A Secure and Dynamic Multi-Keyword Ranked Search Scheme over Encrypted Cloud Data

Project Stage II (CS851PC)

Submitted

in partial fulfilment of the

requirements for the award of the

degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

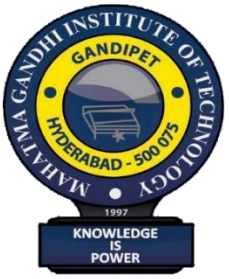
SHIVAYALA MANJUSHA (21261A0555)

SHUCHITA PRAKASH (21261A0557)

Under the guidance of

Mrs. D S BHAVANI,

Assistant Professor (CSE)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY(AUTONOMOUS)

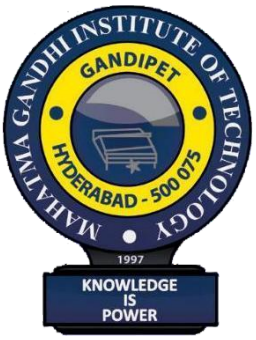
GANDIPET, HYDERABAD-500 075, INDIA

2024-2025

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

(Affiliated to Jawaharlal Nehru Technological University Hyderabad)

Gandipet, Hyderabad-500 075, Telangana (India)



CERTIFICATE

This is to certify that the Report entitled **"A Secure and Dynamic Multi-Keyword Ranked Search Scheme over Encrypted Cloud Data.",** is being submitted by **Ms. Shivayala Manjusha** bearing **Roll No: 21261A0555** and **Ms. Suchitha Prakash** bearing **Roll No: 21261A0557** in partial fulfillment for completion of **Bachelor of Technology** in **Computer Science and Engineering** to **Mahatma Gandhi Institute of Technology** is a record of bona-fide work carried out by them under our guidance and supervision. The results embodied in this project have not been submitted to any other University or Institute for the award of any degree or diploma.

Project Guide
Mrs. D. S. Bhavani
Asst. Professor, Dept. of CSE

Head of the Department
Dr. C.R.K Reddy
Professor, Dept. of CSE

DECLARATION

This is to certify that the work described in this report, entitled **"A Secure and Dynamic Multi-Keyword Ranked Search Scheme over Encrypted Cloud Data"** is a record of work done by us in the Department of Computer Science and Engineering, Mahatma Gandhi Institute of Technology, Hyderabad. No part of the work is copied from books/journals/internet and wherever the portion is taken, the same has been duly referred to in the text. The report is based on the work done entirely by us and not copied from any other source.

SHIVAYALA MANJUSHA

(21261A0555)

SHUCHITA PRAKASH

(21261A0557)

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of people who made it possible because success is the abstract of hard work and perseverance, but steadfast of all is encouraging guidance. So, we acknowledge all those whose guidance and encouragement served as a beacon light and crowned my efforts with success.

.

We would like to express our sincere thanks to **Prof G. Chandra Mohan Reddy, Principal MGIT**, for providing the working facilities in college.

We wish to express our sincere thanks and gratitude to **Dr. C R K Reddy, Professor and HOD**, Department of CSE, MGIT, for all the timely support and valuable suggestions during the period of project.

We are extremely thankful to **Dr. V. Subbaramaiah, Assistant Professor, Department of CSE, MGIT, Mr. G. Nagireddy, Assistant Professor, Department of CSE, MGIT and Ms. M. Mamatha, Assistant Professor, Department of CSE, MGIT**, Project Coordinators for their encouragement and support throughout the project.

Finally, we would also like to thank all the faculty and staff of the CSE Department who helped us directly or indirectly in completing this project.

SHIVAYALA MANJUSHA
(21261A0555)

SHUCHITA PRAKASH
(21261A0557)

ABSTRACT

Due to the increasing popularity of cloud computing, more and more data owners are motivated to outsource their data to cloud servers for greater convenience and reduced cost in data management. However, sensitive data should be encrypted before outsourcing to meet privacy requirements, which makes data utilization like keyword-based document retrieval challenging. In this project, we present a secure multi-keyword ranked search scheme over encrypted cloud data, which simultaneously supports dynamic update operations like deletion and insertion of documents. Specifically, the widely-used TF-IDF model is adopted in the index construction and query generation. The AES algorithm is utilized to encrypt the index and query vectors, while ensuring accurate relevance score calculation between them. Due to the design of our system, the proposed scheme can achieve efficient search performance and flexibly handle the deletion and insertion of documents.

TABLE OF CONTENTS

CHAPTER		PAGE NO.
Certificate		i
Declaration		ii
Acknowledgement		iii
Abstract		iv
Table of Contents		v
List of Figures		vii
List of Tables		viii
CHAPTER 1 INTRODUCTION		**1-5**
1.1	Problem Definition	2
1.2	Existing Solutions	3
1.3	Proposed Solution	3
1.4	Requirements Specification	4
CHAPTER 2 LITERATURE SURVEY		**6-19**
CHAPTER 3 DESIGN AND METHODOLOGY		**20-30**
3.1	Block Diagram of Search Scheme	20
3.2	Architecture of Search Scheme	22
3.3	Flow Diagrams of Search Scheme	25
	Data Owner Flow of Search Scheme	25
	Data User Flow of Search Scheme	26
	Cloud Server Flow of Search Scheme	27
3.4	UML Diagrams of Search Scheme	28
	Use Case Diagram of Search Scheme	28
	Class Diagram of Search Scheme	29
	Sequence Diagram of Search Scheme	30
CHAPTER 4 IMPLEMENTATION		**31-42**
4.1	Tech Stack	31
4.2	Features	32
4.3	Implementation Details	40

CHAPTER 5 TESTING 43

CHAPTER 5 RESULTS 44-52

CHAPTER 7 CONCLUSION AND FUTURE SCOPE 53

REFERENCES 54

APPENDIX 55-83

LIST OF FIGURES

Fig. No.	Description	PageNo.
3.1	Block Diagram of Search Scheme	20
3.2	Architecture of Search Scheme	22
3.3	Data Owner Flow of Search Scheme	25
3.4	Data User Flow of Search Scheme	26
3.5	Cloud Server Flow of Search Scheme	27
3.6	Use Case Diagram of Search Scheme	28
3.7	Class Diagram of Search Scheme	29
3.8	Sequence Diagram of Search Scheme	30
6.1	Project Cover - Navigate to github page and document	44
6.2	Project Overview	45
6.3	Tech Stack	45
6.4	Entities - Navigate to Data Owner	46
6.5	Entities - Navigate to Encrypted Cloud Server	47
6.6	Entities - Navigate to Data User	47
6.7	Data Owner - Upload Documents	48
6.8	Data Owner - Upload documents to cloud	49
6.9	Data Owner - View Documents	49
6.10	Encrypted Cloud Server	50
6.11	Data User - Single keyword search	51
6.12	Data User - Multi Keyword Ranked Search Scheme	52
6.13	Data User - View Downloaded files	52

LIST OF TABLES

Table. No.	Description	Page No.
1	Literature Survey	17
2	Test Cases	43

CHAPTER 1

INTRODUCTION

Cloud computing has been considered as a new model of enterprise IT infrastructure, which can organize huge resources of computing, storage and applications, and enable users to enjoy ubiquitous, convenient and on demand network access to a shared pool of configurable computing resources with great efficiency and minimal economic overhead. Attracted by these appealing features, both individuals and enterprises are motivated to outsource their data to the cloud, instead of purchasing software and hardware to manage the data themselves. Despite the various advantages of cloud services, outsourcing sensitive information (such as e-mails, personal health records, company finance data, government documents, etc.) to remote servers brings privacy concerns. The cloud service providers (CSPs) that keep the data for users may access users' sensitive information without authorization. A general approach to protect the data confidentiality is to encrypt the data before outsourcing. However, this will cause a huge cost in terms of data usability. For example, the existing techniques on keyword-based information retrieval, which are widely used on the plaintext data, cannot be directly applied on the encrypted data. Downloading all the data from the cloud and decrypting locally is obviously impractical. In order to address the above problem, researchers have designed some general-purpose solutions with fully-homomorphic encryption or oblivious RAMs. However, these methods are not practical due to their high computational overhead for both the cloud server and user. On the contrary, more practical special-purpose solutions, such as searchable encryption (SE) schemes have made specific contributions in terms of efficiency, functionality and security. Searchable encryption schemes enable the client to store the encrypted data to the cloud and execute keyword search over the ciphertext domain. So far, abundant works have been proposed under different threat models to achieve various search functionality, such as single keyword search, similarity search, multi-keyword Boolean search, ranked search, multi-keyword ranked search, etc. Among them,

multi-keyword ranked search achieves more and more attention for its practical applicability. Recently, some dynamic schemes have been proposed to support inserting and deleting operations on document collection. These are significant works as it is highly possible that the data owners need to update their data on the cloud server. But few of the dynamic schemes support efficient multi-keyword ranked search.

This project proposes a secure search scheme over encrypted cloud data, supporting multi-keyword ranked search and dynamic operations on the document collection. Specifically, the widely-used "term frequency (TF)–inverse document frequency (IDF)" model is adopted in the index construction and query generation to enable effective multi-keyword ranked search. The AES algorithm is utilized to encrypt the data and queries, while ensuring accurate relevance score calculation between them. The proposed scheme efficiently handles search operations and flexibly supports document deletion and insertion. Extensive experiments are conducted to demonstrate the effectiveness and efficiency of the proposed approach.

1.1 PROBLEM DEFINITION

Problem Definition

With the growing reliance on cloud computing, individuals and enterprises are increasingly outsourcing data to cloud servers due to the convenience and cost-effectiveness of this approach. However, outsourcing sensitive information such as emails, health records, financial data, or government documents raises significant privacy concerns. Although encryption is a standard solution for protecting data confidentiality, it renders traditional data utilization methods, such as keyword-based document retrieval, ineffective.

Existing methods to address this problem, like fully homomorphic encryption and oblivious RAMs, provide strong security but are impractical due to high computational overhead. More efficient approaches, such as searchable encryption (SE) schemes, have been proposed to enable

searching over encrypted data. These schemes have advanced from simple single-keyword searches to more functional multi-keyword ranked searches.

However, current solutions face limitations:

1. **Limited Dynamism:** Many schemes do not support dynamic operations (e.g., insertion and deletion of documents).
2. **Efficiency Challenges:** Achieving both secure and efficient search mechanisms remains complex.
3. **Privacy Threats:** Existing schemes may not adequately protect against advanced attacks, such as statistical frequency analysis.

1.2 EXISTING SOLUTION

Existing solutions for searchable encryption can be broadly categorized into single-keyword search, multi-keyword Boolean search, ranked search, and dynamic search schemes. Early works like those of Song et al. introduced single-keyword searchable encryption, which provided basic functionality but lacked advanced features like ranking or multi-keyword support. Multi-keyword Boolean searches extended functionality by supporting conjunctive, disjunctive, and predicate searches, but these schemes were limited to unranked results. Ranked search schemes, such as those by Cao et al. and Sun et al., enabled relevance-based ranking using vector space models and TF-IDF weighting, though they often suffered from computational inefficiency and did not fully account for keyword importance. Dynamic schemes, like those proposed by Kamara and Papamanthou, addressed document updates but were limited to single-keyword search and lacked efficiency in ranking. While hybrid techniques combining advanced indexing and encryption methods have been proposed, they generally fail to balance efficiency, ranking accuracy, and security, leaving significant scope for improvement.

1.3 PROPOSED SOLUTION

This paper proposes a secure search scheme over encrypted cloud data that supports multi-keyword ranked search and dynamic operations on the

document collection. Specifically, the widely-used "term frequency (TF)–inverse document frequency (IDF)" model is employed during index construction and query generation to provide effective multi-keyword ranked search. The AES encryption algorithm is utilized to secure the data and queries while ensuring accurate relevance score calculation between them. The proposed scheme achieves efficient search performance and supports flexible updates, including the deletion and insertion of documents.

1.4 REQUIREMENTS SPECIFICATION

The requirement specifications outline the software and hardware resources utilized in this project for implementing and testing the dynamic ranked search over encrypted cloud data.

Software Requirements

1. **Operating System:** Microsoft Windows XP or later.
2. **Programming Language:** JavaScript, Java.
3. **Database:** MongoDB.
4. **Development Environment:** Integrated development environment (IDE) supporting J2EE, such as Eclipse or NetBeans, VSCode.
5. **Supporting Libraries:** Required JavaScript libraries for encryption, indexing, and Java Springboot.

Hardware Requirements

1. **System Requirements for Development:**
    a. **Processor:** Pentium IV 2.4 GHz or higher.
    b. **Memory (RAM):** 512 MB or more.
    c. **Display:** 15-inch VGA colour monitor.
    d. **Storage:** 40 GB hard disk with 1.44 MB floppy drive.
    e. **Input Devices:** Logitech mouse and keyboard.

This configuration ensures efficient implementation and testing of the secure multi-keyword ranked search algorithm and dynamic data operations.

CHAPTER 2

LITERATURE SURVEY

1. Software protection and stimulation on oblivious rams

Authors: Oded Goldreich, Rafail Ostrovsky**[1]**

Software protection is one of the most important issues concerning computer practice. There exist many heuristics and ad-hoc methods for protection, but the problem as a whole has not received the theoretical treatment it deserves. In this paper, we provide theoretical treatment of software protection. We reduce the problem of software protection to the problem of efficient simulation on *oblivious* RAM.

A machine is *oblivious* if the sequence in which it accesses memory locations is equivalent for any two inputs with the same running time. For example, an oblivious Turing Machine is one for which the movement of the heads on the tapes is identical for each computation. (Thus, the movement is independent of the actual input.) *What is the slowdown in the running time of a machine, if it is required to be oblivious?* In 1979, Pippenger and Fischer showed how a two-tape *oblivious* Turing Machine can simulate, on-line, a one-tape Turing Machine, with a logarithmic slowdown in the running time. We show an analogous result for the random-access machine (RAM) model of computation. In particular, we show how to do an on-line simulation of an arbitrary RAM by a probabilistic *oblivious* RAM with a polylogaithmic slowdown in the running time. On the other hand, we show that a logarithmic slowdown is a lower bound.

2. A fully homomorphic encryption scheme

Authors: Craig Gentry**[2]**

We propose the first fully homomorphic encryption scheme, solving a central open problem in cryptography. Such a scheme allows one to compute arbitrary functions over encrypted data without the decryption key – i.e., given encryptions E(m1), . . . , E(mt) of m1, . . . , mt , one can efficiently

compute a compact ciphertext that encrypts f(m1, . . . , mt) for any efficiently computable function f. This problem was posed by Rivest et al. in 1978. Fully homomorphic encryption has numerous applications. For example, it enables private queries to a search engine – the user submits an encrypted query and the search engine computes a succinct encrypted answer without ever looking at the query in the clear. It also enables searching on encrypted data – a user stores encrypted files on a remote file server and can later have the server retrieve only files that (when decrypted) satisfy some boolean constraint, even though the server cannot decrypt the files on its own. More broadly, fully homomorphic encryption improves the efficiency of secure multiparty computation. Our construction begins with a somewhat homomorphic "boostrappable" encryption scheme that works when the function f is the scheme's own decryption function. We then show how, through recursive self-embedding, bootstrappable encryption gives fully homomorphic encryption. The construction makes use of hard problems on ideal lattices.

3. Cryptographic cloud storage

Authors: Seny Kamara, Kristin Lauter**[3]**

We consider the problem of building a secure cloud storage service on top of a public cloud infrastructure where the service provider is not completely trusted by the customer. We describe, at a high level, several architectures that combine recent and non-standard cryptographic primitives in order to achieve our goal. We survey the benefits such an architecture would provide to both customers and service providers and give an overview of recent advances in cryptography motivated specifically by cloud storage.

4. Security challenges for public cloud

Authors: J. Athena, V. Sumathy**[4]**

Numerous advancements in the Information Technology (IT) require the proper security policy for the data storage and transfer among the cloud.

With the increase in size of the data, the time required to handle the huge-size data is more. An assurance of security in cloud computing suffers various issues. The evolution of cryptographic approaches addresses these limitations and provides the solution to the data preserving. There are two issues in security assurance such as geographical distribution and the multi-tenancy of the cloud server. This paper surveys about the various cryptographic techniques with their key sizes, time required for key/signature generation and verification constraints. The survey discusses the architecture for secure data transmissions among the devices, challenges raised during the transmission and attacks. This paper presents the brief review of major cryptographic techniques such as Rivest, Shamir Adleman (RSA), Dffie Hellman and the Elliptic Curve Cryptography (ECC) associated key sizes. This paper investigates the general impact of digital signature generation techniques on cloud security with the advantages and disadvantages. The results and discussion section existing in this paper investigate the time consumption for key/signature generation and verification with the key size variations effectively. The initialization of random prime numbers and the key computation based on the points on the elliptic curve assures the high-security compared to the existing schemes with the minimum time consumption and sizes in cloud-based applications.

