Security against Compromised Servers: Verifiable DB Queries

By

Muhammad Waheed Akram

2015-NUST-MS-IS8-119153

Supervisor

Dr. Shahzad Saleem

Department of Information Security

School of Electrical Engineering and Computer Science,

National University of Sciences and Technology (NUST), Islamabad,

Pakistan.

Dedication

To my parents, my wife, daughter, my friends and my supervisor who has always been supportive.

Approval

It is certified that the contents and form of the thesis entitled "**Security against Compromised Servers: Verifiable DB Queries**" submitted by **Muhammad Waheed Akram** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Shahzad Saleem**

Signature: ________________

Date: ____________________

Committee Member 1: **Dr. Abdul Ghafoor Abbasi**

Signature: ________________

Date: ____________________

Committee Member 2: **Madam Haleemah Zia**

Signature: ________________

Date: ____________________

Committee Member 3: **Dr. Naveed Ahmed**

Signature: ________________

Date: ____________________

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at National University of Sciences & Technology (NUST) School of Electrical Engineering & Computer Science (SEECS) or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: Muhammad Waheed Akram

Signature: _______________

Acknowledgements

I am extremely thankful to Allah Almighty. I would be thankful to my family and friends who encouraged me and supported me. I express my enormous gratitude to my supervisor Dr. Shahzad Saleem and my GEC members. I was very lucky that I could work on my thesis in Applied Information Security Lab (AIS) at SEECS-NUST. I want to thank Dr. Shahzad Saleem on his generosity in sharing precious thoughts and knowledge with me.

Last but not least, I want to thank Dr. Naveed Ahmed at SEECS-NUST for sharing his valuable experiences regarding the web & network security with me.

Abstract

In this modern age, we are marching towards automation, data centralization in every field of life. We are dependent on databases for storage of data from applications like medical, banking, and stock exchange. Integrity of data is dependent on the security of the server hosting databases. Security of servers cannot be guaranteed because of sophisticated hacking tools, advance malwares, viruses and widespread vulnerabilities in software. So, there is a need to build secure web based framework that can ensure the integrity of data in fully compromised environment. In fully compromised environment an attacker has full access to the server and can change data in databases. User of an application cannot detect that data coming from server is not corrupted. So, solution for this problem is verifiable queries, which means that user can verify with cryptographic proof at web browser end that the data coming from database against his query is correct. This solution is implemented to provide data integrity in fully compromised environment. In this implementation, proof is calculated at the time of write operation to the database. Write operations consist of insert, update and delete functions. This proof is stored on blockchain. Later on, when a user performs read operation (i.e. read, find, and sum) from databases, user gets results along with some proof helpers. User can verify the integrity of result at browser end by calculating proof from proof helper and comparing with proof from blockchain. Properties of integrity correctness, completeness, and freshness are ensured in our solution. Our solution can be integrated with all web applications with high integrity requirements. However, as a proof of concept, we have integrated this solution with a medical application. The performance evaluation of our solution shows that both proof calculation and verification is efficient in terms of memory requirements and latency.

Table of Contents

List of Figures

List of Tables

List of Equations

Chapter 1

1 Introduction

Databases are used to store users' data. The security of that databases is always dependent on the security of that server where databases reside. For security of that server all type of security controls (i.e. firewall, IPS/IDS) are implemented. During the development phase of these application, security of application is also ensured to avoid common vulnerabilities for example SQL injection, XSS, and CSRF [1]. Communication channel between server and client is also encrypted and secure using TLS/SSL. But according to study conducted by university of Maryland, that attacker on average attacks after 39 seconds which means that there are more chances for attackers to get access of servers [2]. Growth rate of cyber-attacks is increasing every year due to increase in vulnerabilities in software and availability of more advanced and sophisticated hacking tools. These hacking tools can attack and get access of the server. Once attacker gets control of server, he can change the data on databases server. Compromised data can result into erroneous decisions. For a remote user, it is very difficult to ensure the integrity of the response data which he is getting from a database server. Such a scenario can lead to severe consequences in environments with high integrity requirements.

In this thesis, we will talk about data Integrity. So, it is defined as data protection against from illicit change in data. It holds these three properties of integrity: *completeness, correctness,* and *freshness* [3]. We will now try to define and explain these properties with the help of an example.

Completeness is an absolute truth which means information provided is completed, no related information is missed [4]. We can understand with the help of an example; a traveler requests a flight booking site to list all flights from Islamabad to Dubai. For completeness, he needs assurance that the result set for flights from Islamabad to Dubai is complete and there is not any other flight between these two points on the said date.

Correctness is defined as accuracy which means information provided is accurate [5]. For correctness, the traveler will ensure that all the listed flights from Islamabad to Dubai were accurate with respect to all the attributes like flight number, departure time, arrival time, source, destination and operator.

Freshness is defined as up-to-date information which means that information is not replayed [6]. For freshness, he will ensure all the listed flights are up-to-data, means query result is obtained from the latest data and all the rescheduling till the time of query have been accommodated.

In medical web application, patients' data is very important for physicians. If an attacker changes patient's data then physician can prescribe wrong medicine. This may cause harm to health of the patient. Every year many patients are misdiagnosed, and sometime this also leads to death [7]. According to a study in USA from 2013 to 2017, 363 security hacking incidents affected server based electronic records of 13 million patients [8]. So, if server is fully compromised then physician should have some facility to verify patients' data before making correct decision.

In medical application, a query is executed by physician for particular patient and database returns data of that patient. Now physician wants to ensure integrity of query result. If some cryptographic proof is provided for verification of integrity of result then physician can trust the results of query completely. This solution is required in similar cases where databases are outsourced to third party, or data sender is trusted and server is untrusted.

So, if the end client can verify integrity of query results, then he can trust on the data (from query result) on compromised server.

1.1. Motivation

In internet environment, servers remain vulnerable to many types of attack. Many critical applications (i.e. remote medical application, financial application, stock exchange application) where data tampering by an attacker can cause very serious impact on end clients and data owners. The end user trusts on the server after verifying the results of queries executed on the databases with some cryptographic proof. So, there should be some real world solution for query verification that enables the end user to verify the queries result at the client's browser end in real time with minimum overhead. It should help to user to take decisions on the data with full confidence in a fully compromised environment. It should also work with minimal overhead and can be integrated with any new and old web applications.

1.2. Problem Statement

Security of all web applications and databases depends upon the security of the server. Security of these servers cannot be ensured. In case of outsourced databases, the client is not trusting on third party databases. The attacker can corrupt the data on the server in fully compromised environment, so the end client gets wrong query results by executing a query on corrupted databases. The client can take wrong decisions depending upon wrong data results which can cause damage to clients. If the databases are untrusted, then it very difficult for client to trust on query results.

It is a challenge for the client to detect the integrity of data is preserved or not. There should be some mechanism of query verification which helps the client to integrity of data is preserved. This also enables the clients to check that the server is compromised or not, and result of query is not corrupted.

So verifiable database queries provide verification of results at browser end with the help of cryptographic proof in compromised environment. It should have minimum overhead which may not affect the performance of web applications and databases in real time.

1.3. Objectives and Research Goals

Research goals of this thesis are to propose and implement the solution for database query verification in fully compromised environment where the client is not trusting the data results from the server. The client can ensure integrity with its properties i.e. completeness, correctness, and freshness of the query data result at browser end.

The query verification solution should be implemented so that it can be used in real time with minimum overhead and performance of web applications do not suffer too much. Solution should be flexible, so it can be integrated with new web applications in development phase and also with already developed web applications with minimal changes. Solution should also be expressive so it can support most common and widely used simple and aggregated database queries. These following main objectives of this research are shown in figure 1-1:

- *Goal-1:* Implement query verification using authenticated data structures
- *Goal-2:* Integrate solution with test web application i.e. Medical application
- *Goal-3:* Proof creation and verification mechanism
- *Goal-4:* An extensive performance evaluation of proposed solutions

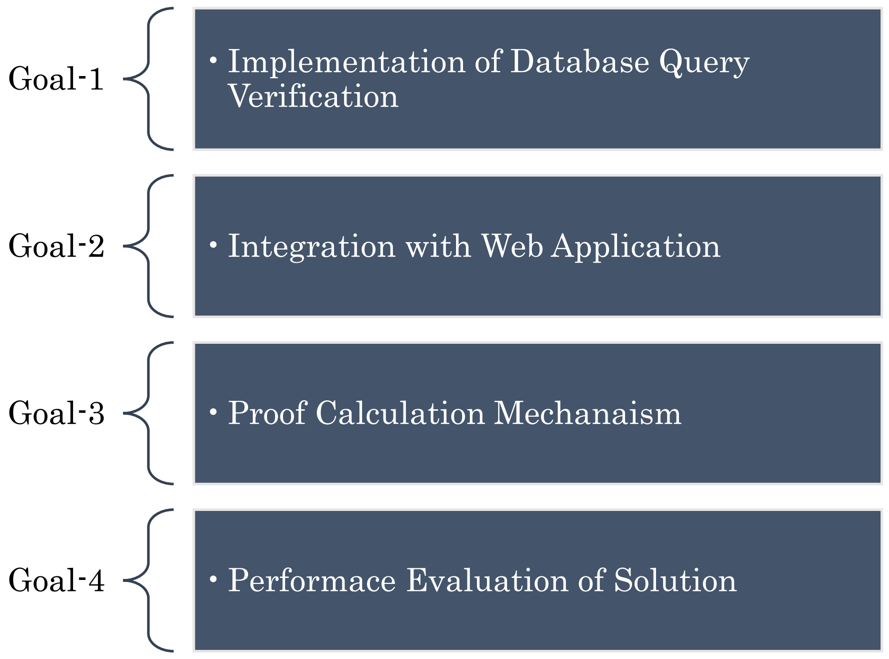

Goal-1
Implementation of Database Query Verification
Goal-2
Integration with Web Application
Goal-3
Proof Calculation Mechanaism
Goal-4
Performace Evaluation of Solution

Figure 1-1 Research model

1.4. Thesis Organization

In chapter 2, background information related to this research problem is discussed and explained. This is helpful for understanding of design of solution. Chapter 3 includes literature review of all existing and related solutions to this research problem. There methodology and issues are discussed in details. Chapter 4 is about the purposed solution and methodology which is used to solve this research problem. Proposed solutions and complete design are also explained. In chapter 5, implementation of proposed solutions is discussed. It also includes integration of query verification solution with test medical web application. Chapter 6 is related to extensive performance evaluation of the solution with respect to latency, and throughput. Performance and cost analysis are discussed with details in this chapter. Conclusion and future work are described in chapter 7. Thesis organization as described above is also shown in figure 1-2.

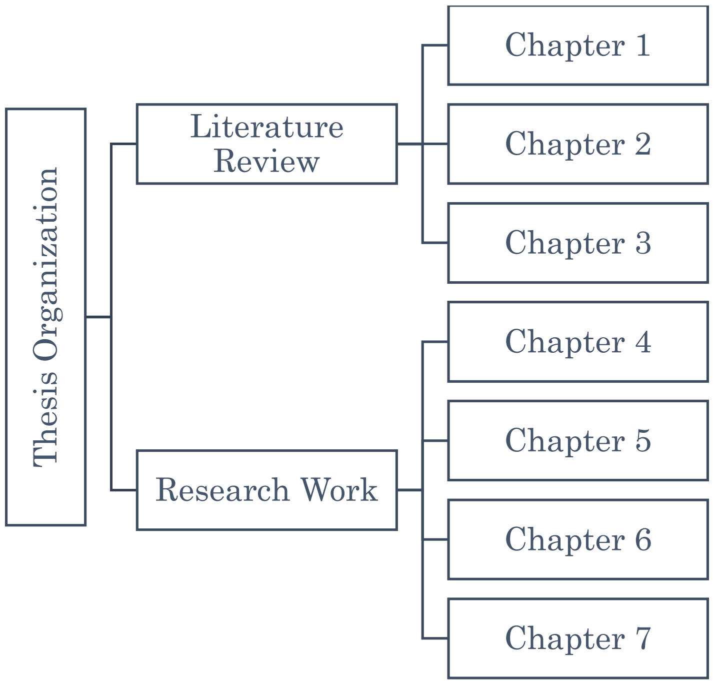


Figure 1-2 Thesis Organization

Chapter 2

2 Background Information

In this chapter, background information which is required for understanding of this thesis is given. This includes background information regarding integrity in databases, authenticated data structures and hashing functions.

2.1. Query Verification in Commercial Databases

According to databases journal, following are five top databases for 2019 [9]:

1. Oracle Database
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL
5. MongoDB

Now, we have discussed the integrity protection of each top five databases. Oracle is a SQL (rational) database, and first commercially available databases. Oracle has introduced many features with each release [10]. They have introduced data encryption using AES and also provide the data integrity feature using SHA-x. But there is an additional cost of encryption and integrity. It also results in increase in finical cost and degradation of performance [10]. If attacker can get access of databases then he can also change the transaction and manipulate results.

MySQL has two versions one is free and other is enterprise. MySQL provides support of encryption of databases and data integrity. Data integrity is independent of encryption but in MySQL, it uses the hash checksum during encryption process. In MySQL, the user has to pay money to buy licenses of security features [11].

Microsoft SQL server is the third commercial solution. It also provides encryption and integrity. Like Oracle and MySQL it also has addition cost for these security features [12].

PostgreSQL is open source and powerful database. It is community-based database. It also has features of encryption and data integrity. But many security issues of this database are reported which are being patched on regular basis against the reported vulnerabilities [13].

MongoDB become popular in very short time. It is no-SQL, and used in many famous web and modern applications. It has security features like TLS/SSL, encryption at rest. [14].

There are also other databases which are not discussed here. All databases have security features like encryption of data (AES 256) and integrity of data (SHA-1). These inbuilt security features may be utilized depending upon the criticality of the application data which is being stored in databases. These security features add some addition financial cost and also overhead on the processing of the server. Encryption, decryption and hashing are functions which increase latency.

In case of compromised environment, servers do not provide information to the client to check integrity of that the data generated as a result of a query. So, results of queries in all databases are not providing proof to the client to verify the results against integrity requirements.

2.2. Authenticated Data Structure (ADS)

Authenticated data structure (ADS) is type of data structure which has two roles to perform operations. One is *prover* and other is *verifier*. **Prover** calculates proofs when data is written. Proofs are provided to **verifier** when he needs to verify the data.

Authenticated data structures are almost thirty years old concept in cryptography. Authenticated data structures are widely used in applications to ensure authenticity. Authenticated data structures are implemented in various forms and each implementation has different performance.

Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi [15] have discussed, implemented and presented "λ•", the first programming language for Authenticated data structures implementation. They use it for different Authenticated data structures like Merkle hash tree, Red black tree, and skip lists.

Authenticated data structures provide more efficient method for verification. Authenticated data structures are used in some previously proposed integrity solution [16], [17]. Authenticated data structures can be implemented in multiple ways to keep the correctness and completeness property intact.

In server client model, the end client verifies the data which is stored on server. Server use Authenticated data structures to make proof and send to the client to verify the data. The end client uses the proof sent by server and verify correctness of the data. Authenticated data structures have very common use in peer to peer communication where anonymity and integrity is required. ADS are also implemented for client server model with centralized and distributed approach [18].

2.3. Merkle Hash Tree

Merkle hash tree [19], [20] is the most common method to verify any type of data. It reduces proof size and makes one root hash. Root hash is built on the hashes of all the data in the tree. Merkle Hash tree uses cryptographic hash functions like Secure Hash Algorithms [21](i.e. SHA-1, SHA-2 and SHA-3) [22] to build the tree.

Secure hash algorithm (SHA) series are the most common and trusted hash functions. Merkle hash tree is a tree of hashes, with top hash or root hash at root of the tree. This scheme is used in many famous applications like bitcoin, blockchain and other applications. Merkle hash tree is graphically represented in figure 2-1.

There are 8 leaves from L1, L2, L3, L4, L5, L6, L7 and L8. H1 is hash of L1 and similarly H2 for L2 and so on. H12 is the hash of H1 and H2 and H34 is the hash of H3 and H4. H14 is hash of H12 and H34. Similarly, this tree is built up in hierarchy, and Root hash of Merkle tree is calculated. Any hash function can be used with any number of resultant hash bits i.e. SHA-1, SHA-2, SHA-3. If any value changes from L1 to L8, then corresponding hash is changed. All dependent hashes are calculated again and new Merkle root hash is calculated. Root hash is the output that come from all node hashes of the tree and it can be used to verify the integrity of data at all the leaves of the tree.

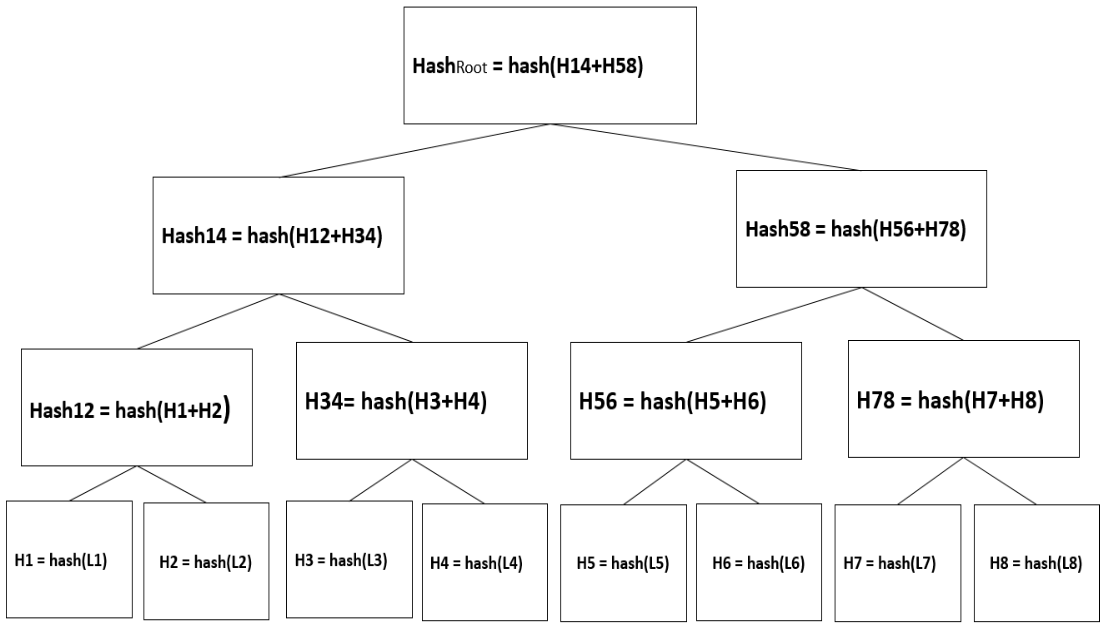


Figure 2-1 Merkle Hash Tree

In case of verification, the client wants to verify that L4 is changed or not as shown in Figure 2-2. Then H3, H12 and H58 is sent as proof helpers to the end client. **Proof helpers** mean all those relevant hash nodes which are required to build Merkle root hash. In this case, H3, H12 and H58 are proof helpers. The end client calculates H4 from L4, H34 from H3 and H4. Then H14 is calculated using H12 and H34. Finally Root hash is calculated from H14 and H58.

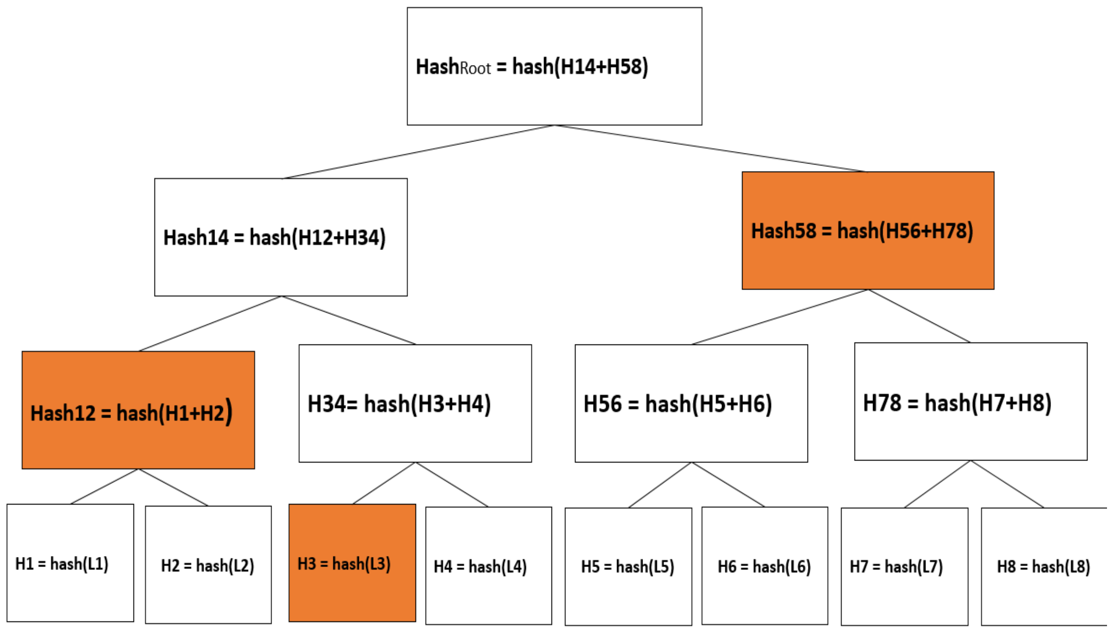


Figure 2-2 Merkle Hash Tree Proof

Newly calculated Merkle root hash is compared with previously calculated Merkle root hash. If newly calculated Merkle root hash is same as previously calculated Merkle root hash then it means data has not been changed or corrupted at leaves. The end client trusts the data in this case. If newly calculated Merkle root hash does not match with previously calculated root hash of tree then it means that data used in hashes at leaves are changed or corrupted. In this case, the end client knows that the requested data has been changed. So, Merkle hash tree provides efficient mechanism to protect the integrity of data.

2.4. Cryptographic Hashes

Hash functions are used to associate the fixed size of data to every input data of different lengths and is subsequently used to prove integrity. In cryptography, output of the hash function (the fixed size of data) must fulfill some properties. The hash functions have following properties: Pre-Image Resistance, Second Pre-Image Resistance, and Collision Resistance [23]. Some of these hash functions are weak candidate for usage for example SHA-1 [24].

National Institute of Standards and Technology (NIST) has also presented hash algorithm series which is compliant with FIPS [25]. This series is famous as secure hash algorithm (SHA), and variants of this series are SHA-0, SHA-1, SHA-2 and SHA-3 [26], [27] SHA-0 and SHA-1 has output length of 160 bits but they are vulnerable to collision. SHA-2 has six different variants which have 224, 256, 384 and 512 bits output. Similarly, SHA-3 has also six different variants which have 224, 256, 384, 512, and arbitrary "d" bits output [22] to protect the integrity.

In Merkle hash tree, SHA-x can used for hash calculation of all nodes. Hard disk integrity checks also use the cryptographic hash functions, and save output for a given data. At verification time, hash of data is calculated again. If both results are matched then it means data has not changed. Many software, files and documents on internet use this technique, so the end client can trust on these files.

Passwords are saved as output of cryptographic hash functions. During verification, when the user enters the password, its hash is taken and compared with already stored hash. As clear text passwords are very much vulnerable, if system gets compromised. Hashing is also used in source code management tools.

Chapter 3

3 Literature Review

In this chapter, we have done a review of all research related to databases integrity, query verification and web application integrity. People have already explored the areas like database integrity and query verification and subsequently proposed different solutions. Some known issues and shortcomings exist in those techniques which are discussed in this chapter.

3.1. Databases Integrity

Databases are collections of information which are hosted on the servers. They can handle all type of data i.e. personal information, pictures, files and related data. Different vendors provide databases and the most common used are Oracle databases, MySQL, and MongoDB. Databases integrity means that data should not be changed by the attacker or any third party.

These days many application are hosted on clouds, so databases are also migrated to clouds. In some cases databases are also outsourced to a third party. In all these scenarios, trust on database service providers is a big question. Similarly, if you have deployed your own databases server, then there are also chances attacker can hack your server. The end client using databases does not get idea that data is verified or not.

HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan [28] implemented the solution for untrusted databases as shown in Figure 3-1 [28]. The solution was implemented in such a way that the end user can verify the results getting from databases. He can also verify completeness and authenticity of data. As already discussed, three properties of the integrity verification of data are completeness, correctness and freshness.

But their solution was not ensuring completeness and freshness. Data result of query can be old or replayed, as it can fulfill correctness property. For authenticity of data, digital signatures were used. The use of digital signatures were also increasing the size of data, which was stored on outsourced or untrusted databases. Computational cost of signature verification was also additional overhead. Attacker can also change query results and proofs (which is used in verification) in fully compromised environment. So, the end user cannot detect that results of queries were corrupted. Implementation was only supported with SQL database.

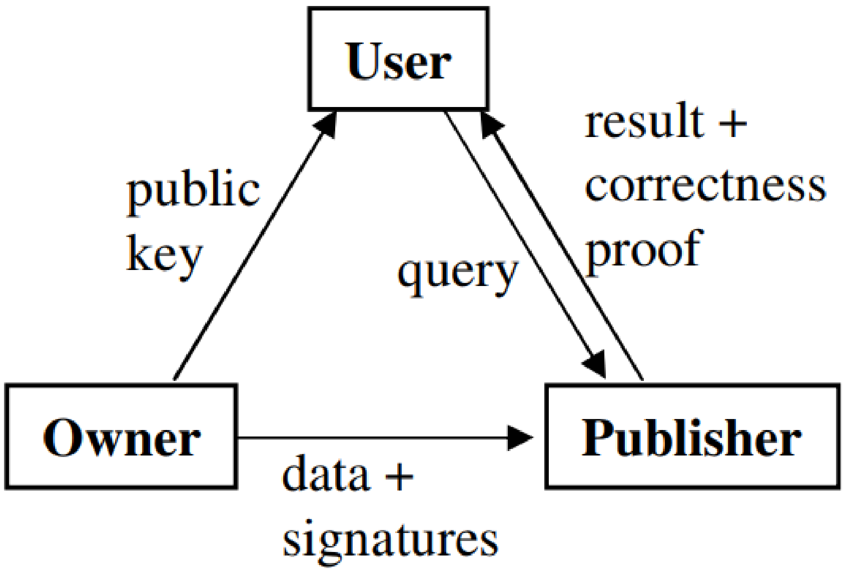


Figure 3-1 Solution for untrusted databases [28]

Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou [29] implemented the IntegriDB. It was also used to ensure integrity of untrusted databases. In this model, data owners were keeping the records. In many applications data owner cannot keep all the records on its local databases. For example, medical devices (remote blood pressure measuring device, pacemaker) of patient are sending data to medical web application. Data owner is patient, and he cannot save all of its data for verification. Even in remote medical devices like pacemaker, cannot store too much data.

