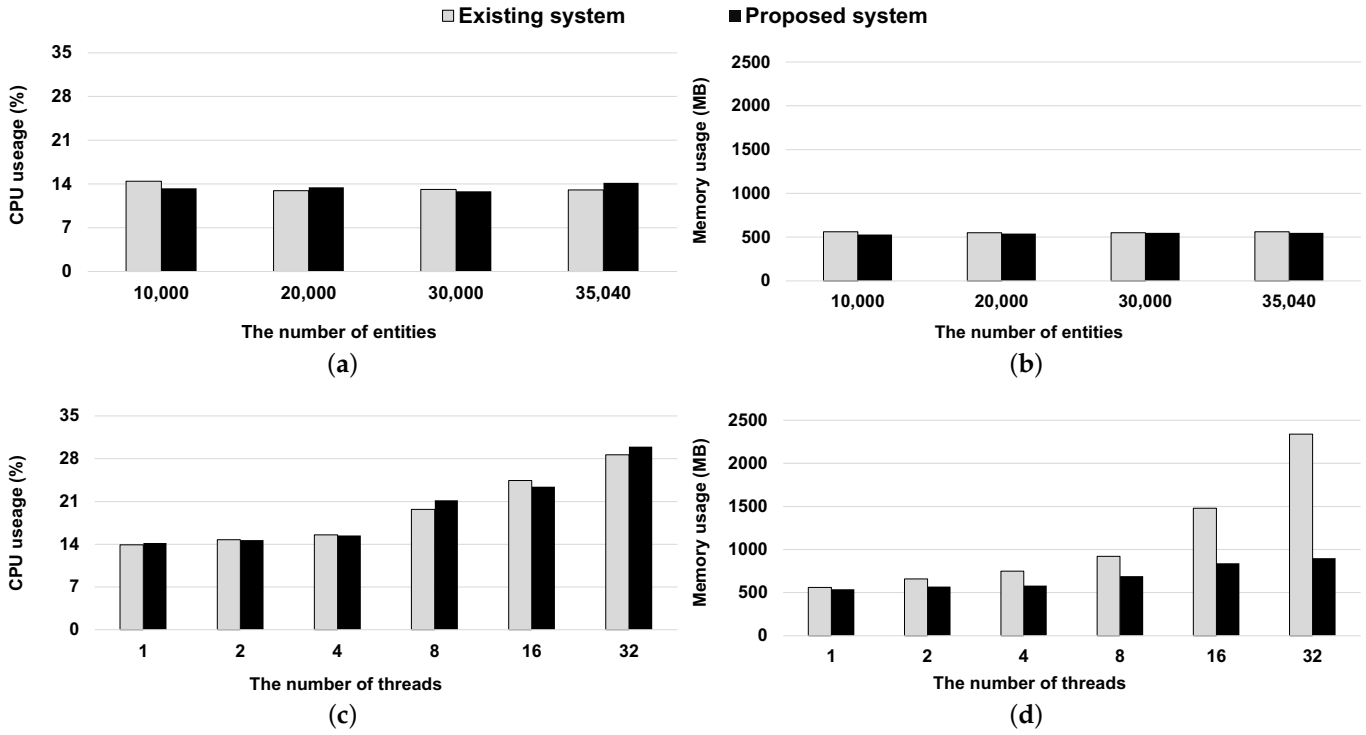

Figure 8. Resource usage in the smart contract. (**a**) CPU usage with different entity numbers; (**b**) memory usage with different entity numbers; (**c**) CPU usage with different thread numbers; (**d**) memory usage with different thread numbers.

As shown in Figure 8c,d the CPU and memory usage increase as the number of threads increases. In this evaluation, multiple threads concurrently handle entities, leading to a simultaneous increase in the required resources. As the number of threads increases, the demanded memory in the current system expands by as much as 2.6 times when compared

to the proposed system. The results show that, in the proposed system, EVM uses more memory than SQLite. The reason is that more EVMs are required to support user-defined data structures when more threads are added to the existing system.

In regular transactions, like smart contracts, as shown in Figure 9a,b, the CPU usage in the proposed system is higher than that of the existing system (with an external database), where the number of entities increases. Even if the CPU usage in the proposed system is higher, the CPU usage of the proposed system is average (about 2%), and it shows that the CPU usage itself is still low. Also, the memory usage in both systems is similar even if the number of entities increases. Figure 9c shows that the CPU usage increases as the number of threads increases in the proposed system. Meanwhile, the existing system (with an external database) does not increase CPU usage. Since the existing system (with an external database) is a more complex architecture (e.g., locking mechanism) compared to SQLite, the CPU usage is almost the same, even if the number of threads increases. Finally, Figure 9d shows that both systems slightly increase memory usage according to the number of threads.

