future internet

MDPI

Article

FODIT: A Filter-Based Module for Optimizing Data Storage in B5G IoT Environments

Bruno Ramos-Cruz [1], Francisco J. Quesada-Real [1,*], Javier Andreu-Pérez [2] and Jessica Zaqueros-Martinez [1]

[1] Computer Science Department, University of Jaen, 23071 Jaén, Spain; brcruz@ujaen.es (B.R.-C.); zaqueros@ujaen.es (J.Z.-M.)
[2] Centre for Computational Intelligencer, School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, UK; j.andreu-perez@essex.ac.uk
* Correspondence: fqreal@ujaen.es

Abstract

In the rapidly evolving landscape of the Internet of Things (IoT), managing the vast volumes of data generated by connected devices presents significant challenges, particularly in B5G IoT environments. One key issue is data redundancy, where identical data is stored several times because it is captured by multiple sensors. To address this, we introduce "*FODIT*", a filter-based module designed to optimize data storage in IoT systems. FODIT leverages probabilistic data structures, specifically filters, to improve storage efficiency and query performance. We hypothesize that applying these structures can significantly reduce redundancy and accelerate data access in resource-constrained IoT deployments. We validate our hypothesis through targeted simulations under a specific and rare configuration: high-frequency and high-redundancy environments, with controlled duplication rates between 4% and 8%. These experiments involve data storage in local databases, cloud-based systems, and distributed ledger technologies (DLTs). The results demonstrate FODIT's ability to reduce storage requirements and improve query responsiveness under these stress-test conditions. Furthermore, the proposed approach has broader applicability, particularly in DLT-based environments such as blockchain, where efficient querying remains a critical challenge. Nonetheless, some limitations remain, especially regarding the current data structure used to maintain consistency with the DLT, and the need for further adaptation to real-world contexts with dynamic workloads. This research highlights the potential of filter-based techniques to improve data management in IoT and blockchain systems, contributing to the development of more scalable and responsive infrastructures.

Keywords: B5G IoT; filters; data storage

Academic Editor: Paolo Bellavista

Received: 14 May 2025
Revised: 26 June 2025
Accepted: 26 June 2025
Published: 30 June 2025

Citation: Ramos-Cruz, B.; Quesada-Real, F.J.; Andreu-Pérez, J.; Zaqueros-Martinez, J. FODIT: A Filter-Based Module for Optimizing Data Storage in B5G IoT Environments. *Future Internet* **2025**, *17*, 295. https://doi.org/10.3390/fi17070295

1. Introduction

Technological advancements have driven the rapid proliferation of Internet of Things (IoT) devices, resulting in highly connected and automated smart environments [1,2]. These devices generate large volumes of real-time data that enable automation and digital transformation across various domains such as supply chain monitoring [3], activity recognition [4], industrial automation [5], healthcare [6–8], smart homes [9], and smart cities [10,11]. However, the exponential growth in data generation poses major challenges in storage and management. Traditional systems often struggle with the volume, velocity, and variety of data, leading to inefficiencies and increasing infrastructure costs.

The emergence of Beyond 5G IoT (B5G IoT) [12,13] has intensified these demands by introducing enhancements in performance, energy efficiency, security, reliability, and

ultra-low latency [14], thus supporting large-scale, latency-sensitive, and mission-critical applications [15]. Key B5G capabilities include massive machine-type communications (mMTC) and ultra-reliable low-latency communications (URLLC) [16–18]. While mMTC enables connectivity for millions of low-bandwidth devices—ideal for distributed sensing and monitoring—URLLC supports applications requiring extreme reliability and real-time responsiveness, such as industrial automation, remote surgery, and autonomous driving. Complementing these capabilities, edge computing brings processing closer to data sources, reducing latency and offloading core network traffic [19,20], and NETWORK SLICING allows the physical network to be partitioned into virtual slices tailored to specific application needs [21].

These technologies converge to enable a next-generation IoT ecosystem that demands adaptive and efficient data management strategies. Among these, data storage efficiency is particularly critical, as redundant data storage leads to resource waste, performance degradation, and higher operational costs [22]. Probabilistic data structures such as Bloom and cuckoo filters offer a promising solution by enabling approximate membership queries with minimal memory usage [23,24]. By accepting a small probability of false positives, these filters significantly reduce storage requirements and accelerate query operations, making them well-suited for resource-constrained environments.