5. Searchable Encryption

Authors: Sunitha Buchade**[5]**

To manage large volume of data, most of the organizations and data owners outsource their data to remote cloud storage servers. Since the remote servers are untrusted party, the data has to be encrypted to achieve security and privacy. But encryption and decryption causes communication overhead for many data utilization operations like searching and updating. To overcome this contradiction, searchable encryption notion was introduced. Searchable encryption (SE) is an ability of a server to search upon the ciphertext and retrieve the data without decrypting it. By searching on the ciphertext it protects the user's tactful data. This article concentrates on study of various resil-ient keyword search schemes in area

of Cloud SE. The main basis for SE evolution is attaining the challenging task of efficient enci-phered data utilization with privacy check. Thus schemes are developed considering their search capability, efficiency and security. In this paper we summarize and analyze various available keyword searchable schemes based on query expressiveness and query correct-ness. We review articles which have extensively researched in the field of SE with various keyword search schemes.

6. Multi-keyword ranked search

Authors: Jiayi Li, Jianfeng Ma, Yinbin Miao, Ruikang Yang, Ximeng Liu, Kim-Kwang Raymond Choo**[6]**

With the explosive growth of data volume in the cloud computing environment, data owners are increasingly inclined to store their data on the cloud. Although data outsourcing reduces computation and storage costs for them, it inevitably brings new security and privacy concerns, as the data owners lose direct control of sensitive data. Meanwhile, most of the existing ranked keyword search schemes mainly focus on enriching search efficiency or functionality, but lack of providing efficient access control and formal security analysis simultaneously. To address these limitations, in this article we propose an efficient and privacy-preserving **Multi-keyword Ranked Search** scheme with **Fine-grained** access control (MRSF). MRSF can realize highly accurate ciphertext retrieval by combining coordinate matching with Term Frequency-Inverse Document Frequency (TF-IDF) and improving the secure k NN method. Besides, it can effectively refine users' search privileges by utilizing the polynomial-based access strategy. Formal security analysis shows that MRSF is secure in terms of confidentiality of outsourced data and the privacy of index and tokens. Extensive experiments further show that, compared with existing schemes, MRSF achieves higher search accuracy and more functionalities efficiently.

7. Dynamic Update

Authors: Jiawei Wan, Shijie Jia. Limin Liu, Yang Zhang**[7]**

Cloud storage is an increasingly popular service of cloud computing, which can provide convenient on-demand data outsourcing services and release the burden of maintaining local data for both individuals and organizations. However, the cloud service providers are not fully trusted by the users. The reason is that the users lose physical control of their cloud data, and the cloud service providers may conceal the status of the data when encountering data loss accidents for reputation. Therefore, it is critical for users to efficiently verify the integrity of cloud data.In this paper, we propose a public integrity verification scheme to support efficient dynamic update of cloud data based on Merkle Hash Tree linked list (MHT-list), which is a novel two-dimensional data structure we designed. This structure utilizes multiple merkle hash trees (MHTs) and a linked list to record data information at the cloud service provider side. Meanwhile, we exploit the structural advantages of the MHT-list to make our scheme more efficient in dynamic update and integrity verification than existing works. Moreover, we formally prove the security of the proposed scheme and evaluate the performance of our scheme by concrete extensive experiments. The results demonstrate that our proposed scheme achieves dynamic update effectively in public integrity verification of cloud data, and outperforms the previous works in computation and communication overhead.

8. Cloud Computing

Authors: Benneth Uzoma, Bonaventure Okhuoya**[8]**

Cloud computing is a key technological development in the information technology industry. It is one of the best techniques for managing and allocating a lot of information and resources across the entire internet. Technically speaking, cloud computing refers to accessing IT infrastructure through a computer network without having to install anything on your personal computer. Businesses can modify their resource levels to match their operational needs by utilizing cloud computing. Organizations and corporations can cut infrastructural costs with the use of cloud computing. Organizations can test their applications more quickly, with better

management, and with less upkeep. The IT team can adapt resources to changing and erratic requirements thanks to cloud computing. There is proof that cloud computing has a role in everyday life thanks to various applications in various contexts. This essay will cover every aspect of cloud computing, including its architecture, traits, types, service models, advantages, and challenges.

9. Multi-owner Secure Encrypted Search Using Searching Adversarial Networks

Authors: Kai Chen, Zhongrui Lin, Jian Wan, Lei Xu, Chungen Xu**[9]**

Searchable symmetric encryption (SSE) for multi-owner models draws much attention as it enables data users to perform searches over encrypted cloud data outsourced by data owners. However, implementing secure and precise query, efficient search and flexible dynamic system maintenance at the same time in SSE remains a challenge. To address this, this paper proposes secure and efficient multi-keyword ranked search over encrypted cloud data for multi-owner models based on searching adversarial networks. We exploit searching adversarial networks to achieve optimal pseudo-keyword padding, and obtain the optimal game equilibrium for query precision and privacy protection strength. Maximum likelihood search balanced tree is generated by probabilistic learning, which achieves efficient search and brings the computational complexity close to {O}(\log N).In addition, we enable flexible dynamic system maintenance with balanced index forest that makes full use of distributed computing. Compared with previous works, our solution maintains query precision above 95% while ensuring adequate privacy protection, and introduces low overhead on computation, communication and storage.

10. Generic Multi-keyword Ranked Search on Encrypted Cloud Data

Authors: Baocheng Zhang, Rongxing Lu, Kui Ren, Zhiguang Qin, Xuemin (Sherman) Shen**[10]**

Although searchable encryption schemes allow secure search over the encrypted data, they mostly support conventional Boolean keyword search, without capturing any relevance of the search results. This leads to a large amount of post-processing overhead to find the most matching documents and causes an unnecessary communication cost between the servers and end-users. Such problems can be addressed efficiently using a ranked search system that retrieves the most relevant documents. However, existing state-of-the-art solutions in the context of Searchable Symmetric Encryption (SSE) suffer from either (a) security and privacy breaches due to the use of Order Preserving Encryption (OPE) or (b) non-practical solutions like using the two non-colluding servers. In this paper, we present a generic solution for multi-keyword ranked search over the encrypted cloud data. The proposed solution can be applied over different symmetric searchable encryption schemes. To demonstrate the practicality of our technique, in this paper we leverage the Oblivious Cross Tags (OXT) protocol of Cash et al. (2013) due to its scalability and remarkable flexibility to support different settings. Our proposed scheme supports the multi-keyword search on Boolean, ranked and limited range queries while keeping all of the OXT's properties intact. The key contribution of this paper is that our scheme is resilient against all common attacks that take advantage of OPE leakage while only a single cloud server is used. Moreover, the results indicate that using the proposed solution the communication overhead decreases drastically when the number of matching results is large.

11. Secure Phrase Search for Intelligent Processing of Encrypted Data in Cloud-Based IoT

Authors: Yue Tong, Ximeng Liu, Jinbo Xiong, Robert H. Deng**[11]**

Phrase search allows retrieval of documents containing an exact phrase, which plays an important role in many machine learning applications for cloud-based IoT, such as intelligent medical data analytics. In order to protect sensitive information from being leaked by service providers, documents (e.g., clinic records) are usually encrypted by data owners before

being outsourced to the cloud. This, however, makes the search operation an extremely challenging task. Existing searchable encryption schemes for multi-keyword search operations fail to perform phrase search, as they are unable to determine the location relationship of multiple keywords in a queried phrase over encrypted data on the cloud server side. In this paper, we propose P3, an efficient privacy-preserving phrase search scheme for intelligent encrypted data processing in cloud-based IoT. Our scheme exploits the homomorphic encryption and bilinear map to determine the location relationship of multiple queried keywords over encrypted data. It also utilizes a probabilistic trapdoor generation algorithm to protect users' search patterns. Thorough security analysis demonstrates the security guarantees achieved by P3. We implement a prototype and conduct extensive experiments on real-world datasets. The evaluation results show that compared with existing multi keyword search schemes, P3 can greatly improve the search accuracy with moderate overheads.

12. Multi-keyword ranked search with access control for multiple data owners in the cloud

Authors: Xiaohua Dai, Zhitao Guan, Yushu Zhang, Zhigang Zhou**[12]**

Secure search is important for promoting the widespread use of cloud computing. As a common situation, when data users search for certain data from multiple data owners, searchable encryption raises the possibility of secure searches over encrypted cloud data. However, most of the existing schemes for the multi-owner model are based on asymmetric searchable encryption, which is inefficient for searching unstructured data. In addition, a Trusted Third-Party (TTP) is often required to assist in encrypting indexes or enforcing access control, which increases the cost and becomes a bottleneck to system security. To overcome these limitations, this paper proposes a secure multi-keyword ranked search scheme with access control for the multi-owner model, named MOMRS-AC. Specifically, we extend the basic secure KNN-based search scheme considering the single-owner model to an efficient multi-keyword ranked search scheme without TTP for the

multi-owner model, and provide a signature algorithm to verify the search results. Also, we construct a fine-grained access control that supports recipient anonymity, large-universe and decentralized authorities. To lower the overhead of data users, we extend MOMRS-AC to support outsourcing of verification and decryption. Security and performance analyses demonstrate that MOMRS-AC is secure and efficient for the multi-owner model in practice.

13. Semantic-Based Multi-Keyword Ranked Search Schemes over Encrypted Cloud Data

Authors: Yan Sun, Zhigang Zhao, Yahui Li, Wei Liu**[13]**

Traditional searchable encryption schemes construct document vectors based on the term frequency-inverse document frequency (TF-IDF) model. Such vectors are not only high-dimensional and sparse but also ignore the semantic information of the documents. The Sentence Bidirectional Encoder Representations from Transformers (SBERT) model can be used to train vectors containing document semantic information to realize semantic-aware multi-keyword search. In this paper, we propose a privacy-preserving searchable encryption scheme based on the SBERT model. The SBERT model is used to train vectors containing the semantic information of documents, and these document vectors are then used as input to the Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) clustering algorithm. The HDBSCAN algorithm generates a soft cluster membership vector for each document. We treat each cluster as a topic, and the vector represents the probability that the document belongs to each topic. According to the clustering process in the schemes, the topic-term frequency-inverse topic frequency (TTF-ITF) model is proposed to generate keyword topic vectors. Through the SBERT model, searchable encryption scheme can achieve more precise semantic-aware keyword search. At the same time, the special index tree is used to improve search efficiency. The experimental results on real datasets prove the effectiveness of our scheme.

14. Cross-Lingual Multi-Keyword Ranked Search over Encrypted Cloud Data

Authors: Cheng Liu, Ximeng Liu, Muhammad Khurram Khan**[14]**

Multi-user multi-keyword ranked search scheme in arbitrary language is a novel multi-keyword rank searchable encryption (MRSE) framework based on Paillier Cryptosystem with Threshold Decryption (PCTD). Compared to previous MRSE schemes constructed based on the k-nearest neighbor searcha-ble encryption (KNN-SE) algorithm, it can mitigate some draw-backs and achieve better performance in terms of functionality and efficiency. Additionally, it does not require a predefined keyword set and support keywords in arbitrary languages. However, due to the pattern of exact matching of keywords in the new MRSE scheme, multilingual search is limited to each language and cannot be searched across languages. In this pa-per, we propose a cross-lingual multi-keyword rank search (CLRSE) scheme which eliminates the barrier of languages and achieves semantic extension with using the Open Multilingual Wordnet. Our CLRSE scheme also realizes intelligent and per-sonalized search through flexible keyword and language prefer-ence settings. We evaluate the performance of our scheme in terms of security, functionality, precision and efficiency, via extensive experiments.

15. Multi-keyword Ranked Search with Fine-grained Access Control over Encrypted Cloud Data

Authors: Jingyu Lei, Jiao Mo**[15]**

With the development of cloud computing, people have become accustomed to using these cheap and convenient services provided by the cloud server for data storage and processing. For protecting data privacy from leaking information to unauthorized users, probably including the cloud service provider (CSP), the data must be stored in an encrypted form before outsourcing, which leads to some traditional data services being unavailable, for example, data search. Therefore, to design a

privacy-preserving data search scheme is an extremely challenging problem. Meanwhile in order to keep data from using (and even searching) by unauthorized users, it also needs to make sure that the search scheme could achieve fine-grained access control on data users, according to the access policies made by the date owner (DO). In this paper, considering both the control of user's access authority and the privacy protection in data search, we have designed a multi-keyword searchable scheme with access control over encrypted cloud data. By using the technology of inverted indexes, we can rapidly filter out data files that users own authority to access, while methods of coordinate matching and secure inner product computation have also been introduced to realize the privacy-preserving multi-keyword search. Through analysis of privacy and efficiency, we prove that our scheme achieves the requirements of privacy protection under two different adversary models, and that it has relatively high efficiency from the perspective of time complexity.

Table 1 Literature Survey

S.No	Authors	Title	Year	Journal/ Source	Methodology	Merits	Demerits
1	Oded Goldreich, Rafail Ostrovsky	Software Protection on Oblivious RAM	1996	ACM	Theoretical treatment of software protection using oblivious RAM simulation	Establishes a theoretical foundation for secure software execution.	Focuses primarily on theoretical analysis; lacks practical applicability or experimental evaluation.
2	Craig Gentry	A Fully Homomorphic Encryption Scheme	2009	Stanford University	Development of fully homomorphic encryption using ideal lattices	Enables computation on encrypted data, enhancing privacy and security for remote processing.	Computational overhead and implementation complexity in real-world scenarios.
3	Seny Kamara, Kristin Lauter	Cryptographic Cloud Storage	2010	Springer	Architectural analysis combining cryptographic primitives for secure cloud storage	High-level exploration of benefits and challenges in cryptographic cloud storage.	Absence of detailed technical implementations or experimental results.
4	J. Athena, V. Sumathy	Security Challenges for Public Cloud	2017	SCIRP	Survey of cryptographic techniques and their impact on cloud security	Highlights time and key size optimizations in cryptographic techniques like RSA, ECC.	Lacks real-world implementation insights and comparative performance analysis across techniques.
5	Sunitha Buchade	A Study on Searchable Encryption Schemes	2019	Research Gate	Analysis of various resilient keyword search schemes in cloud SE	Explores SE schemes for efficient encrypted data utilization with privacy checks.	Lacks implementation details and practical applications.
6	Jiayi Li, Jianfeng Ma, Yinbin Miao, Ruikang Yang, Ximeng Liu, Kim-Kwang Raymond Choo	Multi-keyword Ranked Search	2020	IEEE Xplore	Proposed MRSF scheme combining TF-IDF and secure kNN method for privacy-preserving search	Provides fine-grained access control and higher search accuracy with formal security analysis.	Focuses primarily on keyword ranking; does not explore other data types or user behaviors.

7	Jiawei Wan, Shijie Jia, Limin Liu, Yang Zhang	Dynamic Update	2021	IEEE Xplore	Public integrity verification with a Merkle Hash Tree (MHT-list) data structure	Efficient dynamic update and integrity verification with formal security proof and extensive testing.	Complexity in implementing and managing the MHT-list structure for large-scale applications.
8	Benneth Uzoma, Bonaventure Okhuoya	Cloud Computing	2022	Research Gate	Overview of cloud computing architecture, traits, types, and challenges	Comprehensive review of cloud computing applications and benefits in various contexts.	Limited focus on security-specific challenges and solutions in cloud computing.
9	Kai Chen, Zhongrui Lin, Jian Wan, Lei Xu, Chungen Xu	Multi-owner Secure Encrypted Search Using Searching Adversarial Networks	2019	CANS	Uses searching adversarial networks and probabilistic balanced trees for efficient, privacy-preserving encrypted search in multi-owner settings.	High query precision (>95%), strong privacy, low overhead, and support for dynamic updates	Increased system complexity and limited evaluation on large-scale real-world data.
10	Baocheng Zhang, Rongxing Lu, Kui Ren, Zhiguang Qin, Xuemin (Sherman) Shen	Generic Multi-keyword Ranked Search on Encrypted Cloud Data	2014	IEEE	Introduces a generic multi-keyword ranked search scheme leveraging the OXT protocol to enhance efficiency and security in encrypted cloud data search.	Supports ranked multi-keyword search without relying on Order-Preserving Encryption, ensuring better privacy and reduced communication overhead.	May involve increased computational complexity and requires careful implementation to maintain performance.
11	Yue Tong, Ximeng Liu, Jinbo Xiong, Robert H. Deng	Secure Phrase Search for Intelligent Processing of Encrypted Data in Cloud-Based IoT	2019	IEEE Internet of Things Journal	Proposes P3, a privacy-preserving phrase search scheme using homomorphic encryption and bilinear maps to determine keyword positions over encrypted data.	Enables accurate phrase search over encrypted IoT data with strong privacy guarantees and moderate overhead.	Relies on complex cryptographic operations, which may impact performance in resource-constrained environments.