At verification time, communication channel between data owner and the end client is not always available in web applications. For example, pacemaker cannot be online when physician needs to verify the data. IntegriDB had client-server model. IntegriDB client was used to verify query results with proofs which were stored on the databases. IntegriDB setup time was also very high. This solution was providing only correctness and completeness of resultant data. Freshness was not provided. Proof and result were returned in the response of IntegriDB query from IntegriDB server.

One serious issue with IntegriDB was, if the server was compromised, then client could not able to verify the query results. IntegriDB does not work in fully compromised environment. If attacker has full access, he can send wrong data with wrong proof. For example, if end user has requested value blood pressure of patient at time 9pm then attacker has ability to return the value at 10 pm which can be verified from data owner too. IntegriDB was also only supporting for rational databases (SQL). We can see the flow of IntegriDB, as shown in figure 3-2 [29].

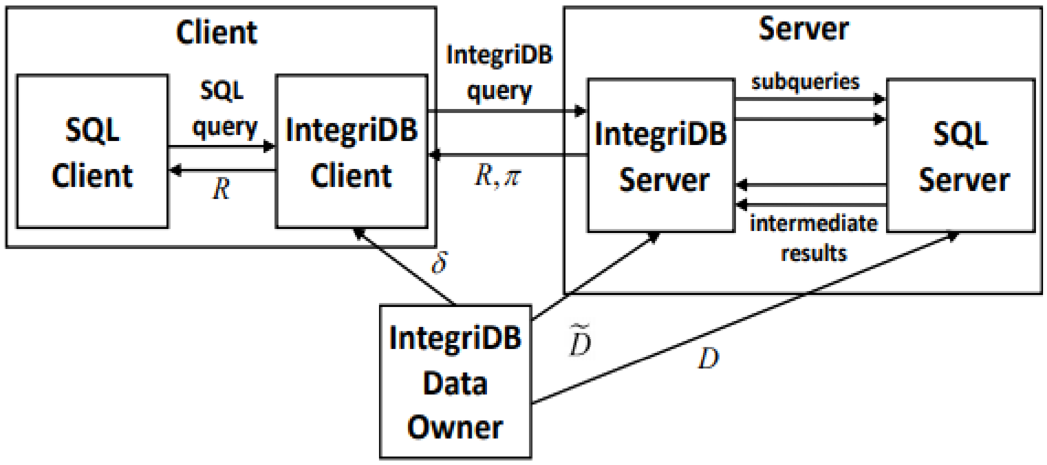


Figure 3-2 Implementation of IntegriDB [29]

CryptDB [30] was presented to solve the security issues of the databases. It was implemented to solve the confidentially issues of database. According to the published result, it was found to be quite real time. It was not providing the integrity protection of databases for databases queries results.

Palazzi Bernardo, Pizzonia Maurizio and Pucacco Stefano [31] proposed the solution for only completeness of resultant data from databases. They were using skip list as authenticated data structures to provide completeness of data results. It was implemented on the SQL based databases. It was also supporting only basic databases queries and provides only completeness. This solution was not covering correctness and freshness of SQL queries. In compromised environment, it does not provide completeness feature of integrity.

3.2. Query Verification

Data is stored in databases, where the end user interaction with database is depend upon the databases queries. There are mainly two type of operations on data: *read* and *write*. **Read** queries interact with database and get data from database. Example of read queries are project, sum, average, max, min, count as given in table 4-1. **Write** queries enable user to write some new data, update or delete existing data from databases. Write queries are insert, update and remove/delete as given in table 4-2 [32]. Format of these queries may vary from one database to another database. This is dependent upon the vendor or the type of databases (SQL or NO-SQL).

Any application integrated with the database uses these queries to perform operations on databases. The end user cannot trust on the results of queries in case untrusted databases. Attacker or third party (managing database) can corrupt the results of read and write queries.

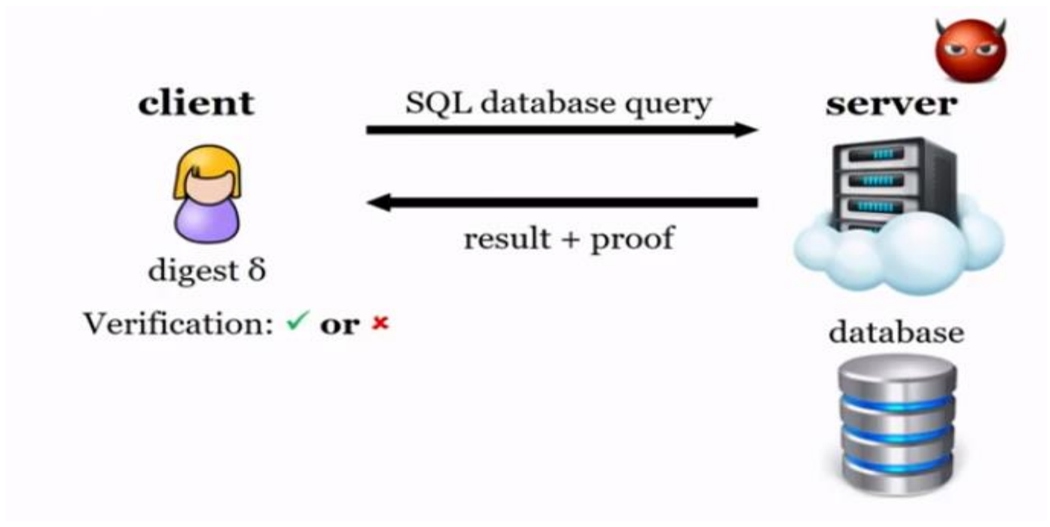


Figure 3-3 vSQL [33]

Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos and Charalampos Papamanthou [33] presented the paper on vSQL, which was providing SQL queries verification in case of untrusted databases on cloud. They introduced verification of SQL queries when databases were on cloud or on third party's untrusted servers as shown in figure 3-2. Their solution was less efficient, because it had very large setup time. According to given results, it was more expressive but less efficient than the solutions based on authenticated data structures. Proof calculation was on the cloud server which would require the client to purchase more resources for complex cryptographic operations and thus increasing the cost and overhead. Moreover, the end user had to maintain local database which was additional cost for him.

Hweehwa Pang and Kian-Lee Tan [34] discussed the solution for completeness of database queries using authenticated data structures. This solution was only providing completeness of query result. It was built for relational databases and it was supporting only limited number of queries. This solution was not verifying query results when the database server was compromised. This solution was also for SQL.

Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese [35] proposed efficient solution for query integrity for databases in outsourced environment. As in figure 3-4 [35], DB querier was communicating with cloud databases for proof. Their solution was supporting selection, projection and only few aggregated queries of the databases. The proposed solution was for only SQL databases.

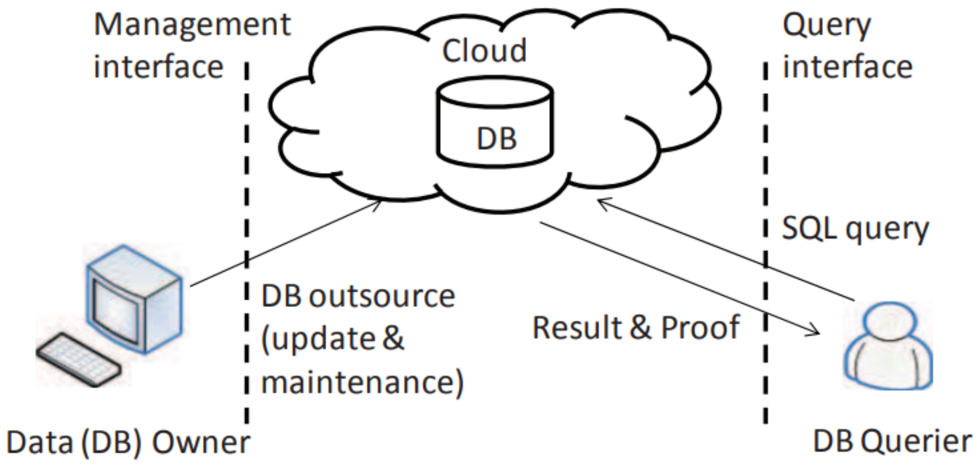


Figure 3-4 Query Integrity for outsourced databases [35]

Solution for authenticity and integrity [36] was proposed for SQL based databases in outsourced environment. There were multiple issues with assumptions and final design. The end client was getting proof from outsourced databases and from data owner. Usually, the end clients do not communicate with the data owners. Data owner may not have enough storage to keep the copy of data and proof with themselves.

Grisha Weintraub and Ehud Gudes [37] proposed solution for No-SQL databases on clouds using probabilistic approach to provide correctness and completeness only. Freshness is an important part of the integrity and it was not ensured by them in their solution.

In outsource databases, the end user sometime prefers to store encrypted data. When client executes query on database, he gets encrypted results which are decrypted at the client end. To handle this scenario solution [38] was proposed and implemented with proxy re-encryption. Proof was also stored on the cloud database. This solution did not provide freshness of results. If the cloud database is compromised then proof can also be changed by intruder. Similarly, untrusted third party can also change data and proof. The end user cannot detect the change in this case.

3.3. Web Application Integrity

Web applications are of two types: static and dynamic. In static web applications content of web pages is remain same every time. Integrity protection of such web pages is very easy. Just store hash of each web page using cryptographic hash functions [22]. The end user accesses these pages in browser. He calculates the hash of web pages. He has to match hash with already stored hash. If both hashes are same, there are no change is web pages and the integrity of web application preserves.

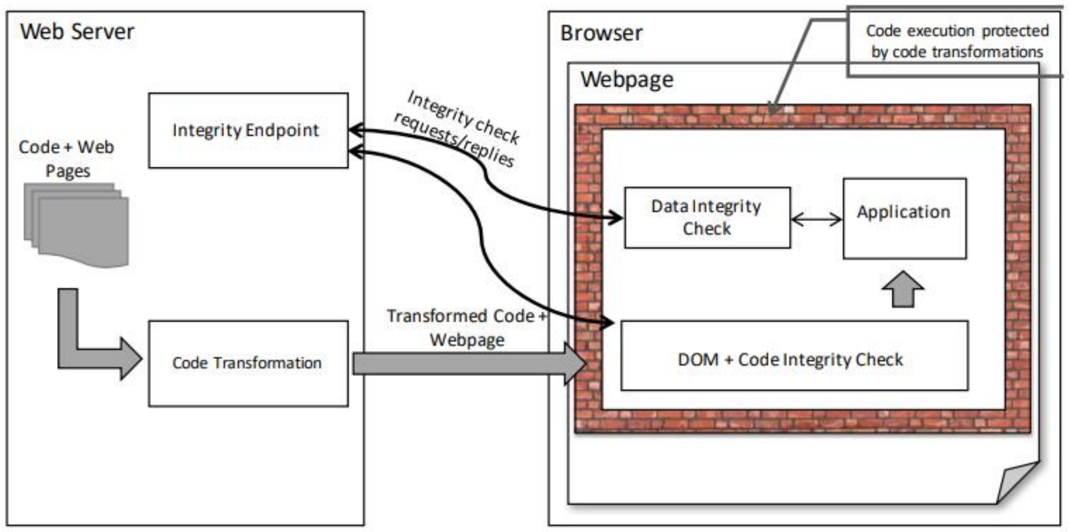

Web Server
Code + Web Pages
Integrity Endpoint
Code Transformation
Integrity check requests/replies
Transformed Code + Webpage
Browser
Code execution protected by code transformations
Webpage
Data Integrity Check
Application
DOM + Code Integrity Check

Figure 3-5 Static Web Application Security [39]

Pedro Fortuna, Nuno Pereira and Ismail Butun [39] proposed the solution for web applications integrity. In their solution, it was comparing web application code on the server and on the client side as shown in figure 3-5 [39]. This solution was only for the static web applications.

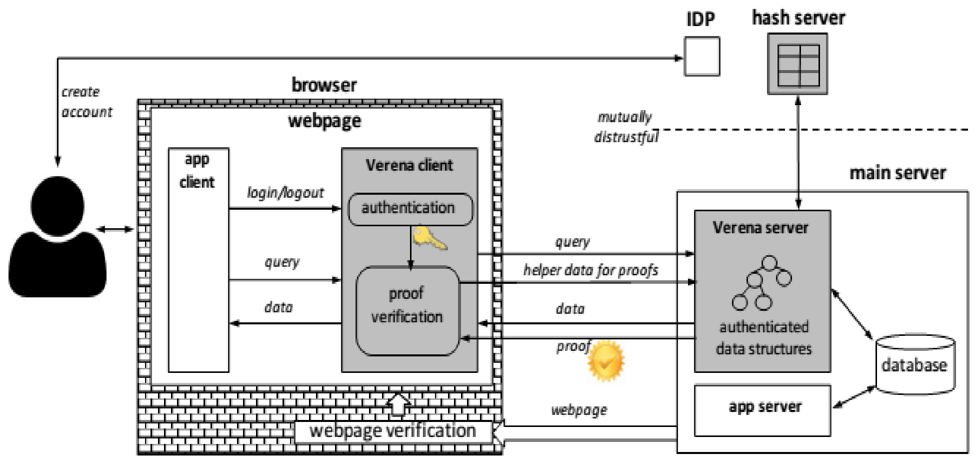


Figure 3-6 Verena Framework [3]

In the second case we have dynamic web applications, where web pages are not static, they change every time when the end user accesses the pages of web application. These pages get data from databases and display on web pages. Now integrity protection in dynamic web application becomes more challenging. The end user should have some proof from the server which he can use to verify the data.

Nikolaos Karapanos, Alexandros, Filios, Raluca Ada Popa, Srdjan Capkun [3] presented the solution. In their solution, they proposed protection of integrity in fully compromised environment. They had introduced a new server which used to store the proof (hash) of each user as shown in figure 3-6 [3]. They had used authenticated data structures to build proofs from data. Red black binary Merkle hash tree was used to calculate proof for data. Root of each tree was saved on the hash server. Main advantage of hash server was, if main server was compromised then the end user could also detect the integrity of data from proofs from main server and hash server.

This model was not very much expressive in term of databases queries. Aggregation queries did not have proof for its results, for proof verification it was dependent upon the result of simple queries. In this solution, there was an issue that client (who is operating at browser end) had to communicate with hash server via main server. In case of compromised server, attacker can return wrong hash against user-id to the end client. As it was already mentioned in the vSQL [33] that authenticated data structures are more efficient, but it supports limited number of database queries.

Chapter 4

4 Research Methodology

In this chapter, we have discussed the methodology of database query verification solution in fully compromised environment.

4.1. Problem Overview

Security of web applications and databases is always dependent on the security of servers. Once server is compromised, the end user using web application cannot detect at browser end that he is getting verified data. For this, database query verification is required. In the worst case, when attacker has full access to server, this solution should detect that server is compromised. If the end user sends a query for execution, attacker can return wrong data or data of another user. This wrong result can have serious impact on the end client. This problem is widely faced in many applications in which data sender is trusted but server (storing data) is not trusted. So, there should be database query verification, so the end user can verify results in fully compromised environment.

4.2. Proposed Solution

In our solution, we have used Merkle Hash tree based Authenticated data structures, which are used for construction of proof (in write operation) and proof verification (in read operation). In Chapter 2, section 2.3 Merkle hash tree is explained in detailed. During database write operation Merkle hash tree is processed and root hash of tree is updated after write operation. This root hash is sent to blockchain for storage. For database read operation Merkle hash tree is used for verification of the end user's data which is returned from databases after executing the query. Query result with proof helpers (Chapter 2, Section 2.2) are sent to the end client. The end user verifies the result with help of proof helpers, data return from query and root hash from blockchain. Root hash is calculated by the end user from data and proof helpers, and it is compared with root hash from blockchain. Our customized implementation of Merkle hash tree helps us to verify aggregates queries efficiently.

4.3. Design and Architecture

Figure 4-1 shows the architecture of verifiable database query solution.

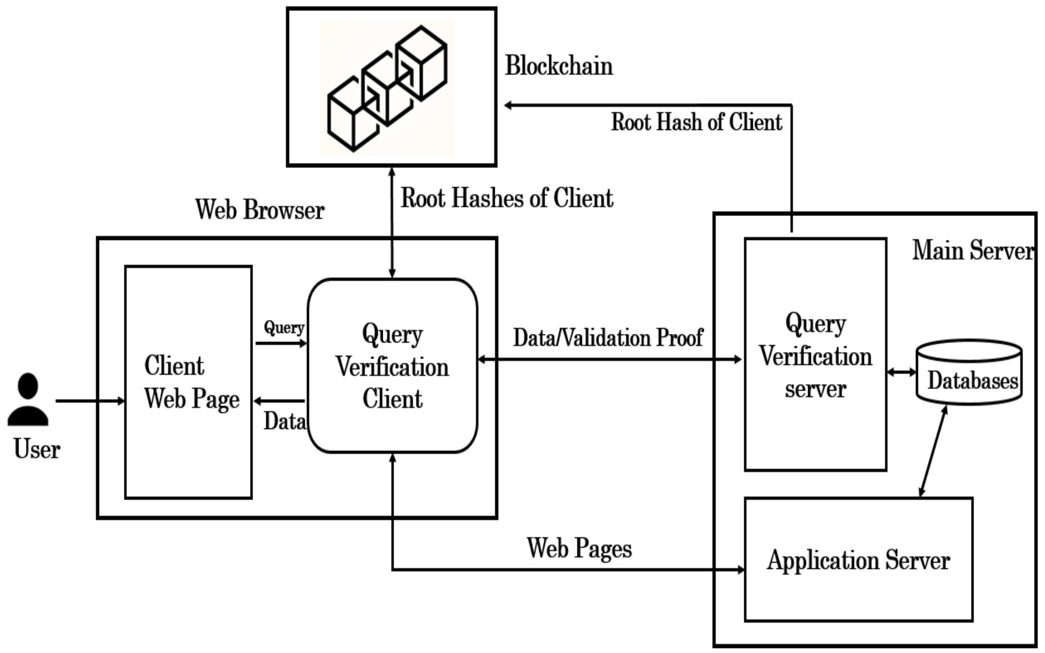


Figure 4-1 Database Query verification architecture

Now, we discuss each component of the solution in detail.

4.3.1. User

The user is end client who is using web application. He can view, modify and delete his data which is stored on database. Users can have different roles like one user can not view data of another user, or one user can view data of another user but cannot delete it. User roles are implemented in web applications according to requirements. Like in the medical web applications, physician can view patient data to prescribe medicines. A patient cannot view, update data of another patient.

4.3.2. Client Web Page

Client web page is the simple HTML page which is used to display client information with help of CSS, JavaScript, Ajax and jQuery in web browser. These web pages are generated by server-side application. This application can be implemented in any technology like PHP, java servlets, ASP .Net, node.js. Web pages can be static or dynamic web pages generated by web application server. The end user can interact with databases with help of client web pages in his web browser. He can request data by sending query to databases however interface data query is dependent on the implementation of application. Client web pages are generated by web server and sent to the end client for view.

4.3.3. Query Verification Client

This is main component on the end user side, which communicates with application server and blockchain for proof verification of query results (sent by the database). Query is executed on database server, and result is returned to the query verification client with proof helpers (Chapter 2, Section 2.2). The proof helpers from untrusted server are verified with the help of the proof resides on the blockchain. Query verification client constructs Merkle hash tree using data (query result) and proof helpers, so root hash is calculated and matched with root hash which is coming from blockchain. Blockchain ensures two basic properties of integrity i.e. freshness and correctness. So, if blockchain proof is verified with proof helpers from untrusted server then it means that the resultant data is verified.

4.3.4. Databases

Web applications use databases to store data on it. Databases can be of any vendor depending upon the requirements of application. Databases may reside on same server where web application is running. It can also be outsourced to third party [40]. In our architecture, database is residing on same server where web application is deployed. Our design is independent of location of database, but it should be accessible to web application server over the network. Web application can communicate with database server. Popular databases are already discussed in chapter 2, section 2.1. The end user requests any data from database. Dynamic web pages are built using the results of query (executed on database) and displayed to the end user in his web browser.

4.3.5. Application Server

Web application is deployed on web application server. It communicates with databases to access the stored data. All requests by client are entertained by the application server. It returns requested web pages for the end client. Dynamic web pages are generated by web application server using data from databases. It is sent to browser of the end client where he can view these web pages. There are many types of web application servers. It depends upon the requirements of application for choosing web application server. Most common web servers are Apache HTTP Server, Ngixn, and Microsoft IIS. [41]. Server end of any web application can be developed using any programming language. These servers provide TLS/SSL based encrypted communication cannel with the end client.

4.3.6. Query Verification Server

This is the most important component in architecture of query verification. This query verification server is used for proof creation, proof updation and proof helpers (Chapter 2, Section 2.2) calculation for verification of database query. In verification process, this component is used to calculate and send proof helpers to the end client. Authenticated data structures are used for proof verification and creation process. Authenticated data structures are used in our design are based on Merkle hash tree.

In section 2.2 of chapter 2, it is explained that Merkle hash tree is built on the hashes of data on leaves. For every end user, who needs query verification solution, Merkle hash tree is built for that end user. So, there is forest of trees on main server. For four users, forest of Merkle hash tree is shown in figure 4-2.

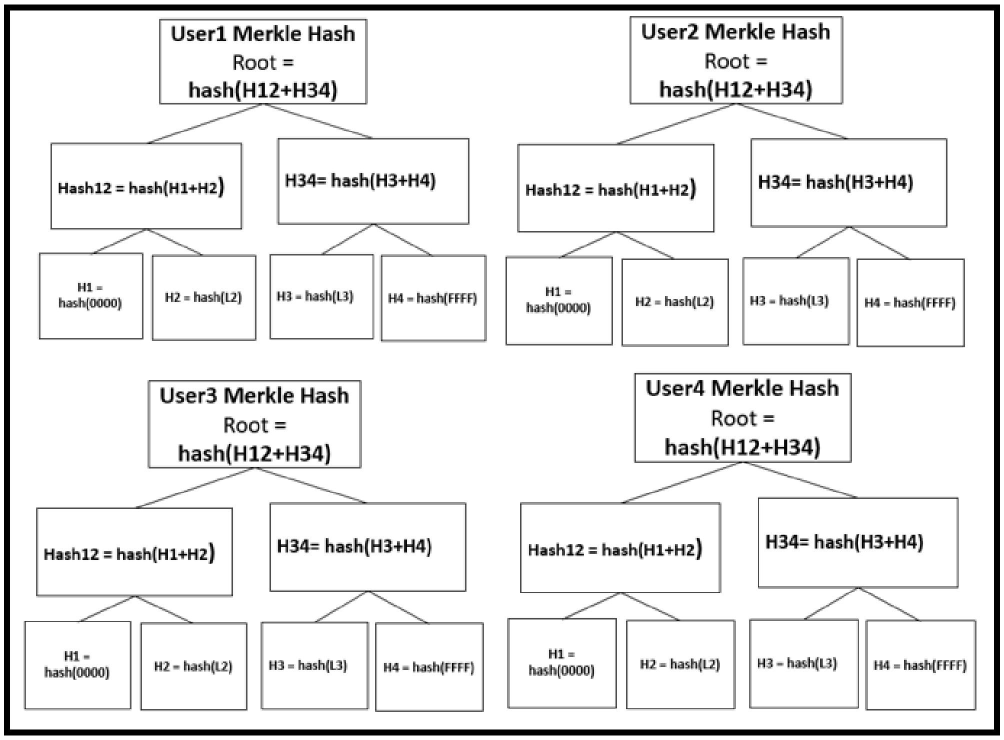

User1 Merkle Hash Root = hash(H12+H34)
Hash12 = hash(H1+H2)
H34= hash(H3+H4)
H1 = hash(0000)
H2 = hash(L2)
H3 = hash(L3)
H4 = hash(FFFF)
User2 Merkle Hash Root = hash(H12+H34)
User3 Merkle Hash Root = hash(H12+H34)
User4 Merkle Hash Root = hash(H12+H34)

Figure 4-2 Forest of Merkle hash trees

There are two types of queries: *simple* and *aggregated*. Simple queries are those in which the user gets data at particular matching field of data collection in databases i.e. “find record of user at 9pm, 10th January 2018”. Aggregated queries are those in which the end user get data on specified range field i.e. "find sum of values for a user from morning time to evening". Aggregated queries are listed in table 4-3.

In databases, there are two types of operation *write* and *read*. **Write** operation means when the data is changed in databases, it may be insert, update or delete. Some example write operation in databases are given table 4-2. In **read** operation, user can get data from database. Common read operation for database are given in table 4-1

Table 4-1 Database Query Read operations

Read Query	**Detail Operation**
Find	Get value from no-SQL database
Select	Get value from SQL database
Sum	Get sum on range
Average	Get Average on range
Max	Maximum value on range
Min	Minimum value on range
Count	Number of values on range

Write operation on databases are given in table 4-2

Table 4-2 Database Query Write operations

Write Query	**Detail Operation**
Insert	Add new value in database
Update/Modify	Update existing value in databases
Delete	Delete existing value from database

We explain the design of proof creation, which is used in write operation. If the end user performs a write operation, (insert something in database) then data of the end user is also sent to query verification server. In query verification, Merkle root hash tree is checked for the end user in the databases. If tree already exists, then data is hashed and added as new leaf in existing tree.

Table 4-3 Database Aggregate Queries

Aggregate Query	**Detail Operation**
Sum	Get sum on range
Count	Number of values on range
Average	Get Average on range
Max	Maximum value on range
Min	Minimum value on range

In this process new root hash of Merkle hash tree is calculated. This root hash is sent to blockchain. If user is performing write operation, then new Merkle hash tree is created. To ensure completeness, two leafs are added in Merkle tree one is starting leaf which is “0000” and end leaf is “FFFF”. After creating new Merkle hash tree, new data is added as leaf as shown in figure 4-3 and 4-4.

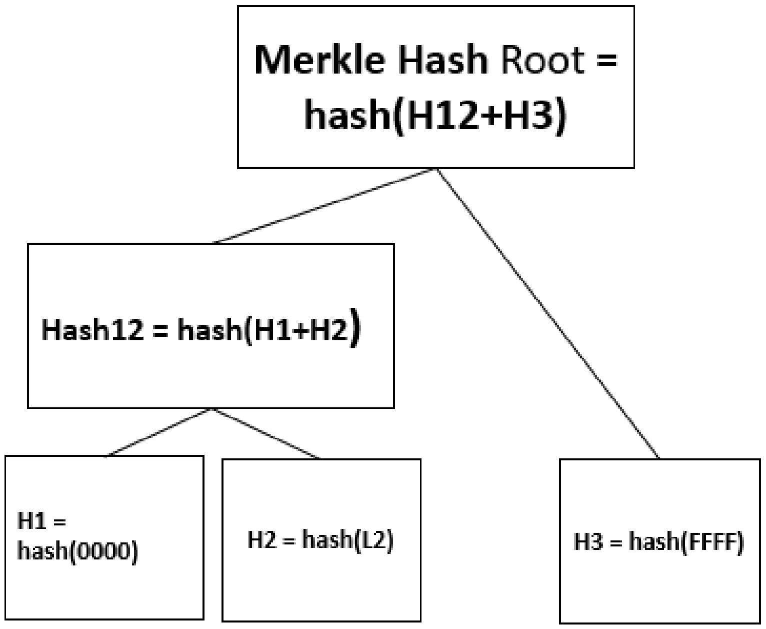


Figure 4-3 Add leaf in Merkle hash tree

New root hash is calculated that is sent to blockchain. If the end user wants to update a value, then tree of user is traversed to find the value which is going to update.