We hypothesize that *filter-based approaches optimize IoT data storage by avoiding duplication and reducing query time*. To support this claim, we explore the foundations of probabilistic data structures and demonstrate their applicability in IoT scenarios that demand efficient and scalable data management.

This paper introduces FODIT (filter-based optimization for data storage in IoT), a lightweight module that applies advanced filtering techniques to prevent redundant data storage and improve query performance while preserving data integrity. We evaluate its effectiveness by analyzing FODIT's architecture and validating its performance through simulations of real-world case studies involving data storage in local databases, cloud-based systems, and distributed ledger technologies (DLTs). The results demonstrate substantial improvements in both storage efficiency and overall system responsiveness.

The remainder of this paper is structured as follows. Section 2 introduces the IoT architecture within B5G environments and presents the fundamental concepts of filters, with a focus on cuckoo filters. Section 3 reviews the most relevant related work in the field. Section 4 analyses current IoT storage models, highlighting their specific characteristics. Section 5 describes the proposed FODIT module, detailing its architecture and workflow, and presenting its integration into the Phonendo framework [25]. Section 6 presents the evaluation case studies, experiments, results, and limitations. Finally, Section 7 summarizes the main contributions and outlines directions for future research.

2. Background

This section presents foundational concepts related to IoT data storage, highlighting the limitations of current storage models and the evolving role of B5G technologies. It also introduces probabilistic data structures as an enabler for efficient storage management.

2.1. IoT Architecture in B5G Environments

The IoT has evolved into a critical enabler of intelligent systems, with applications ranging from smart homes and cities to industrial automation and autonomous transport. At its core, IoT architecture is composed of five key components—devices and sensors, connectivity, data processing, user interface, and security—organized across four hierarchical layers: perception, network, edge computing, and application, as shown in Figure 1.

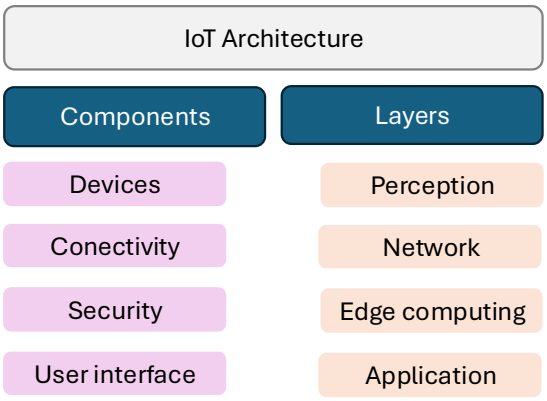

Figure 1. The figure displays IoT architecture.

The emergence of B5G technologies significantly augments this architecture by providing ultra-low latency, massive connectivity, and enhanced network intelligence. B5G introduces features such as AI-native networking [26], intelligent slicing [27], and advanced edge computing capabilities [20], which directly support dynamic and real-time IoT workloads. These advancements open new possibilities for distributed intelligence, collaborative edge processing, and context-aware data handling. However, they also exacerbate the challenge of managing the exponential growth in data volume generated by billions of interconnected devices.

2.2. Filters

Filters are employed to optimize time and space efficiency [28]. These filters are widely used in big data analytics, networking, the Internet of Things, database management, and other domains that require large-scale data processing and rapid response times.

Definition 1. *The filters are space-efficient data structures used to represent a set's elements and allow membership tests.*

The membership test determines whether a given element belongs to a set. If the result of the query is positive, the element is assumed to be in the set, although this may occur with a certain probability of a false positive. The false positive rate depends on the design of the specific filter.

Currently, there are several variants of filters. In this article, we compare four types, namely, the Bloom filter [23], the counting Bloom filter [29], the sliding Bloom filter [30], and the cuckoo filter [24]. Each of these filters supports a distinct set of operations. For example, as shown in Table 1, all the filters support insertion; however, only the counting Bloom filter and the cuckoo filter support deletion. Additionally, the table presents the false positive rate for each filter and outlines the computational complexity of insertion, query, and deletion operations. Typically, these filters rely on hash functions for their implementation. The complexity and scalability of each filter depend on its specific design.