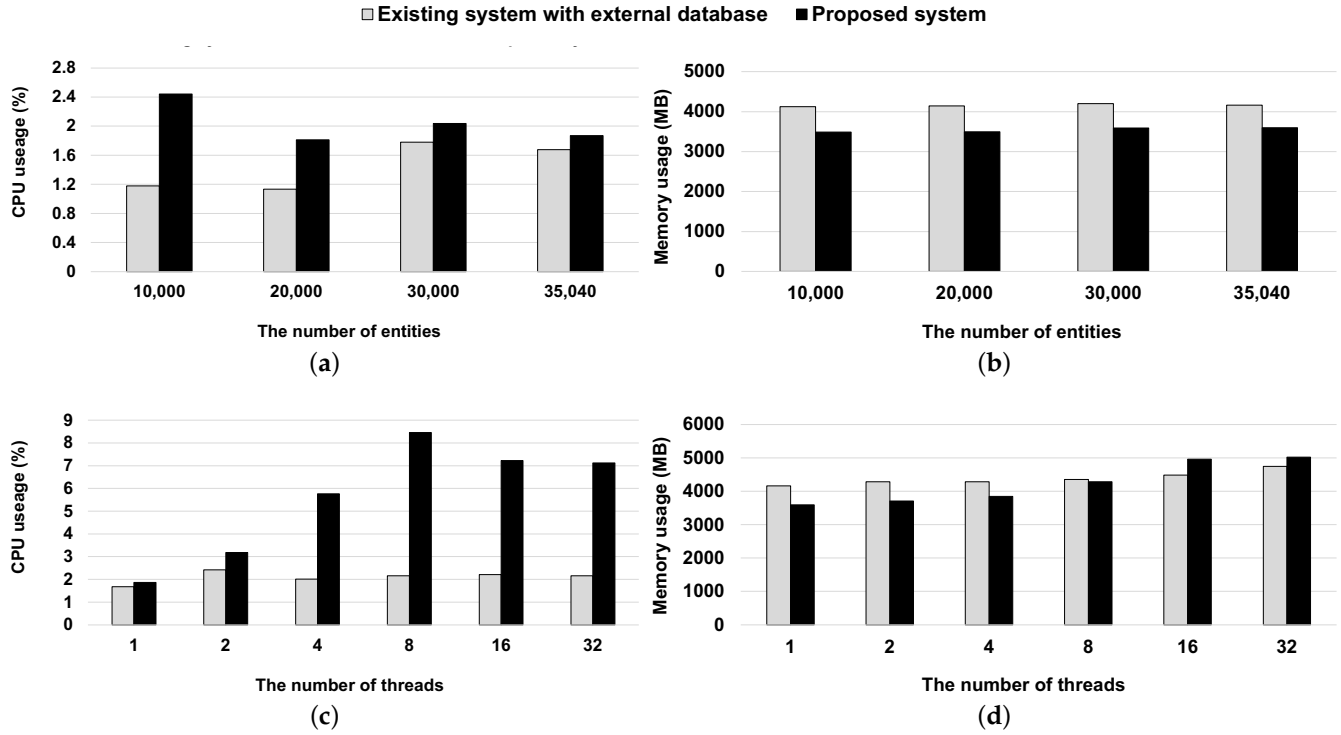

Figure 9. Resource usage in a regular transaction. (**a**) CPU usage with different entity numbers; (**b**) memory usage with different entity numbers; (**c**) CPU usage with different thread numbers; (**d**) memory usage with different thread numbers.

4.5. Comparative Analysis

To provide a more comprehensive perspective on the advantages and efficiencies of our proposed embedded relational database within the blockchain, we present a detailed comparison table (Table 1). This table contrasts the features and performance metrics of our system with traditional blockchain databases and those using external databases. This comparative analysis aims to elucidate the distinct enhancements and innovations introduced in our proposed system.

This comparison underscores the elevated performance, enhanced security, and increased query versatility of our proposed system. While traditional blockchain databases offer high security, they often lack performance and scalability. Blockchain systems utilizing external databases enhance query versatility but can compromise security and data integrity. Our proposed system amalgamates the strengths of both models, ensuring op-

timal performance, robust security, and extensive query capabilities while mitigating the complexities and vulnerabilities associated with external databases.