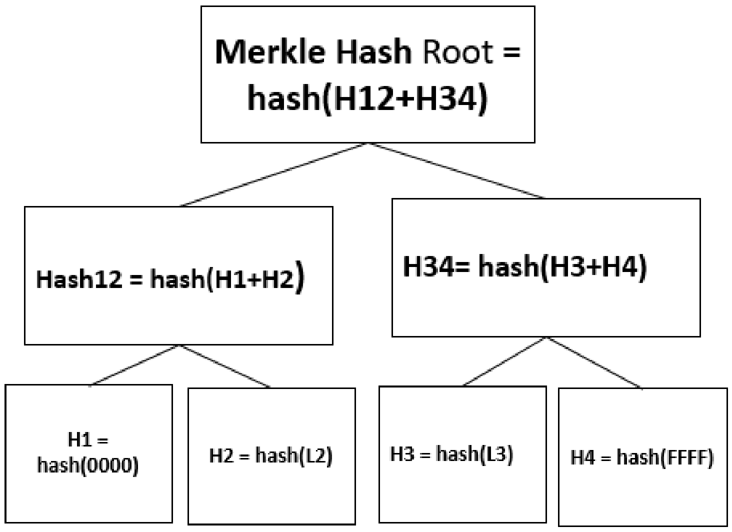


Figure 4-4 Merkle hash tree with four leafs

The existing leaf hash is replaced with new hash of updated data, root hash of Merkle tree is also updated. This updated Merkle root hash is also sent to blockchain. Similarly, if the end user wants to run some delete query, then relevant hash value is searched in Merkle hash tree of the end user. The leaf containing deleted hash value is removed as leaf. So, Merkle root hash is calculated again and this is updated on blockchain.

Now come to verification part of the data, if the end user performs a read operation, then query verification server calculates proof helpers (Chapter 2, Section 2.2) against required the data. These proof helpers are hash values in adjacent leaf and nodes on upper order of tree. If the total number of leaves are “n” then “$\log_2(n)$” is the numbers of proof helpers. This is the benefit of Merkle hash tree that number of nodes in proof helpers are less as compare to the total number of leaves. So, these leaves are sent to query verification client (at browser of the end user) with the resultant data from databases. Now query verification client constructs root hash with help of proofs as explain in section 4.3.3.

To provide the integrity for aggregated queries, we have proposed two types of the solutions to solve this problem. So, the best solution is recommended after performance evaluation of solutions.

4.3.7. Proposed Solution-1:

In **1st solution**, the end user has two Merkle hash trees, one tree for simple queries and another tree for aggregated queries. Both trees are updated on insert. In 1st Merkle hash tree only one value gets hash and added as leaf in the tree. In 2nd Merkle hash tree (for aggregated queries), sum, average, min, max and count is taken with using pervious data. These values are hashed together, and added as leaf in the tree (Merkle hash tree for aggregated queries). Both trees have their root hashes. These are sent on the blockchain. Theoretically, the benefit of this design is aggregated and simple trees have separated proofs which provide more trust on query result. For example, we have table of “user1” value is store against key as shown in table 4-4:

Table 4-4 Sample Data Values

Key	Value
1	X
2	Y

In 1st solution we have two Merkle hash trees as shown in figure 4-5. Key and value is hashed, and this hashed valued is added in leaf of 1st Merkle hash tree (implemented for simple queries. This tree grows as the data of user is increased. This tree is saved in databases on main server for better performance. Root hash of tree is sent to blockchain as proof, which is used in verification step.

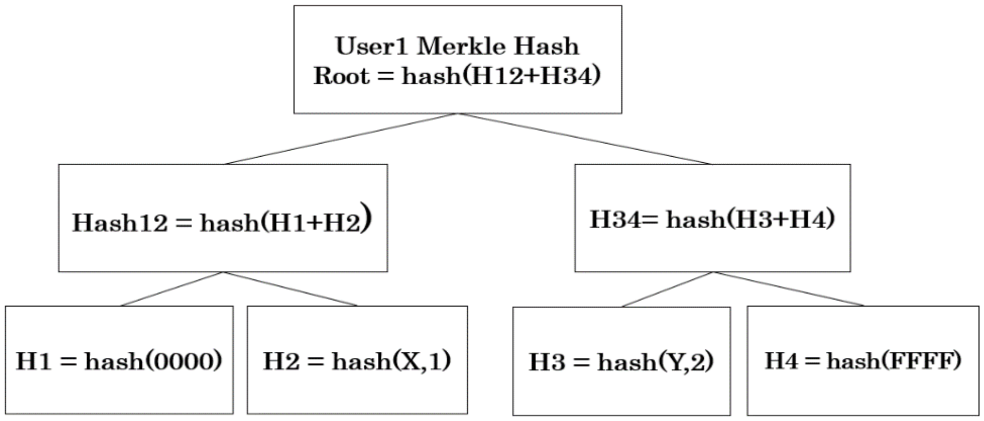


Figure 4-5 Proposed Tree for Simple Queries

2nd Merkle hash tree is used for query verification of aggregated queries. 2nd tree is shown in figure 4-6.

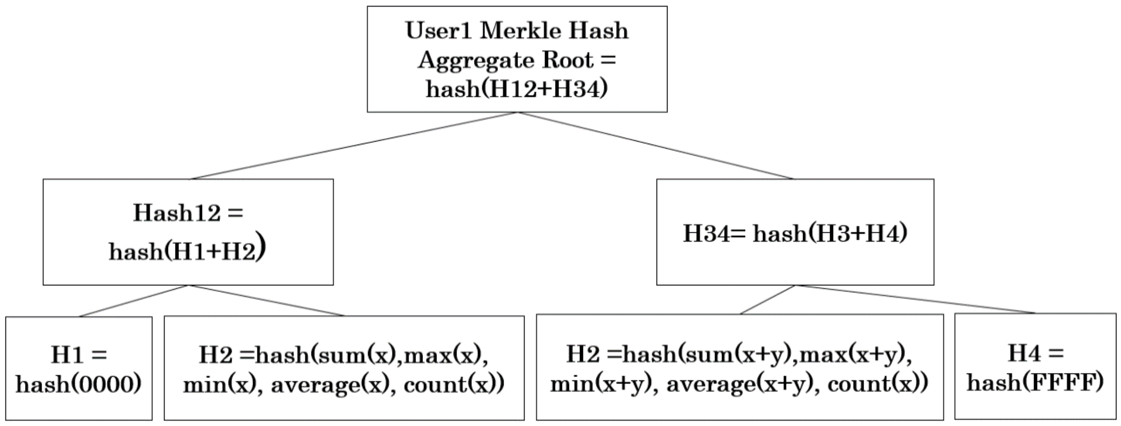

User1 Merkle Hash Aggregate Root = hash(H12+H34)
Hash12 = hash(H1+H2)
H34= hash(H3+H4)
H1 = hash(0000)
H2 =hash(sum(x),max(x), min(x), average(x), count(x))
H2 =hash(sum(x+y),max(x+y), min(x+y), average(x+y), count(x))
H4 = hash(FFFF)

Figure 4-6 Proposed Tree for Aggregated Queries

In this tree, we have shown sum, count, average, maximum and minimum. These result are pre-calculated using current data and pervious data. This tree grows as data is increased. This tree is also stored in database. Tree is loaded, when proof helpers (Chapter 2, Section 2.2) are required for query verification client. Root hash of this tree is also sent to blockchain.

In this solution, we have to maintain two tree for each user. We have two root hashes on the blockchain for each user. Query verification client requests root hash depending upon the type of query. If user sends simple query for execution, then query verification client requests root hash of 1st Merkle hash tree which is used simple queries.

4.3.8. Proposed Solution-2:

Our proposed **solution-2** is designed with only one Merkle has tree, to provide proofs for simple and aggregated queries in our query verification solution. In this case, only Merkle root hash, which is used for verification of both simple and aggregated queries. We calculate sum, average, min, max and count on the end user data with pervious data of same end user. This is aggregated data is concatenated with the data from the end user and hashed. This hash value is added as leaf in Merkle hash tree (for aggregated queries). We have constructed Merkle hash tree for "User1" using data in table 4-4. Tree of "User1" is shown in figure 4-7:

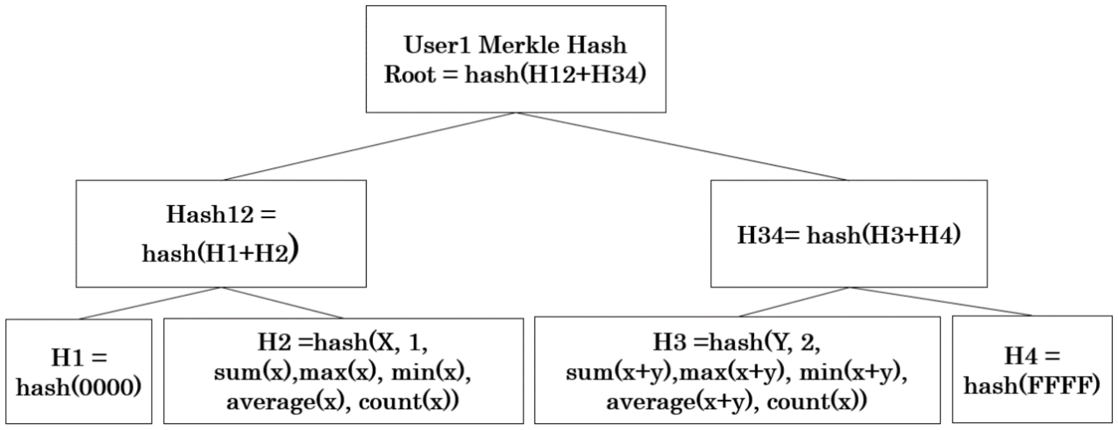

User1 Merkle Hash Root = hash(H12+H34)
Hash12 = hash(H1+H2)
H34= hash(H3+H4)
H1 = hash(0000)
H2 =hash(X, 1, sum(x),max(x), min(x), average(x), count(x))
H3 =hash(Y, 2, sum(x+y),max(x+y), min(x+y), average(x+y), count(x))
H4 = hash(FFFF)

Figure 4-7 Proposed solution-2

This tree also grows when data increases. This solution provides verification of database queries for simple as well as aggregated queries. If aggregated query has ranges that is not matched with exact value in leaves, then this query is parsed into subparts for verification of data.

Query verification server provides proof helpers (Chapter 2, Section 2.2) when client needs to verify the result of database query. In case of solution-2, for any type of query (simple or aggregated) only one tree is used. For example, "User1" wants to verify the result where key is "2". In this case, query verification client asks for root hash of user from blockchain, and also proof helpers from query verification server. Query verification server returns H3, H4 and H12 as proof helpers as shown in figure 4-8.

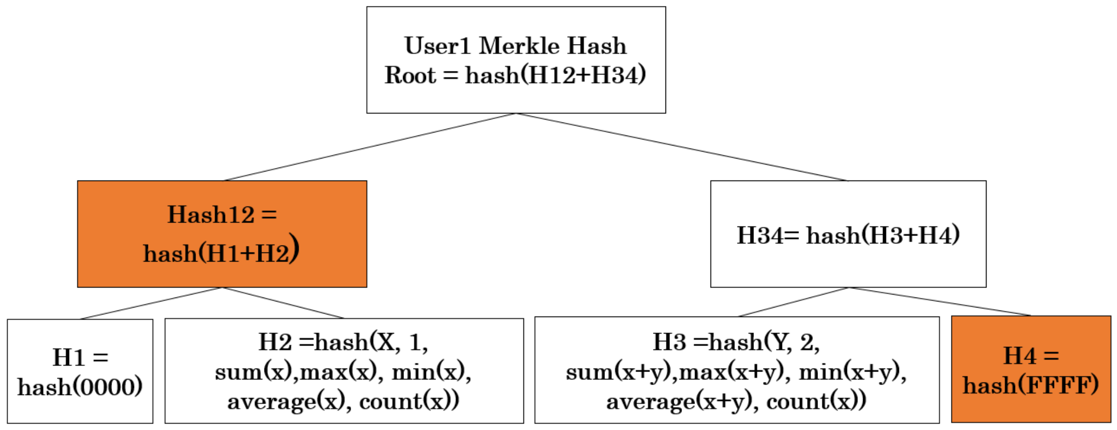

User1 Merkle Hash Root = hash(H12+H34)
Hash12 = hash(H1+H2)
H34= hash(H3+H4)
H1 = hash(0000)
H2 =hash(X, 1, sum(x),max(x), min(x), average(x), count(x))
H3 =hash(Y, 2, sum(x+y),max(x+y), min(x+y), average(x+y), count(x))
H4 = hash(FFFF)

Figure 4-8 Proof helpers from Merkle hash tree in solution-2

Query verification client calculates root using query data, H4 and H12.

$$Root\ hash\ =\ Hash\ (H12\ +\ Hash\ (H3 +\ H4))$$

Equation 4-1 Root Hash Calculation

This root hash is compared with root hash from blockchain. If both are matched then result is verified. We match the result from database with result stored in tree. If aggregated query range is not matched exactly with the tree, then we split the query in sub queries. Proof helpers (Chapter 2, Section 2.2) are found from tree for those sub queries. If these sub queries are verified, then main query is also verified.

4.3.9. Blockchain

Blockchain is used in our design to store Merkle root hashes of each end user. When the end user generates Merkle root hash after each write operation, new root hash is calculated and sent to blockchain. Query verification client access root hashes of each end user from blockchain to verify the poof helpers that are returned from query verification server. Blockchain ensures freshness and correctness of the proof.

4.4. Attacker Model

Our solution works in very strong attacker model for the main application server. Main server is untrusted and attacker can access it completely. Attacker can return wrong queries result to the end user. It can manipulate the data in databases. In real world, attacker may hack the main server where databases and web applications reside. Databases are outsourced to third party that cannot be trusted by the end user. As we already discuss that blockchain are used to verify queries at client end. Blockchain is trusted, as integrity of the data on blockchain is ensured. So, root hashes on blockchain are trusted. If they are changed by some attacker, then it can be detected easily. In this attacker model, adversary has full control of main server. Main server is treated as untrusted and block chain are treated as trusted.

4.5. Communication between Client, Server and Block Chain

The end user is defined as client, he is user of the web application. He communicates with main server where databases and web applications are deployed. Client also communicates with blockchain to get root hashes stored on it. Server also communicates with blockchain to write the root hash after every write operation. Sequence of communication for write and read operation is explained.

4.5.1. Write Operation

When there is write operation, client sends data to server, server writes that data in databases. Flow of communication between web browser, server and blockchain for write operation is shown in figure 4-9.

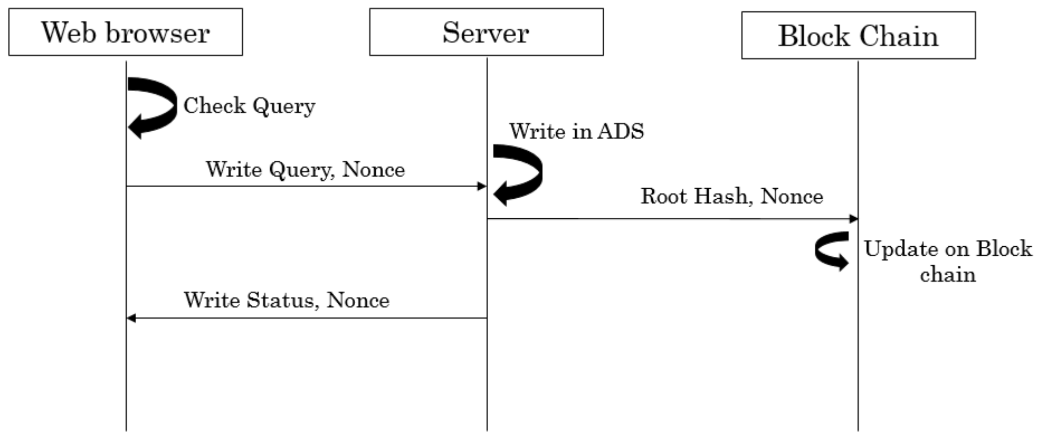


Figure 4-9 Communication for Write Query

Query verification client checks the query first, if write query, then it forwards query with nonce to query verification server. Query verification server identifies relevant Merkle hash tree (ADS) for that client. It performs update, delete or insert operation on the tree. So, new Merkle root hash is calculated after every write operation. This new root hash is sent to blockchain where it is shared among nodes on blockchain. Pseudo code for Query verification client is given in figure 4-10. CheckQuery is method which identifies the type of query operation (read or write). WriteStatus Okay means write operation is completed by query verification server.

```
1- Input: Query of the user
2- checkQuery() // Query verification client checks query entered by user
3- If queryType () == Write // tell about read or write operation
      Then
            Forward to Query verification server
4- WriteStatus == OK // get write status from query verification server
5- End
```

Figure 4-10 Pseudo code of Write Operation for Query verification Client

```
1- Input: **Query** from Query verification Client
2- If OperationType () == Write // Query verification Server checks Write Operation type
     Then
          Write a new node in ADS
3- If OperationType () == update
     Then
          Update existing node in ADS
4- If OperationType () == Delete
     Then
          Delete existing node in ADS
5- Roothash = getRootHash()   // Get New root hash from ADS after write operation
6- SendtoBlockChain (Roothash, user_id) // Send new root hash to blockchain
7- Send WriteStatus OK to query verification client
End
```

Figure 4-11 Pseudo code of Write Operation for Query verification Server

Pseudo code for Query verification server is given in figure 4-11. When a user performs insert, update and delete operation, ADS are updated accordingly. SendtoBlockChain is used to send root hash of user to block chain against his user_id. Write status is send to user, which indicates to user that write operation is performed successfully.

4.5.2. Read Operation

For read operation, client sends query for the data from databases. Query verification client checks query and forwards to query verification server. It also request blockchain to send root hash of the client.

Query verification server calculates proof helpers (Chapter 2, Section 2.2) which explained in section 4.3.6. This proof helpers, and data from query results are returned to query verification client. Query verification client receives requested root hash from block chain, the data (query result), and the proof helpers from main server. It performs calculation with help of proof helpers and data and calculates the new root hash. This new root hash is compared with root hash returned from blockchain. If both are matched then message is displayed on web page that "data is verified successfully" otherwise error message is shown to user. The complete communication is shown in figure 4-12.

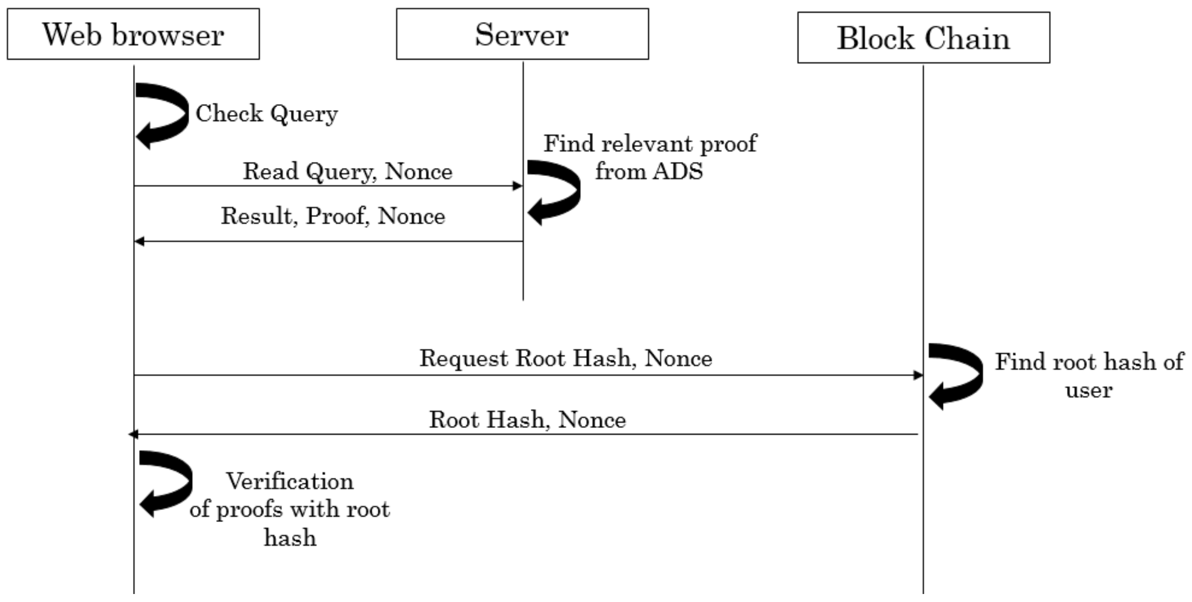


Figure 4-12 Communication for Read Query

Pseudo code of read operation for query verification client is shown in figure 4-13. CheckQuery detects read or write operation.

```
1- Input: Query from user
2- CheckQuery () // Query verification client checks query entered by user
3- If queryType () == Read // tell about read or write operation
        Then
        i-      rootblockchain = get Root hash From Block Chain(user_id)
        ii-     {Proof Helpers, Data} = Get Proof Helpers (user_id)
4- NewRoot = CalculateRoot (Proof helpers, data) // Root is calculated using proof helps and data of query
5- If rootblockchain == newRoot
        Then
                Integrity of Data: Pass // user is notified with integrity pass status
        Else
                Integrity of Data: Fail // user is notified with integrity fail status
End
```

Figure 4-13 Pseudo code of Read Operation for Query verification Client

Pseudo code of read operation for query verification server is shown in figure 4-14. GetProofHelper is a method which is describing the mechanism of get proof helpers from the ADS. If for aggregated queries range parameters are not matched with stored in ADS, then query verification server splits the query in sub queries. These sub queries are verified using proof helpers from ADS. Main query is said to be verified, if it all sub queries are verified.

```
1- Input: **Query** from Query verification Client
2-  **If** queryType () == Read // tell about read or write operation
      **Then**
            ProofHelpers = **GetProofHelper** (user_id)
3-  SendProofHelper (ProofHelpers, user_id) // send proof helpers to query verification client
**End**
-----------------------------------------------------------------
1- **GetProofHelper** (user_id) // GetProofHelper is used to find proofs for a user
2- **If** query == simple
      **Then**
      Find nodes from ADS
3-  **elseif** query == aggregated
      **Then**
            **If** ranges exist
               **return** nodes
            **else**
               **Split** query in sub query
               **find** proof helpers for sub query
               **return** nodes
**End**
```

Figure 4-14 Pseudo code of Read Operation for Query verification server

Chapter 5

5 Prototype Implementation

In this chapter, we have discussed the implementations of our proposed solutions for databases query verification and its, integration with medical web application.

5.1. System Overview

We have discussed the building blocks of our solution in chapter 2 and chapter 4. Medical web application is used to integrate with our query verification solution. Medical web application has two clients, one is the patient and second is the physician. In this test medical web application, we have patients' categorization according to their disease. Physician can view data of each patient of his specialty, for example only cardiac physician can view heart patients. Server end of medical web application is integrated with databases. This is dynamic web application, where web pages depend upon the databases queries result.

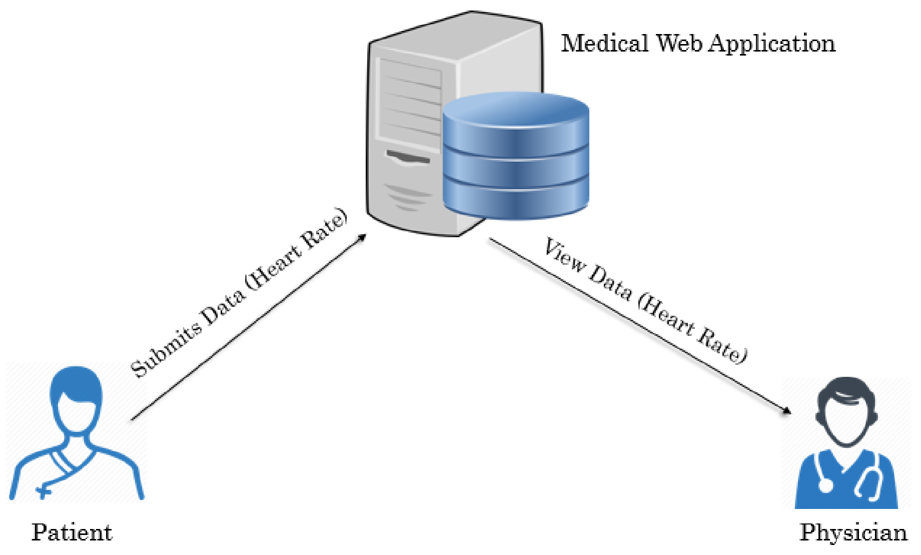


Figure 5-1 System Overview

Data of patient is added by some remote measuring devices like pace maker, blood pressure measuring devices [42]. It is stored in databases. The remote monitoring is more useful in medical for both physicians and patients. Authenticated data structures (Merkle Hash Tree) is built when data is added, when physician gets the data of patient, also gets proof helpers for that patient. These proof helpers and data are used for verification of executed query in medical web application. If some adversary changed the data of patient, then physician is notified. Figure 5-

1 shows that the patient send data to web server where it is stored in databases. Physician can query on data of patient via graphical interface of medical web application.

5.2. Implementation Tools

Medical web application is developed in Node.js [43] using Meteor framework [44]. MongoDB [45] is used for data storage. Meteor [44] uses Node.js on server-side of medical web application. Meteor is modern platform to develop modern web applications. MongoDB is NO-SQL database which is used in many famous web applications. MongoDB is document oriented database. It uses JSON (JavaScript Object Notation). It is more useful when some integration with JSON based API is required [46], [47]. Meteor builds web pages on HTML, CSS and JavaScript. JavaScript is a scripting language which is used in Node.js.

Node.js has node package manager (npm), which has already developed packages that are free to use. It is used globally by around 11,000,000 [48] JavaScript developers and it has around 60,000 [49] packages available, which can be used anyone by importing it in his application. We have used "Atom" [50] source code editor for development of medical web application. Atom is open source utility, it has support for Node.js as well many other programming languages. We have used SJCL library [51] For cryptographic hash functions (like SHA-1, SHA-256). SJCL is developed using JavaScript by the Emily Stark, Mike Hamburg and Dan Boneh in Stanford University [51]. BlazeJs [52] is used for development of web interface of medical web application. We have created reactive HTML templates with help of this library. It also manipulates and merges these templates.