This paper focuses on the cuckoo filter as a tool to optimize the B5G IoT storage and queries, which provides a lower false positive rate, resulting in more accurate membership tests. This makes it a suitable choice for applications where minimizing false positives is critical. Furthermore, as shown in Table 1, the cuckoo filter reduces processing time by requiring fewer hash functions compared to other filters.

Table 1. Comparison of Probabilistic Filters.

Feature	Bloom Filter	Counting Bloom Filter	Sliding Bloom Filter	Cuckoo Filter
Supports Deletions	No	Yes	No	Yes
False Positives	$\left(1-e^{-\frac{kn}{m}}\right)^k$	$\left(1-e^{-\frac{kn}{m}}\right)^k$	$\mathrm{n}\left\lceil\frac{\epsilon u}{n}\right\rceil \leq \epsilon u + n$	$1-\left(1-\frac{1}{2^f}\right)^{2b} = \frac{2b}{2^f}$
Insertion Speed	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(\log n)$
Query Speed	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Deletion Speed	Not supported	$\mathcal{O}(k)$	Not supported	$\mathcal{O}(1)$
Implementation	Simple	Moderate (counter management)	Complex (window management)	Moderate (cuckoo hashing)
Scalability	Limited (fixed size)	Limited (counter overhead)	Better (sliding window)	Best (resizable, low overhead)

Cuckoo filter

The cuckoo filter is a probabilistic data structure that can be represented as an array $CF[b]$ of b bins, where each bin contains d entries used to represent the elements of a set A, with all entries initially empty. This filter supports insertion, search, and deletion operations. To insert an element x into the cuckoo filter $CF[b]$, Algorithm 1 is used [24]. The first step of the algorithm is to compute the fingerprint f_x of the element using cryptographic hash functions, as follows:

$$f_x = fingerprint(x) \tag{1}$$

In the next step, the partial-key cuckoo hashing technique shown in Equations (2) and (3) is used to obtain the indexes h_1 and h_2 of the candidate bins:

$$h_1 = hash(x) \tag{2}$$

$$h_2 = h_1 \oplus hash(f_x) \tag{3}$$

If the bin with index h_1 has any empty entry, then the fingerprint f_x is added, and the process ends. Otherwise, it will look in h_2. If h_2 has any empty entry, then the fingerprint f_x is added, and the process ends.

If h_1 and h_2 do not have any empty entries, a bin i ($i = h_1$ or $i = h_2$) is randomly chosen, and then an entry e from bin i is selected (see Algorithm 1). Once the selection is made, the fingerprint f_e stored in the entry e is retrieved, and the fingerprint f_x is inserted. An alternative position j is now calculated, indicating the bin into which the fingerprint f_e can be inserted:

$$j = i \oplus hash(f_e) \tag{4}$$

If the bin j has any empty entries, then the fingerprint f_e is inserted, and the insertion process ends. This process is repeated until a bin with an empty entry is found or until a maximum number of relocations is reached. After the relocations, if an empty bin is not found, then the cuckoo filter $CF[b]$ is considered full and no items can be inserted.

The search operation of an element x in the filter $CF[b]$ is performed by a membership query. Unlike the insertion process, the search process does not need to handle displaced elements; instead, it can look directly at the two guaranteed bins $h1$ and $h2$. To do this, Algorithm 2 is used, where the fingerprint of the element is computed using Equation (1). Subsequently, the positions of the candidate bins are calculated using Equations (2) and (3). If the obtained bins have any entry with the fingerprint f_x, then the membership query returns true with a probability of false positive; otherwise, it returns false.

Finally, in the deletion operation to remove an element x from the cuckoo filter $CF[b]$, the process is similar to the search. Using the function $SF(x)$ in Algorithm 2, the item x is searched. If there exists an entry that contains the fingerprint f_x in the obtained bins, then a copy of it is deleted as shown in Algorithm 3.