12	Xiaohua Dai, Zhitao Guan, Yushu Zhang, Zhigang Zhou	Multi-keyword Ranked Search with Access Control for Multiple Data Owners in the Cloud	2019	Future Generation Computer Systems	Proposes MOMRS-AC, extending KNN-based search to a multi-owner model with fine-grained access control and result verification without relying on a trusted third party.	Supports secure, efficient multi-keyword ranked search with decentralized access control and result verification, eliminating the need for a trusted third party.	Potentially increased computational overhead due to complex access control and verification mechanisms.
13	Yan Sun, Zhigang Zhao, Yahui Li, Wei Liu	Semantic-Based Multi-Keyword Ranked Search Schemes over Encrypted Cloud Data	2022	Research Gate	Utilizes SBERT for semantic vector generation, HDBSCAN for clustering, and TTF-ITF for keyword-topic vectors to enhance search precision.	Achieves more accurate semantic-aware keyword search with improved efficiency using a specialized index tree.	Incorporates complex models like SBERT and HDBSCAN, potentially increasing computational overhead.
14	Cheng Liu, Ximeng Liu, Muhammad Khurram Khan	Cross-Lingual Multi-Keyword Ranked Search over Encrypted Cloud Data	2020	IEEE	Proposes CLRSE, a scheme leveraging Open Multilingual WordNet and Paillier Cryptosystem with Threshold Decryption to enable cross-lingual multi-keyword ranked search over encrypted data.	Supports multilingual search without predefined keyword sets, enhancing functionality and efficiency over previous MRSE schemes.	Exact keyword matching may limit semantic search capabilities across languages.
15	Jingyu Lei, Jiao Mo	Multi-keyword Ranked Search with Fine-grained Access Control over Encrypted Cloud Data	2016	ICMMCT	Utilizes inverted indexes, coordinate matching, and secure inner product computation to enable privacy-preserving multi-keyword search with fine-grained access control over encrypted cloud data.	Achieves efficient and secure multi-keyword search with fine-grained access control without relying on a trusted third party.	Potential computational overhead due to complex cryptographic operations and access control mechanisms.

CHAPTER 3

DESIGN AND METHODOLOGY

3.1 BLOCK DIAGRAM OF SEARCH SCHEME

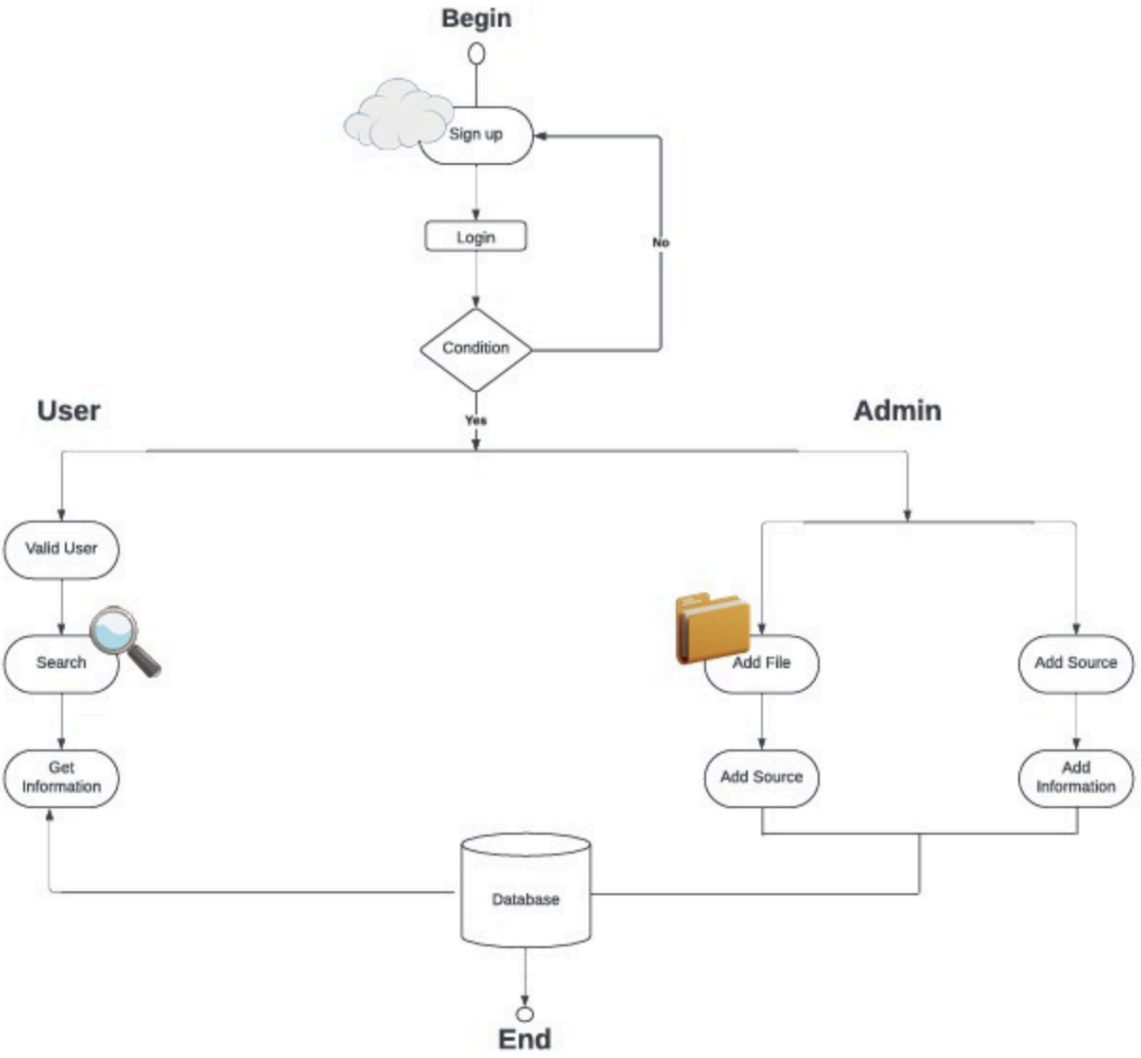


Fig.3.1 Block Diagram of Search Scheme

This block diagram represents the workflow of a system with Searchable Encryption, detailing the interactions between users, admins, and the underlying database. Here's a breakdown of its components and flow:

1. Start:

    1. The process begins with the sign-up or login functionality.
    2. A condition check ensures that only authenticated users or admins can proceed.

2. User Workflow:

    1. Valid User Check: After authentication, the user is validated.
    2. Search: The user performs a search operation on encrypted data.
    3. Retrieve Information: Based on the search query, the system retrieves relevant data from the encrypted database and presents it securely.

3. Admin Workflow:

    1. Add File: Admins can upload files to the system.
    2. Add Source: Admins can manage data sources related to the files.
    3. Add Information: Additional metadata or related information can also be provided by the admin.
    4. These actions help in building and maintaining the searchable encrypted database.

4. Database:

    1. The central component where encrypted data is stored.
    2. Facilitates secure storage and retrieval for both users and admins.

5. End:

    1. The workflow concludes after the user retrieves the desired information or the admin updates the database.

3.2 ARCHITECTURE OF SEARCH SCHEME

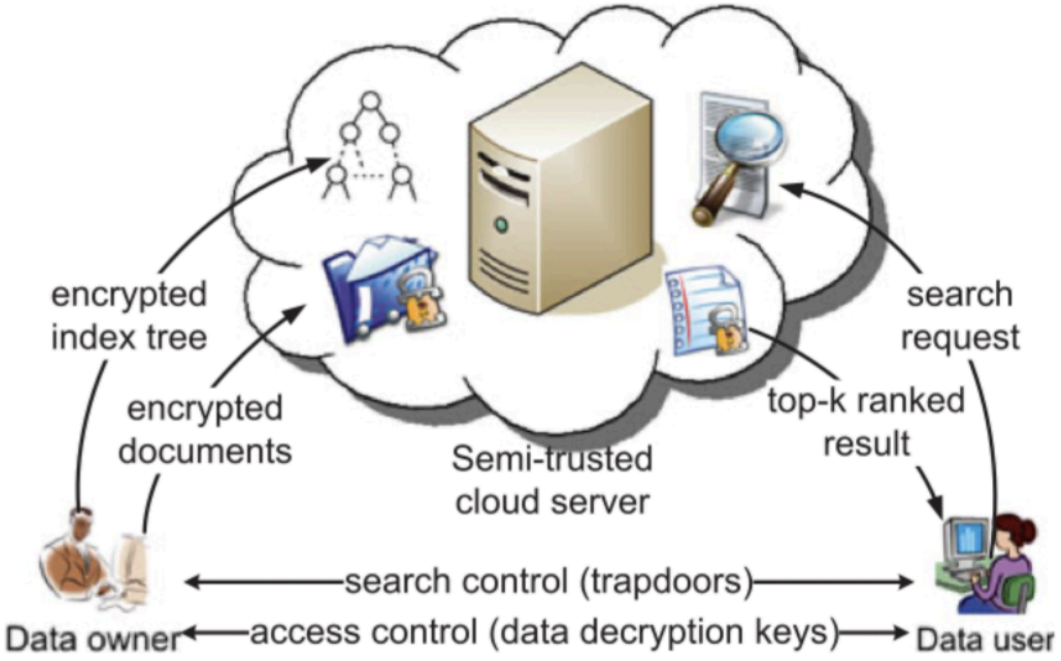


Fig. 3.2 Architecture of Search Scheme

This architecture diagram outlines a Searchable Encryption system using a semi-trusted cloud server to store and manage encrypted data, allowing secure search functionality. Here's a detailed breakdown of the components and flow:

Key Components:

1. Data Owner:
    a. The entity responsible for creating and encrypting data (e.g., documents) before uploading them to the cloud server.
    b. Generates the encrypted documents and encrypted index tree to facilitate secure search.
    c. Manages search control (trapdoors) and access control (decryption keys) to ensure that only authorized users can access the data.
2. Semi-Trusted Cloud Server:
    a. Stores the encrypted documents and index tree securely.

b. Processes search requests from authorized users without accessing or decrypting the data directly.
c. Returns the top-k ranked results, ensuring relevance based on the user's query.

3. Data User:
   a. An authorized user who interacts with the cloud server to search for specific information.
   b. Uses trapdoors (encrypted search queries) to perform secure searches on the encrypted data.
   c. Receives the top-k ranked results, which can then be decrypted using access control keys provided by the data owner.

Workflow:

1. Data Preparation (by Data Owner):
   a. The data owner encrypts the documents and creates an encrypted index tree that maps searchable terms to the encrypted data.
   b. The encrypted data and index tree are uploaded to the cloud server.
2. Search Request (by Data User):
   a. The data user generates a trapdoor, an encrypted representation of the search query, and sends it to the cloud server.
   b. The user cannot directly access or decrypt the entire database but can perform secure searches using the trapdoor.
3. Query Processing (by Cloud Server):
   a. The semi-trusted cloud server uses the encrypted index tree to process the search request without decrypting the data.
   b. It returns the top-k ranked results, ensuring only the most relevant matches are sent back to the user.
4. Decryption and Access Control:
   a. The data user receives the encrypted results and uses the decryption keys provided by the data owner to access the actual information.

b. The data owner ensures fine-grained control over which users can decrypt and access specific data.

Key Features:

1. Search Control with Trapdoors: Enables secure querying without exposing data content to the cloud server.
2. Access Control: Maintains data confidentiality by restricting decryption keys to authorized users.
3. Top-k Ranking: Improves user experience by returning only the most relevant results.
4. Semi-Trusted Cloud Server: Ensures that even if the server is partially compromised, the data remains secure due to encryption.

This architecture effectively balances security, functionality, and performance, making it suitable for use cases like encrypted document storage, privacy-preserving search, and data sharing in sensitive domains.

3.3 FLOW DIAGRAMS OF SEARCH SCHEME

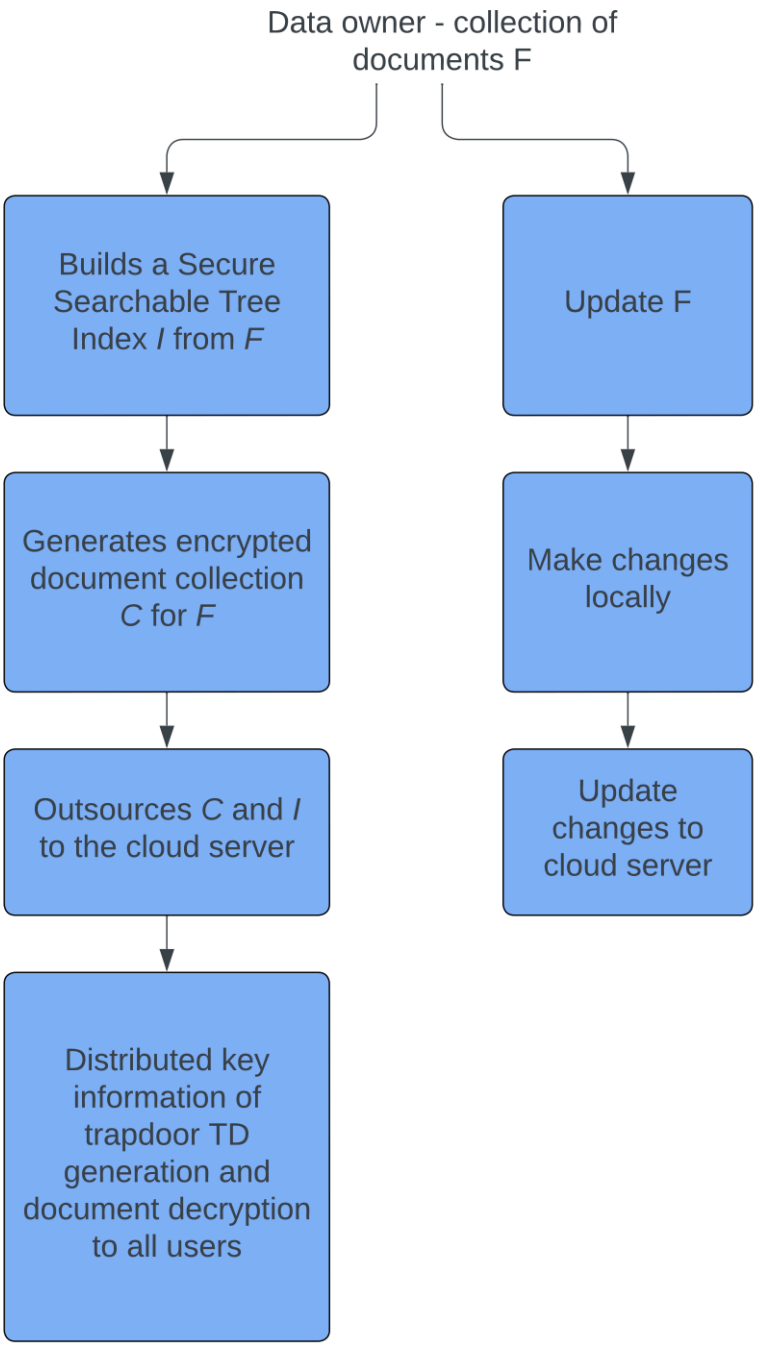


Fig. 3.3 Data Owner Flow of Search Scheme

This flowchart describes the process for a data owner managing encrypted data for secure cloud storage and searchable encryption. The data owner first builds a secure searchable tree index (I) and generates an encrypted document collection (C) from the document set (F). These are then outsourced to a cloud server. The owner updates documents

locally, synchronizes the changes to the cloud server, and distributes key information (for generating trapdoors and decrypting documents) to authorized users, ensuring secure search and controlled access. This system ensures privacy while enabling efficient and flexible data updates and searches.