We have implemented authenticated data structures as Merkle Hash tree. Merkle-tools [53] is the npm package for Merkle hash tree which is used in this medical web application. It has methods to create Merkle hash tree, proofs and verification of proofs. We have stored authenticated data structures in MongoDB for better performance [54]. Merkle-tools provides methods for adding leaf, get leaf, get proof, and validate proof. Blockchain are used to have root hash of each user. When query verification client needs root hash for verification, client can get from blockchain. Blockchain has different implementations, we are using private blockchain implemented HyperLadger [55].

5.3. Medical Web Application

Medical web application has two roles, patient and physician. Patient and physician are registered by using on registration web page of medical web application. In web medical application, patients are registered using their email address, password and disease group. In test medical web application, disease groups are classified into three type, heart, diabetic and blood pressure.

Physicians can also register using their email address, password and his specialty i.e. heart, diabetic and blood pressure. Physician of same specialty can view patients of only same category, like heart specialist can see only heart patient. Patient can add reading (of heart rate, sugar level, blood pressure) against the timestamp. These readings are added by patient itself or remote reading devices (pacemaker, blood pressure measuring devices) can send data to medical web application. When a physician logs on to medical web application, he can search a patient records and add remarks to it. He can execute aggregated queries like, sum, average on medical record of patient for different range of timestamp. Medical web application returns the resultant data from databases.

To provide verifiable database queries, we have integrated query verification client at the client end of this application, and query verification server is at server end of this application. Medical web application is designed with two collections in MongoDB. These collections names are "Users" and "Records". We have created two more collections named "Users_MHT" and "Users_MHTAggr" to implement proposed solution-1. But we have created only one collection named "Users_MHT2" for implementation of the proposed solution-2. Three collections are created which are used in implementation of query verification. Details of all collections is given in table 5-1.

Table 5-1 Databases Collections

Collection Name	**Details**
Users	Two type of users i.e. Physician and Patient
Records	This collection contains records of patients
Users_MHT	It contains Merkle hash tree of users for simple queries (proposed solution-1)
Users_MHTAggr	It contains Merkle hash tree of users for aggregation queries (proposed solution-1)
Users_MHT2	It contains Merkle hash tree of users for both simple ad aggregated queries (proposed solution-2)

Every collection has fields which are used to store different values which are used in query verification solution. We have discussed each collection and its fields in detail as given in table 5-2.

Table 5-2 Collection Fields

Collection Name	Collection Fields
Users	(user_id(email), CreatedAt, password, login_token, profile{account_type, disease_group})
Records	(user_id, disease_group, reading, remarks, timestamp)
Users_MHT	(user_id, patient_MHT, timestamp)
Users_MHTAggr	(user_id, Agr_Array , Agr_Node_Hash, timestamp)
Users_MHT2	(user_id, data_Array , MHT_Node_Hash, timestamp)

5.3.1. Collection: Users

This collection contains the information of users which may be physician or patient. *User_id* is unique id of user which is email address. *CreatedAt* is time at which user is created. This time can be used for user verification time. *Password* is used to store the password of the user and it is saved in database as encrypted data. *Login_token* is implemented to save session of user. In Profile, *account_type* field is used to classify the patient and physician, and *disease_group* is saving types of the diseases (heart, diabetic or blood pressure). This collection grows as number of patients or physician increased in medical web application.

5.3.2. Collection: Records

This collection is used to store patients' reading. *User_id* is id of the patient which is linked from collection: Users. *Disease_group* defines the categories of disease. *Reading* is the measurement value which is coming from remote medical devices or patient can add it. *Remarks* is used to save comments entered by the physician for a patient. *Timestamp* is used to save measurement time of the reading. This collection is also growing, as patients' readings are increased. Patient can only update *reading*, *timestamp* fields. Physician can only update the *remarks* against *reading* of patient. Physician can view data of patient by searching data in this collection in medical web application.

5.3.3. Collection: Users_MHT

This collection is used to store the authenticated data structures which are implemented as Merkle hash tree of every patient for simple queries in proposed solution-1. Merkle hash tree is serialized before storing in the database. When Merkle hash tree is loaded from database, we have to un-serialized tree first. Number of rows in collection is equal to number of patients. One patient has only one Merkle has tree in database. We store tree of the patient in field *patient_MHT* against it *user_id*.

When patient enters new record, same row of patient is updated in Users_MHT. When a new reading is entered by patient against user_id in collection: Records, we check Users_MHT that it contains tree for that *user_id* or not. If *user_id* is exists, then we read *patient_MHT* against that *user_id*. This tree is un-serialized and loaded in memory. New leaf is added in this loaded tree, and new root hash of the tree is calculated by query verification server. New root hash is sent to blockchain and this new Merkle hash tree is serialized again and updated in Users_MHT. This process is also repeat for updating any reading for patient in Records collection.

When patient delete reading in records, corresponding leaf of Merkle hash tree is deleted by query verification server and new updated Merkle hash tree is updated in Users_MHT. Patient's loaded tree is traversed, relevant leaf is identified and removed, and new root hash is calculated. This root hash is updated on blockchain and tree is updated in Users_MHT.

For each patient, if number of readings are increasing, row for that patient remains one in this collection, but size of patient_MHT is also growing. This collection is used for query verification server to calculation of proof helpers for simple query. When query verification server receive simple read query, it loads the relevant tree from this collection. It finds relevant proof helpers as explained in chapter 2, section 2.2.

5.3.4. Collection: Users_MHTAggr

This collection is used to store Merkle hash tree which is built for aggregated queries. This collection has user id (*user_id*), array of aggregation (*Agr_Array*) which contains sum, average, maximum, minimum, and count, and hash of array of aggregation (*Agr_Node_Hash*). We have stored *timestamp* in this collection, which is used in range queries. This collection is used to provide proof for aggregation queries. This collection grows as Records collection grow for each patient. If a reading is inserted in records, then aggregated array is calculated by new reading with last entry of same patient in Users_MHTAggr. New reading is added in sum, count is incremented, and average is calculated on new sum and new count. Maximum and minimum are also calculated by comparing new reading with pervious maximum and minimum. This newly calculated aggregated array, hash of this aggregated array and timestamp are also inserted in this collection with patient's user_id.

When some reading is updated in records against some timestamp, we find the already existing row from this collection against timestamp and user_id. Updated reading is used to find new updated aggregated array for that timestamp, and we calculate hash of updated aggregated array, then these new values are updated in this collection. We have to update the all next Agr_Array from timestamp. These are updating using pervious reading and new updated reading. For example, new sum is calculated by minus the old reading from sum and then adding new updated reading. Hashes are also calculated again.

When a patient deletes some reading, particular row at that timestamp is deleted and values at next timestamp are updated, and new hash is taken of updated aggregated array, which is stored in database. This collection is used in proposed solution-1 for aggregated queries and this is used by query verification server to calculate proof for aggregated queries.

5.3.5. Collection: Users_MHT2

This collection is used to store leaves nodes of Merkle hash tree for each patient. This is growing collection, number of rows increase as readings of patient increase in Records collection. This collection has these fields: *user_id*, *data_Array*, *MHT_Node_Hash*, and *timestamp*. Field *user_id* has id of each patient which is used to find the data of each patient. Field *data_Array* contains, reading (reading from Records collection), sum, average, count, maximum and minimum. Sum, average, maximum and minimum are calculated till that *timestamp* using pervious value of *data_Array*. Sum is calculated using new reading and pervious sum from data_Array. *MHT_Node_Hash* is hash of *data_Array*, and this is leaf of Merkle hash tree of that patient. *Timestamp* is used for time at which reading of patient is measured.

This collection is used for proposed solution-2. In our solution-2, we build only for Merkle hash tree for both simple and aggregated databases queries. If new *reading* value is inserted in Records collection, then a new record is also inserted in Users_MHT2. At that timestamp, we calculate sum, average, count, maximum and minimum, and update in *data_Array*. This sum is calculated by using data_Array at pervious timestamp. Hash of data_Array is taken and stored in *MHT_Node_Hash*. These all values are inserted in this collection. On update and delete operation, row in collection at that timestamp is deleted or replaced with updated reading. This collection is used by query verification server. Query verification server uses this collection for calculation of proof helpers for simple and aggregated queries. These proofs are sent to query verification client, where physician verify query results with these proof helpers.

5.4. Query Verification Solution Implementation

We have integrated queries verification solution with this medical web application. **Query verification client** is integrated at web browser and **Query verification server** is integrated at the server end. Both query verification client and server are developed in node.js. When a patient is registered, Merkle hash tree is created for it with two leaves one is 0000 (starting leaf) and other FFFF (end leaf). These values are used for checking the completeness of data. It helps physician to verify that result returned from database is complete. When a new record is inserted by patient, new leaf is added between 0000 and FFFF as explained in chapter 4.

Root hash is sent to blockchain using its API. SHA-256 bit is used for hashing of data for leaves of Merkle hash tree. When patient logs out then Merkle hash tree is saved in database in serialized form. When a patient logs in medical web application, patient is authenticated, and Merkle hash tree is loaded for that patient from database by query verification server. Process of tree loading is very simple. In database, Merkle hash tree is in serialization form. This Merkle hash tree is un-serialized, assigned to Merkle hash tree object. So that new leaf can be inserted, updated or deleted.

Patient inserts, updates or deletes some reading, loaded Merkle hash is updated accordingly. For update of record against timestamp, leaf node of Merkle hash tree is searched and updated with new hash of updated record, and root hash of tree is also updated on blockchain. If delete query is sent by patient, then query verification server also deletes the leaf after searching the leaf in Merkle hash tree for which delete is called. New root hash is calculated and stored on blockchain. Patient logouts from medical web application its tree is also saved in database after serialization. This is generic flow of implementation for proposed solution-1 and solution-2.

5.4.1. Proposed Solution-1 Implementation:

We have explained generic flow in section 5.4. Now we are explaining solution-1 implementation and its flow. There are two Merkle hash trees for each patient. In **proposed solution-1**, query verification server uses two database collections (Users_MHT and Users_MHTAggr) for proof calculation and verification.

Patient registers in medical web application, and patient's two Merkle hashes are created with 0000 and FFFF leaves. When he logs in, it's both trees are loaded from databases. Patient inserts a new reading in medical web application. Reading is stored in records collection of medical application using insert query. Query verification server takes reading, timestamp and user_id (patient id). Hash of reading is calculated and added in the Merkle hash tree as new leaf. New Merkle root hash tree is calculated. Root hash is pushed to blockchain.

If patient wants to update or delete reading at any timestamp, then delete/update is perform on Records collection. Due to delete or update, Merkle hash tree (1st for simple queries) is also updated. It is traversed and leaf is identified, and leaf is updated with new hash of updated reading or leaf is deleted when reading is delete. Merkle hash tree calculates its new root hash, which is updated on block chain.

Merkle hash tree is serialized and stored in Users_MHT as patient_MHT against user_id and timestamp (at which it is updated). As discussed earlier in section 5.3 Users_MHT is used to provide proof helpers (explained in Chapter 2 and pseudo code in Chapter 4 section 4.5.4) for simple queries. At same time, Users_MHTAggr is used to provide proof helpers for aggregated queries. To provide proof helpers for aggregated queries, we have used a different Merkle hash tree. When a new reading is entered, Users_MHTAggr is also updated with Agr_Array, Agr_Node_Hash, timestamp and user_id. Agr_Array contains sum, average, count, maximum, and minimum of reading at that timestamp. Query verification server gets previously inserted Agr_Array at last timestamp, it calculates new Agr_array using new reading as:

```
Agr_Array.Sum = Agr_Array.Sum (pervious) + reading;
Agr_Array.Count = Agr_Array.Count (pervious) + 1;
Agr_Array.Average = Agr_Array.Sum/Agr_Array.Count;
if (reading > Agr_Array (pervious).Maximum) {Agr_Array.Maximum=reading;}
if (reading < Agr_Array (pervious).Minimum) {Agr_Array.Minimum=reading;}
```

This newly calculated array is hashed with SHA-256 and result hash is assigned to Agr_Node_Hash. This hash is also added to already existing 2nd Merkle hash tree (aggregated tree) of patient and it is stored on Users_MHTAggr. New Merkle root hash is calculated and stored on blockchain. This root hash is used in verification of aggregated queries.

In case of updated reading, 2nd Merkle hash tree (aggregated tree) is updated with new Agr_Array at that timestamp, and all next Agr_Array against timestamps also updated. In case of delete, Agr_Array at that timestamp is deleted, and all next Agr_Array against timestamps are calculated again. So tree is updated, and new root hash is calculated. Which is updated on blockchain.

For verification part of our **solution-1**, Physician requests some readings of patient from medical web application. Query verification client checks type of the query, is it a simple? Or aggregated? If it's simple query, then query verification client gets Merkle root hash for that patient from the block chain. Query verification server returns proof helpers from Merkle hash tree of patient loaded from Users_MHT. If query is aggregated, then query verification client gets aggregated Merkle root hash of patient from blockchain. Query verification gets proof helpers from Merkle hash tree loaded from Users_MHTAggr.

Query verification client uses the proof helpers, and data to calculate root hash of tree. This newly calculated root hash is compared with root hash returned from blockchain. If both are matched, verification status is shown to the physician.

5.4.2. Proposed Solution-2 Implementation:

We have used only one Merkle hash tree in **proposed solution-2**. Users_MHT2 is the collection which is used for storage of leaves of Merkle hash tree for each patient. In this solution, we have implemented Merkle hash tree in such a way that it can provide proof helpers for simple as well as aggregated queries. We have used technique for completeness of data which is similar to our proposed solution-1. Merkle hash tree has two leaves with 0000 and FFFF. Tree is created when a patient registers on medical web application. When a patient logs in, it is loaded from database collection Users_MHT2. Data_Array is calculated after new reading, it depends upon the pervious reading and new reading. It is very similar to Agr_Array of solution-1, but Data_Array contains reading of patient too. Query verification server calculates Data_Array as:

```
Data_Array.reading = reading;
Data_Array.Sum = Data_Array.Sum (pervious) + reading;
Data_Array.Count = Data_Array.Count (pervious) + 1;
Data_Array.Average = Data_Array.Sum/ Data_Array.Count;
if(reading > Data_Array (pervious).Maximum) { Data_Array.Maximum=reading;}
if(reading < Data_Array (pervious).Minimum) { Data_Array.Minimum=reading;}
```

MHT_Node_Hash is hash of Data_Array, and it is added to leaf of Merkle hash tree. New Merkle root hash is calculated and it is pushed to blockchain.

In case of updated reading, Merkle hash is updated with new Data_Array at that timestamp, and all next Data_Array against timestamps also updated.

In case of delete, Data_Array at that timestamp is deleted, and all next Data_Array against timestamps are calculated again. So tree is updated, and new root hash is calculated. These updated leaves of Merkle hash tree are also updated in Users_MHT2. New root hash is updated on blockchain. At the time of verification in solution-2, query verification client gets root hash for that patient from blockchain. Query verification server sends proof helpers (from loaded Merkle hash tree of patient if not loaded load from Users_MHT2) to query verification client.

5.4.3. Implementation of Proof Verification for Queries:

When physician logs in, he is verified from databases. He sends a query for a patient, query verification client send request to blockchain for root hash of that patient. Query is forwarded to main server, where query is executed on databases and data is taken against that query. Query verification server checks the Merkle hash tree of that patient and get proof helpers (explained in chapter 2), if Merkle hash tree is not loaded, query verification server loads it from Users_MHT (simple query) and Users_MHTAggr (if aggregated query) for **solution-1**. For **solution-2** Merkle hash tree is loaded from Users_MHT2 by query verification server.

Proof helpers are sent by query verification server. Proof helpers are hashes from different nodes of Merkle hash tree (explained in chapter 2, Section 2.2). These proof helpers, and data is returned to query verification client. Query verification client calculates new root hash with help of poof helpers and data. New root hash is compared with root hash of patient from blockchain. If both are matched, then a message is displayed to physician that requested record is verified. If it is not matched, then physician is notified that data is corrupted.

Physician enter “from” and “to” values of timestamp for aggregated queries on range. Query verification server checks the ranges of timestamp entered by physician. For example, he wants to aggregate patient data from start to particular time “x”.

5.4.3.1. Proof Verification in Solution-1:

In our **proposed solution-1**, query verification server get *Agr_Array, Agr_Node_Hash* from Users_MHTAggr collection where timestamp is matched with “x”. Agr_Array contains all aggregated results from start to timestamp “x”. Proof helpers against Agr_Array is calculated from Merkle hash tree loaded from Users_MHTAggr. Query verification server sends proof helpers, *Agr_Array*, *Agr_Node_Hash* and query result from database to query verification client.

Query verification client matches root hash (from block chain) with root hash constructed from proof helpers and data. Aggregated query result from database is also compared with Agr_Array. If both results (query result and Agr_Array) are matched and root hashes (root hash from proof helpers and root hash from blockchain) are matched, then aggregated query result are verified.

If physician wants to find the reading from timestamp “x” to timestamp “y”. Database returns the result of aggregate at that time range after executing the query. In solution-1, query verification server splits this query into two queries for proof helpers. Instead of “x”, “x-1” is used because at “x” we have sum from start to value “x” and at “y” we have sum from start to “y”, now sum between “x” and “y” is calculated by subtracting sum at “x-1” from sum at “y”. It gets Agr_Array, Agr_Node_Hash at timestamp at “x-1” and gets proof helpers from Merkle hash tree loaded from Users_MHTAggr in solution-1 and similar for timestamp “y”.

Now query verification client gets proof helpers at timestamp “x-1” and timestamp “y”. Agr_Array$_{(x)}$, Agr_Node_Hash$_{(x-1)}$ and Proof helpers$_{(x-1)}$ and Agr_Array$_{(y)}$, Agr_Node_Hash$_{(y)}$ and Proof helpers$_{(y)}$ is sent to query verification client. Query verification client verify Merkle root hash from blockchain with Merkle root hash from proof helpers. If root hash from block chain and new root hash from proof helpers are verified for “x-1” and “y”, then integrity of data is preserved. But we also check the aggregate query result to match with result calculated from Agr_Array of “x-1” and “y”.

$$Sum\ from\ x\ to\ y\ =\ Agr\_Array.Sum\ (y)\ -\ Agr\_Array.Sum\ (x-1);$$

Equation 5-1 Sum on different ranges in solution-1

We have described the calculation of sum using “y” and “x-1” in equation 5-1. So, Merkle root hash and this sum both can guarantee the integrity for aggregate query results for solution-1.

5.4.3.2. Proof Verification in Solution-2:

In our **proposed solution-2**, query verification server gets *Data_Array*, MHT_Node_Hash from Users_MHT2 collection with timestamp is matched with “x”. *Data_Array* contains all aggregated results from start to timestamp “x”. Proof helpers against required *Data_Array* are calculated from Merkle hash tree which is loaded from Users_MHT2. Proof helpers, *Data_Array*, MHT_Node_Hash and query result from database are sent to query verification client. Query verification client uses root hash from block chain and new root hash which is constructed from proof helpers and query result. Aggregated result of database is also compared with *Data_Array*. If results (query result and *Data_Array*) are matched and root hashes (root hash from proof helpers and root hash from blockchain) are matched, then aggregated query result are verified and physician is notified.

If physician wants to find the reading from timestamp “x” to timestamp “y”. Database returns the result of aggregate at that time range after executing the query. We use the same technique mentioned in solution-1 to implement for solution-2. Proof for “x-1” and “y” is got from Merkle tree, which is loaded from Users_MHT2. It is sent to query verification client. At query verification client, Merkle root hash (from blockchain) is matched with new Merkle root hash (calculated from proof helpers). If root hashes for “x-1” and “y” are verified, then integrity of data is preserved. We are ensuring it more with comparing result from database with result calculated from Data_Array of “x-1” and “y”.

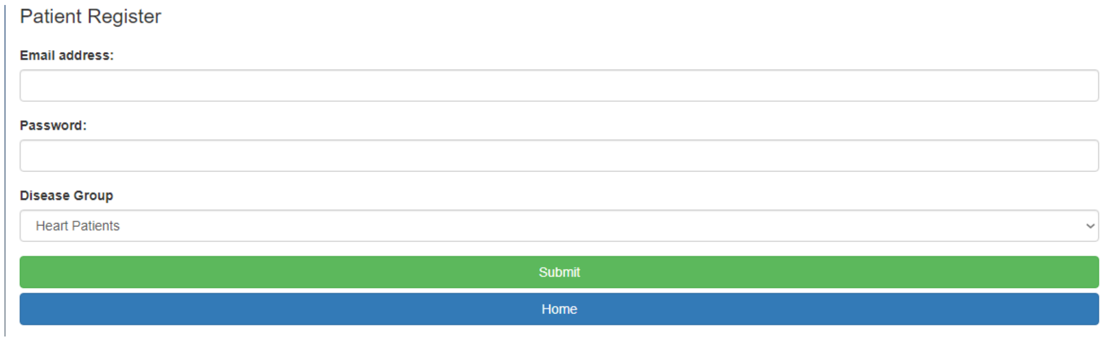
$$Sum\ from\ x\ to\ y\ =\ Data\_Array.Sum\ (y)\ -\ Data\_Array.Sum\ (x-1);$$

Equation 5-2 Sum on different ranges in solution-2

If sum from query result and sum from Data_Array are same, and also Merkle root hash from proof helpers and root from blockchain are same, then query verification client shows the message to physician that query is verified. If hash root is verified for “x-1” but not for “y”, then physician is notified that data is corrupted.

5.4.4. Implemented Medical Web Application

Medical web application was developed and query verification solutions were integrated with it. Patient and physician create their account as shown in figure 5-2 and 5-4.


Patient Register

Email address:

Password:

Disease Group

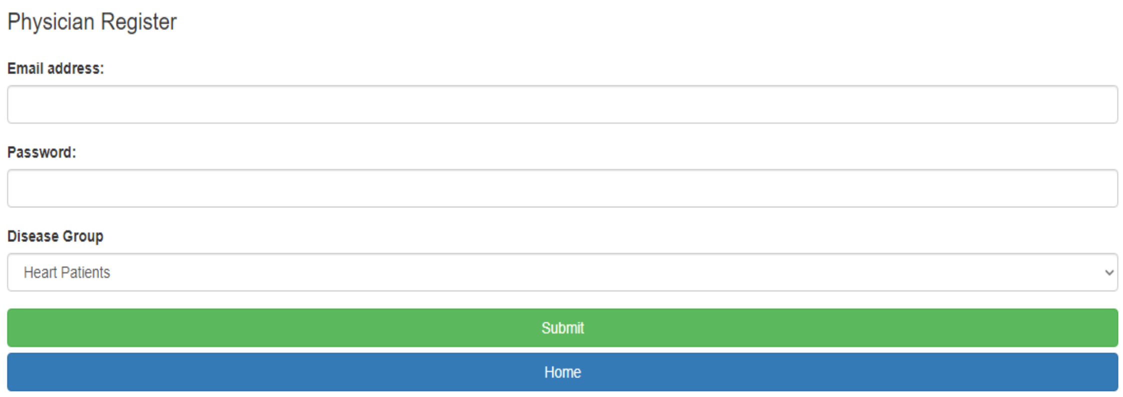
Heart Patients

Figure 5-2 Patient Register

Patient needs email address (user_id), password and disease group for registering in medical web application. Pseudo code for patient is given in figure 5-3. Patient does not have type of visibility to query verification solution. Query verification client and server work in background.

```
1- If patient is new // if patient is new
      Then
            Enter username // email address
            Enter password
            Select Disease group // blood pressure, heart, diabetes
            Click submit to register
2- Enter username and password // information required for patient login
3- Click login
4- If user query ==Insert  // add new reading of patient
      Then
            Enter new reading
            Enter timestamp
            Click submit
5- If user query == delete // delete reading at given timestamp
      Then
            Enter timestamp
            Click delete
6- If user query == update  // update reading at given timestamp
      Then
            Enter new reading
            Enter timestamp
            Click update
End
```

Figure 5-3 Pseudo code for patient


Physician Register

Email address:

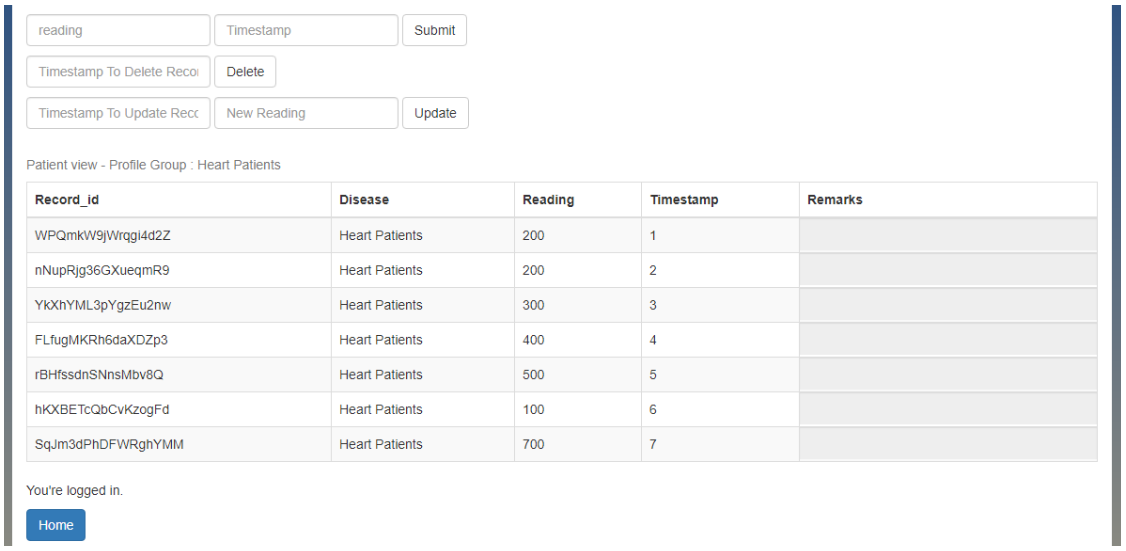
Password:

Disease Group

Heart Patients

Figure 5-4 Physician Register

When patient logs in, he can add, delete and update the reading against the timestamp as shown in figure 5-5. When add, delete or update is performed by patient, Merkle hash tree gets update and root hash is calculated which is sent to blockchain.



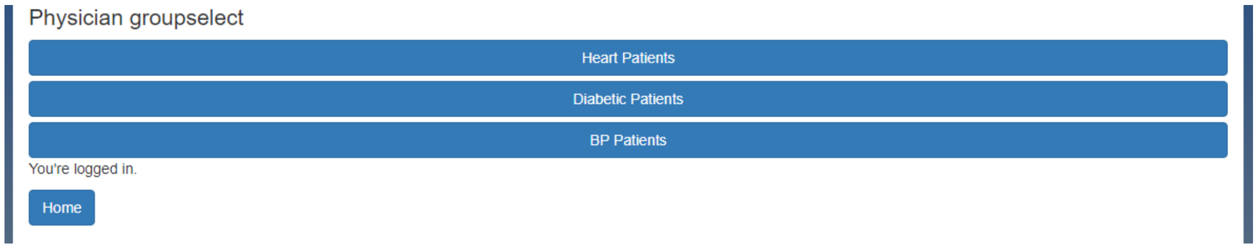
Patient view - Profile Group : Heart Patients

Record_id	Disease	Reading	Timestamp	Remarks
WPQmkW9jWrqgi4d2Z	Heart Patients	200	1	
nNupRjg36GXueqmR9	Heart Patients	200	2	
YkXhYML3pYgzEu2nw	Heart Patients	300	3	
FLfugMKRh6daXDZp3	Heart Patients	400	4	
rBHfssdnSNnsMbv8Q	Heart Patients	500	5	
hKXBETcQbCvKzogFd	Heart Patients	100	6	
SqJm3dPhDFWRghYMM	Heart Patients	700	7	

You're logged in.