Algorithm 1 Insert (x)

Input : x ;
Output : Cuckoo filters F_i ;
function CF(x)
  $f_x = fingerprint(x)$;
  $h_1 = hash(x)$;
  $h_2 = h_1 \oplus hash(f_x)$;
  if $bin[h_1]$ has an empty entry **then**
    $bin[h_1] \leftarrow f_x$
    return Done;
  else
    if $bin[h_2]$ has an empty entry **then**
      $bin[h_2] \leftarrow f_x$
      return Done;
    end if
  end if
  $i =$ randomly chosen h_1 or h_2;
  for $count = 1$ **to** $MaxNumAttempts$ **do**
    randomly select an entry e from $bin[i]$;
    swap f_x and the fingerprint f_e stored in entry e ;
    $j \leftarrow i \oplus hash(f_e)$
    if $bin[j]$ has an empty entry **then**
      $bin[j] \leftarrow f_e$;
      return Done ;
    end if
  end for
  return Failure ;
  return F_i ;
end function

Algorithm 2 Search (x)

Input : x, F_i ;
Output : $found, notfound$;
function SF(x, F_i)
  $f_x = fingerprint(x)$;
  $h_1 = hash(x)$;
  $h_2 = h_1 \oplus hash(f_x)$;
  if $bin[h_1]$ has a f_x **then**
    return $found$;
  else
    if $bin[h_2]$ has a f_x **then**
      return $found$;
    end if
  end if
  return $Notfound$;
end function

Algorithm 3 Delete (x)

Input : x, F_i ;
Output : *Yes*, *No* ;
function DF(x, F_i)
  $Search \leftarrow SF(x)$;
  if $Search == found$ **then**
    Remove $f(x)$
    return Yes;
  else
    return *No*;
  end if
end function

3. Related Work

A range of filter-based modules has been proposed for optimizing data storage in IoT environments. Singh et al. [31] introduce the accommodative Bloom filter (ABF), a variant of the scalable Bloom filter, which outperforms existing variants in terms of false positive rates and query complexity. Podnar et al. [32] focus on IoT data management methods and optimization algorithms, emphasizing the use of local sub-servers and publish/subscribe mechanisms. Jeong et al. [33] propose a secure cloud storage service for IoT environments, using a provable data possession model and Bloom filters. Finally, Singh et al. [34] present the fuzzy-folded Bloom filter (FFBF) for big data storage in the cloud, which extends the standard BF to incorporate a fuzzy-enabled folding approach, effectively reducing storage requirements without affecting the false positive rate and query time. These studies collectively highlight the potential of filter-based modules in enhancing data storage and retrieval in IoT environments.

The optimization of data storage in IoT environments has been a significant area of research, given the explosive growth of data generated by IoT devices. Various approaches have been explored to address the challenges associated with efficient data storage, including data compression [35], deduplication [36], and the use of probabilistic data structures [24,28].

Data compression techniques aim to reduce the size of data before storage, thus saving space and potentially improving transmission efficiency [37]. Techniques such as lossless and lossy compression have been widely studied and applied in IoT environments. Lossless compression algorithms like SZ lossy [38] are used to ensure that the original data can be perfectly reconstructed, which is crucial for applications requiring high data integrity. However, these methods can be computationally intensive and may not always provide sufficient compression ratios for the large volumes of data generated by IoT devices.

Data deduplication is another approach to optimize storage by eliminating duplicate copies of repeating data. Techniques like chunk-based deduplication and file-level deduplication have been implemented in various storage systems. For instance, the work by Altowaijri et al. [39] introduced the aggregation of sensor data, followed by preprocessing steps to filter out irrelevant or noisy data, which significantly reduced storage requirements by identifying and eliminating redundant data. While effective, deduplication can introduce additional complexity in data retrieval and management, especially in dynamic IoT environments where data is continuously updated [36].