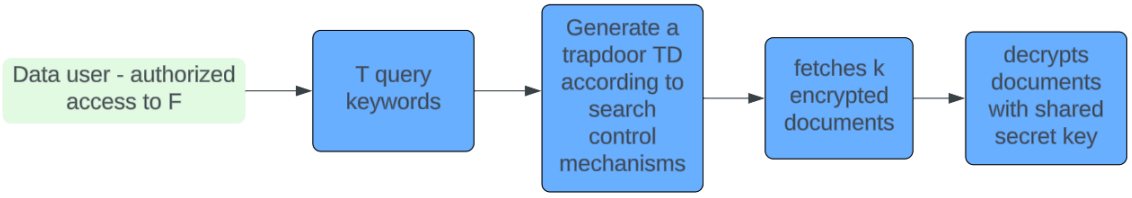


Fig. 3.4 Data User Flow of Search Scheme

This flowchart outlines the process for a data user to securely access documents stored on a cloud server. The user inputs query keywords (T) and generates a trapdoor (TD) based on search control mechanisms. The trapdoor is used to retrieve k encrypted documents from the cloud server. Finally, the user decrypts the documents using a shared secret key, ensuring secure and authorized access to the requested data.

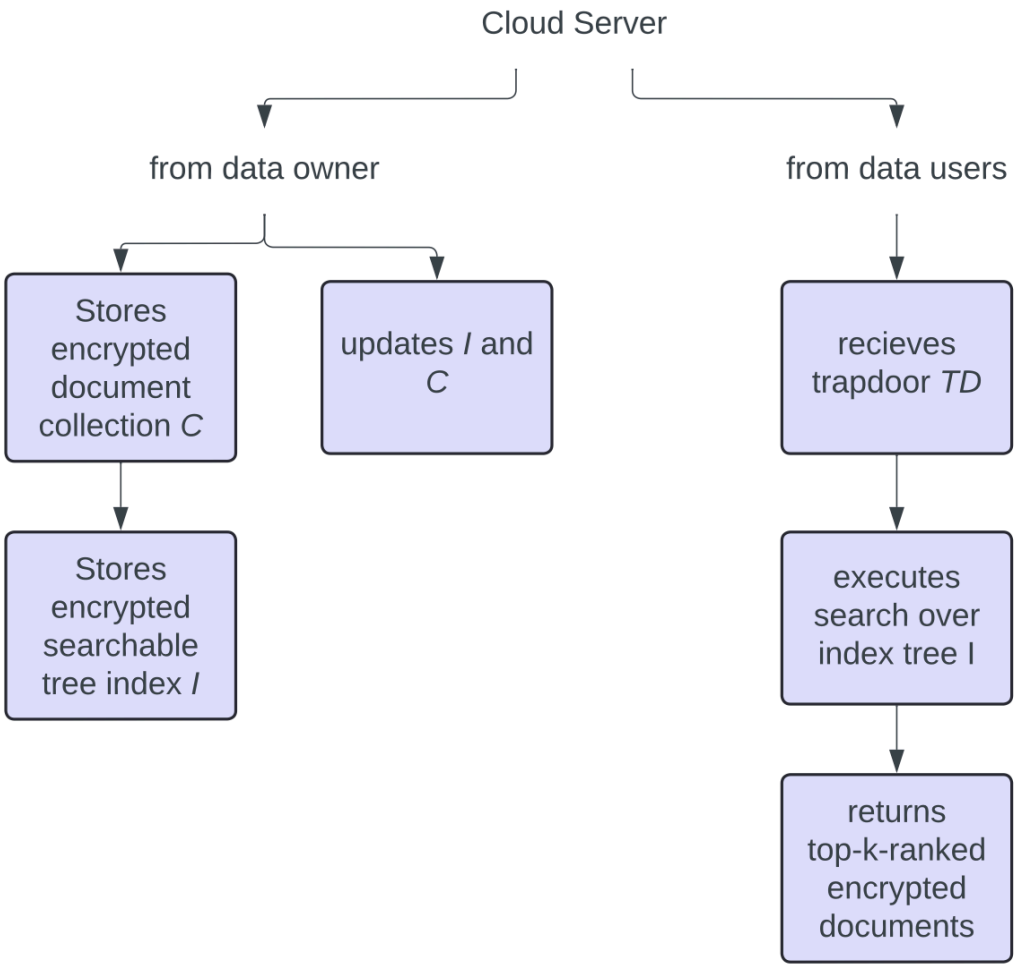


Fig. 3.5 Cloud Server Flow of Search Scheme

This diagram illustrates a searchable encryption system where a cloud server facilitates secure data management and search functionalities. The data owner uploads an encrypted document collection CCC and a searchable encrypted index tree III to the server, which can also handle updates to CCC and III. Data users interact with the server by sending a trapdoor TDTDTD (an encrypted search query). The server uses TDTDTD to search the index III securely and returns the top-k ranked encrypted documents, ensuring data confidentiality throughout the process.

3.4 UML DIAGRAMS OF SEARCH SCHEME

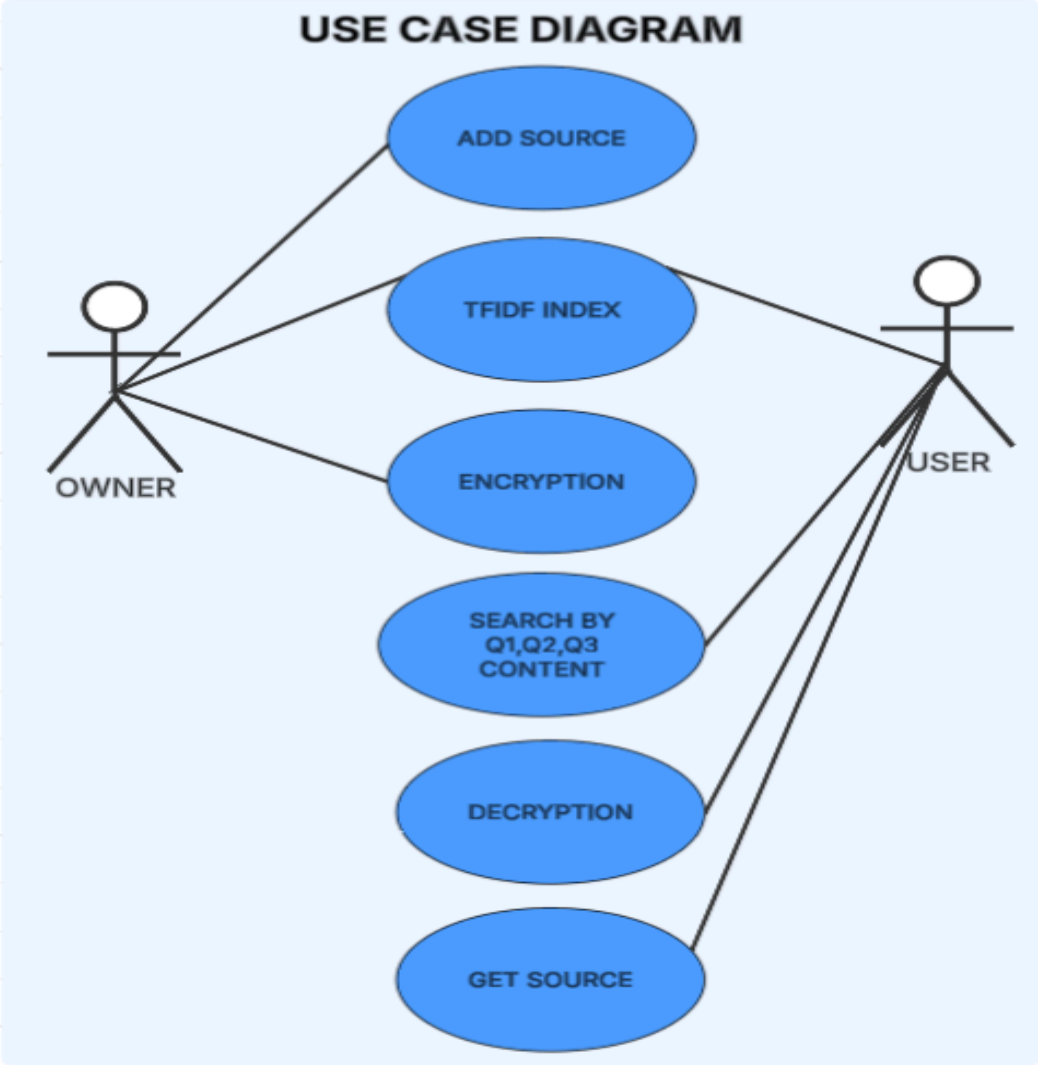


Fig. 3.6 Use Case Diagram of Search Scheme

This is a Use Case Diagram for a system enabling Encrypted Search over Cloud Data, illustrating the interaction between users and the system. It showcases how a Data Owner and a User perform tasks like document encryption, keyword indexing, and secure querying. The diagram highlights essential functionalities such as adding sources, creating TF-IDF indexes, encryption, querying by content, decryption, and retrieving results. It effectively represents the roles and actions needed for secure, privacy-preserving cloud data access.

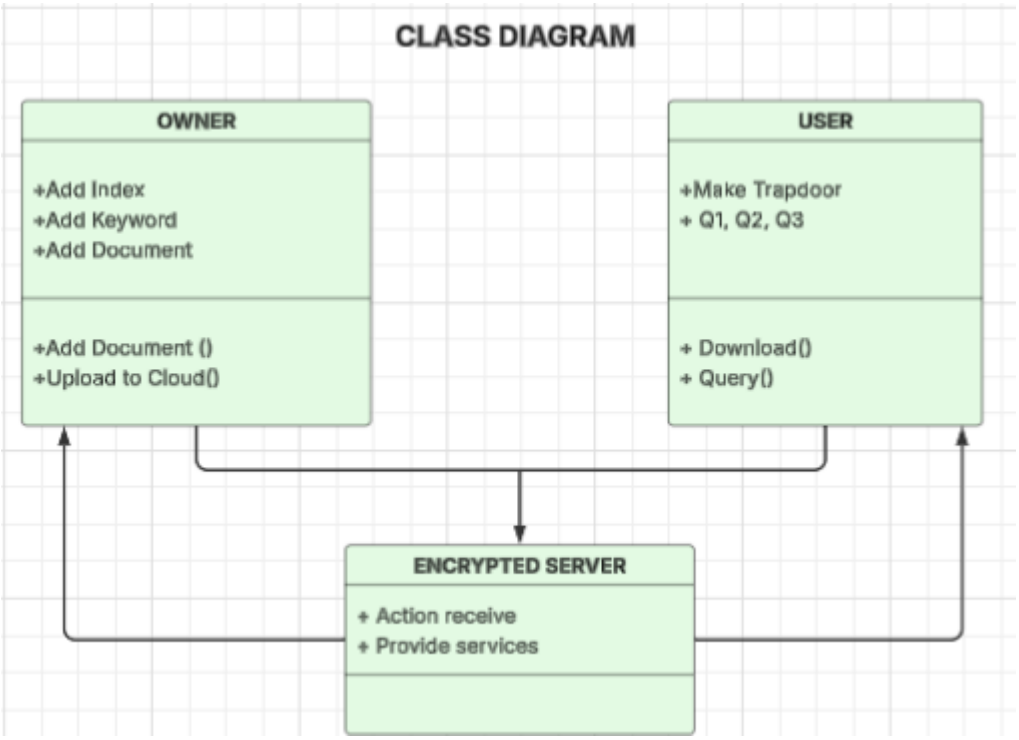


Fig. 3.7 Class Diagram of Search Scheme

This is a Class Diagram for a system involving Encrypted Search over Cloud Data, commonly used in secure cloud storage and searchable encryption projects. The Owner adds indexes, keywords, and documents, then uploads them to the Encrypted Server. The User generates a secure trapdoor (search token) and sends a query to the server. The Encrypted Server receives the trapdoor, searches the encrypted data, and processes the request. Matching encrypted results are returned to the User, who can then download the relevant documents.

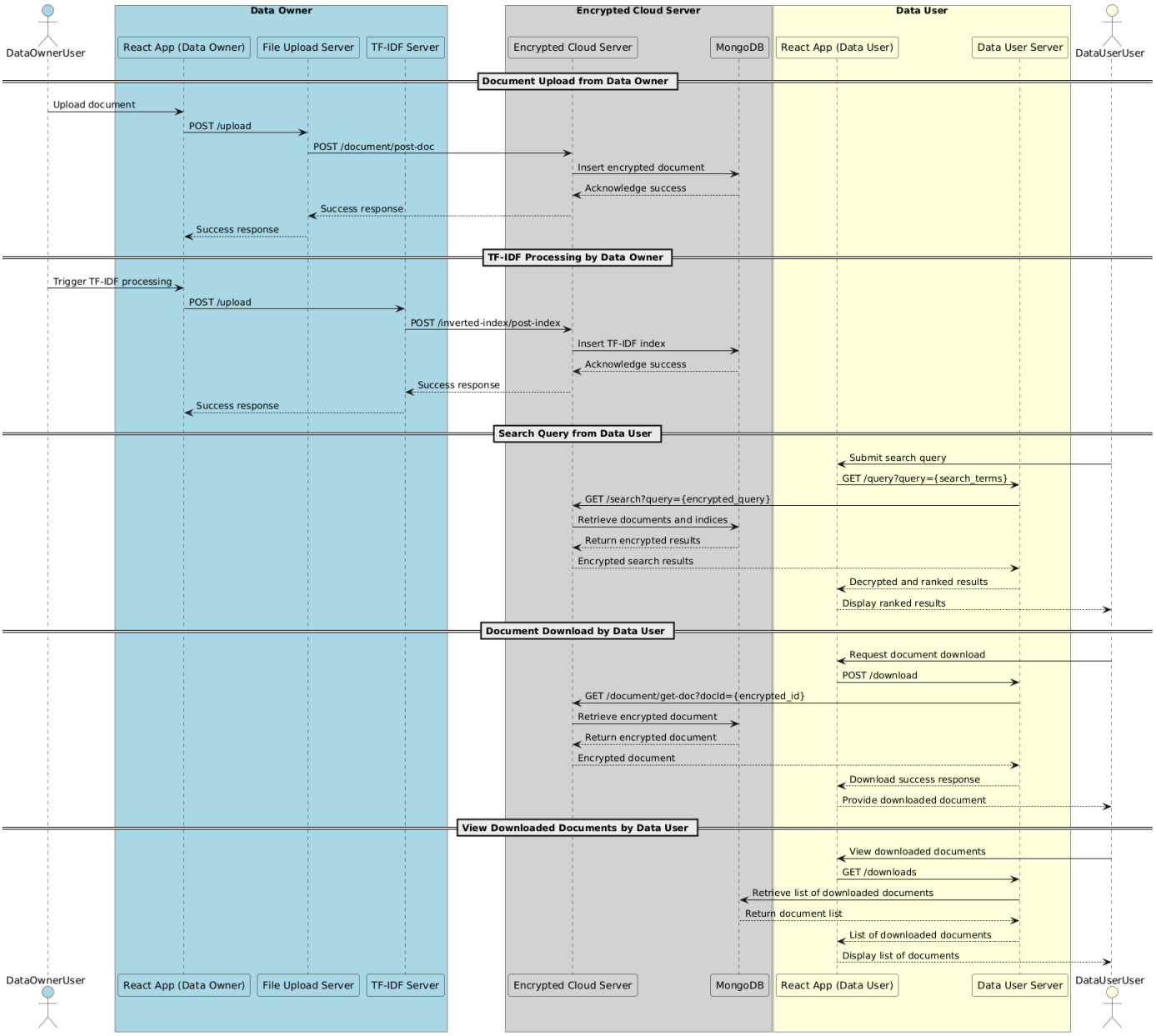


Fig. 3.8 Sequence Diagram of Search Scheme

The sequence diagram illustrates the secure document management system's interactions between the Data Owner, Encrypted Cloud Server, and Data User. It shows processes like uploading documents, performing TF-IDF analysis, searching encrypted data, and downloading documents. MongoDB operations are integrated within the Encrypted Cloud Server for storing and retrieving encrypted data. The diagram highlights secure communication and role-based workflows for both data owners and users.

CHAPTER 4

IMPLEMENTATION

4.1 TECH STACK

The development of this project involved a full-stack architecture combining modern web technologies, cryptographic libraries, and secure backend services to implement a dynamic and secure multi-keyword ranked search system over encrypted cloud data.