Figure 5-5 Patient View

When physician gets data of patient, query verification client performs verification process with help of query result, and proof helpers (sent by query verification server). New root hash is calculated and matched with Merkle root of patient from blockchain.


Physician groupselect

You're logged in.

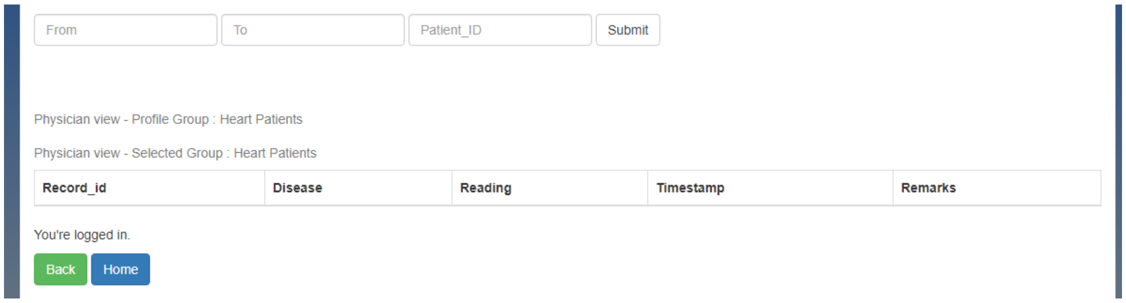
Figure 5-6 Physician View

```
1- If Physician is new // if physician is new
        Then
                Enter username // email address
                Enter password
                Select Disease group // Physician specialty blood pressure, heart, diabetes
                Click submit to register
2- Enter username and password // information required for Physician login
3- Click login
4- Select patient's disease group // according to physician specialty
5- Enter Patient_ID and Click submit, physician get
            a. Get all record of patient
            b. Query Verification status = true or false
6- Enter Patient_ID, "from" and "to" and Click submit, physician get
        If "from" == "to" // timestamp "from" and "to" fields are same
           Then
            a. Get record of patient at given timestamp
            b. Aggregated queries result from start to given timestamp
            c. Query Verification status = true or false
        Else // timestamp "from" and "to" fields are not same
         Then
            a. Get record of patient "from" timestamp till "to" timestamp
            b. Aggregated queries result "from" timestamp till "to" timestamp
            c. Query Verification status = true or false
7- Physician can enter remarks for patient
End
```

Figure 5-7 Pseudo code for physician

Physician can select patients according to his specialty, he can search records of patient at particular timestamp, range or can get all data. If he enters only patient id with empty "from" and "to" timestamp field then, he gets all records of selected patient. Integrity of all record is also be ensured.

Physician needs email address (user_id), password and disease group for registering in medical web application. Pseudo code for physician is given in figure 5-7.



Physician view - Profile Group : Heart Patients

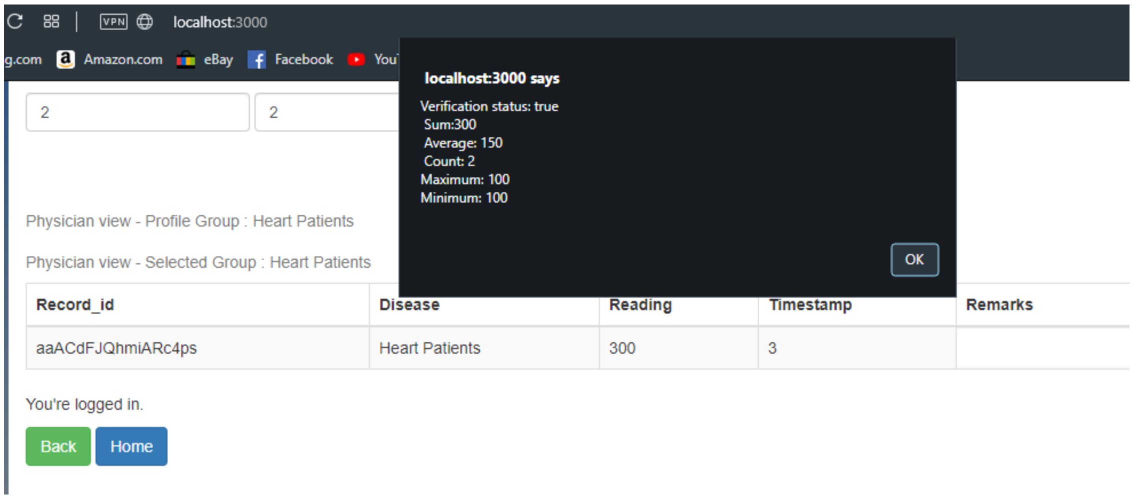
Physician view - Selected Group : Heart Patients

Record_id	Disease	Reading	Timestamp	Remarks

You're logged in.

Figure 5-8 Patient records view for Physician

If he enters same "from" and "to" timestamp field with patient id, then he gets value of the record at that particular timestamp as shown in figure 5-8. It also returns aggregation from start to this timestamp as shown in figure 5-9.



localhost:3000 says

```
Verification status: true
 Sum:300
 Average: 150
 Count: 2
Maximum: 100
Minimum: 100
```

Physician view - Profile Group : Heart Patients

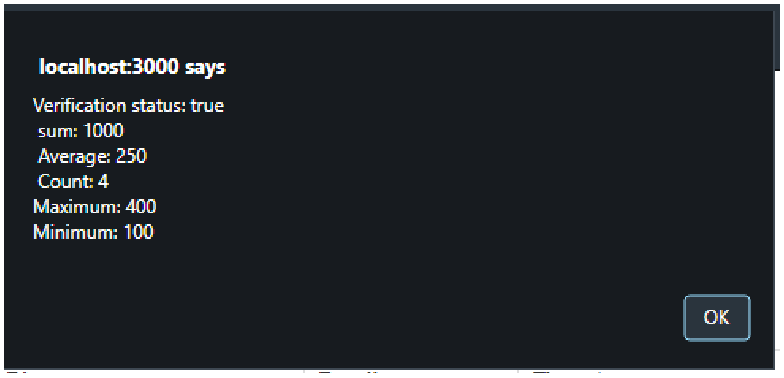
Physician view - Selected Group : Heart Patients

Record_id	Disease	Reading	Timestamp	Remarks
aaACdFJQhmiARc4ps	Heart Patients	300	3	

You're logged in.

Figure 5-9 Verification of patient's searched data

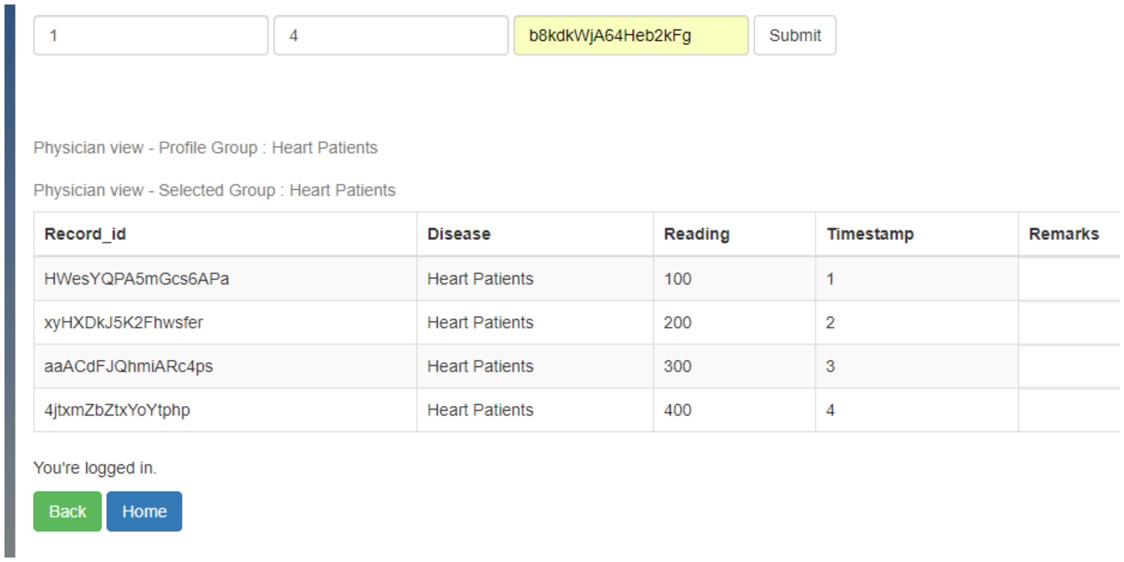
Verification status informs physician that data is verified or not. If verification status is false, it means that data stored is not preserving integrity. It is changed by some other malicious entity. If user enters "from" and “to” timestamp, then he gets result with verification status as shown in figure 5-10.



```
localhost:3000 says
Verification status: true
 sum: 1000
 Average: 250
 Count: 4
Maximum: 400
Minimum: 100
```

Figure 5-10 Verification Status with Aggregation results

After pressing OK, he gets given range data on web page as shown in figure 5-11.



1	4	b8kdkWjA64Heb2kFg	Submit

Physician view - Profile Group : Heart Patients

Physician view - Selected Group : Heart Patients

Record_id	Disease	Reading	Timestamp	Remarks
HWesYQPA5mGcs6APa	Heart Patients	100	1	
xyHXDkJ5K2Fhwsfer	Heart Patients	200	2	
aaACdFJQhmiARc4ps	Heart Patients	300	3	
4jtxmZbZtxYoYtphp	Heart Patients	400	4	

You're logged in.

Figure 5-11 Aggregate results after verification status

Verification solution 1 and 2 both have same front end medical web application, just backend implementation varies.

Chapter 6

6 Evaluation of Research Work

In this chapter, we have discussed the performance evaluation of implemented prototype of query verification solution. Performance comparison is also done with some pervious solution. Implemented prototype is tested and evaluated according to cost and performance. In performance analysis we have measured performance of test web application without and with query verification solutions. In cost analysis, we have discussed that how much addition cost is required to meet with query verification solution.

6.1. Performance Analysis

We have calculated end to end latency, throughput, storage overhead of Merkle hash tree, and proof size send to blockchain for solution-1 and solution-2 in performance analysis of prototype implementation of query verification solutions. We have also compared the results for the read and write operation with and without applying database query verification solution. Main server was deployed on Lenovo laptop with core i5 2.60GHz processor, 4.0 GB RAM and with hard drive of 1TB. Query verification server was also integrated with application on same machine. Web pages were accessed via Google chrome browser on another laptop of same specification. Blockchain node was installed on another Dell machine OptiPlex. These machines were communicating via LAN network.

6.1.1. Latency

Latency is time delay to access data in read or write operation. We have evaluated the query verification for multiple records for single patient. So, one Merkle hash tree was added, updated and deleted. We have performed some read operation i.e. find one reading of patient, find reading on range with "from" and "to" timestamps, and also aggregate queries like sum, count, average, max and min. We have calculated the performance of these operations over 1000 iteration of records for single patient. We have compared results for solution-1 and solution-2.

We have used PhantomJS [56] which is headless browser to send multiple request. Netsniff.js[57] is utility of PhantomJS which is used to test the load time of web pages and response time of all imports in the web pages. It exports the results in HAR format. HAR viewer [58] is used to visualize the results of network activity from Netsniff.js.

Table 6-1 Latency

Operation	**Without Query Verification (ms)**	**Latency Solution-1 (ms)**	**Latency Solution-2 (ms)**
Insert	20	30	27
Update	25	80	70
Delete	24	87	75
Find (single value)	20	38	40
Find (range)	32	49	51
Sum	35	50	52
Average	34	48	50
Count	30	45	48
Maximum	32	47	51
Min	33	46	50

In figure 6-1 we have shown the graph of these operations with number of records 1000 and results are also given table 6-1. Average latency access of

solution-2 is better than **solution-1** of all operations. But in solution-1 we have two separate Merkle hash trees, that's why performance of aggregation is better than solution-2. Node.js function *"performane.now()"* was also used for these results. We conclude the results of each query.

For ***insert*** operation, **solution-1** takes 10% more time than without query verification. **Solution-2** takes 7% more time than without query verification solution. **Solution-1** takes 3% more time than **solution-2**.

For ***update*** operation, **solution-1** takes 55% more time than without query verification. **Solution-2** takes 45% more time than without query verification solution. **Solution-1** takes 10% more time than **solution-2**.

For ***delete*** operation, **solution-1** takes 63% more time than without query verification. **Solution-2** takes 51% more time than without query verification solution. **Solution-1** takes 12% more time than **solution-2**.

For *find (single value)*, **solution-1** takes 18% more time than without query verification. **Solution-2** takes 20% more time than without query verification solution. **Solution-1** takes 2% less time than **solution-2**.

For *find (range)*, **solution-1** takes 17% more time than without query verification. **Solution-2** takes 19% more time than without query verification solution. **Solution-1** takes 2% less time than **solution-2**.

For *sum query,* **solution-1** takes 15% more time than without query verification. **Solution-2** takes 17% more time than without query verification solution. **Solution-1** takes 2% less time than **solution-2**.

For *average query,* **solution-1** takes 14% more time than without query verification. **Solution-2** takes 16% more time than without query verification solution. **Solution-1** takes 2% less time than **solution-2**.

For *count query,* **solution-1** takes 15% more time than without query verification. **Solution-2** takes 18% more time than without query verification solution. **Solution-1** takes 3% less time than **solution-2**.

For *max query,* **solution-1** takes 15% more time than without query verification. **Solution-2** takes 19% more time than without query verification solution. **Solution-1** takes 4% less time than **solution-2**.

For *min query,* **solution-1** takes 13% more time than without query verification. **Solution-2** takes 17% more time than without query verification solution. **Solution-1** takes 4% less time than **solution-2**.

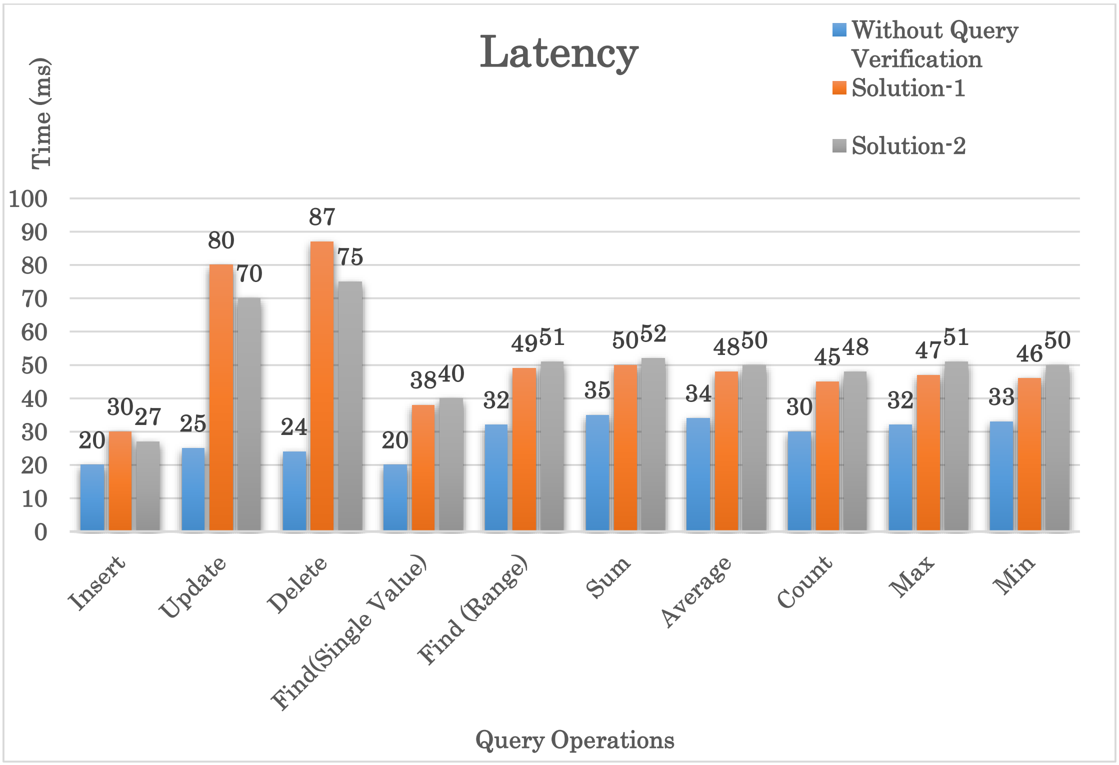


Figure 6-1 Latency Comparison Chart

Delete and update operation in both solutions take more times because *Agr_Node_Hash* (as explained in chapter 5) in both solution depends upon the reading, sum, max and min. If reading a reading is deleted at timestamp, then all next nodes are updated. Same happen in update operation, Agr_Node_Hash gets update form current update to next all records.

We have measured the result of load time of registration of patient, login view, add, delete, and update view of patient. We have measured physician data view for physician. Query verification solution creates Merkle hash tree for each patient at time of registration, so delay is added in registration process of patient. Similarly, when patient logs on, Merkle hash tree is loaded at that time from databases. Table 6-2 describes load time of our **solution-1** and **solution-2** for our medical web application.

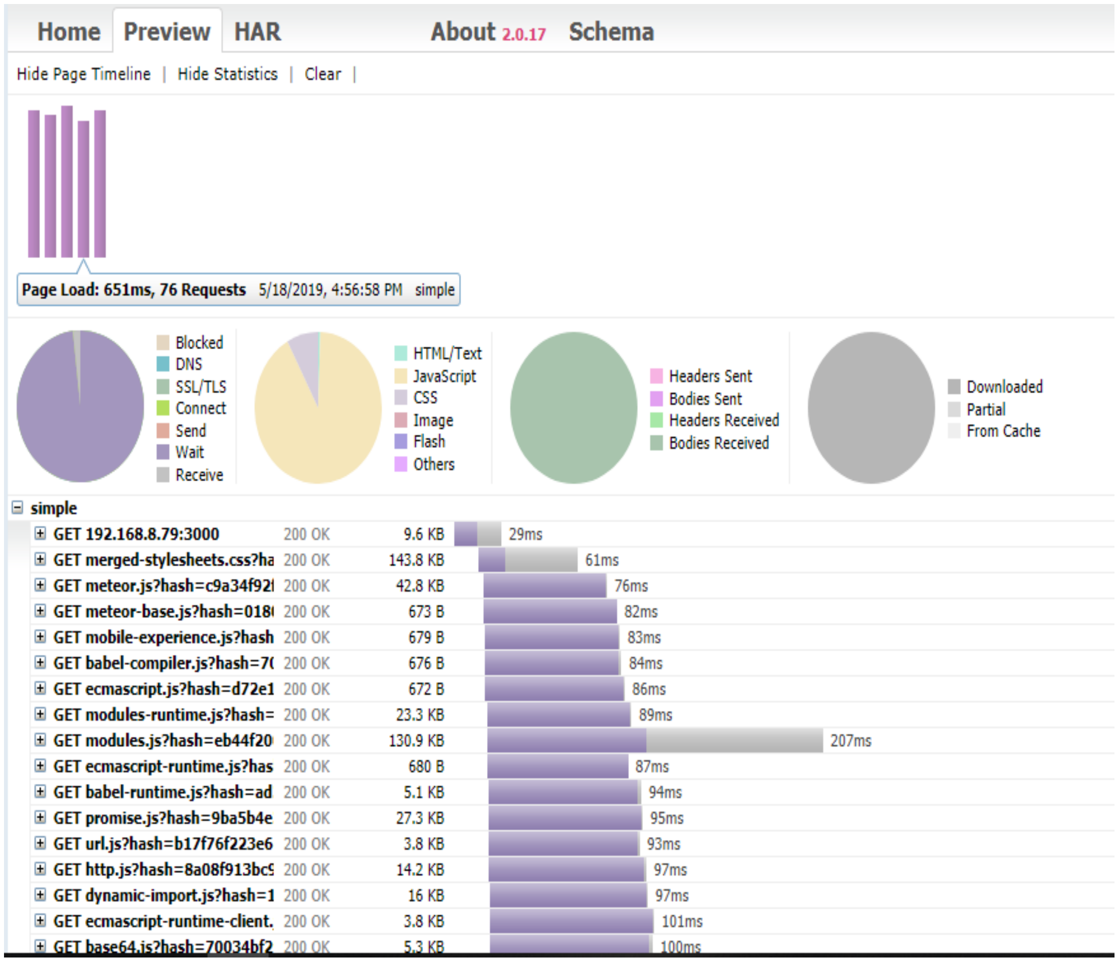
Table 6-2 Load Time

Application View	**Load Time (ms) without solution**	**Load Time (ms) with solution-1**	**Load Time (ms) with solution-2**
Patient Registration	10	17	12
Patient login	12	19	16
Patient record	20	34	29
Physician view	25	40	36

Patient Registration, **solution-1** takes 7% more time than without query verification. **Solution-2** takes 2% more time than without query verification solution. **Solution-1** takes 5% more time than **solution-2**. Patient login, **solution-1** takes 7% more time than without query verification. **Solution-2** takes 4% more time than without query verification solution. **Solution-1** takes 3% more time than **solution-2**.

Patient record, **solution-1** takes 14% more time than without query verification. **Solution-2** takes 9% more time than without query verification solution. **Solution-1** takes 5% more time than **solution-2**. Physician view, **solution-1** takes 15% more time than without query verification. **Solution-2** takes 9% more time than without query verification solution. **Solution-1** takes 4% more time than **solution-2**.

Figure 6-2 shows the HAR (HTTP Archive) [58] view of access time of web pages of medical web application with query integration **solution-2** This data was generated using PhantomJS. Figure 6-4 shows the HAR view of access time of web pages of medical web application with query integration **solution-1**. These results for 1000 records of readings for a patient and load time is complete time to load all data and all imported files like CSS and JS, and jQuery.



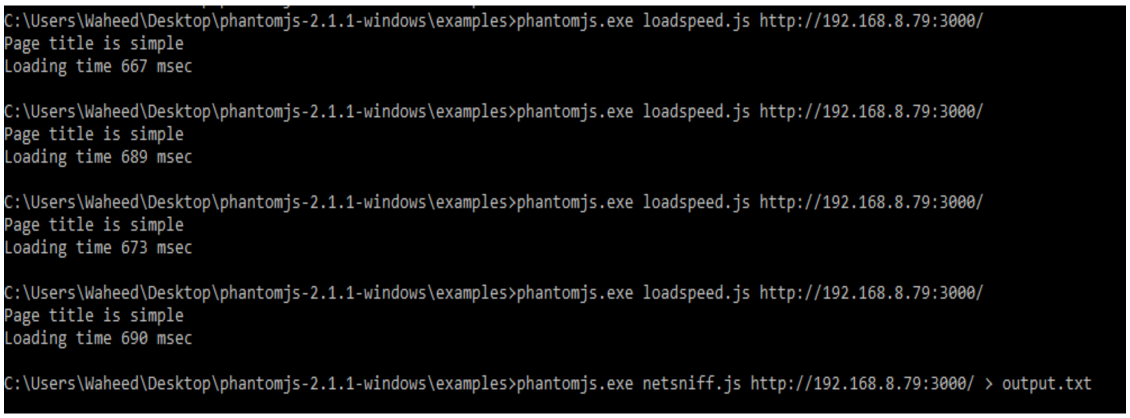
Page Load: 651ms, 76 Requests 5/18/2019, 4:56:58 PM simple

simple

GET 192.168.8.79:3000	200 OK	9.6 KB	29ms
GET merged-stylesheets.css?ha	200 OK	143.8 KB	61ms
GET meteor.js?hash=c9a34f92	200 OK	42.8 KB	76ms
GET meteor-base.js?hash=018	200 OK	673 B	82ms
GET mobile-experience.js?hash	200 OK	679 B	83ms
GET babel-compiler.js?hash=7	200 OK	676 B	84ms
GET ecmascript.js?hash=d72e1	200 OK	672 B	86ms
GET modules-runtime.js?hash=	200 OK	23.3 KB	89ms
GET modules.js?hash=eb44f20	200 OK	130.9 KB	207ms
GET ecmascript-runtime.js?has	200 OK	680 B	87ms
GET babel-runtime.js?hash=ad	200 OK	5.1 KB	94ms
GET promise.js?hash=9ba5b4e	200 OK	27.3 KB	95ms
GET url.js?hash=b17f76f223e6	200 OK	3.8 KB	93ms
GET http.js?hash=8a08f913bc	200 OK	14.2 KB	97ms
GET dynamic-import.js?hash=1	200 OK	16 KB	97ms
GET ecmascript-runtime-client.	200 OK	3.8 KB	101ms
GET base64.js?hash=70034bf2	200 OK	5.3 KB	100ms

Figure 6-2 HAR View of Solution-2

In figure 6-4, we can see the load time of web application which is hosted at local server and accessed with http://192.168.8.70:3000/, page size 9.6KB and response time is 29ms. Other GET requests are for loading other imports and required JavaScript in meteor. Figure 6-3 shows testing of load speed of medical web application. This test was run four time with PhantomJS using loadspeed.js script. Every time result was different from other. So average value was taken for multiple iterations. Netsniff.js was used to create logs which can be viewed from HAR viewer as shown in figure 6-2. Output of netsniff.js for web application was given as input to HAR viewer. There are many HAR viewer [58][59] which are used to show the graphical representation of network logs. Some of them are available as google chrome extensions [60].