Probabilistic data structures, such as Bloom filters [23] and cuckoo filters [24], have been increasingly recognized for their potential in managing large-scale data efficiently. Bloom filters, introduced by Bloom [23], allow for fast membership queries with a controlled false positive rate, making them suitable for applications where space efficiency is critical. Cuckoo filters, proposed by Fan et al. [24], build on Bloom filters by offering lower false positive rates and supporting deletions, thus providing more flexibility for dynamic datasets. In the context of B5G IoT, several studies have explored the use of these probabilistic structures to enhance data storage and retrieval. For example, Kumar et al. [40] designed a proposal to remove redundant data on the final layer and improve access times for stored data. Kumar proposed a Bloom filter-based solution at different layers of IoMT (edge-fog-cloud).

Despite advancements in probabilistic data structures, their application to optimizing data storage in B5G IoT environments remains relatively underexplored. This gap serves as one of the main motivations for this work.

4. Analysis of Storage in IoT Environments

In B5G-enabled IoT ecosystems, storage becomes a critical bottleneck due to the need for real-time responsiveness and scalable data retention. This section examines three primary storage scenarios in such contexts: local databases, cloud-based services, and DLT (see Figure 2). Each scenario presents unique benefits and challenges in terms of storage efficiency, data accessibility, and resource management.

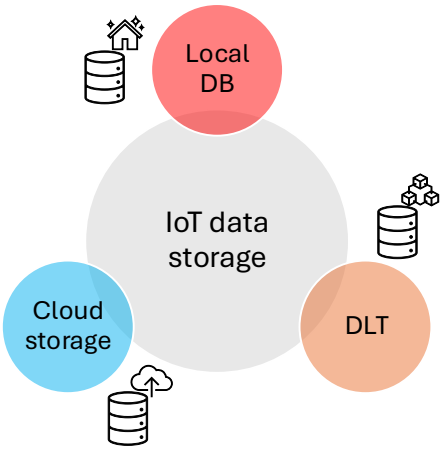

Figure 2. The figure displays IoT data storage scenarios.

- *Local databases* provide on-premises storage solutions, allowing IoT devices to store and manage data without relying on external networks [41]. This approach ensures low-latency access and enhanced security, as data remains within a controlled environment. However, scalability is a major limitation, as the storage capacity is constrained by the physical hardware available. Additionally, local databases may face challenges in data synchronization and remote accessibility, making them less suitable for large-scale, distributed IoT systems.
- *Cloud-based storage* solutions offer scalable, on-demand storage managed by third-party providers. This model enables seamless data access, redundancy, and backup capabilities, making it ideal for IoT applications that require high availability [42]. The key benefits of cloud storage include elastic scalability, reduced infrastructure costs, and remote accessibility. Nevertheless, reliance on cloud services introduces concerns regarding data privacy, network dependency, and latency. Additionally, subscription costs and compliance with data regulations must be carefully managed when adopting cloud-based solutions.
- *DLT* such as blockchain [43], offers a decentralized approach to data storage, enhancing security and transparency. In an IoT context, DLT ensures data integrity, immutability, and traceability, which are crucial for applications requiring high levels of trust. However, DLT-based storage systems often face scalability challenges, as maintaining a distributed ledger across multiple nodes demands significant computational and storage resources [44]. Furthermore, the transaction processing time in blockchain-based systems can introduce latency, making it less suitable for real-time IoT applications [8,45].

Each of these storage approaches plays a vital role in addressing the requirements of IoT data storage. In Table 2, the pros and cons are summarized. Local databases offer security and low latency, but lack scalability. Cloud storage provides flexibility and scalability while raising concerns about data privacy and network dependence. DLT ensures data integrity and security at the cost of scalability and processing efficiency. A

hybrid approach that integrates these storage solutions based on specific application needs can help optimize data management in IoT ecosystems.

The volume and heterogeneity of IoT data in B5G environments necessitate a paradigm shift in how data is filtered, prioritized, and stored. The convergence of edge computing and B5G allows for intelligent, on-device decision-making to reduce unnecessary data transfer and storage. Probabilistic data structures, such as *filters*, offer lightweight mechanisms for approximate membership tests and summarization, which are especially valuable when bandwidth or storage resources are constrained.

Table 2. Comparison of storage scenarios in B5G IoT environments.