1. Languages Used

Java: Used for implementing cryptographic operations, including encryption of the document collection, secure index generation, and handling backend operations (especially on the cloud server side).

JavaScript: Powered the front-end applications (Data Owner and Data User interfaces) and supported Node.js-based server components.

2. Front-End

React.js: Developed the user-facing components for both Data Owner and Data User portals, offering a dynamic and responsive interface for document upload, keyword input, and result visualization.

HTML5 & CSS3: Provided structure and styling for all web interfaces, ensuring accessibility and responsiveness.

3. Back-End

Node.js: Served as the primary backend runtime for handling API requests, processing user inputs, and interfacing with MongoDB.

Express.js: A minimalist web framework used to build RESTful APIs for Data Owner and Data User applications.

Java (Spring Boot): Deployed on the cloud server to perform secure index search operations, manage encrypted data, and support dynamic updates.

4. **Database**

MongoDB: A NoSQL database used to store the encrypted index and encrypted documents. Its flexible document schema allowed seamless updates and efficient keyword-based querying on the server side.

5. **Security & Encryption**

Post-Quantum Cryptography: The project incorporates ML-DSA (Dilithium) from the noble-post-quantum library, ensuring secure signing and verification resistant to quantum attacks.

TF-IDF Model: Used to create a ranked searchable index over the encrypted document collection. The score-based indexing supports relevance-based retrieval.

Trapdoor Mechanism: Enables privacy-preserving keyword search over the encrypted index without revealing sensitive information to the cloud server.

6. **Development Tools**

Git & GitHub: For version control and collaboration.

Visual Studio Code: The primary code editor for front-end and backend development.

Postman: Used to test API endpoints and simulate user behavior.

4.2 FEATURES

1.Data Owner

This component implements the data owner's side of a secure multi-keyword ranked search system over encrypted cloud data. As the data owner, this

system enables you to securely store and manage your documents in the cloud while maintaining complete control over your data's privacy and security. The system handles document encryption, indexing, and secure transmission to the cloud server, ensuring that your sensitive information remains protected even when stored in a third-party cloud environment.

System Architecture

The data owner component consists of three main parts:

1. File Upload Server - Handles document encryption and secure upload to the cloud
2. TF-IDF Server - Creates searchable indices for encrypted documents
3. React Frontend - Provides an intuitive interface for document management

Components

1. File Upload Server (file-upload-server.js)

1. Port: 5001
2. Functionality:
    a. Handles file uploads from the frontend
    b. Encrypts documents using AES encryption
    c. Stores documents locally
    d. Sends encrypted documents to the cloud server
3. API Endpoints:
    a. POST /upload - Upload and encrypt a single file
    b. GET /documents - Retrieve all uploaded documents

2. TF-IDF Server (tfidf-server.js)

1. Port: 5000
2. Functionality:

a. Creates TF-IDF (Term Frequency-Inverse Document Frequency) models
b. Processes documents for search indexing
c. Sends encrypted indices to the cloud server

3. API Endpoints:
   a. GET /tfidf - Get current TF-IDF data
   b. POST /add-document - Add a new document to the index
   c. POST /upload - Process and upload all documents

3. Encryption Module (encrypt.js)

1. Uses AES encryption with CBC mode
2. Implements secure key management
3. Provides encryption functions for both documents and indices

Security Features

1. AES-256 encryption for all documents
2. Secure key management
3. Encrypted search indices
4. Secure document transmission
5. Privacy-preserving search capabilities

Security Considerations

1. All documents are encrypted before being sent to the cloud
2. Search indices are encrypted to prevent information leakage
3. Secure key management practices are implemented
4. Regular security audits are recommended

2. Data User

This project implements the client-side component of a secure multi-keyword ranked search system over encrypted cloud data. As a data

user, this system allows you to securely search through encrypted documents stored in the cloud without compromising data privacy. The data owner (separate server) handles the encryption and storage of documents, while this client enables secure search operations and document retrieval.

System Architecture

The data user system consists of two main components:

1. Data User Server (Port 5002)
    a. Handles secure search operations
    b. Communicates with the Search Server (port 8080) for encrypted index retrieval
    c. Manages document downloads and local storage
    d. Provides API endpoints for search and document management
    e. Implements encryption/decryption for secure communication
2. React Frontend
    a. Provides user interface for search operations
    b. Displays search results and document previews
    c. Manages document downloads
    d. Handles user authentication and session management

Technical Implementation

Encryption/Decryption Module

The system uses AES-256-CBC encryption with:

1. 32-byte secret key (shared with data owner)
2. 16-byte initialization vector (IV)
3. PKCS7 padding

Key files:

1. encrypt.js: Handles encryption using CryptoJS

2. decrypt.js: Handles decryption using Node.js crypto module

Dependencies

1. Node.js
2. Express.js
3. CryptoJS
4. Axios
5. TF-IDF Search library

Security Features

1. End-to-end encryption of search queries using shared symmetric key
2. Trapdoor generation for secure query transmission
3. TF-IDF based document ranking on encrypted data
4. Secure document retrieval and decryption
5. Local document storage management
6. Privacy-preserving search results

Use Cases

1. Secure document search in enterprise environments
2. Privacy-preserving research document retrieval
3. Secure access to encrypted business documents
4. Confidential document management systems

3. Encrypted Cloud Server

Welcome to the Encrypted Server repository! This project is the cloud server component of a secure document management system. It is built as a Spring Boot application and provides robust backend services for handling document uploads, retrievals, and processing using state-of-the-art methods. Below, you'll find a detailed explanation of the technical aspects, functionalities, and how the system operates.

Features

1. Secure Document Upload and Retrieval: Facilitates encrypted document uploads and retrievals between the data owner and data user.
2. Cloud Integration: Hosted on a cloud server, leveraging MongoDB for database management.
3. TF-IDF Model Integration: Employs a TF-IDF (Term Frequency-Inverse Document Frequency) model for document analysis and processing.
4. Frontend Connectivity: Connected to a React-based frontend application for user interaction and visualization.
5. Scalable Architecture: Designed to handle multiple users and large datasets efficiently.

Encryption and Zero-Knowledge Principles

This system is built with end-to-end encryption and adheres to zero-knowledge principles, ensuring the highest levels of security and privacy. Here's how it works:

1. End-to-End Encryption:
    a. All documents, queries, and responses are encrypted on the client side before being transmitted to the server.
    b. The server only processes and stores encrypted data, ensuring that no sensitive content is ever exposed.
2. Zero-Knowledge Principles:
    a. The server has no knowledge of the document content, query keywords, or TF-IDF model results in their unencrypted form.
    b. All computations (e.g., TF-IDF analysis) are performed on encrypted data.
    c. This ensures that even if the server is compromised, sensitive information remains secure.

3. By combining encryption and zero-knowledge techniques, this system provides a robust framework for secure and private document management.

Technical Architecture

1. 1. Spring Boot Backend
2. The backend is developed using Spring Boot, a powerful Java framework for building scalable and production-ready applications. It serves as the core of this server-side application, providing the following features:
3. RESTful APIs: Exposes endpoints for document upload, retrieval, and processing.
4. Security: Implements robust authentication and authorization mechanisms to ensure data integrity and confidentiality.
5. Integration with MongoDB: Seamlessly connects to the MongoDB cloud database for storing and retrieving document metadata and content.
6. 2. Database: MongoDB
7. Cloud-hosted MongoDB: The application uses a MongoDB database hosted in the cloud for storing:
    a. Document metadata.
    b. User information.
    c. TF-IDF model results.
8. High Performance: MongoDB's schema-less and scalable architecture ensures efficient handling of large datasets.
9. 3. Frontend: React Application
10. React.js: The frontend is built using React, a popular JavaScript library for building user interfaces. It connects to the backend server to:
    a. Display uploaded documents.
    b. Facilitate user interactions for uploading and retrieving documents.

c. Present TF-IDF analysis results in a user-friendly manner.

11. 4. TF-IDF Model
12. The backend integrates a TF-IDF (Term Frequency-Inverse Document Frequency) model for processing and analyzing document content.
13. This model is particularly useful for:
    a. Text analysis.
    b. Keyword extraction.
    c. Content-based document retrieval.

System Workflow

1. Document Upload:
    a. The data owner uploads a document via the React frontend.
    b. The document is encrypted on the client side and securely transmitted to the Spring Boot backend.
    c. Metadata and content are stored in the MongoDB database in encrypted form.
2. Document Retrieval:
    a. The data user requests a document through the frontend.
    b. The backend fetches the encrypted document from MongoDB and sends it to the user.
    c. Encrypted transmission ensures data security.
3. TF-IDF Model Processing:
    a. Documents are analyzed using the TF-IDF model on encrypted data.
    b. The processed results are also stored and displayed securely.

Prerequisites

To run this project, you need:

1. Java: Version 11 or higher.
2. Spring Boot: Pre-configured in the project.

3. MongoDB: A cloud-hosted MongoDB instance.
4. Node.js and NPM: For running the React frontend.
5. Cloud Hosting Provider: The backend server should be deployed on a cloud server (e.g., AWS, GCP, Azure).

4.3 IMPLEMENTATION DETAILS

1. Encryption and Decryption

Encryption

1. Algorithm: AES-256 in CBC mode.
2. Implementation:
    a. The encryption module (encrypt.js) encrypts documents and indices before uploading them to the cloud server.
    b. Each document and its associated TF-IDF index are encrypted using a securely managed key.

```
const CryptoJS = require("crypto-js");
const SECRET_KEY =
CryptoJS.enc.Utf8.parse("1234567890abcdef1234567890abcdef");
// 32 bytes
const IV = CryptoJS.enc.Utf8.parse("0000000000000000");
// 16 bytes
const encrypt = (text) => {
  return CryptoJS.AES.encrypt(text, SECRET_KEY, {
    iv: IV,
    mode: CryptoJS.mode.CBC,
    padding: CryptoJS.pad.Pkcs7,
  }).toString();
};
module.exports = { encrypt };
```

Decryption

1. Algorithm: AES-256 in CBC mode.
2. Implementation:
    a. The decryption module complements the encryption process, enabling secure retrieval and usage of documents and indices.

```
const CryptoJS = require("crypto-js");
const SECRET_KEY =
CryptoJS.enc.Utf8.parse("1234567890abcdef1234567890abcdef");
// Same key as used in encryption
const IV = CryptoJS.enc.Utf8.parse("0000000000000000");
 // Same IV as used in encryption
const decrypt = (cipherText) => {
  const bytes = CryptoJS.AES.decrypt(cipherText, SECRET_KEY, {
    iv: IV,
    mode: CryptoJS.mode.CBC,
    padding: CryptoJS.pad.Pkcs7,
  });
  return bytes.toString(CryptoJS.enc.Utf8);
};
module.exports = { decrypt };
```

2. TF-IDF Model

The TF-IDF (Term Frequency-Inverse Document Frequency) model enables ranked keyword-based search. It calculates the relevance of keywords in each document and stores this as an encrypted index.

Implementation

1. The tfidf-server.js processes documents to compute the TF-IDF indices.

2. These indices are encrypted and uploaded to the cloud server.

```
const TfIdf = require("tf-idf-search");
const tf_idf = new TfIdf();
// Add document to the TF-IDF model
tf_idf.addDocumentFromString(document);
// Upload encrypted index to server
const postIndexResponse = await axios.post(
  "http://cloud-server-url/inverted-index/post-index",
  { index: encrypt(JSON.stringify(tf_idf)) }
);
```

CHAPTER 5

TESTING

Table 2: Test Cases

S.No	Test Case	Input	Expected Output	Obtain Output	Result
1	Encrypt Document	"Confidential data", AES key, IV	Encrypted string ≠ input	Encrypted string (e.g., U2FsdGVkX1+)	Pass
2	Decrypt Document	Encrypted string, AES key, IV	“Confidential data”	Garbled text	Fail
3	TF-IDF Index Creation	["Quick fox", "Lazy dog"]	Scores for terms	Scores for terms	Pass
4	React Upload API	"example.txt", "Sample content"	Success message	Error: File not found	Fail
5	Execute Search Query	"keyword1 keyword2"	Ranked results	Ranked results	Pass
6	Download Document	"example.txt", encrypted content	File with decrypted content	File saved locally	Pass
7	Retrieve by ID (API)	Valid document ID	Metadata and content	Metadata and content	Pass
8	Upload (Backend API)	Valid document payload	Document ID returned	Null response	Fail
9	Fetch TF-IDF Index	GET request	Inverted index JSON	Inverted index JSON	Pass
10	Spring Boot Startup	Start application	No exceptions	No exceptions	Pass

This table summarizes the testing outcomes for the encrypted document search system. It covers functionalities like encryption/decryption, TF-IDF index creation, file upload/download APIs, search query execution, and server startup validation. Each test lists input, expected output, actual output, and whether it passed or failed. The results help identify modules that require further debugging or improvements.

CHAPTER 6

RESULTS

Github repository -

https://github.com/stars/s-prak/lists/major-project-final

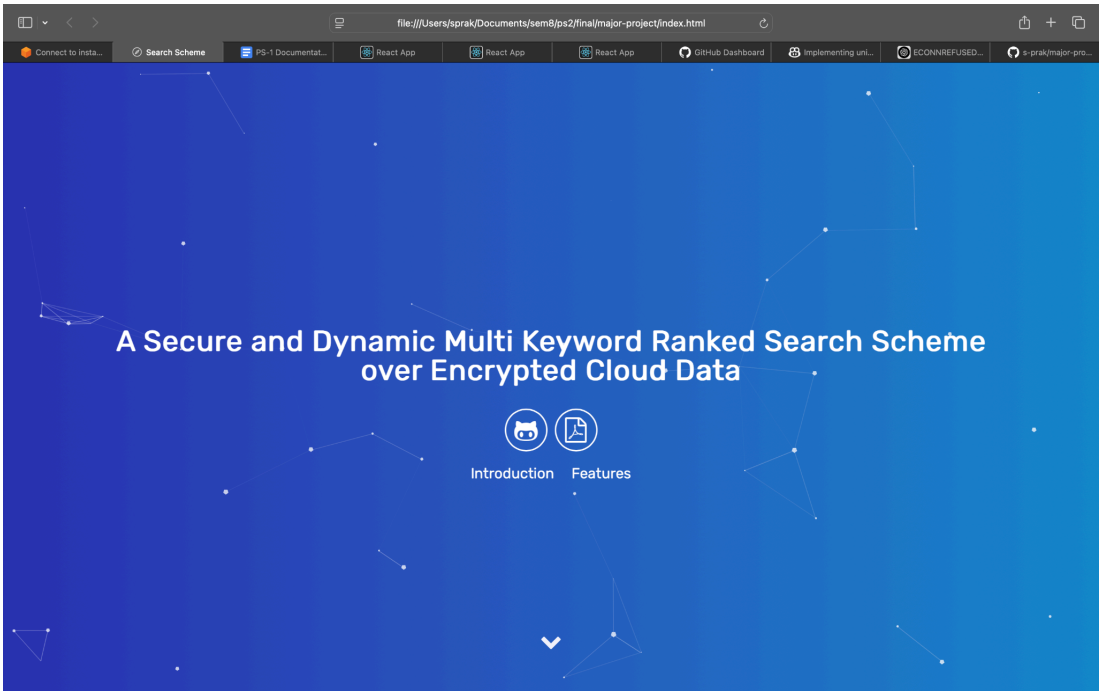


Fig 6.1 Project Cover - Navigate to github page and document

This is the project cover, it displays the title of the project. It also gives links to the github page and the documentation. Apart from this it also redirects to the Introduction and Features section of this project cover.

Fig. 6.2 Project Overview

This section gives a brief overview of the project. It talks about which context this project is useful and then also specifies some of the applications of the project. An architectural layout of the project is also given.