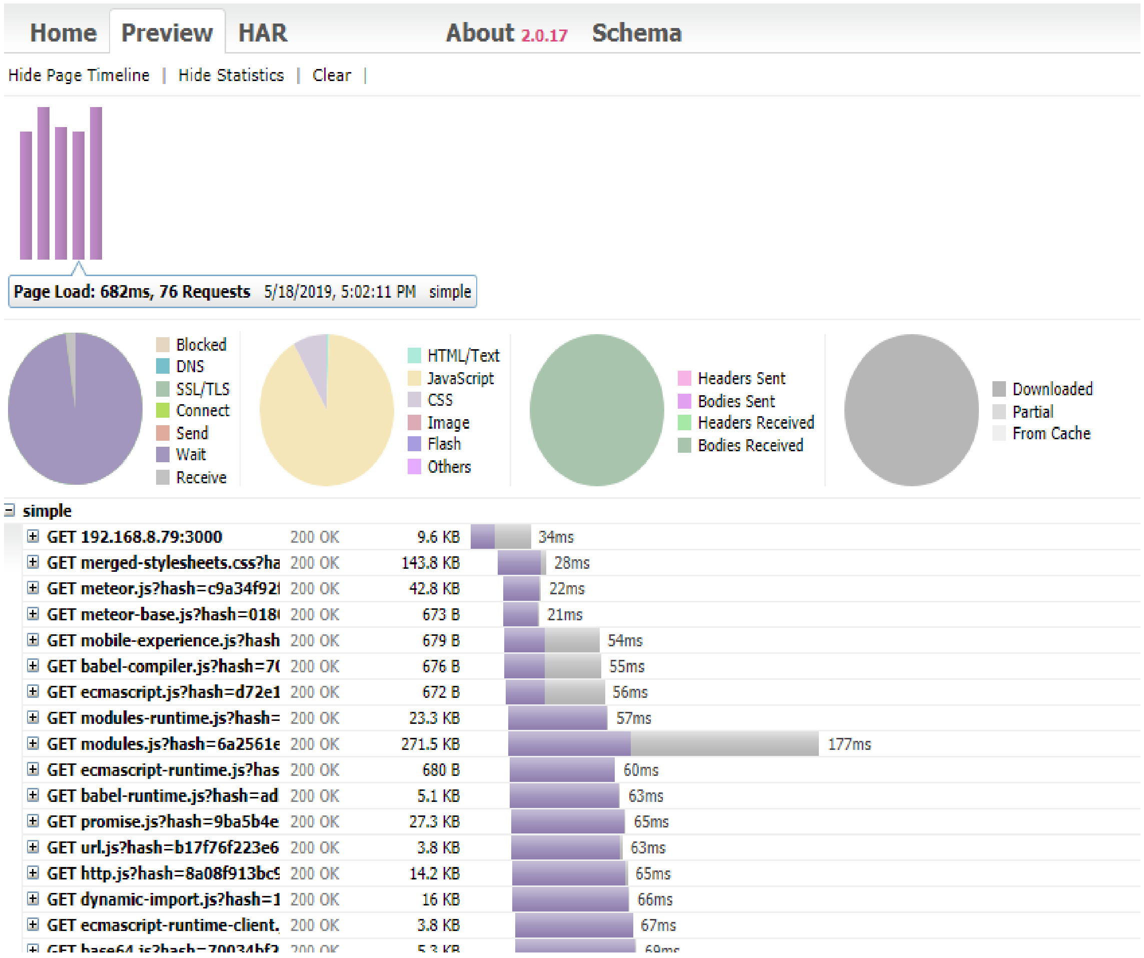


```
C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 667 msec

C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 689 msec

C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 673 msec

C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 690 msec

C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe netsniff.js http://192.168.8.79:3000/ > output.txt
```

Figure 6-3 PhantomJS Test for Solution-2

In figure 6-4, HAR view of logs from solution-1. Response time was 34ms for web application page. Overall page load time was 651ms for solution-2 and 682 ms for solution-1. Figure 6-5 shows load time testing of medical web application with query verification **solution-1** using PhantomJS. In table 6-2 results were measured by take difference between times from start of page till that end of page. These results show load time of solution-2 is less than solution-1.

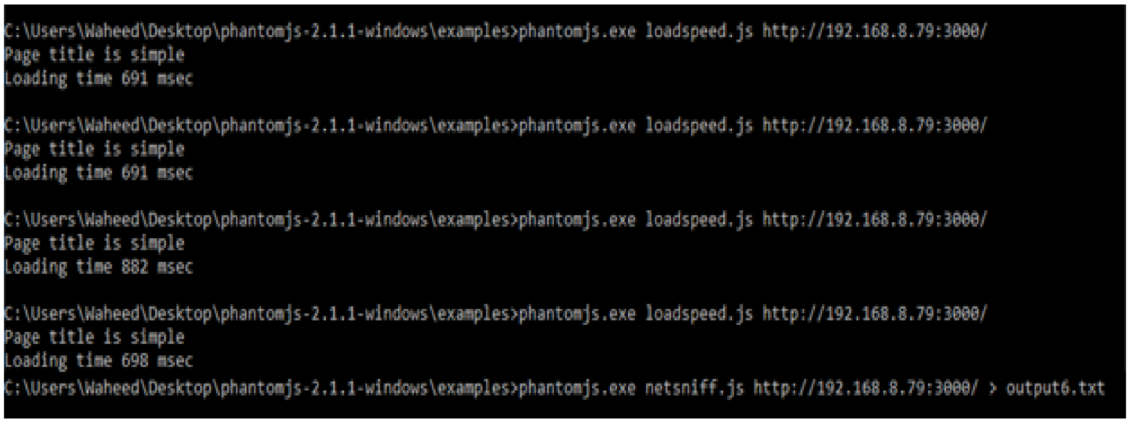
Solution-1 has two Merkle hash trees which takes more time than **solution-2**. In **solution-1** is faster than **solution-2** without aggregation queries, because **solution-1** without aggregation does not have pre-hash calculations like aggregate. For aggregation in solution-1, it has pre-hash calculation which make it expensive than solution-2.



Page Load: 682ms, 76 Requests 5/18/2019, 5:02:11 PM simple

```
simple
GET 192.168.8.79:3000                 200 OK     9.6 KB   34ms
GET merged-stylesheets.css?ha         200 OK   143.8 KB   28ms
GET meteor.js?hash=c9a34f92           200 OK    42.8 KB   22ms
GET meteor-base.js?hash=018           200 OK     673 B    21ms
GET mobile-experience.js?hash         200 OK     679 B    54ms
GET babel-compiler.js?hash=7(         200 OK     676 B    55ms
GET ecmascript.js?hash=d72e1          200 OK     672 B    56ms
GET modules-runtime.js?hash=          200 OK    23.3 KB   57ms
GET modules.js?hash=6a2561e           200 OK   271.5 KB   177ms
GET ecmascript-runtime.js?has         200 OK     680 B    60ms
GET babel-runtime.js?hash=ad          200 OK     5.1 KB   63ms
GET promise.js?hash=9ba5b4e           200 OK    27.3 KB   65ms
GET url.js?hash=b17f76f223e6          200 OK     3.8 KB   63ms
GET http.js?hash=8a08f913bc9          200 OK    14.2 KB   65ms
GET dynamic-import.js?hash=1          200 OK      16 KB   66ms
GET ecmascript-runtime-client.        200 OK     3.8 KB   67ms
GET base64.js?hash=70034bf2           200 OK     5.3 KB   69ms
```

Figure 6-4 HAR View of Solution-1



```
C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 691 msec

C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 691 msec

C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 882 msec

C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe loadspeed.js http://192.168.8.79:3000/
Page title is simple
Loading time 698 msec
C:\Users\Waheed\Desktop\phantomjs-2.1.1-windows\examples>phantomjs.exe netsniff.js http://192.168.8.79:3000/ > output6.txt
```

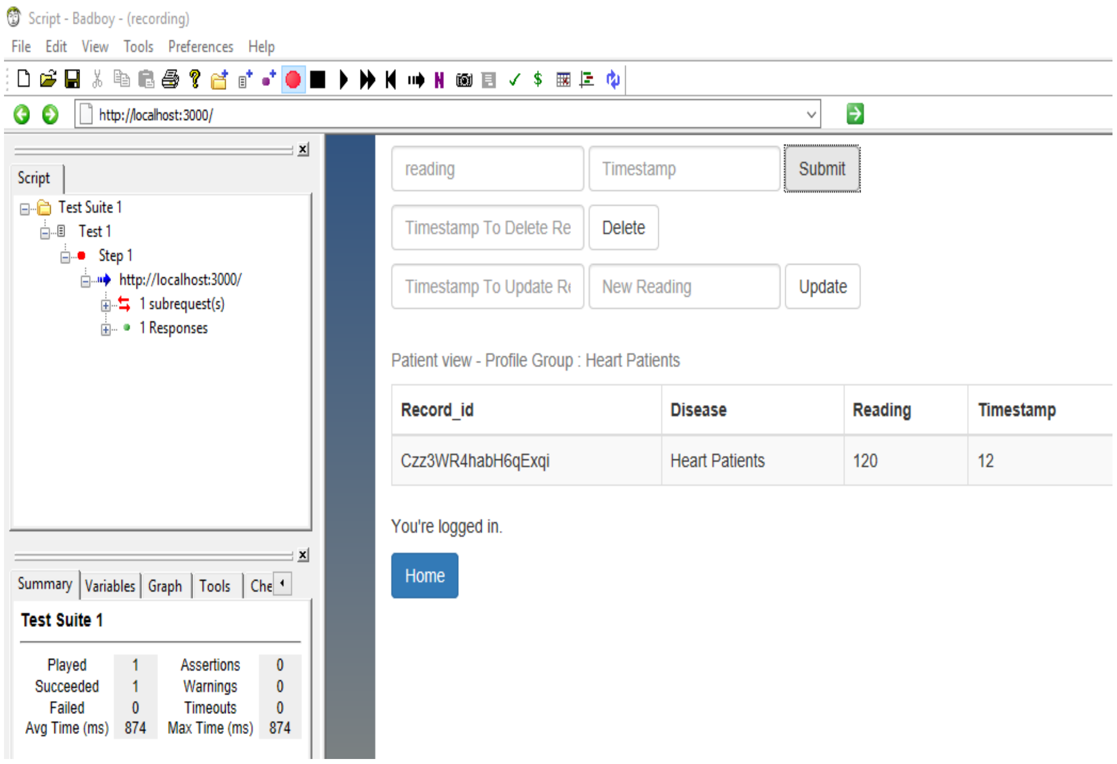
Figure 6-5 PhantomJS Test for Solution-1

6.1.2. Throughput

We have measured throughput of query verification solution for both implemented prototypes, and it was compared without query verification solution. Throughput is also dependent upon the specification of server machine where application is deployed. If server has very good processing specification then application has good throughput. We have already explained that our application is deployed on laptop with core i5 processor.

Throughput was measured by using Apache JMeter [61] and Gatling [62]. Gatling is tool use to record the flow of web applications and then simulate it and produce the results. Recording was done for write (insert, update, delete), read (find (single value), find range, sum, count, average, maximum and minimum) and read/write both for query verification **solution-1** and **solution-2**. These recordings were simulated by Gatling, it produced reports on all requests simulated by it and generated report in HTML.

Apache JMeter is open source which is used for performance testing of web applications. Throughput (requests/seconds) are measured from it. It has also different testing parameters too. We were required to configure all test scenario on JMeter and also guided it about the flow of web application. This process could be done manually and automatically. Badboy Software [63] is used to record the web application flow, which is used in Apache JMeter. Process of recording to web application flow is shown in figure 6-6. After recording the flow, there was option of export to JMeter, which was used to export the script of recording for JMeter. Apache JMeter run the flows that were recorded. It also has options to simulate recording against multiple users and can iterate each test case multiple time. Figure 6-7 shows JMeter performance testing. It has ability to iterate the simulation multiple times and results is as average of these iterations.


Script - Badboy - (recording)

File Edit View Tools Preferences Help

http://localhost:3000/

Script
- Test Suite 1
  - Test 1
    - Step 1
      - http://localhost:3000/
        - 1 subrequest(s)
        - 1 Responses

reading | Timestamp | Submit

Timestamp To Delete Re | Delete

Timestamp To Update Re | New Reading | Update

Patient view - Profile Group : Heart Patients

Record_id	Disease	Reading	Timestamp
Czz3WR4habH6qExqi	Heart Patients	120	12

You're logged in.

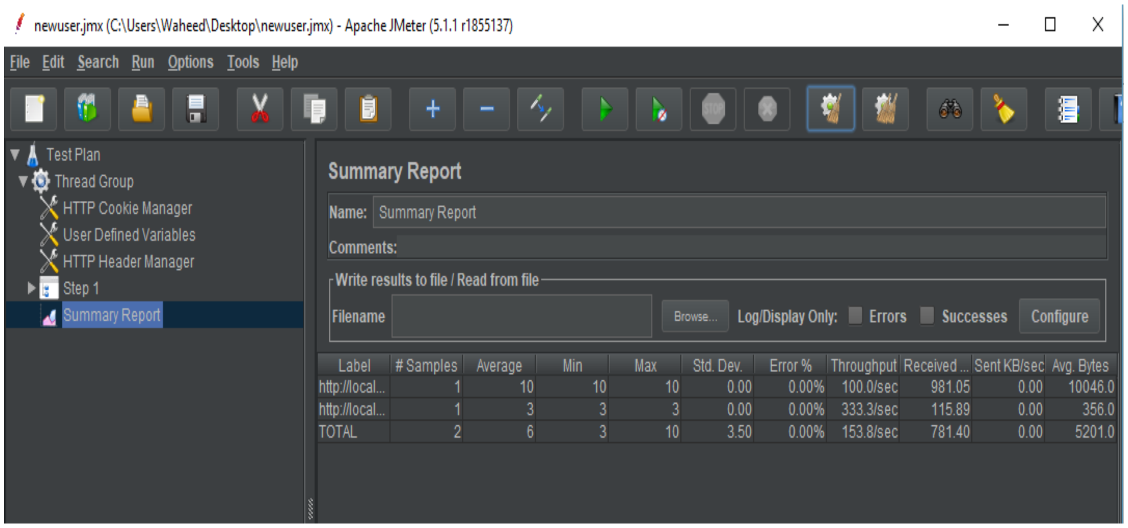
Home

Summary | Variables | Graph | Tools | Che

Test Suite 1

Played	1	Assertions	0
Succeeded	1	Warnings	0
Failed	0	Timeouts	0
Avg Time (ms)	874	Max Time (ms)	874

Figure 6-6 Badboy Recording for JMeter


newuser.jmx (C:\Users\Waheed\Desktop\newuser.jmx) - Apache JMeter (5.1.1 r1855137)

File Edit Search Run Options Tools Help

- Test Plan
  - Thread Group
    - HTTP Cookie Manager
    - User Defined Variables
    - HTTP Header Manager
    - Step 1
    - Summary Report

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: | Errors | Successes | Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
http://local...	1	10	10	10	0.00	0.00%	100.0/sec	981.05	0.00	10046.0
http://local...	1	3	3	3	0.00	0.00%	333.3/sec	115.89	0.00	356.0
TOTAL	2	6	3	10	3.50	0.00%	153.8/sec	781.40	0.00	5201.0

Figure 6-7 JMeter performance testing

Table 6-3 shows the throughput for both implemented solutions and without query verification solution. Throughput of solution-2 is more than solution-1. In *read operation*, **solution-2** has 18% more throughput than **solution-1**. In *write operation,* **solution-2** has 36% more throughput than **solution-1**.

Table 6-3 Throughput

Operation	Throughput Without sol (Request/sec)	Throughput Solution-1 (Request/sec)	Throughput Solution-2 (Request/sec)
Read	800	172	190
Write	1200	220	256
Read & Write	600	187	223

The graph of comparison without query verification and with solution-1 and solution-2 is shown in figure 6-8.

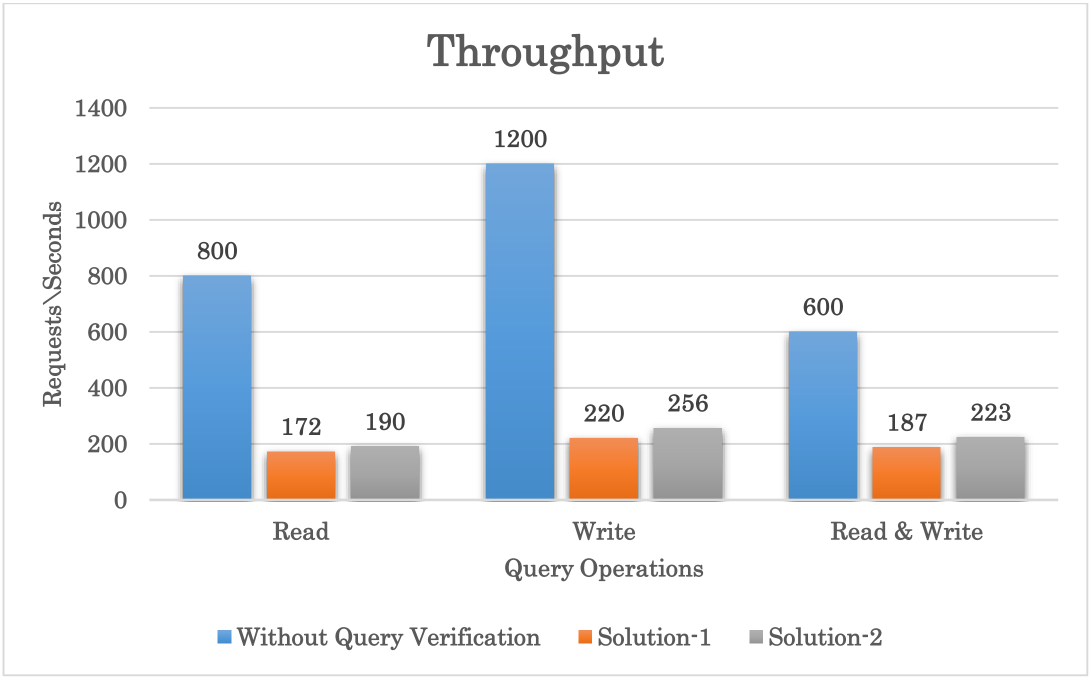
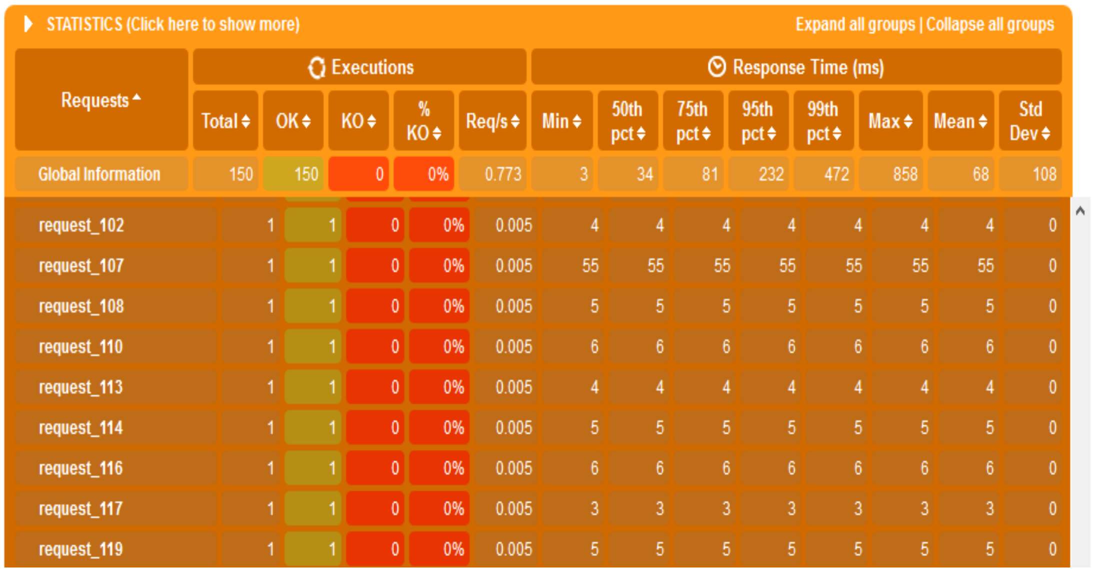


Figure 6-8 Throughput Comparison Chart

In *read/write operation,* **solution-2** has also 36% more throughput than **solution-1**. These results are generated by Apache JMeter. Solution-2 has more throughput than solution-1. Now, we conclude the results of medical web application without applying any solution with **solution-2**. Throughput is degraded around 4 times, when we apply query verification **solution-2**.

Results produced from *Gatling* are HTML reports. We had recorded read, write and read/write mix operation with Gatling recorder for both query verification solution-1 and solution-2. These recordings were simulated and results were produced. HTML reports were containing details of all requests simulated by it and response time. It had generated different types of graphs in HTML reports. It had categorized results on the base of request response time.



STATISTICS (Click here to show more) Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	150	150	0	0%	0.773	3	34	81	232	472	858	68	108
request_102	1	1	0	0%	0.005	4	4	4	4	4	4	4	0
request_107	1	1	0	0%	0.005	55	55	55	55	55	55	55	0
request_108	1	1	0	0%	0.005	5	5	5	5	5	5	5	0
request_110	1	1	0	0%	0.005	6	6	6	6	6	6	6	0
request_113	1	1	0	0%	0.005	4	4	4	4	4	4	4	0
request_114	1	1	0	0%	0.005	5	5	5	5	5	5	5	0
request_116	1	1	0	0%	0.005	6	6	6	6	6	6	6	0
request_117	1	1	0	0%	0.005	3	3	3	3	3	3	3	0
request_119	1	1	0	0%	0.005	5	5	5	5	5	5	5	0

Figure 6-9 Read/Write Operation Solution-2

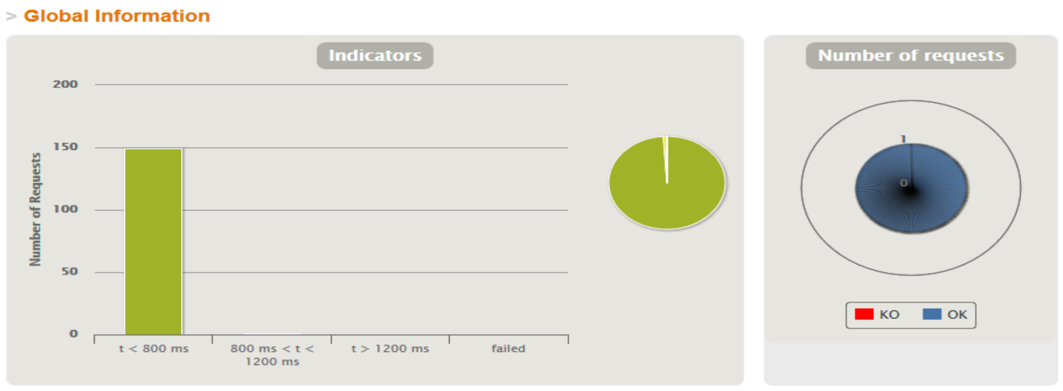


Figure 6-10 Read/Write Operation Sol-2

For **query verification solution-2**, results for read, write, and read/write mixed are shown in figure 6-9, 6-13 and 6-17. Results were showing the mean response time with standard deviation and indicators of request time. Results were showing that write took more time than read operation. We had created some read and write mix queries at specific value and range. KO requests were those which were not responded by web application server and they were marked as failed.

Read/Write operation of **solution-2** shows that all request's response time was less than 800ms and mean time was 68ms. 150 requests were handled at 0.773 request/seconds as shown in figure 6-9.

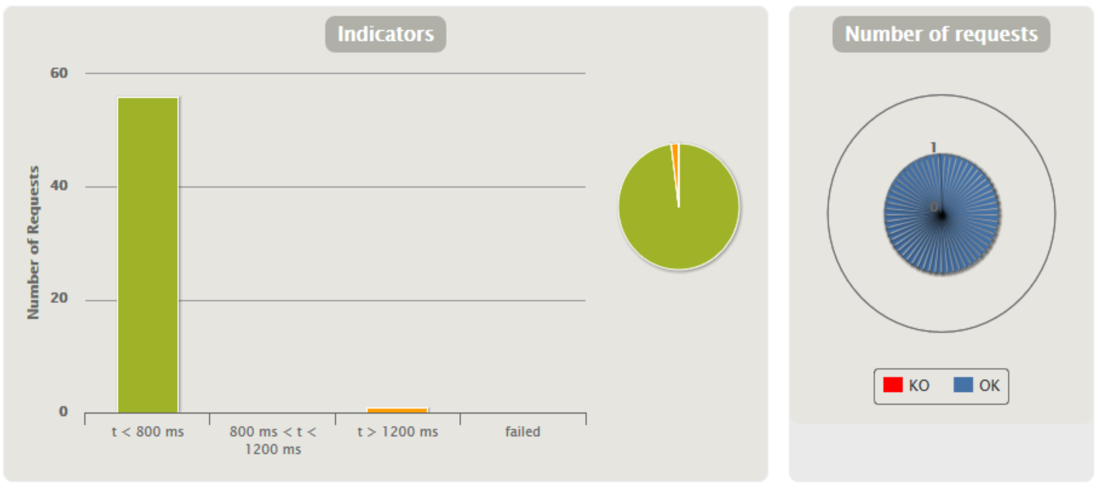
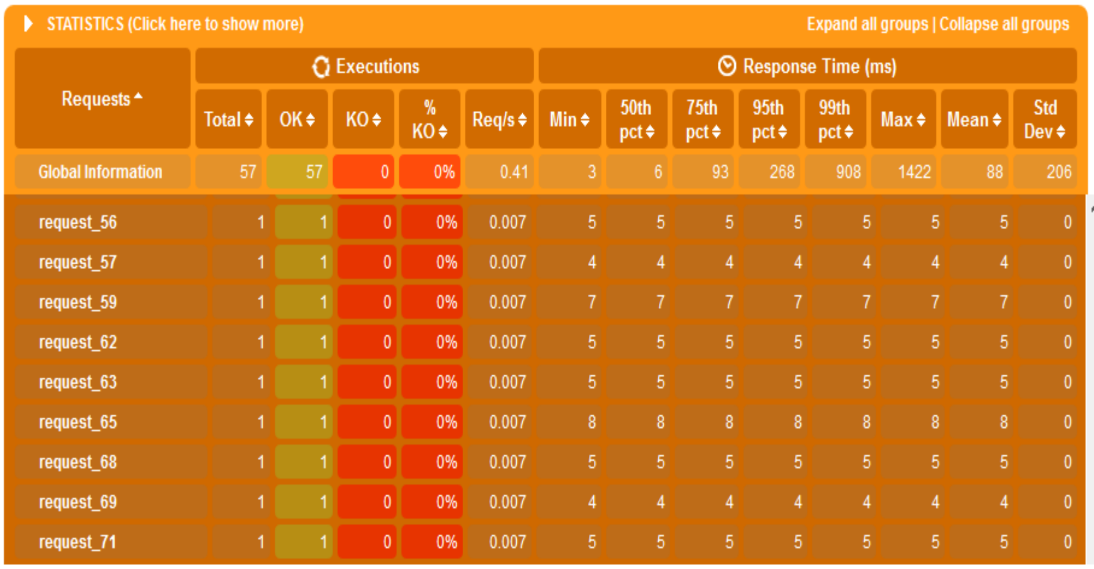


Figure 6-11 Read/Write Operation Solution-1



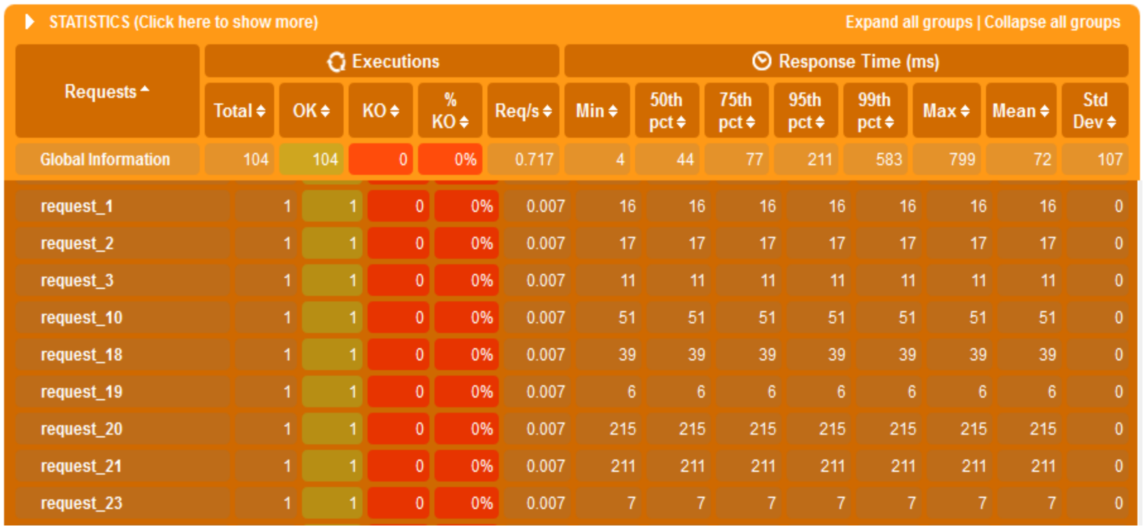
STATISTICS (Click here to show more) Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	57	57	0	0%	0.41	3	6	93	268	908	1422	88	206
request_56	1	1	0	0%	0.007	5	5	5	5	5	5	5	0
request_57	1	1	0	0%	0.007	4	4	4	4	4	4	4	0
request_59	1	1	0	0%	0.007	7	7	7	7	7	7	7	0
request_62	1	1	0	0%	0.007	5	5	5	5	5	5	5	0
request_63	1	1	0	0%	0.007	5	5	5	5	5	5	5	0
request_65	1	1	0	0%	0.007	8	8	8	8	8	8	8	0
request_68	1	1	0	0%	0.007	5	5	5	5	5	5	5	0
request_69	1	1	0	0%	0.007	4	4	4	4	4	4	4	0
request_71	1	1	0	0%	0.007	5	5	5	5	5	5	5	0

Figure 6-12 Read/Write Operation Solution-1

Figure 6-11 and 6-12 simulation results for read/write operations for query verification **solution-1**. Mean response time was 88ms and requests were handled at 0.41 requests/second. We conclude from both results that solution-2 handles more requests than solution-1. Throughput of **Solution-2** is 36% more than **solution-1** for read/write operation.