Table 1. Comparative analysis of different blockchain database systems.

Features/Metrics	Traditional Blockchain DB	Blockchain with External DB	Proposed System
Performance	Moderate	High	High
Scalability	Limited	Moderate	Enhanced
Security	High	Variable (depends on external DB security)	Very High
Complexity	Low	High (due to management of external DB)	Moderate
Cost Efficiency	High	Low (due to additional DB maintenance costs)	High
Query Versatility	Limited	High	Very High
Data Integrity	High	Moderate (risk of discrepancies between blockchain and external DB)	Very High
Real-Time Processing	Moderate	High	High

5. Discussion

In this section, we conduct a comprehensive discussion and present the multifaceted applications. The benefits spanning various industries underscore the robustness and versatility of our proposed model. Also, we list the limitations and potential challenges of the current system through the integration of an embedded relational database within a blockchain system and talk about future work to solve them.

5.1. Impact on Applications

The proposed system can be used in several industries by combining the immutable and transparent nature of the blockchain with the efficiency of SQL queries. We list the benefits that can be achieved when the proposed system is applied in various industries. In supply chain management, the ability to execute SQL query operations directly within the blockchain ensures instant data retrieval, which is essential for real-time tracking and verification. This feature mitigates the challenges of delayed data access and complexities associated with tracking goods at each logistic point, directly addressing the Oracle problem and ensuring data authenticity. Voting systems benefit from enhanced data integrity and real-time accessibility. The dual storage mechanism of LevelDB and SQL within the blockchain ensures that each vote is securely recorded and can be quickly retrieved and verified. This diminishes the risk of vote tampering and ensures electoral transparency and integrity. The healthcare sector experiences improved efficiency in records management. The embedded SQL queries enable healthcare providers to access and share encrypted patient data securely and in real time. This direct access to blockchain-stored data eliminates the need for intermediary data retrieval layers, enhancing data security and access speed. Financial services are streamlined, with our system ensuring that transaction data are not only securely recorded in the blockchain but are also instantly accessible via SQL queries. In particular, for cross-border transactions, this feature eliminates delays associated with data retrieval, ensuring real-time transaction verification and enhancing financial security.

5.2. Limitations of the Proposed System

Despite the significant advancements achieved with our proposed method, we recognize several limitations that warrant future exploration. One prominent constraint is the overhead associated with storing data both in LevelDB and the embedded SQL database within the blockchain. This dual storage mechanism, while enhancing query efficiency, potentially increases the storage requirements and impacts the system's overall performance. Additionally, the current implementation is tailored for Ethereum-based blockchain

systems. As the blockchain ecosystem is diverse and rapidly evolving, the adaptability of our approach to other blockchain platforms remains an unexplored avenue.

5.3. Future Work

In future work, we will aim to systematically address the identified limitations. In particular, the overhead associated with the dual storage mechanism is a primary concern. To mitigate this, we are exploring the development of more efficient data indexing and compression algorithms that would allow for rapid data retrieval and reduced storage space, eliminating redundancy and enhancing the overall system's efficiency. Ensuring fast and efficient data retrieval and maintaining the integrity and security inherent in blockchain technology will be the guiding principles.

Also, to tackle the limitations associated with the system's specificity to the Ethereum blockchain, we plan to extend the applicability of our approach to other blockchain platforms. To achieve this, we will conduct a series of empirical studies, tests, and validations to adapt and optimize our method to the unique architectural and operational nuances of different blockchain ecosystems. Our focus will be on ensuring that the embedded SQL database integration remains efficient, secure, and performative across the blockchain environments. By doing this, we anticipate broadening the scope and usability of our method, making it a universally applicable solution for enhanced data query operations within diverse blockchain systems.

6. Related Work

In several academic fields, blockchain technology presents many challenges. For example, there are studies aimed at enhancing the efficiency of the blockchain, including the performance of retrieval procedures.

Systems using external databases. To increase retrieval performance, previous systems [8,26,27] have used external databases for making indirect queries within a blockchain system. Etherscan [8], which is a search, API, and analytics platform for Ethereum, employs an external database. It enables users to browse the Ethereum blockchain in search of transactions, addresses, tokens, prices, and other Ethereum-related activity. Etherchain [26] is an explorer for the Ethereum blockchain; it extends the native Ethereum API. It offers basic statistical information, like the transaction count and block time. Additionally, it enables users to investigate smart contract transactions, search transactions, and monitor user account balances. Ethstats [27] is a visual interface used for monitoring the Ethereum network's health; it offers the most recent data on a block number, connected node information, pending transactions, gas prices, etc. FlureeDB [28] and BigchainDB [29] provide developing blockchain database solutions to support SQL-like queries. However, in those systems (e.g., Etherchain [26], EtherScan [8], and Ethstats [27]), users cannot verify whether the results of the external database and the blockchain data are identical since they use external databases outside of the blockchain system. Also, external database systems have significant maintenance costs to manage.