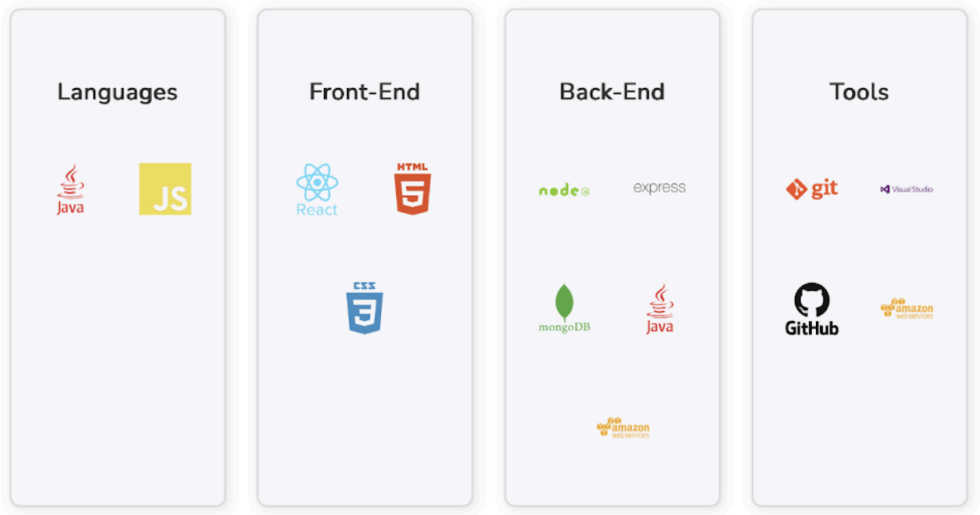


Fig. 6.3 Tech Stack

This figure shows the Technology stack used to build the project. Java and Javascript are the programming languages used. The frontend used react, html and css. For the backend a wide range of frameworks, languages, databases and runtime environments were used. These include NodeJS, express for server, springboot, java, AWS EC2 instance, and mongoDB. Apart from these the various tools used for development are git, github, VS code, and AWS.

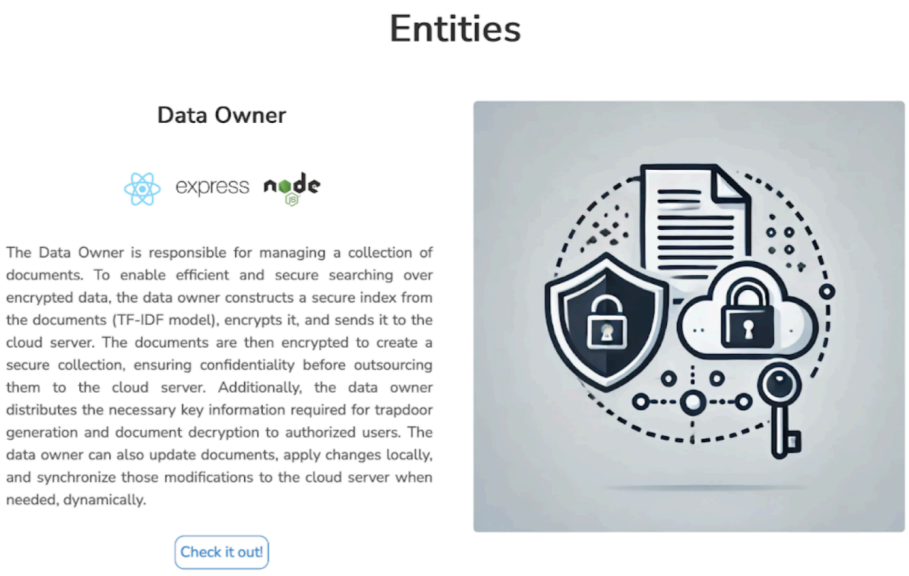


Fig. 6.4 Entities - Navigate to Data Owner

This figure shows the Data Owner entity, and helps us to navigate to the Data Owner. It also includes symbols of the tech stack the Data Owner comprises.

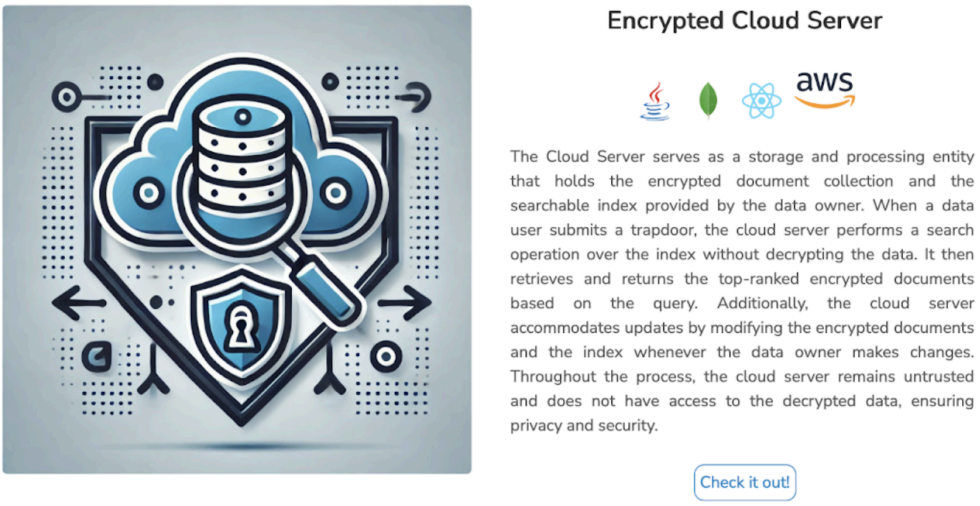


Fig. 6.5 Entities - Navigate to Encrypted Cloud Server

This figure shows the Encrypted Cloud Server entity, and helps us to navigate to the Encrypted Cloud Server. It also includes symbols of the tech stack the Encrypted Cloud Server comprises.

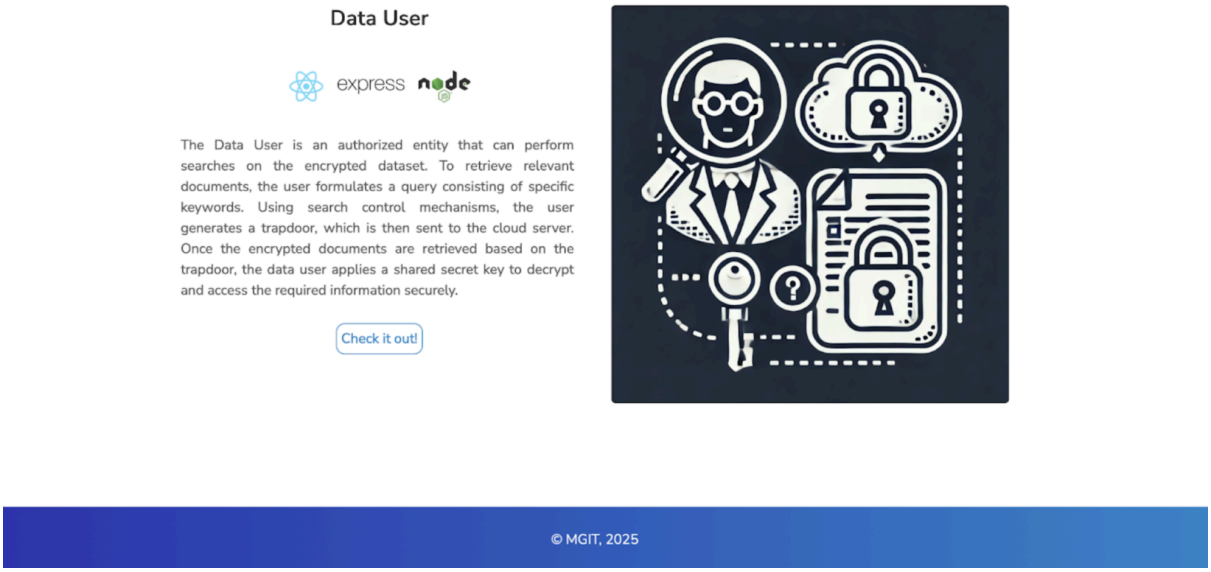


Fig. 6.6 Entities - Navigate to Data User

This figure shows the Data User entity, and helps us to navigate to the Data User. It also includes symbols of the tech stack the Data User comprises.

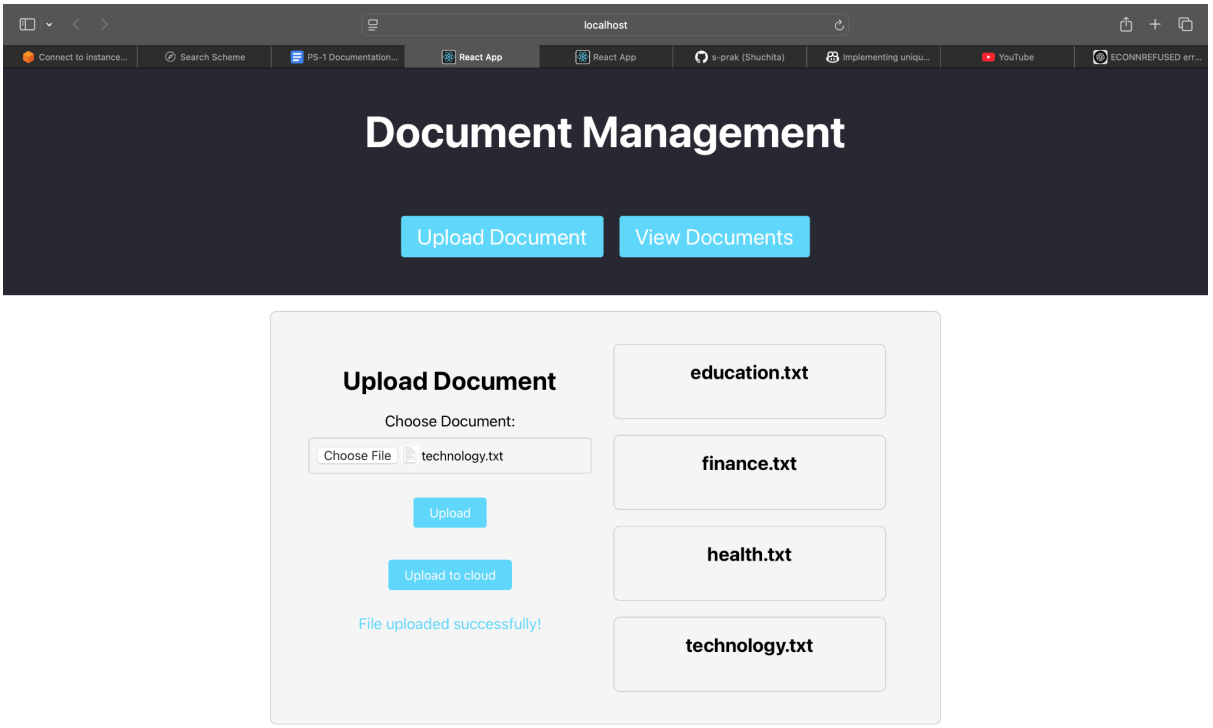


Fig. 6.7 Data Owner - Upload Documents

This figure shows the page of the Data Owner, it contains a navigation bar, through which we can either upload documents or view documents. In the Upload Document page, we can choose a file from the disk and upload it, we can also upload multiple files. After uploading the multiple files, we can upload these files to the Cloud Server. On the left side, the messages of either success or failure are logged. The Documents uploaded in this session can be seen on the right side.

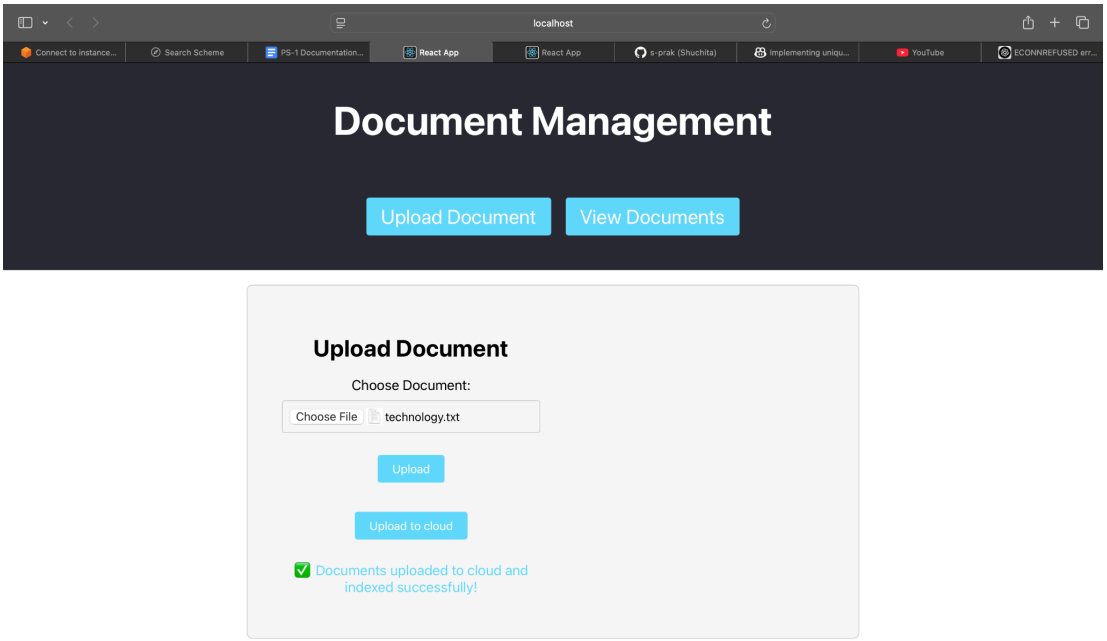


Fig. 6.8 Data Owner - Upload documents to cloud

This figure shows a use case, where the Data Owner has been successful to upload documents to the cloud. This implies good connectivity to the cloud.

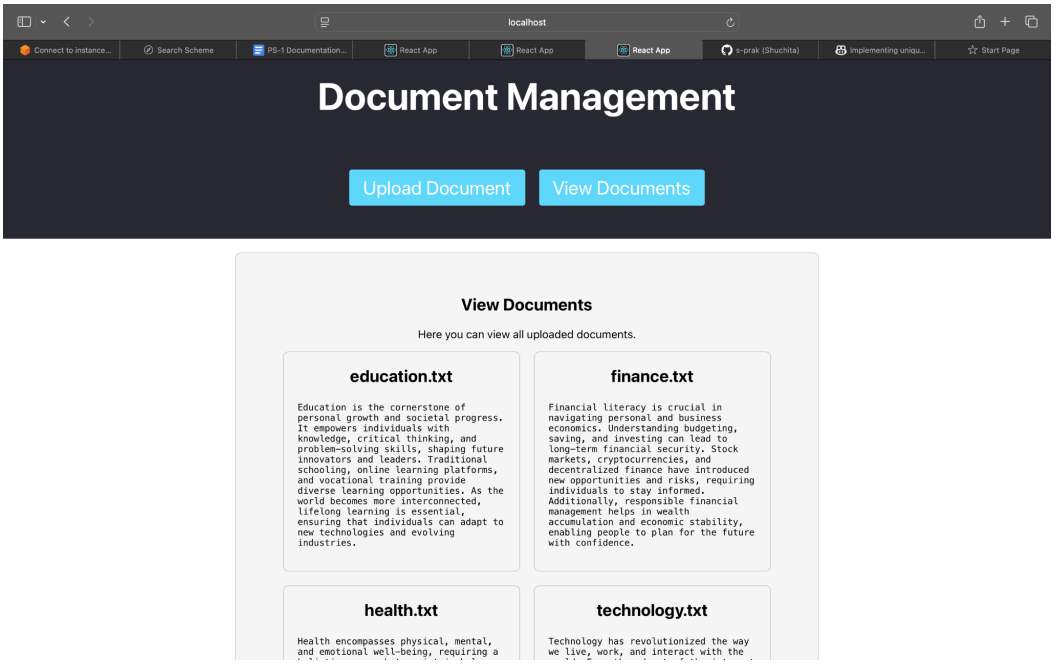


Fig. 6.9 Data Owner - View Documents

This is the other page of the Data Owner, which is used to view the documents. Here the documents that are available locally with the Data Owner are displayed.