<u>Write operation</u> of **solution-2**, response time was less than 800ms and mean response time was 72ms, and 104 requests were handled at 0.717 requests/seconds as shown in figure 6-13 and 6-14.



STATISTICS (Click here to show more) Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	104	104	0	0%	0.717	4	44	77	211	583	799	72	107
request_1	1	1	0	0%	0.007	16	16	16	16	16	16	16	0
request_2	1	1	0	0%	0.007	17	17	17	17	17	17	17	0
request_3	1	1	0	0%	0.007	11	11	11	11	11	11	11	0
request_10	1	1	0	0%	0.007	51	51	51	51	51	51	51	0
request_18	1	1	0	0%	0.007	39	39	39	39	39	39	39	0
request_19	1	1	0	0%	0.007	6	6	6	6	6	6	6	0
request_20	1	1	0	0%	0.007	215	215	215	215	215	215	215	0
request_21	1	1	0	0%	0.007	211	211	211	211	211	211	211	0
request_23	1	1	0	0%	0.007	7	7	7	7	7	7	7	0

Figure 6-13 Write Operation Solution-2

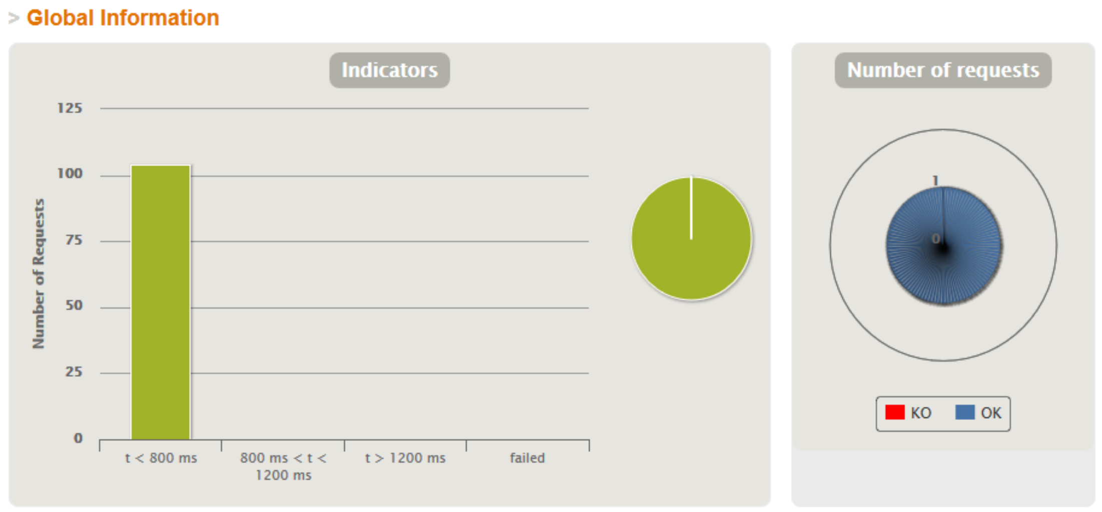


Figure 6-14 Write Operation Solution-2

Figure 6-15 and 6-16 show the results after simulation by Gatling for write operation with **solution-1** of query verification. Mean time for write operation was 137ms. Response time of all requests was under 800ms except one which was great than 1200ms. Request/seconds for this test was 0.33.

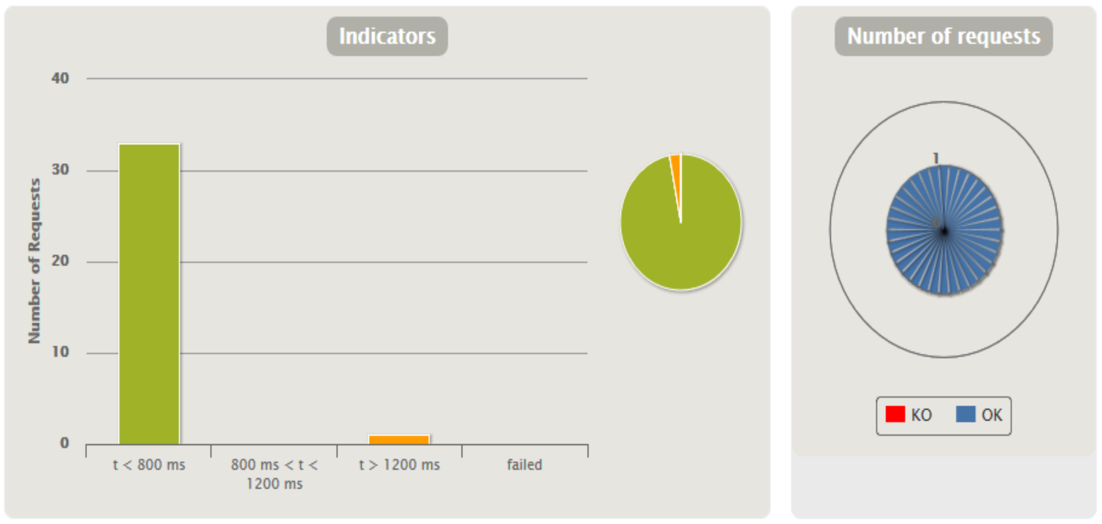
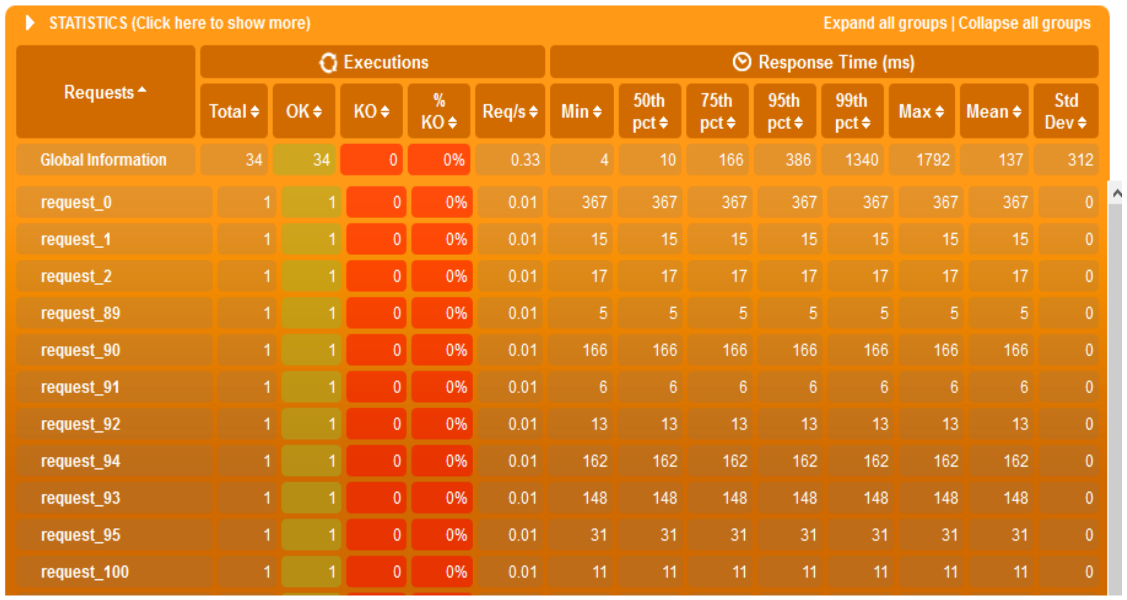


Figure 6-15 Write Operation Solution-1

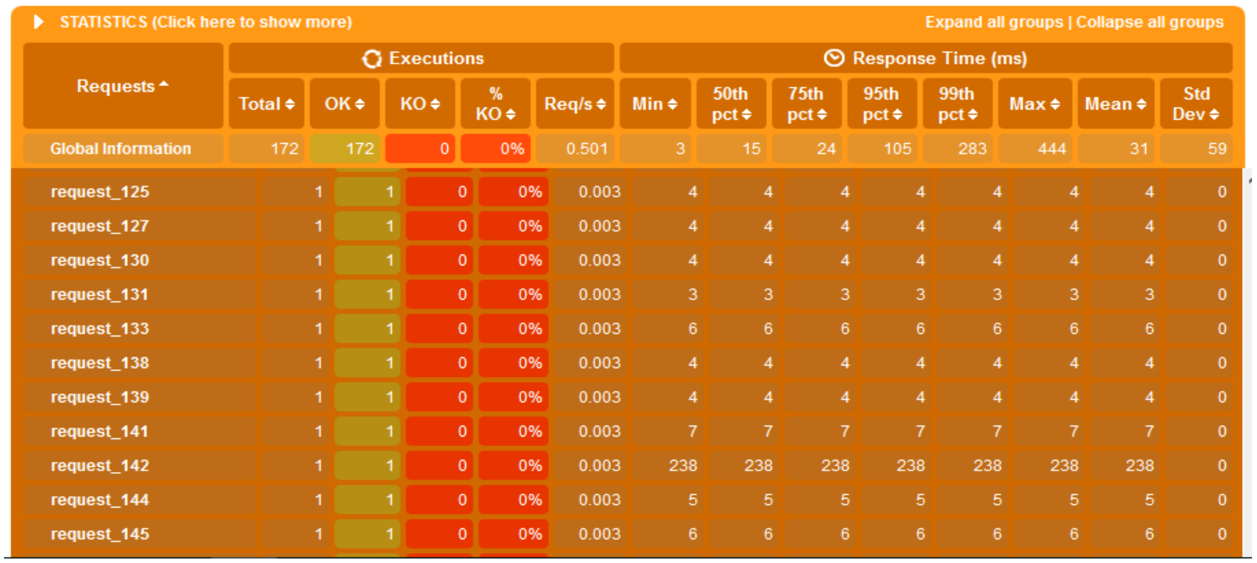

STATISTICS (Click here to show more)
Expand all groups | Collapse all groups

Requests	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	34	34	0	0%	0.33	4	10	166	386	1340	1792	137	312
request_0	1	1	0	0%	0.01	367	367	367	367	367	367	367	0
request_1	1	1	0	0%	0.01	15	15	15	15	15	15	15	0
request_2	1	1	0	0%	0.01	17	17	17	17	17	17	17	0
request_89	1	1	0	0%	0.01	5	5	5	5	5	5	5	0
request_90	1	1	0	0%	0.01	166	166	166	166	166	166	166	0
request_91	1	1	0	0%	0.01	6	6	6	6	6	6	6	0
request_92	1	1	0	0%	0.01	13	13	13	13	13	13	13	0
request_94	1	1	0	0%	0.01	162	162	162	162	162	162	162	0
request_93	1	1	0	0%	0.01	148	148	148	148	148	148	148	0
request_95	1	1	0	0%	0.01	31	31	31	31	31	31	31	0
request_100	1	1	0	0%	0.01	11	11	11	11	11	11	11	0

Figure 6-16 Write Operation Solution-1

Results of solution-1 also depict that write operations are more expensive as compare to read operation when data from query result is verified for every ready operation. We conclude from results that throughput of write operation for solution-2 is 38% more than solution-1.

Read operation for **solution-2**, response time was less than 800ms for all requests with mean response time was 31ms. For 172 requests, web application was handling 0.501 request/seconds as shown in figure 6-17 and 6-18.



STATISTICS (Click here to show more) Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	172	172	0	0%	0.501	3	15	24	105	283	444	31	59
request_125	1	1	0	0%	0.003	4	4	4	4	4	4	4	0
request_127	1	1	0	0%	0.003	4	4	4	4	4	4	4	0
request_130	1	1	0	0%	0.003	4	4	4	4	4	4	4	0
request_131	1	1	0	0%	0.003	3	3	3	3	3	3	3	0
request_133	1	1	0	0%	0.003	6	6	6	6	6	6	6	0
request_138	1	1	0	0%	0.003	4	4	4	4	4	4	4	0
request_139	1	1	0	0%	0.003	4	4	4	4	4	4	4	0
request_141	1	1	0	0%	0.003	7	7	7	7	7	7	7	0
request_142	1	1	0	0%	0.003	238	238	238	238	238	238	238	0
request_144	1	1	0	0%	0.003	5	5	5	5	5	5	5	0
request_145	1	1	0	0%	0.003	6	6	6	6	6	6	6	0

Figure 6-17 Read Operation Solution-2

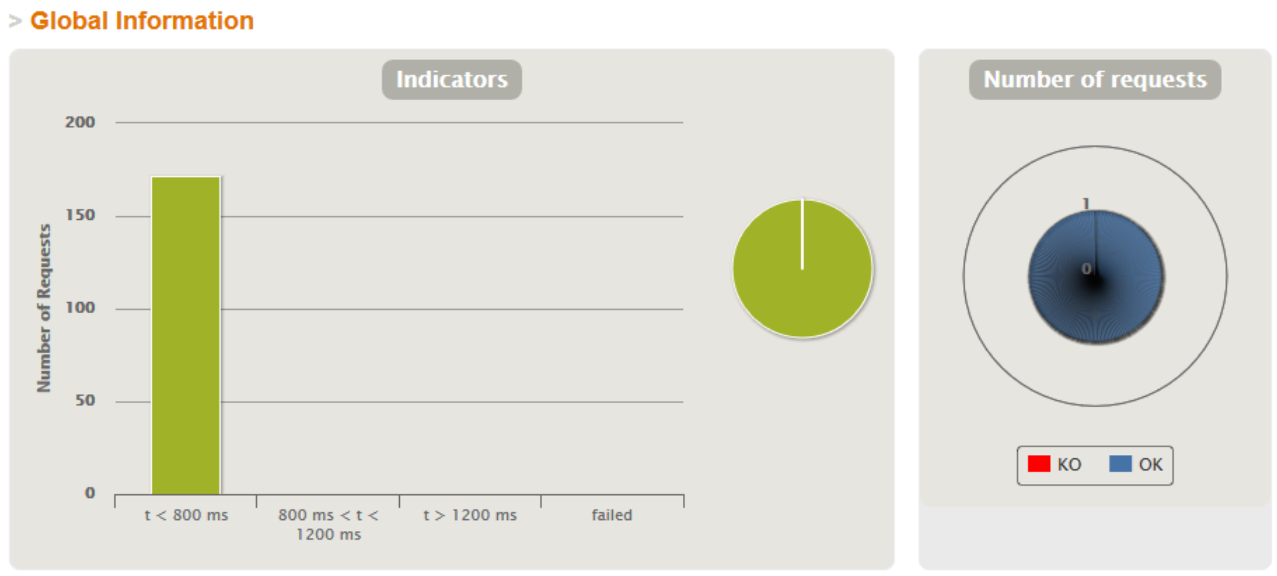


Figure 6-18 Read Operation Solution-2

Figure 6-19 and 6-20 show results of read operation on **solution-1**. Response time of all 138 read requests were less than 800ms mean response time was 31ms and requests were handled at 0.515 requests/seconds.

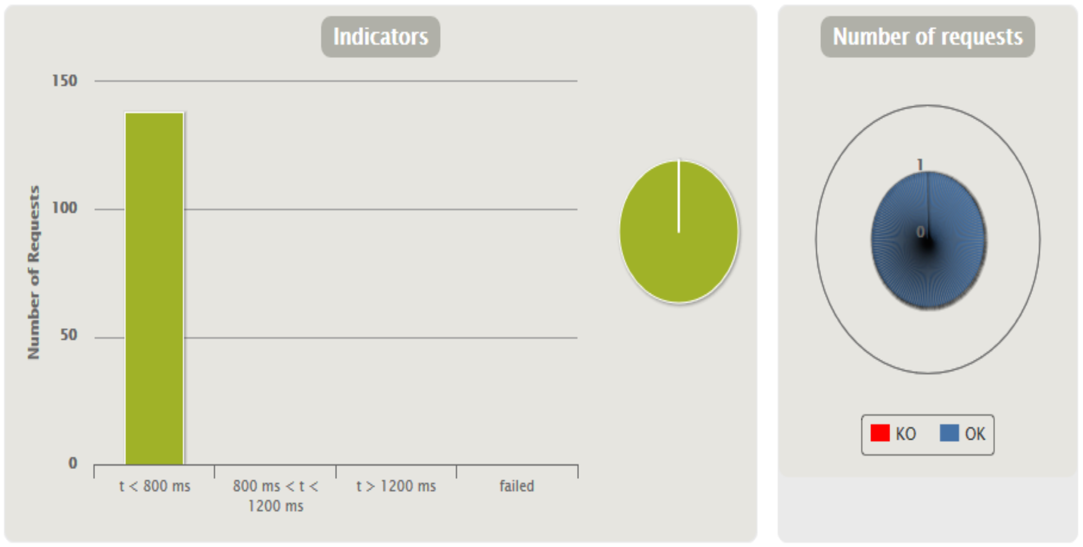
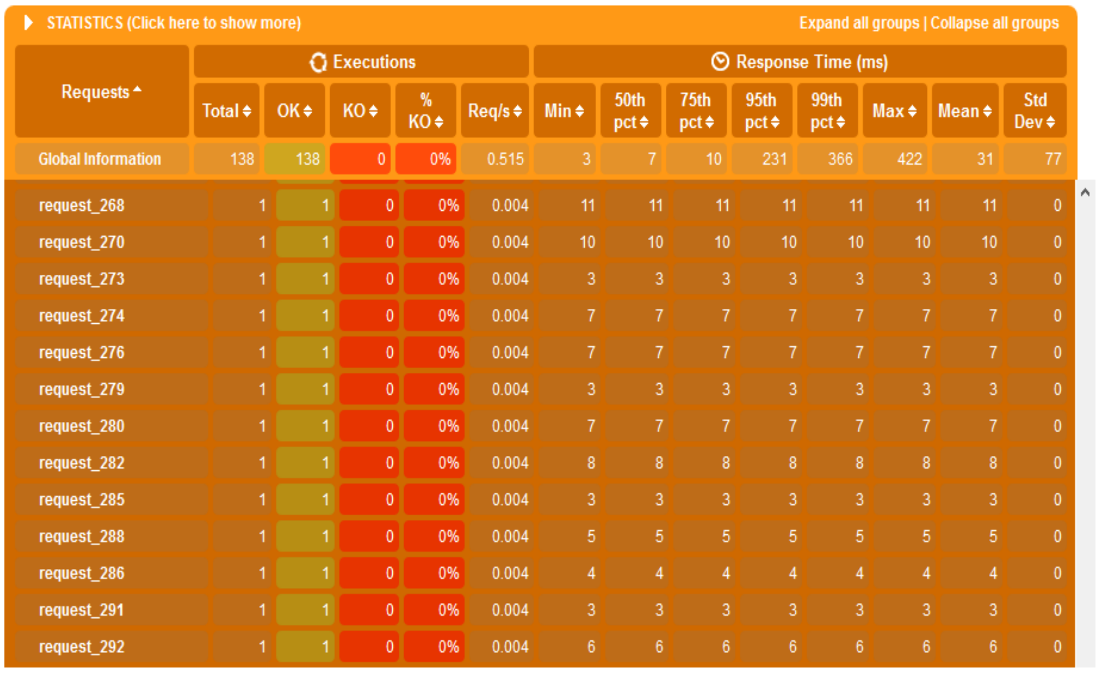


Figure 6-19 Read Operation Solution-1



STATISTICS (Click here to show more) Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	138	138	0	0%	0.515	3	7	10	231	366	422	31	77
request_268	1	1	0	0%	0.004	11	11	11	11	11	11	11	0
request_270	1	1	0	0%	0.004	10	10	10	10	10	10	10	0
request_273	1	1	0	0%	0.004	3	3	3	3	3	3	3	0
request_274	1	1	0	0%	0.004	7	7	7	7	7	7	7	0
request_276	1	1	0	0%	0.004	7	7	7	7	7	7	7	0
request_279	1	1	0	0%	0.004	3	3	3	3	3	3	3	0
request_280	1	1	0	0%	0.004	7	7	7	7	7	7	7	0
request_282	1	1	0	0%	0.004	8	8	8	8	8	8	8	0
request_285	1	1	0	0%	0.004	3	3	3	3	3	3	3	0
request_288	1	1	0	0%	0.004	5	5	5	5	5	5	5	0
request_286	1	1	0	0%	0.004	4	4	4	4	4	4	4	0
request_291	1	1	0	0%	0.004	3	3	3	3	3	3	3	0
request_292	1	1	0	0%	0.004	6	6	6	6	6	6	6	0

Figure 6-20 Read Operation Solution-1

Throughput of **solution-1** is slightly greater than **solution-2**. Requests/second of solution-1 is 1.4% more than solution-2. We conclude from these results that throughput for read operation in **solution-1** is more than **solution-2**.

We have compare all results of read/write, write and read operation. Solution-2 has more throughput than solution-1 in read/write and write, but it is slightly less than solution-1 in read operation.

6.1.3. Storage Overhead

In solution-1, we have two collections Users_MHT and Users_MHTAggr which are used to save the two Merkle hash trees of the user for simple and aggregate queries. We have inserted thousands reading to check the storage overhead of Merkle hash trees. We have serialized the Merkle hash trees to save in MongoDB. For each user, there was only one row in Users_MHT collection. We have used SHA-256 for hashing. Each node in Merkle hash tree was of 256 bits. For a perfect Merkle hash tree, for L leaves, it has 2L-1 number of nodes.

$$Size\ of\ perfect\ Merkle\ hash\ tree\ =\ (2L-1)\ *\ 256\ bits$$

Equation 6-1 Size of Perfect Merkle hash tree

For 1000 records size of Merkle hash tree is:

$$Size\ =\ (2*1000\ -1)*\ 256\ =\ 511{,}744\ bits\ =\ 63.968\ Kbytes$$

This was the size of Merkle tree for 1000 records, but we had serialized the Merkle hash tree in such a way that we had used only L*256 bits space.

$$Implemented\ size\ =\ L\ *\ 256\ bits\ =\ 1000*\ 256\ =\ 256{,}000\ bits\ =\ 32\ Kbytes$$

So total size storage overhead for **solution-1** is the sum of both size of Users_MHT and size of Users_MHTAggr.

$$Total\ size\ =\ size\ of\ Users\_MHT\ +\ size\ of\ Users\_MHTAggr$$

Equation 6-2 Total Storage size in solution-1

Users_MHT size for one user was equal to data in that row of the user.