Embedded Blockchain systems. There are other ways to increase retrieval performance; previous studies [11,14,30–32] have proposed new layers and new language for the blockchain. To enhance the efficiency of select queries, EtherQL [14], for instance, adds a querying layer to the Ethereum client. It offers a variety of queries, such as the top K queries and range queries for transactions and blocks. VQL [31] provides efficient query services by extracting transactions from the underlying blockchain system and adding a middleware layer. EQL [32] proposes a query language that can retrieve information from the blockchain written in the programming language (Smalltalk) [33]. Through this query language, users can obtain block and transaction data. Pratama et al. [11] add three main query functions (retrieval query, aggregate query, and aggregate query) so that users and developers can easily access blockchain data.

For users or analysts who often generate transactions, these earlier systems and studies enhance the performance of searching for information about Ethereum's blocks,

transactions, and accounts. Our study is in line with these studies in terms of investigating the performances of select operations in the blockchain system. On the other hand, our work focuses on obtaining the data from both smart contracts and transactions utilizing an embedded relational database within a blockchain system that is based on Ethereum.

7. Conclusions

This article focuses on enabling SQL query operations within a blockchain system. We combine an embedded relational database with an Ethereum-based blockchain system to provide SQL queries. This enables range queries for smart contracts without any user-defined data structures and decreases the management costs for regular transactions without any external database. We implemented the proposed scheme on an Ethereum-based blockchain system and evaluated the proposed system using a synthetic benchmark. Our experimental results show that the proposed system—in smart contracts—can improve performance by up to about 22x compared with the existing system. Also, our system shows a similar search performance compared with the existing system, including an external database in regular transactions.

We show a significant optimization of SQL query operations within a blockchain system, particularly within the Ethereum-based blockchain. However, challenges remain for further enhancement. The overhead associated with storing data in both LevelDB and SQL is a primary concern. Therefore, in future work, a pivotal focus will be centered on reducing this overhead. By addressing these challenges, we aim to advance the embedded relational database within the blockchain, enhancing its efficiency, scalability, and performance in blockchain systems.

Author Contributions: Methodology, J.H., S.L. and Y.S. (Yongseok Son); Software, Y.S. (Yunhyeong Seo); Writing—original draft, S.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIT) (No. 2021R1C1C1010861, 2022R1A4A5034130, RS-2022-00166541), (Corresponding Author: Yongseok Son).

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Satoshi, N. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://assets.pubpub.org/d8wct41f/31611263538139.pdf (accessed on 18 September 2023).
2. Buterin, V. A Next-Generation Smart Contract and Decentralized Application Platform. 2014. Available online: https://finpedia.vn/wp-content/uploads/2022/02/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf (accessed on 18 September 2023).
3. Yli-Huumo, J.; Ko, D.; Choi, S.; Park, S.; Smolander, K. Where is current research on blockchain technology?—A systematic review. *PLoS ONE* **2016**, *11*, e0163477. [CrossRef] [PubMed]
4. Samaniego, M.; Deters, R. Blockchain as a Service for IoT. In Proceedings of the 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Chengdu, China, 15–18 December 2016; pp. 433–436.
5. Mengelkamp, E.; Notheisen, B.; Beer, C.; Dauer, D.; Weinhardt, C. A blockchain-based smart grid: Towards sustainable local energy markets. *Comput. Sci.-Res. Dev.* **2018**, *33*, 207–214. [CrossRef]
6. Ali, M.; Nelson, J.C.; Shea, R.; Freedman, M.J. Blockstack: A Global Naming and Storage System Secured by Blockchains. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, 22–24 June 2016; pp. 181–194.
7. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
8. etherscan. 2018. Available online: https://etherscan.io (accessed on 18 September 2023).
9. Ethereum State Trie Architecture Explained. 2019. Available online: https://medium.com/@eiki1212/ethereum-state-trie-architecture-explained-a30237009d4e (accessed on 18 September 2023).