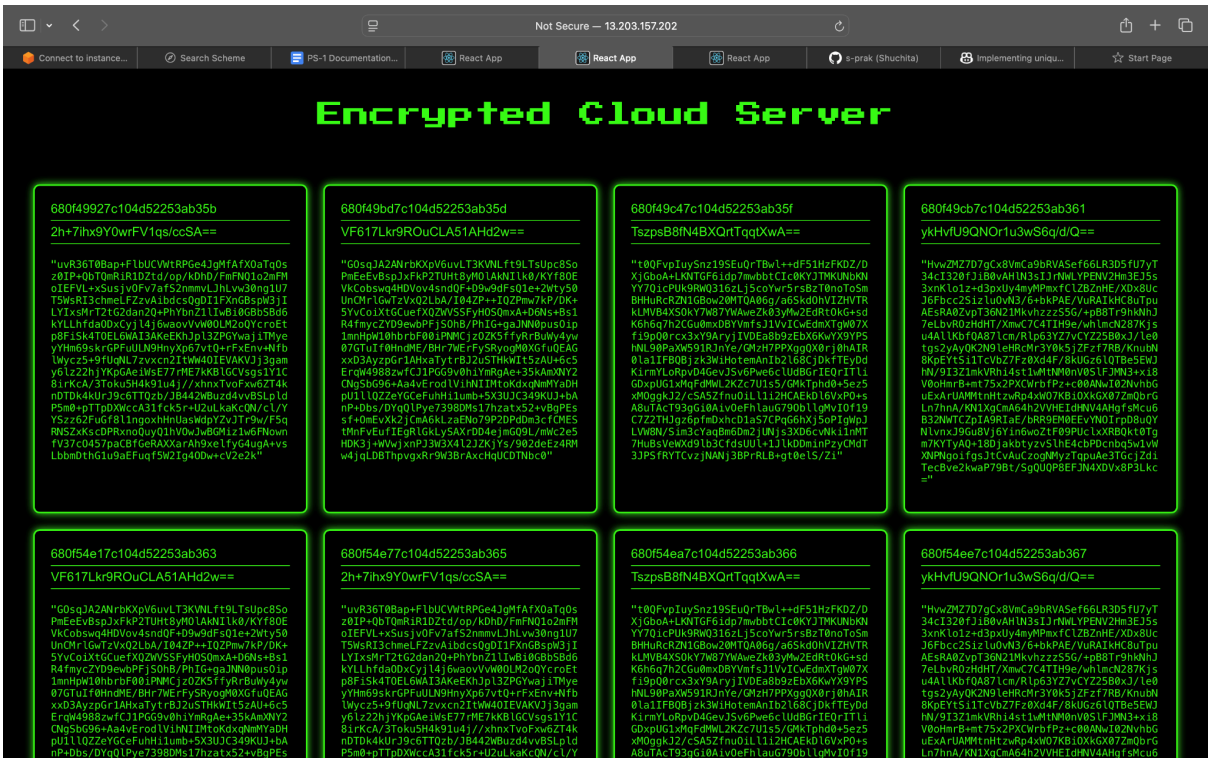


Fig. 6.10 Encrypted Cloud Server

The above figure shows a peek into the Encrypted Cloud Server. As visible, all the information that resides in the server is encrypted, the keyword, identifier, content and filename, all are encrypted.

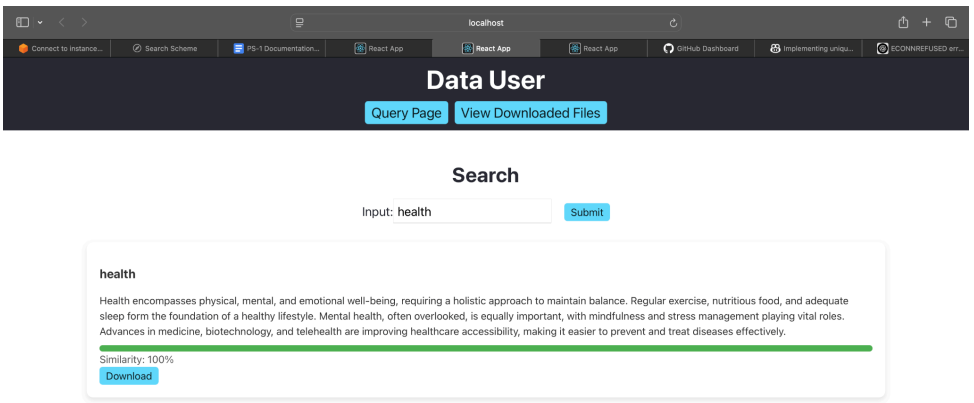


Fig. 6.11 Data User - Single keyword search

This figure shows the page of the Data User, it contains a navigation bar, through which we can either retrieve documents via a query or view documents. In the Query page, search for keywords, either single keywords or multiple keywords. The results of the closest documents according to the query are retrieved and rendered on the screen. These documents can be downloaded based on the interest of the Data User. The results are also ranked according to the similarity with the query.

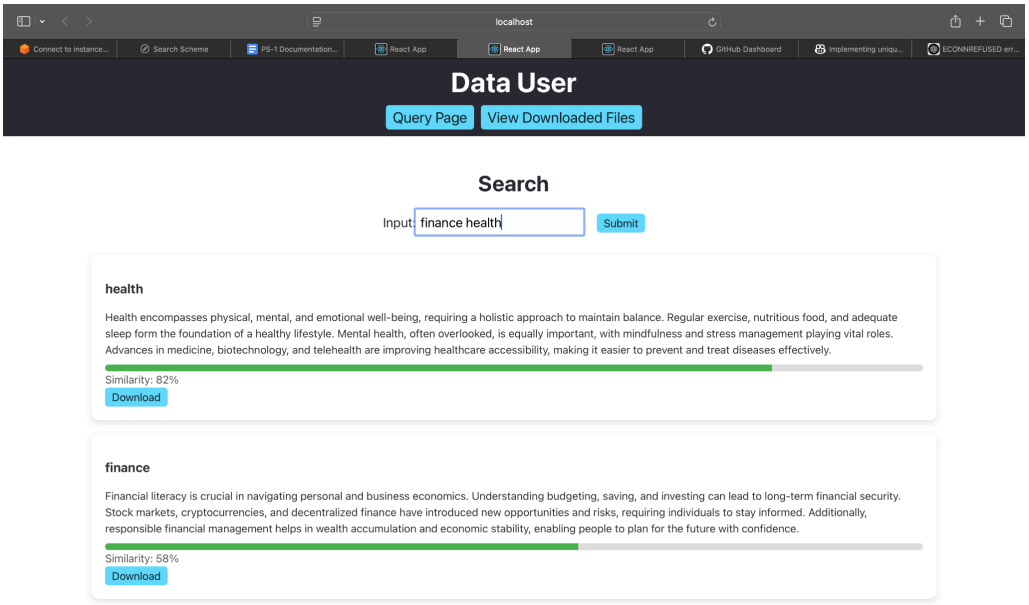


Fig. 6.12 Data User - Multi Keyword Ranked Search Scheme

This figure shows a successful query search of over a multi ranked keyword search scheme, over encrypted data.

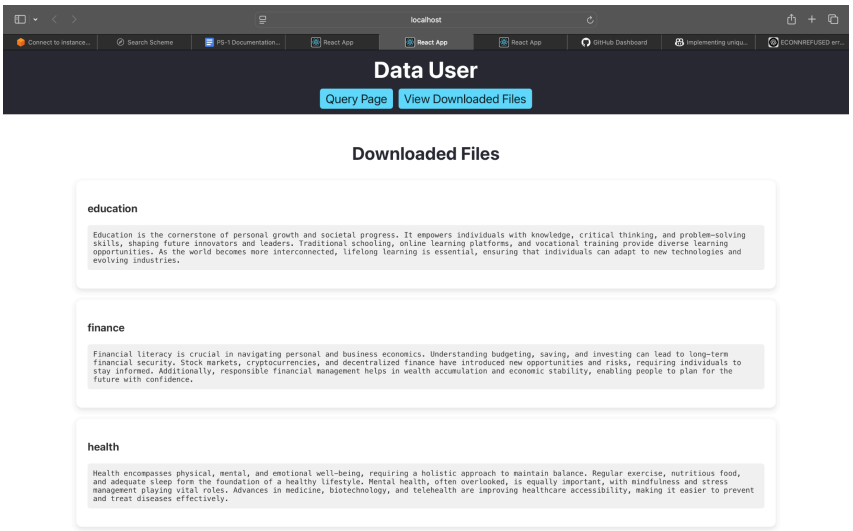


Fig. 6.13 Data User - View Downloaded files

This figure shows the downloaded files of the Data User, files that are downloaded and saved locally.

CHAPTER 7

CONCLUSION AND FUTURE SCOPE

Conclusion

This report introduces a secure and efficient dynamic search scheme that supports accurate multi-keyword ranked search, along with the ability to dynamically delete and insert documents. The adoption of the TF-IDF model for index construction and query generation enables effective ranked search. Furthermore, the scheme's support for parallel search processes contributes to reduced time costs, as demonstrated by experimental results.

Future Scope

Several meaningful and challenging future directions exist for enhancing this work. One key area involves designing a dynamic searchable encryption scheme where update operations (insertion and deletion) can be performed solely by the cloud server, thereby minimizing the data owner's storage overhead for auxiliary information like IDF recalculation data. Addressing security challenges in multi-user environments also presents a significant avenue for future research. This includes tackling issues such as user revocation without requiring a complete index rebuild and key redistribution. Moreover, exploring methods to mitigate the risks posed by potentially dishonest users within a multi-user system, such as preventing the leakage of decrypted documents or the unauthorized distribution of secure keys, warrants further investigation.

REFERENCES

1. Goldreich, O., & Ostrovsky, R. (1996). Software protection and simulation on oblivious RAMs. *ACM*.
2. Gentry, C. (2009). A fully homomorphic encryption scheme. *Stanford University*.
3. Kamara, S., & Lauter, K. (2010). Cryptographic cloud storage. In *Proceedings of the International Conference on Financial Cryptography and Data Security* (pp. 136-149). Springer.
4. Athena, J., & Sumathy, V. (2017). Security challenges for public cloud. *SCIRP*.
5. Buchade, S. (2019). A study on searchable encryption schemes. *ResearchGate*.
6. Li, J., Ma, J., Miao, Y., Yang, R., Liu, X., & Choo, K. K. R. (2020). Multi-keyword ranked search. *IEEE Xplore*.
7. Wan, J., Jia, S., Liu, L., & Zhang, Y. (2021). Dynamic update: Efficient public integrity verification scheme for cloud storage. *IEEE Xplore*.
8. Uzoma, B., & Okhuoya, B. (2022). Cloud computing: A key technological development. *ResearchGate*.
9. Chen, K., Lin, Z., Wan, J., Xu, L., & Xu, C. (2019). Multi-owner secure encrypted search using searching adversarial networks. *CANS*.
10. Zhang, B., Lu, R., Ren, K., Qin, Z., & Shen, X. (2014). Generic multi-keyword ranked search on encrypted cloud data. *IEEE*.
11. Tong, Y., Liu, X., Xiong, J., & Deng, R. H. (2019). Secure phrase search for intelligent processing of encrypted data in cloud-based IoT. *IEEE Internet of Things Journal*.
12. Dai, X., Guan, Z., Zhang, Y., & Zhou, Z. (2019). Multi-keyword ranked search with access control for multiple data owners in the cloud. *Future Generation Computer Systems*.
13. Sun, Y., Zhao, Z., Li, Y., & Liu, W. (2022). Semantic-based multi-keyword ranked search schemes over encrypted cloud data. *ResearchGate*.
14. Liu, C., Liu, X., & Khan, M. K. (2020). Cross-lingual multi-keyword ranked search over encrypted cloud data. *IEEE*.
15. Lei, J., & Mo, J. (2016). Multi-keyword ranked search with fine-grained access control over encrypted cloud data. *ICMMCT*.

APPENDIX

data_owner/app/src/components/FileUpload.js

```
import React, { useState } from 'react';
import axios from 'axios';
import '../styles/FileUpload.css';

const FileUpload = () => {
  const [selectedFile, setSelectedFile] = useState(null);
  const [uploadStatus, setUploadStatus] = useState('');
  const [uploadedDocuments, setUploadedDocuments] = useState([]);

  const handleFileChange = (event) => {
    setSelectedFile(event.target.files[0]);
  };

  const handleSubmit = async (event) => {
    event.preventDefault();
    if (selectedFile) {
      const formData = new FormData();
      formData.append('file', selectedFile);

      try {
        const response = await axios.post('http://localhost:5001/upload',
formData, {
          headers: {
            'Content-Type': 'multipart/form-data',
          },
        });
        if (response.status === 200) {
```

```
        setUploadStatus('File uploaded successfully!');
        setUploadedDocuments([...uploadedDocuments,
selectedFile.name]);
      } else {
        setUploadStatus(`Failed to upload file: ${response.statusText}`);
      }
    } catch (error) {
      console.error('Error uploading file:', error);
      setUploadStatus(`Error uploading file: ${error.message}`);
    }
  } else {
    setUploadStatus('Please select a file first.');
  }
};


const handleUploadToCloud = async () => {
  setUploadStatus('⏳ Uploading documents to cloud...');
  try {
    const uploadResponse = await
axios.post('http://localhost:5000/upload');
    console.log(uploadResponse);
    if (uploadResponse.status === 200) {
      setUploadStatus('✅ Documents uploaded to cloud and indexed
successfully!');
      setUploadedDocuments([]);
    } else {
      setUploadStatus('❌ Failed to upload documents to cloud.');
    }
  } catch (error) {
    console.log(error);
    setUploadStatus(`❌ Error: ${error.message}`);
  }
};
```

```
  return (
    <div className="file-upload-container">
      <div className="upload-section">
        <h2 className="upload-title">Upload Document</h2>
        <form onSubmit={handleSubmit} className="upload-form">
          <div className="file-input-container">
            <label htmlFor="fileInput" className="file-input-label">Choose
Document:</label>
            <input type="file" id="fileInput" className="file-input"
onChange={handleFileChange} />
          </div>
          <button type="submit"
className="upload-button">Upload</button>
        </form>
        <button onClick={handleUploadToCloud}
className="cloud-upload-button">Upload to cloud</button>
        {uploadStatus && <p
className="upload-status">{uploadStatus}</p>}
      </div>
      <div className="uploaded-documents">
        {uploadedDocuments.map((doc, index) => (
          <div key={index} className="document-card">
            <h3>{doc}</h3>
          </div>
        ))}
      </div>
    </div>
  );
};

export default FileUpload;
```

data_owner/app/src/components/ViewDocuments.js

```
import React, { useEffect, useState } from 'react';
import '../styles/ViewDocuments.css';

function ViewDocuments() {
  const [documents, setDocuments] = useState([]);

  useEffect(() => {
    fetch('http://localhost:5001/documents')
      .then(response => response.json())
      .then(data => setDocuments(data))
      .catch(error => console.error('Error fetching documents:', error));
  }, []);

  return (
    <div className="view-documents">
      <h2>View Documents</h2>
      <p>Here you can view all uploaded documents.</p>
      <div className="documents-container">
        {documents.map((doc, index) => (
          <div key={index} className="document-card">
            <h3>{doc.fileName}</h3>
            <pre>{doc.content}</pre>
          </div>
        ))}
      </div>
    </div>
  );
}
```

```
export default ViewDocuments;
```

data_owner/app/src/App.js

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes, Link } from
'react-router-dom';
import FileUpload from './components/FileUpload';
import ViewDocuments from './components/ViewDocuments';
import './App.css';

function App() {
  return (
    <Router>
      <div className="App">
        <header className="App-header">
          <h1>Document Management</h1>
          <nav>
            <ul>
              <li><Link to="/">Upload Document</Link></li>
              <li><Link to="/view-documents">View Documents</Link></li>
            </ul>
          </nav>
        </header>
        <main>
          <Routes>
            <Route exact path="/" element={<FileUpload />} />
            <Route path="/view-documents" element={<ViewDocuments />}
/>
          </Routes>
        </main>
      </div>
```

```
    </Router>
  );
}

export default App;
```

data_owner/file-upload-server.js

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const cors = require('cors');
const axios = require('axios');
const { encrypt } = require("./encrypt");

const app = express();
const port = 5001;

// Enable CORS
app.use(cors());

// Set storage engine
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    const uploadPath =
'/Users/sprak/Documents/sem8/ps2/final/docs';
    if (!fs.existsSync(uploadPath)) {
      fs.mkdirSync(uploadPath, { recursive: true });
    }
    cb(null, uploadPath);
  },
  filename: (req, file, cb) => {
```

```
    cb(null, file.originalname);
  },
});

// Init upload
const upload = multer({ storage: storage });

// Upload endpoint
app.post('/upload', upload.single('file'), async (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded.');
  }

  try {
    // Read file content
    const fileContent = fs.readFileSync(req.file.path, 'utf8');
    const filename= req.file.originalname.split('.')[0];
    // Prepare request body
    const requestBody = {
      docId: encrypt(filename), // Use original filename
      doc: encrypt(fileContent)
    };
    console.log(filename);
    console.log(requestBody);

    // Send file data to the API
    const response = await
axios.post('http://13.203.157.202:8080/document/post-doc',
requestBody, {
      headers: { 'Content-Type': 'application/json' }
    });
```

```
    res.status(200).send(`File uploaded and sent successfully:
${response.data}`);
  } catch (error) {
    console.error('Error processing file:', error);
    res.status(500).send('Error uploading file.');
  }
});

// Get all documents endpoint
app.get('/documents', (req, res) => {
  const uploadPath = '/Users/sprak/Documents/sem8/ps2/final/docs';
  fs.readdir(uploadPath, (err, files) => {
    if (err) {
      return res.status(500).send('Unable to scan files');
    }
    const documents = files.map(file => {
      const filePath = path.join(uploadPath, file);
      const content = fs.readFileSync(filePath, 'utf-8');
      return { fileName: file, content: content };
    });
    res.status(200).json(documents);
  });
});

// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something went wrong!');
});

app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
```