Storage	Pros	Cons
Local DB	Low latency High security Network independence	Limited scalability Sync complexity Limited remote access
Cloud storage	Scalable Cost-efficient Remote access B5G-enhanced bandwidth	Latency Privacy concerns Network reliance Subscription cost
DLT	Immutable Decentralized trust Transparent	High overhead Scalability issues Transaction latency

5. FODIT Overview

In this section, we introduce a module named FODIT. A general description is presented in Section 5.1, and Section 5.2 discusses the workflow.

5.1. Architecture Overview

The FODIT module operates in two distinct phases. The first phase handles data insertion into the server provider, while the second phase performs data queries. Figure 3 illustrates this process.

In the insertion phase, before storing data directly in the server provider, a cuckoo filter is initialized. This filter serves as a mechanism to optimize both storage and the insertion process. Each time a data item is to be inserted into the server provider, it is first checked against the cuckoo filter. If the data already exists in the filter, this implies that it has already been inserted into the server provider, and thus the system avoids inserting it again, preventing duplicate storage. A special case arises when data is intended to be stored on a DLT, as we cannot guarantee that the data has been successfully stored until the corresponding transaction is confirmed. To avoid inconsistencies between the data stored in the filter and that stored on the DLT, FODIT integrates a data structure where data is held until confirmation is received. Two possible scenarios may occur:

- *Transaction confirmation is received within the maximum time window defined for the temporary structure.* In this case, the data is removed from the temporary structure and inserted into the filter.
- *Transaction confirmation is not received within the defined time window.* In this case, the data is discarded from the temporary structure and is not inserted into the filter.

It is possible that redundant data may be sent for storage in the DLT. In such cases, the hash of the data is stored as a key in the data structure, and a queue of duplicate entries is maintained under that key. If the first entry in the queue is successfully written to the DLT, the remaining duplicates are discarded. However, if the first transaction fails because

the confirmation is not received within the defined time window, the system attempts to submit the second entry, and so on, until a transaction is confirmed or the queue is exhausted without any entry being successfully stored in the DLT.

In the query phase, data is retrieved from the server provider. To do this, the system first checks whether the data exists in the cuckoo filter. If the filter confirms its presence, a search is performed in the server provider, which then returns the requested information. If the data is not found in the filter, it is assumed not to exist in the server provider, thus avoiding unnecessary queries and optimizing the retrieval process.

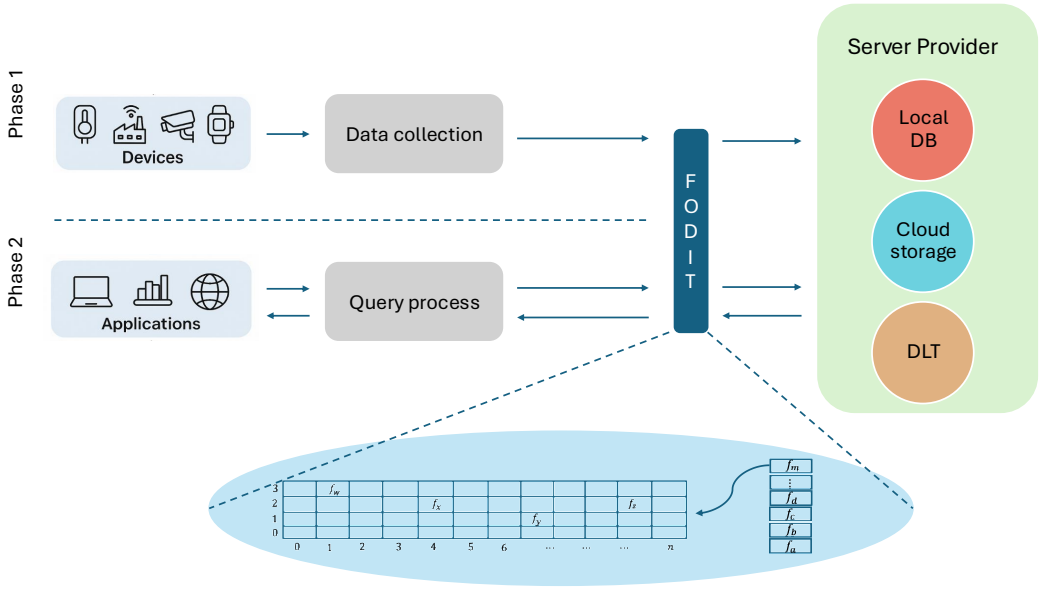

Figure 3. The figure displays the phases for the FODIT module.