10. Is It Possible to Access Storage History from a Contract in Solidity? 2016. Available online: https://ethereum.stackexchange.com/questions/11545/is-it-possible-to-access-storage-history-from-a-contract-in-solidity (accessed on 18 September 2023).
11. Pratama, F.A.; Mutijarsa, K. Query support for data processing and analysis on ethereum blockchain. In Proceedings of the 2018 International Symposium on Electronics and Smart Devices (ISESD), Bandung, Indonesia, 23–24 October 2018; pp. 1–5.
12. The Graph. 2020. Available online: https://thegraph.com (accessed on 18 September 2023).
13. SQLite. 2018. Available online: https://www.sqlite.org/index.html (accessed on 18 September 2023).
14. Li, Y.; Zheng, K.; Yan, Y.; Liu, Q.; Zhou, X. EtherQL: A query layer for blockchain system. In Proceedings of the International Conference on Database Systems for Advanced Applications, Suzhou, China, 27–30 March 2017; Springer: Cham, Switzerland, 2017; pp. 556–567.
15. Han, J.; Kim, H.; Eom, H.; Coignard, J.; Wu, K.; Son, Y. Enabling SQL-query processing for ethereum-based blockchain systems. In Proceedings of the 9th International Conference on Web Intelligence, Mining and Semantics, Seoul, Republic of Korea, 26–28 June 2019; pp. 1–7.
16. TRANSACTIONS. 2022. Available online: https://ethereum.org/ko/developers/docs/transactions/ (accessed on 18 September 2023).
17. DPRating Crypto Rankings (Based on GitHub Activity) June 2018 Report: EOS, Cardano, TRON, and Ethereum Tied for First Place. 2018. Available online: https://www.cryptoglobe.com/latest/2018/07/dprating-crypto-rankings-based-on-github-activity-june-2018-edition/ (accessed on 18 September 2023).
18. Szabo, N. The Idea of Smart Contracts. *Nick Szabo'S Pap. Concise Tutor.* **1997**, *6*, 199.
19. Chishti, M.S.; Sufyan, F.; Banerjee, A. Decentralized On-Chain Data Access via Smart Contracts in Ethereum Blockchain. *IEEE Trans. Netw. Serv. Manag.* **2021**, *19*, 174–187. [CrossRef]
20. LevelDB. 2019. Available online: https://github.com/google/leveldb (accessed on 18 September 2023).
21. OpenSea Statistics 2023: How Many Users Does OpenSea Have? 2023. Available online: https://thesmallbusinessblog.net/opensea-statistics/ (accessed on 18 September 2023).
22. Caldarelli, G. Understanding the blockchain oracle problem: A call for action. *Information* **2020**, *11*, 509. [CrossRef]
23. What Is the Difference between Embedded Databaseand Ordinary Database Like MySql or Oracle. 2018. Available online: https://goo.gl/oV9x7b (accessed on 18 September 2023).
24. Wikipedia: Embedded Database. 2018. Available online: https://en.wikipedia.org/wiki/Embedded_database (accessed on 18 September 2023).
25. Apache JMeter. 2022. Available online: https://jmeter.apache.org/ (accessed on 18 September 2023).
26. Etherchain. 2018. Available online: https://www.etherchain.org (accessed on 18 September 2023).
27. Ethstats. 2018. Available online: https://ethstats.net (accessed on 18 September 2023).
28. Platz, B.; Filipowski, A.; Doubleday, K. Flureedb: A Practical Decentralized Database. 2017.
29. McConaghy, T.; Marques, R.; Müller, A.; De Jonghe, D.; McConaghy, T.; McMullen, G.; Henderson, R.; Bellemare, S.; Granzotto, A. Bigchaindb: A scalable blockchain database. *White Pap. BigChainDB* **2016**, 53–72.
30. EthereumJ. 2018. Available online: https://github.com/ethereum/ethereumj (accessed on 18 September 2023).
31. Peng, Z.; Wu, H.; Xiao, B.; Guo, S. VQL: Providing query efficiency and data authenticity in blockchain systems. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), Macao, China, 8–12 April 2019; pp. 1–6.
32. Bragagnolo, S.; Rocha, H.; Denker, M.; Ducasse, S. Ethereum query language. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, Gothenburg, Sweden, 27 May–3 June 2018; pp. 1–8.
33. Goldberg, A.; Robson, D. *Smalltalk-80: The Language and Its Implementation*; Addison-Wesley Longman Publishing Co., Inc.: Reading, MA, USA, 1983.