});

data_owner/tfidf-server.js

```
const express = require("express");
const cors = require("cors");
const fs = require("fs");
const path = require("path");
const TfIdf = require("tf-idf-search");
const axios= require('axios');
const { encrypt } = require("./encrypt");

const app = express();
const port = 5000;

// ✅ Allow requests from frontend (localhost:3001)
app.use(cors());

// Middleware to parse JSON request body
app.use(express.json());

const tf_idf = new TfIdf();
const docsDirectory =
"/Users/sprak/Documents/sem8/ps2/final/docs";

// API Endpoint to Get TF-IDF Data
app.get("/tfidf", (req, res) => {
  res.json(tf_idf);
});

// API Endpoint to Add a Document
app.post("/add-document", (req, res) => {
  const { document, docID } = req.body;
```

```
  if (!document || !docID) {
    return res.status(400).json({ error: "Both 'document' and 'docID' are
required" });
  }

  tf_idf.addDocumentFromString(document);

  console.log("Document added successfully");

  res.json({ message: "Document added successfully", docID, document });
});

// API Endpoint to Upload and Process Documents from Directory
app.post("/upload", async (req, res) => {  // <-- Make this async
  try {
    const files = fs.readdirSync(docsDirectory);

    files.forEach((file) => {
      const filePath = path.join(docsDirectory, file);
      tf_idf.addDocumentFromPath(filePath);
      //console.log(`Added document: ${filePath}`);
    });

    console.log(JSON.stringify(tf_idf));

    // ✅ Now `await` works because this function is async
    const postIndexResponse = await axios.post(
      "http://13.203.157.202:8080/inverted-index/post-index",
      { index: encrypt(JSON.stringify(tf_idf)) }
    );
```

```
    res.json({ message: "Documents uploaded successfully", files });
  } catch (err) {
    console.error("Error in upload:", err);
    res.status(500).json({ error: "Failed to process documents", details: err.message });
  }
});

// Start Server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

encrypted-server/src/main/resources/application.properties

```
spring.application.name=encryptedSearch
spring.data.mongodb.database=${MONGO_DATABASE}
spring.data.mongodb.uri=mongodb://${MONGO_HOST}:${MONGO_PORT}
server.address=0.0.0.0
server.port=8080
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/EncryptedSearchApplication.java

```
package dev.sprak.encryptedSearch;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class EncryptedSearchApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(EncryptedSearchApplication.class, args);
    }

}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/doc/DocController.java

```
package dev.sprak.encryptedSearch.doc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@CrossOrigin(origins = "*")
@RestController
@RequestMapping("/document")
public class DocController {

    @Autowired
    private DocService docService ;

    @GetMapping("/get-doc")
    public ResponseEntity<DocModel> getDocument(@RequestParam String docId) {
        Optional<DocModel> document = docService.getDocumentByDocId(docId);
        return document.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());
    }

    @PostMapping("/post-doc")
```

```
    public ResponseEntity<DocModel> postDocument(@RequestBody
DocModel newDocument) {
        DocModel savedDocument = docService.saveDocument(newDocument);
        return ResponseEntity.ok(savedDocument);
    }

    @GetMapping("/get-all-docs")
    public ResponseEntity<List<DocModel>> getAllDocuments() {
        List<DocModel> documents = docService.getAllDocuments();
        return ResponseEntity.ok(documents);
    }

}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/doc/DocModel.java

```
package dev.sprak.encryptedSearch.doc;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "document")
@Getter
@Setter
public class DocModel {
    @Id
    private String id;
    private String docId;
    private String doc;
}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/doc/DocRepository.java

```
package dev.sprak.encryptedSearch.doc;

import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.List;
import java.util.Optional;

public interface DocRepository extends MongoRepository<DocModel,String>
{
    Optional<DocModel> findByDocId(String docId);
    List<DocModel> findAll();
}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/doc/DocService.java

```
package dev.sprak.encryptedSearch.doc;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class DocService {

    @Autowired
    private DocRepository docRepository;

    public Optional<DocModel> getDocumentByDocId(String docId) {
```

```
        return docRepository.findByDocId(docId);
    }

    public DocModel saveDocument(DocModel newDocument) {
        return docRepository.save(newDocument);
    }

    public List<DocModel> getAllDocuments() {
        return docRepository.findAll();
    }

}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/tfidf/IndexController.java

```
package dev.sprak.encryptedSearch.tfidf;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Optional;

@RestController
@RequestMapping("/inverted-index")
public class IndexController {

    @Autowired
    private IndexService indexService;

    @GetMapping("/get-index")
    public ResponseEntity<IndexModel> getIndex() {
```

```
        Optional<IndexModel> index = indexService.getIndex();
        return index.map(ResponseEntity::ok).orElseGet(() ->
ResponseEntity.noContent().build());
    }

    @PostMapping("/post-index")
    public ResponseEntity<IndexModel> postIndex(@RequestBody
IndexModel newIndex) {
        IndexModel savedIndex = indexService.saveIndex(newIndex);
        return ResponseEntity.ok(savedIndex);
    }
}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/tfidf/IndexModel.java

```
package dev.sprak.encryptedSearch.tfidf;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "tfidf")
@Getter
@Setter
public class IndexModel {
    @Id
    private String id;
    private String index;
}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/tfidf/IndexRepository.java

```
package dev.sprak.encryptedSearch.tfidf;

import org.springframework.data.mongodb.repository.MongoRepository;

public interface IndexRepository extends
MongoRepository<IndexModel,String> {
}
```

encrypted-server/src/main/java/dev/sprak/encryptedSearch/tfidf/IndexService.java

```
package dev.sprak.encryptedSearch.tfidf;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service
public class IndexService {
    @Autowired
    private IndexRepository repository;

    public Optional<IndexModel> getIndex() {
        return repository.findAll().stream().findFirst();
    }

    public IndexModel saveIndex(IndexModel newIndex) {
        repository.deleteAll(); // Ensure only one index exists
        return repository.save(newIndex);
    }
}
```

encrypted-server/app/src/DocsList.js

```
import React, { useEffect, useState } from 'react';

const DocsList = () => {
  const [docs, setDocs] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch('http://13.203.157.202:8080/document/get-all-docs')
      .then(response => response.json())
      .then(data => {
         console.log(data);
        setDocs(data);
        setLoading(false);
      })
      .catch(error => {
        console.error('Error fetching documents:', error);
        setLoading(false);
      });
  }, []);

  if (loading) {
    return <div>Loading...</div>;
  }

  return (
    <div className="docs-container">
      {docs.map((doc, index) => (
        <div className="doc-card" key={index}>
          <div className="doc-ids">
            <div className="doc-id">{doc.id}</div>
```

```
            <div className="doc-id">{doc.docId}</div>
          </div>
          <pre className="doc-content">{JSON.stringify(doc.doc, null,
2)}</pre>
        </div>
      ))}
    </div>
  );
};

export default DocsList;
```

data_user/app/src/components/SearchPage.js

```
import React, { useState } from 'react';
import '../styles/SearchPage.css';
import axios from 'axios';

function SearchPage() {
  const [inputValue, setInputValue] = useState('');
  const [results, setResults] = useState([]);
  const [downloads, setDownloads] = useState([]);

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const res = await
axios.get(`http://localhost:5002/query?query=${encodeURIComponent(inpu
tValue)}`);
      const filteredResults = res.data.results.filter(result =>
result.similarityIndex > 0);
      const sortedResults = filteredResults.sort((a, b) => b.similarityIndex -
a.similarityIndex);
      const updatedResults = await Promise.all(sortedResults.map(async
(result) => {
```

```
      try {
        return {
          filename: result.filename,
          content: result.content,
          similarityIndex: result.similarityIndex
        };
      } catch (error) {
        console.error(`Error fetching document for ${result.filename}:`, error);
        return {
          filename: result.filename,
          content: 'Error fetching document',
          similarityIndex: result.similarityIndex
        };
      }
    }));

    setResults(updatedResults);
  } catch (error) {
    console.error('Error fetching data:', error);
    setResults([{ error: 'Error fetching data' }]);
  }
};

const handleDownload = async (result) => {
  try {
    await axios.post('http://localhost:5002/download', {
      filename: result.filename,
      content: result.content
    });
    setDownloads(prevDownloads => [...prevDownloads, result]);
    alert(`File ${result.filename} downloaded successfully.`);
```

```
  } catch (error) {
    console.error('Error downloading file:', error);
    alert('Error downloading file.');
  }
};

return (
  <div className="SearchPage">
    <h1>Search</h1>
    <form onSubmit={handleSubmit}>
      <label>
        Input:
        <input
          type="text"
          value={inputValue}
          onChange={(e) => setInputValue(e.target.value)}
        />
      </label>
      <button type="submit">Submit</button>
    </form>
    <div className="results">
      {results.map((result, index) => (
        <div key={index} className="result-card">
          {result.error ? (
            <p className="error-text">{result.error}</p>
          ) : (
            <>
              <h3 className="filename">{result.filename}</h3>
              <p className="content">{result.content}</p>
              <div className="similarity-bar">
                <div
```

```
                  className="similarity-bar-fill"
                  style={{ width: `${result.similarityIndex * 100}%` }}
                ></div>
              </div>
              <p className="similarity-text">Similarity:
{Math.round(result.similarityIndex * 100)}%</p>
              <button onClick={() =>
handleDownload(result)}>Download</button>
            </>
          )}
        </div>
      ))}
    </div>
  </div>
 );
}

export default SearchPage;
```

data_user/app/src/components/ViewDownloadsPage.js

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
import '../styles/ViewDownloadsPage.css';

function ViewDownloadsPage() {
 const [downloads, setDownloads] = useState([]);

 useEffect(() => {
  const fetchDownloads = async () => {
   try {
    const response = await axios.get('http://localhost:5002/downloads');
    setDownloads(response.data);
```

```
    } catch (error) {
      console.error('Error fetching downloaded files:', error);
    }
  };

  fetchDownloads();
}, []);

return (
  <div className="ViewDownloadsPage">
    <h1>Downloaded Files</h1>
    <div className="downloads">
      {downloads.map((download, index) => (
        <div key={index} className="download-card">
          <h3 className="filename">{download.filename}</h3>
          <pre>{download.content}</pre>
        </div>
      ))}
    </div>
  </div>
);
}

export default ViewDownloadsPage;
```

data_user/app/src/App.js

```
import React, { useState } from 'react';
import './App.css';
import SearchPage from './components/SearchPage';
import ViewDownloadsPage from './components/ViewDownloadsPage';
```

```
function App() {
  const [currentPage, setCurrentPage] = useState('search');

  const handleNavClick = (page) => {
    setCurrentPage(page);
  };

  return (
    <div className="App">
      <header className="App-header">
        <h1>Data Owner</h1>
        <nav>
          <button onClick={() => handleNavClick('search')}>Query
Page</button>
          <button onClick={() => handleNavClick('downloads')}>View
Downloaded Files</button>
        </nav>
      </header>
      <main>
        {currentPage === 'search' && <SearchPage />}
        {currentPage === 'downloads' && <ViewDownloadsPage />}
      </main>
    </div>
  );
}

export default App;
```

data_user/server.js

```
const express = require("express");
const cors = require("cors");
const TfIdf = require("tf-idf-search");
```

```
const axios = require("axios");
const { decrypt } = require("./decrypt");
const { encrypt } = require("./encrypt");
const fs = require("fs");
const path = require("path");

const app = express();
const port = 5002;

const downloadLocation =
"/Users/sprak/Documents/sem8/ps2/final/downloads";

app.use(cors());
app.use(express.json());

app.get("/query", async (req, res) => {
  const query = req.query.query; // ✅ Extract query correctly
  console.log(`🔍 Received query: ${query}`);

  if (!query) {
    console.log("❌ Missing query parameter");
    return res.status(400).json({ error: "Query parameter 'query' is required"
});
  }

  try {
    //1️⃣ Make a GET request to retrieve the index
    console.log("📡 Fetching inverted index from backend...");
    const response = await
axios.get("http://13.203.157.202:8080/inverted-index/get-index");

    //2️⃣ Extract the "index" from the response
    const encryptedIndex = response.data.index;
```

```
    console.log("🔐 Encrypted index received:", encryptedIndex ? "✅ Yes" :
"❌ No");

    if (!encryptedIndex) {
      console.log("❌ No index found in response");
      return res.status(500).json({ error: "No index found in response" });
    }

    //3️⃣ Decrypt the index
    console.log("🔓 Decrypting the index...");
    const decryptedIndex = decrypt(encryptedIndex);
    console.log("✅ Decryption successful");

    //4️⃣ Parse the decrypted index
    const tfidfIndex = JSON.parse(decryptedIndex);
    console.log("📂 Parsed TF-IDF index:", tfidfIndex);

    //5️⃣ Load the index into a TfIdf model
    console.log("📊 Initializing TF-IDF model...");
    const tfidfModel = Object.assign(new TfIdf(), tfidfIndex);
    console.log("✅ TF-IDF model initialized");

    //6️⃣ Perform the search
    console.log(`🔍 Searching for: "${query}" in TF-IDF model...`);
    const searchResults = tfidfModel.rankDocumentsByQuery(query);
    console.log("🔎 Search results:", searchResults);

    const fileTracker = tfidfIndex.tracker;

    const formattedResults = await Promise.all(searchResults.map(async
(result) => {
      const fileEntry = fileTracker.find(entry => entry.index === result.index);
```

```
      if (!fileEntry) return null;


      // Extract filename without extension
      const filename = fileEntry.document.split('/').pop().split('.').slice(0, -1).join('.');
      const encryptedFilename = encrypt(filename);


      console.log(`📂 Fetching content for: ${filename} (Encrypted: ${encryptedFilename})`);


      try {
        const encodedFilename = encodeURIComponent(encryptedFilename);
        const contentResponse = await axios.get(`http://13.203.157.202:8080/document/get-doc?docId=${encodedFilename}`);
        return {
          filename,
          content: decrypt(contentResponse.data.doc),
          similarityIndex: result.similarityIndex
        };
      } catch (error) {
        console.error(`❌ Failed to fetch content for ${filename}`, error.message);
        return { filename, content: "Error fetching content", similarityIndex: result.similarityIndex };
      }
    }));


    console.log("📁 Final search results:", formattedResults.filter(Boolean));
    //7️⃣ Return the results
    res.json({ query, results: formattedResults.filter(Boolean) });
  } catch (error) {
    console.error("❌ Error processing query:", error);
```

```
    res.status(500).json({ error: "Failed to process query", details:
error.message });
  }
});

app.post("/download", async (req, res) => {
  const { filename, content } = req.body;

  if (!filename || !content) {
    return res.status(400).json({ error: "Filename and content are required" });
  }

  const filePath = path.join(downloadLocation, filename);

  try {
    fs.writeFileSync(filePath, content);
    console.log(`✅ File saved to ${filePath}`);
    res.json({ message: `File saved to ${filePath}` });
  } catch (error) {
    console.error(`❌ Error saving file to ${filePath}:`, error);
    res.status(500).json({ error: "Failed to save file", details: error.message });
  }
});

app.get("/downloads", async (req, res) => {
  try {
    const files = fs.readdirSync(downloadLocation);
    const downloads = files.map((file) => {
      const content = fs.readFileSync(path.join(downloadLocation, file), 'utf-8');
      return { filename: file, content };
    });
    res.json(downloads);
```

```
  } catch (error) {
    console.error("❌ Error fetching downloaded files:", error);
    res.status(500).json({ error: "Failed to fetch downloaded files", details:
error.message });
  }
});

app.listen(port, () => {
  console.log(`✅ Server running at http://localhost:${port}`);
});
```