Size of Users_MHT = Size of the auto index + Size of user_id + Size of patient_MHT + Size of timestamp

*Size of Users_MHTAggr = (Size of the auto index + Size of user_id + Size of Agr_Array + Size of Agr_Node_Hash +Size of timestamp) * no of records*

Equation 6-3 Size of Users_MHT and Users_MHTAggr

For 10^3 records of a user:

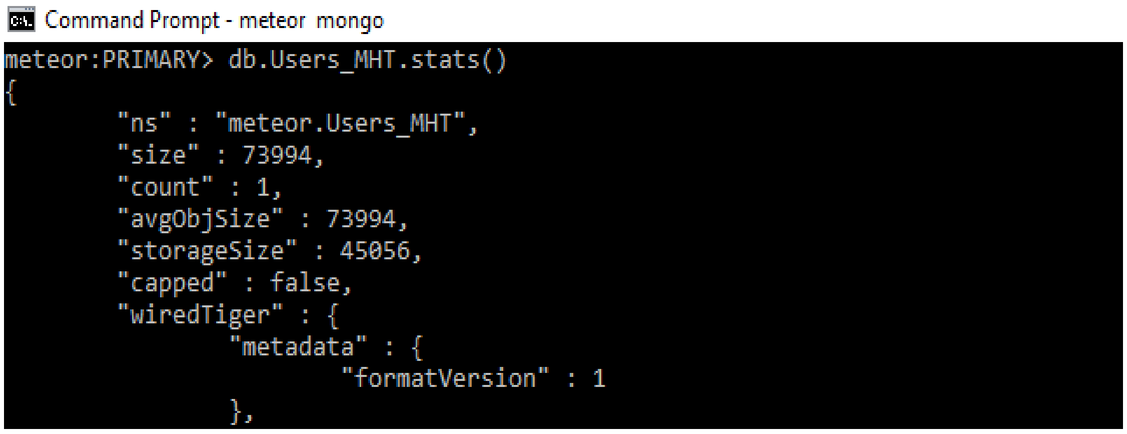
$$Size\ of\ Users\_MHT\ for\ single\ user\ = 40 + 40 + 73 * 1002 + 728 + 40\ =\ 73994\ bytes$$
$$= 0.074MB$$

$$Size\ of\ Users\_MHTAggr\ for\ single\ user\ =\ (27\ + 20\ + 89 + 32 + 22)\ * 1000$$
$$=\ 0.16\ Mbytes$$

For 10^3 records,

$$Total\ size\ is\ =\ 0.074\ +\ 0.16\ =\ 0.234\ Mbytes$$

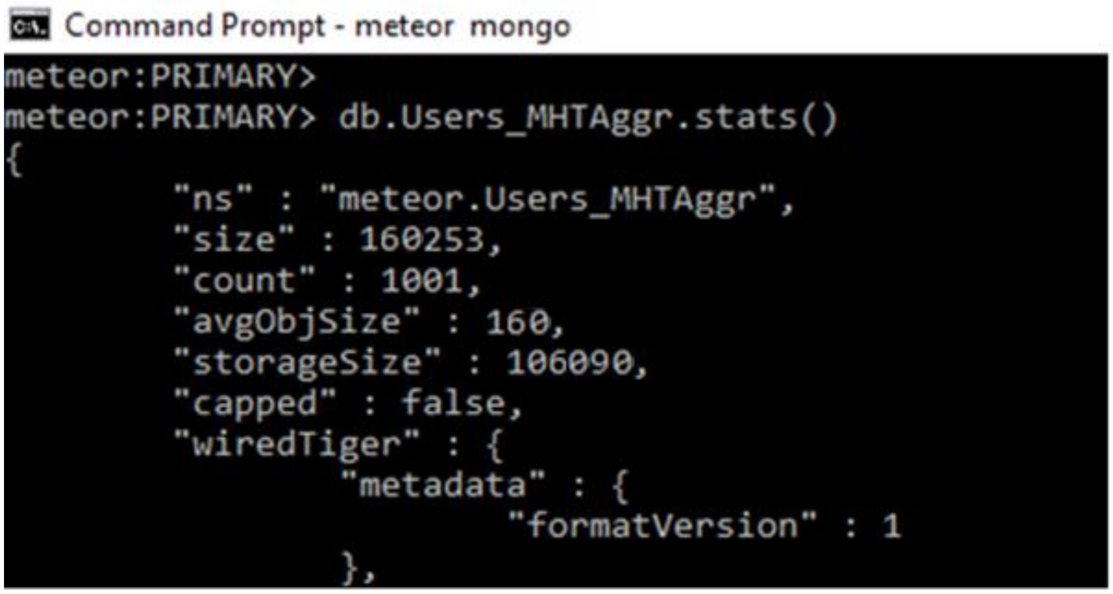
As this was verified from MongoDB command, "*db.Users_MHT.stats()*". Result of command is shown in figure 6-21.



```
Command Prompt - meteor  mongo
meteor:PRIMARY> db.Users_MHT.stats()
{
        "ns" : "meteor.Users_MHT",
        "size" : 73994,
        "count" : 1,
        "avgObjSize" : 73994,
        "storageSize" : 45056,
        "capped" : false,
        "wiredTiger" : {
                "metadata" : {
                        "formatVersion" : 1
                },
```

Figure 6-21 Users_MHT stats

Size was 73994 bytes (0.073994 MB) in Users_MHT collection for 1000 records of single user. Count one means only one user was in Users_MHT. We can see stats for Users_MHTAggr in figure 6-22



```
Command Prompt - meteor  mongo
meteor:PRIMARY>
meteor:PRIMARY> db.Users_MHTAggr.stats()
{
        "ns" : "meteor.Users_MHTAggr",
        "size" : 160253,
        "count" : 1001,
        "avgObjSize" : 160,
        "storageSize" : 106090,
        "capped" : false,
        "wiredTiger" : {
                "metadata" : {
                        "formatVersion" : 1
                },
```

Figure 6-22 Users_MHTAggr stats

Result of the stats command was 165253 bytes (0.165MB) in Users_MHTAggr collection for 1000 records of each user. All calculation were proved by MongoDB commands of collection stats as shown in figure 6-21 and 6-22.

We have calculated the storage overhead for **solution-2**. In solution-2, we have only Users_MHT2 table which was used to store the single Merkle hash tree. This single tree was used for both aggregation and simple queries verification. In this solution, we have also used SHA-256 for hashing. For a perfect Merkle hash tree, for L leaves, it has 2L-1 number of nodes.

$$Size\ of\ perfect\ Merkle\ hash\ tree\ =\ (2L-1)\ *\ 256\ bits$$

Equation 6-4 Size of Merkle hash tree

For 1000 records, size of Merkle hash tree:

$$Size\ =\ (2*1000\ -1)*\ 256\ =\ 511{,}744\ bits\ =\ 63.968\ Kbytes$$

This is the size of Merkle tree for 1000 records, but we have serialized the Merkle hash tree in such way that we need only L*256 bits space.

$$Implemented\ size\ =\ L\ *\ 256\ bits\ =\ 1000*\ 256\ =\ 256{,}000\ bits\ =\ 32\ Kbytes$$

$$Total\ size\ =\ size\ of\ Users\_MHT2$$

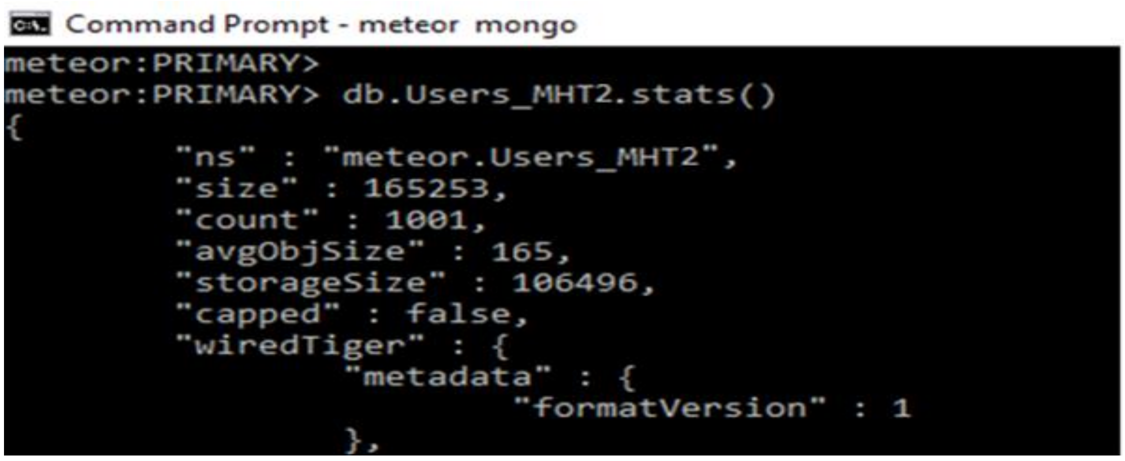
Size of Users_MHT2 = (Size of the auto index + Size of user_id + Size of Agr_Array + Size of Agr_Node_Hash +Size of timestamp) * no of records

Equation 6-5 Total Storage Size in Solution-2

For 10^3 records,

$$Total\ size\ is\ =\ (27\ +25\ +89+32+22)\ *1000\ =\ 0.163\ Mbytes$$

This result was verified from MongoDB command collection stat as shown in figure 6-23.



```
Command Prompt - meteor mongo
meteor:PRIMARY>
meteor:PRIMARY> db.Users_MHT2.stats()
{
        "ns" : "meteor.Users_MHT2",
        "size" : 165253,
        "count" : 1001,
        "avgObjSize" : 165,
        "storageSize" : 106496,
        "capped" : false,
        "wiredTiger" : {
                "metadata" : {
                        "formatVersion" : 1
                },
```

Figure 6-23 Users_MHT2 Stats

Table shows the storage size of solution-1 and solution-2:

Table 6-4 Storage Size Comparison

Number of Records	**Storage Size Solution-1**	**Storage SizeSolution-2**
10^3	0.234 MB	0.165 MB
10^4	2.34 MB	1.65 MB
10^5	23.4 MB	16.5 MB
10^6	234 MB	165 MB

So results of storage size showed that solution-2 was using less space as compare to solution-1. **Solution-1** takes 1.42 times more storage than

solution-2. The comparison graph for solution-1 and solution-2 is shown in figure 6-24.

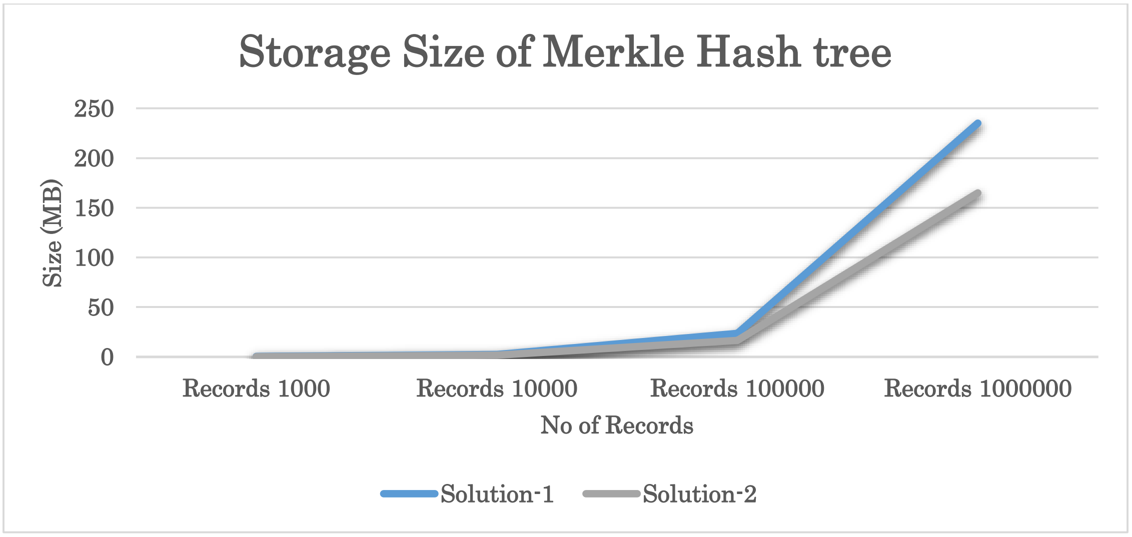


Figure 6-24 Storage Size Comparison Chart

6.1.4. Proof Size

Proof size is the size of data which is used to send to block chain and get from blockchain at time of verification. Proof size is independent of the number of records because root hash of Merkle hash tree is always 256 bits. The data sent to blockchain was Merkle root hash which was of 256 bits because of SHA-256. Root hash was sent with user id, and timestamp at which it was calculated. In solution-1, two Merkle root hashes were sent to blockchain, so proof size in solution-1 is:

$$Proof\ Size\ Sol-1\ =\ Root\ hash\ of\ Users\_MHT\ +\ Root\ Hash\ of\ Users\_MHTAggr$$

Equation 6-6 Proof size for Solution-1

Proof size in solution-2 was size of only one Merkle root hash. So proof size in solution-2 is:

$$Proof\ Size\ Sol-2\ =\ Root\ hash\ of\ Users\_MHT2$$

Equation 6-7 Proof size for solution-2

By comparing proof size of both solutions, solution-2 gets advantage over solution-1. Proof size of the **solution-2** is half of proof size of **solution-1**.

Table 6-5 Proof Size Comparison

Number of Records	**Proof Size Solution-1**	**Proof SizeSolution-2**
For any numbers of records	512 bits	256 bits

6.2. Comparison of Proposed Solutions

We have compared the proposed solution-1 and solution-2 with respect to all parameters discussed above. Latency comparison shows that solution-1 is faster than the solution-2. Throughput comparison shows that solution-2 can handle more requests per second as compare to solution-1. Storage overhead of solution-2 is less than solution-1. Proof size of solution-2 is also less than solution-1. Solution-1 is faster in term of verification latency due to two different authenticated data structures but solution-2 causes less overhead with respect to storage size and proof size.

Table 6-6 shows the recommended solution to use according its performance parameters.

Table 6-6 Proof Size Comparison

Performance Parameters	**Solution-1**	**Solution-2**	**Recommendation**
Latency (Read)	Fast	Slow	Solution-1
Latency (Write)	slow	Fast	Solution-2

Load speed (Verification)	Slow	Fast	Solution-2
Latency (Simple query)	Fast	Slow	Solution-1
Latency (Aggregated Query)	Fast	Slow	Solution-1
Throughput (Read)	Slow	Fast	Solution-2
Throughput (Write)	Slow	Fast	Solution-2
Throughput (Mix)	Slow	Fast	Solution-2
Storage Overhead	2.3 MB	1.65 MB	Solution-2
Proof Size	512 bit	256 bit	Solution-2

6.3. Comparison with existing solutions

In Verena [3], latency, throughput, and storage overhead are measured for their medical web application. As they have deployed their application on different specification machine, still our results are better than their solution. Storage overhead is calculated and compared with Verena, our results are very good. Table 6-7 shows the comparison.

In our **solution-2** only one Merkle hash tree is used for both aggregate and simple queries. Throughput, latency is also improved from Verena, but for actual comparison of latency and throughput testing should be done at same machine with same resources. As their code is not publicly available to test and evaluate. We have compared our results of storage size with results in the paper [3].

Table 6-7 Comparison with Verena

Number of Records	**Query type**	**Verena** [3]	**Storage Size Solution-1**	**Storage Size Solution-2**
10^4	Simple	1.64 MB	0.74 MB	1.65 MB (One tree is used for both)
10^4	Aggregate	1.95 MB	1.60 MB	
Total Size		3.59 MB	2.34 MB	1.65 MB

For 10^4 records of a user, query verification **solution-1** has total 2.34MB storage size. Query verification **solution-2** has total 1.65MB, to provide verification of simple and aggregated queries. Verena [3] has 2.2 times more storage size than our query verification solution-2. Verena has 1.5 times more storage size than our query verification solution-1. So, solution-1 and solution-2 have less storage size than Verena. It means that our solutions are more practical to use with real world web applications to provide verifiable databases queries.

6.4. Cost Analysis

In cost analysis, we have discussed how much addition cost is added to the existing solution. Cost is analyzed in term of the addition requirement for query verification solution. As we have seen storage overhead of query verification solution is very minimal. We do not need to buy any additional database for storage of Merkle hash tree. In proposed solutions, client need to communicate with blockchain for proof verification. So, in the internet environment, we need static IP address so that the end client can communicate with blockchain for proof verification. Blockchain service. We have used blockchain to store root hashes of the end users, so there is cost of blockchain service provider.

Chapter 7

7 Conclusion and Future Work

7.1. Conclusion

Database query verification is very important in all those scenarios where databases and web applications are residing on untrusted third party server or the attacker has full access of the server. This is the most common case in cloud based applications where databases are provided as application as service. In case of internet of things (IoT), databases are not hosted privately, due to high maintenance and licensing cost. The proposed solution covers all type of these cases, and provides query verification with bearable overhead. Our proposed solution enables the end user of web application to verify easily the result of queries which are executed on database. It also enables the end user to verify even main server (web application and databases) is complete compromised. Performance and cost analysis of query verification solution-2 show that it can be implemented with integrity sensitive web applications in real time. In our query verification solution, it is very practical to implement the solution very existing or newly developing web applications.

Database query verification solution enables the end user to make decisions on data with full confidence. It also helps the end user to trust on result of queries (simple and aggregated). In worst case, when an attacker changes the data in databases, it can easily be detected. Similarly, if result is incomplete and not fresh, it is also detected and notified to the end user. Our solution guarantees integrity with all its properties i.e. correctness, completeness and freshness. Authenticated data structures used in our solution are based on Merkle hash tree. It is more efficient, and gives $\log_2(n)$

number of proof helpers (Chapter 2, section 2.2) to verify any leave from total n numbers of leaves. Blockchain is very useful tool for keeping proof on it because it ensures the data on it is not changed and fresh (up to date). It also has traces of data changes with respect to time. Our solution is practical, fast. It is easy to integrate with existing web applications. New web applications can also integrate with query verification solution during development phase. This research on database query verification provides solution to all integrity sensitive web applications and databases in fully compromised environment.

7.2. Future Work

Our research in database query verification for fully compromised environment, opens many directions of modern technology research. Our research gives idea for development of new web servers (Apache, Microsoft IIS) which support authenticated data structures as its inbuilt feature. This web server can enable database query verification via its configuration file. This web server has different implementations of authenticated data structure like Merkle hash tree and skip lists. So, application owner can decide what type of authenticated data structure he wants to use according to his requirements in his web application. This should be independent of databases vendor and type (SQL or No-SQL).

In blockchain, there should be development required for authenticated data structure which can be stored on blockchain easily even they have very large numbers of data hashes. This research gives direction for open source database for development of authenticated data structures as its inbuilt feature. So, it can save authenticated data structures most efficiently. This helps the user to implement query verification solution with improvement in performance.

Trusted hardware modules (for example TPM 1.2) [64] which are used for secure computations. It is possible to use that hardware for database query verification. Main component of our solution on main server is query verification server which is used for proofs calculation and verification of queries. This query verification server should use trusted hardware on main server. It is idea to implement authenticated data structures on trusted hardware. It already provides cryptographic functions like hash (SHA-256, SHA-1), and signatures [65]. This isolates query verification server from main server logically on a compromised server. This trusted hardware module is secured against attacker, and very hard to extract data from it. In term of performance, specialized trusted hardware module has better performance than software base modules.

As this solution is for query verification means ensuring integrity of database, but for confidentiality it can be integrated with Mylar [66]. It can be easily integrated using Node.js. It can also be integrated with other frameworks which provide database encryption.

8 Bibliography

[1] D. Wichers, "Owasp top-10 2013," *OWASP Found. Febr.*, 2013.

[2] M. Cukier, "Study: Hackers Attack Every 39 Seconds | A. James Clark School of Engineering, University of Maryland." [Online]. Available: https://eng.umd.edu/news/story/study-hackers-attack-every-39-seconds.

[3] N. Karapanos, A. Filios, R. Popa, and S. Capkun, "Verena: End-to-End Integrity Protection for Web Applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 895–913.

[4] A. Motro, "Integrity = validity + completeness," *ACM Trans. Database Syst.*, vol. 14, no. 4, pp. 480–502, 1989.

[5] G. Weintraub and E. Gudes, "Data Integrity Verification in Column-Oriented NoSQL Databases BT - Data and Applications Security and Privacy XXXII," 2018, pp. 165–181.

[6] C. Braghin, A. Cortesi, and R. Focardi, *Freshness Analysis in Security Protocols*. 2003.

[7] J. H. Fisher, "The Dangers Of Misdiagnosis Can Sometimes Lead To Death - John H. Fisher, P.C.," 2019. [Online]. Available: https://protectingpatientrights.com/blog/the-dangers-of-misdiagnosis-can-sometimes-lead-to-death/.

[8] J. G. Ronquillo, J. Erik Winterholler, K. Cwikla, R. Szymanski, and C. Levy, "Health IT, hacking, and cybersecurity: national trends in data breaches of protected health information," *JAMIA Open*, vol. 1, no. 1, pp. 15–19, 2018.

[9] H. DuPreez, "Top 10 Databases for 2019 — DatabaseJournal.com,"

2019. [Online]. Available: https://www.databasejournal.com/features/oracle/slideshows/top-10-2019-databases.html.

[10] Oracle, "Introducing Oracle Database Security." [Online]. Available: https://docs.oracle.com/cd/B28359_01/network.111/b28531/intro.htm#DBSEG001.

[11] MySql, "MySQL :: MySQL Enterprise Encryption." [Online]. Available: https://www.mysql.com/products/enterprise/encryption.html.

[12] Microsoft, "SQL Server Security | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/sql-server-security.

[13] Joshua Otwell, "Top PostgreSQL Security Threats | Severalnines." [Online]. Available: https://severalnines.com/blog/top-postgresql-security-threats.

[14] Mongodb, "Encryption at Rest — MongoDB Manual." [Online]. Available: https://docs.mongodb.com/manual/core/security-encryption-at-rest/#encryption-at-rest.

[15] A. Miller *et al.*, "Authenticated data structures, generically," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14*, 2014, vol. 49, no. 1, pp. 411–423.

[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, 2006, p. 121.

[17] M. T. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an

authenticated dictionary with skip lists and commutative hashing," *Proc. - DARPA Inf. Surviv. Conf. Expo. II, DISCEX 2001*, vol. 2, pp. 68–82, 2001.

[18] J. Xu, F. Zhou, M. Yang, F. Li, and Z. Zhu, "Hierarchical Hash list for distributed query authentication," *Jisuanji Yanjiu yu Fazhan/Computer Res. Dev.*, vol. 49, pp. 1533–1544, 2012.

[19] G. Becker, "Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis," 2019.

[20] R. C. Merkle, "A Certified Digital Signature," in *Proceedings on Advances in Cryptology*, 1989, pp. 218–238.

[21] B. Preneel and Bart, "The State of Hash Functions and the NIST SHA-3 Competition," in *Information Security and Cryptology*, Springer-Verlag, 2009, pp. 1–11.

[22] H. Handschuh, "SHA Family (Secure Hash Algorithm)," in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg, Ed. Boston, MA: Springer US, 2005, pp. 565–567.

[23] R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets," *Commun. ACM*, vol. 20, no. 11, pp. 841–850, Nov. 1977.

[24] V. Rijmen and E. Oswald, "Update on SHA-1," in *Proceedings of the 2005 International Conference on Topics in Cryptology*, 2005, pp. 58–71.

[25] Q. H. Dang, "Secure Hash Standard," Gaithersburg, MD, Jul. 2015.

[26] M. J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," Gaithersburg, MD, Jul. 2015.

[27] M. Dworkin, "Hash Functions | CSRC," 2017. [Online]. Available:

https://csrc.nist.gov/projects/hash-functions.

[28] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan, “Verifying completeness of relational query results in data publishing,” p. 407, 2005.

[29] Y. Zhang, J. Katz, and C. Papamanthou, “IntegriDB: Verifiable SQL for Outsourced Databases,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1480–1491.

[30] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, “CryptDB: Protecting Confidentiality with Encrypted Query Processing,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 85–100.

[31] B. Palazzi, M. Pizzonia, and S. Pucacco, “Query Racing: Fast Completeness Certification of Query Results,” in *Data and Applications Security and Privacy XXIV*, 2010, pp. 177–192.

[32] M. Jarke and Y. Vassiliou, “A Framework for Choosing a Database Query Language,” *ACM Comput. Surv.*, vol. 17, no. 3, pp. 313–340, Sep. 1985.

[33] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, “vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 863–880.

[34] H. Pang and K.-L. Tan, “Verifying Completeness of Relational Query Answers from Online Servers,” *ACM Trans. Inf. Syst. Secur.*, vol. 11, no. 2, pp. 5:1--5:50, May 2008.

[35] Q. Zheng, S. Xu, and G. Ateniese, “Efficient Query Integrity for Outsourced Dynamic Databases,” in *Proceedings of the 2012 ACM*

Workshop on Cloud Computing Security Workshop, 2012, pp. 71–82.

[36] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and Integrity in Outsourced Databases," *Trans. Storage*, vol. 2, no. 2, pp. 107–138, May 2006.

[37] G. Weintraub and E. Gudes, "Data Integrity Verification in Column-Oriented NoSQL Databases: 32nd Annual IFIP WG 11.3 Conference, DBSec 2018, Bergamo, Italy, July 16–18, 2018, Proceedings," 2018, pp. 165–181.

[38] Z. Gao, B. Wang, H. Liu, K. Lu, and Y. Zhan, "Verifiable Auditing Protocol with Proxy Re-Encryption for Outsourced Databases in Cloud," *Wuhan Univ. J. Nat. Sci.*, vol. 23, no. 2, pp. 120–128, Apr. 2018.

[39] P. Fortuna, N. Pereira, and I. Butun, "A Framework for Web Application Integrity," in *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, 2018, pp. 487–493.

[40] H. Schuldt, "Application Server," in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, p. 104.

[41] B. Samatha, "List of Top 5 Most Popular Open Source Web Servers." [Online]. Available: https://www.technotification.com/2019/01/open-source-web-servers.html.

[42] P. Kortum and S. C. Peres, "Evaluation of Home Health Care Devices: Remote Usability Assessment.," *JMIR Hum. factors*, vol. 2, no. 1, p. e10, Jun. 2015.

[43] Nodejs, "Node.js." [Online]. Available: https://nodejs.org/en/.

[44] Meteor, “Build Apps with JavaScript | Meteor.” [Online]. Available: https://www.meteor.com/.

[45] Mongodb, “The most popular database for modern apps | MongoDB.” [Online]. Available: https://www.mongodb.com/.

[46] D. Warnock, *MongoDB: Learn MongoDB in a Simple Way!* USA: CreateSpace Independent Publishing Platform, 2016.

[47] K. Chodorow, *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.

[48] Npm, “npm | build amazing things.” [Online]. Available: https://www.npmjs.com/.

[49] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, “Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 385–395.

[50] Atom, “Atom.” [Online]. Available: https://atom.io/.

[51] M. H. and D. B. Emily Stark, “SJCL: a Javascript crypto library.” [Online]. Available: https://crypto.stanford.edu/sjcl/.

[52] Blazjs, “Blaze | BlazeJS.” [Online]. Available: http://blazejs.org/api/blaze.html.

[53] J. Bukowski, “merkle-tools - npm.” [Online]. Available: https://www.npmjs.com/package/merkle-tools.

[54] M. S. Niaz and G. Saake, “Merkle hash tree based techniques for data integrity of outsourced data,” *CEUR Workshop Proc.*, vol. 1366, pp. 66–71, 2015.

[55] Hyperledger, “Hyperledger – Open Source Blockchain Technologies.” [Online]. Available: https://www.hyperledger.org/.

[56] A. Hidayat, "PhantomJS - Scriptable Headless Browser." [Online]. Available: http://phantomjs.org/.

[57] A. Hidayat, "Network Monitoring with PhantomJS." [Online]. Available: http://phantomjs.org/network-monitoring.html.

[58] J. Odvarko, "HTTP Archive Viewer 2.0.17." [Online]. Available: http://www.softwareishard.com/har/viewer/.

[59] Google, "HAR Analyzer | GSuite Toolbox." [Online]. Available: https://toolbox.googleapps.com/apps/har_analyzer/.

[60] E. J. Duran, "Chrome HAR Viewer." [Online]. Available: https://ericduran.github.io/chromeHAR/.

[61] Apache, "Apache JMeter™." [Online]. Available: https://jmeter.apache.org/.

[62] Gatling, "Gatling Open-Source Load Testing." [Online]. Available: https://gatling.io/.

[63] badboy, "Badboy Software." [Online]. Available: https://www.badboy.com.au/.

[64] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*, 1st ed. Berkely, CA, USA: Apress, 2015.

[65] M. Hell, L. Karlsson, B. Smeets, and J. Mirosavljevic, "Using TPM Secure Storage in Trusted High Availability Systems," in *Revised Selected Papers of the 6th International Conference on Trusted Systems - Volume 9473*, 2015, pp. 243–258.

[66] R. A. Popa *et al.*, "Building Web Applications on Top of Encrypted Data Using Mylar," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 157–172.