5.2. Workflow

The FODIT model can be represented as a tuple (D, SP, I_a, Q_a), where D is a set of IoT devices, SP is the server provider, I_a is the insertion algorithm, and Q_a is the search (query) algorithm. In the following paragraphs, each component of the tuple is explained in detail.

IoT devices ($D = d_1, d_2, d_3, \ldots, d_n$): This represents a set of n electronic devices. Through these devices, a data collection denoted by Dc is generated and subsequently stored in the server provider SP.

Server provider (SP): The server provider is an entity responsible for storing and managing the data collection Dc. In this paper, SP may refer to a local database, cloud storage, or a DLT.

Insertion algorithm (I_a): This is the algorithm used to perform data insertion in the FODIT module. It takes the data collection Dc, acquired from the IoT devices D, as input. Let $x \in Dc$; then x is inserted into the cuckoo filter $CF[b]$ using Algorithm 1. It is important to note that the cuckoo filter does not store the data x directly; instead, it stores a fingerprint f_x, computed as shown in Equation (2), to optimize storage. Once the fingerprint is inserted into the filter, the data x is stored in the server provider SP, completing the insertion process. For the particular case where SP is the DLT, the insertion algorithm first stores the data in the structure $AuxS$. If the data x is correctly recorded in the DLT, then the data is inserted into the filter $CF[b]$ and is deleted from the structure $AuxS$.

If there is another data item $y \in Dc$ to be inserted into SP, the algorithm first checks whether y is already in the filter $CF[b]$ using the search function SF, as described in Algorithm 2. If y is found in the filter, it is not inserted into SP, ensuring that duplicate

data is not stored. This process is repeated for every element to be inserted into SP. The complete process is summarized in Algorithm 4.

Query algorithm (Q_a): This algorithm performs the data query operation in the FODIT module. Let z be the data item to be retrieved from SP. The algorithm takes z as input and invokes the search function SF, as described in Algorithm 2, to check for the presence of z in the cuckoo filter $CF[b]$. If z is found in the filter, it can be retrieved from SP; otherwise, z is not present in SP. The complete process is summarized in Algorithm 5.

Algorithm 4 Insertion Algorithm I_a

Input : Data Dc, $CF[b]$, SP;
Output : Updated $CF[b]$ and SP with unique data;
function $I_a(Dc, CF[b], SP)$
  for each $x \in Dc$ **do**
    if SF(x, $CF[b]$) = not found $\wedge$ $SP = DLT$ **then**
      Store x in $AuxS$
      if x is recorded in DLT **then**
        $result \leftarrow$ CF(x, $CF[b]$);
        Delete x from $AuxS$;
      end if
    else
      $result \leftarrow$ CF(x, $CF[b]$);
      if $result =$ Done **then**
        Store x in SP;
      else
        if $result =$ Failure **then**
          handle insertion failure;
        end if
      end if
    end if
  end for
end function

Algorithm 5 Query algorithm Q_a

Input : Item z, $CF[b]$, SP
Output : Retrieved data z or null
function $Q_a(z, CF[b], SP)$
  if SF(z, $CF[b]$) = found **then**
    Retrieve z from SP
    return z
  end if
  return null ▷ z not found
end function

5.3. Integrating FODIT in Phonendo

To illustrate a practical use case, we present the integration of the FODIT module within a real-world framework called *Phonendo* [25] (see Figure 4). Phonendo is an open-source framework designed to support the configuration and deployment of trustworthy smart environments. Its modular architecture and open-source nature enable straightforward customization and extension, allowing developers to easily adapt the framework to specific use cases by incorporating new modules. One such example is CertifIoT [46], which builds upon Phonendo to provide a solution for data certification by combining IoT and DLT technologies. Although Phonendo was originally conceived to facilitate seamless integration with DLTs, it can also function independently, without requiring data to be published to a DLT.

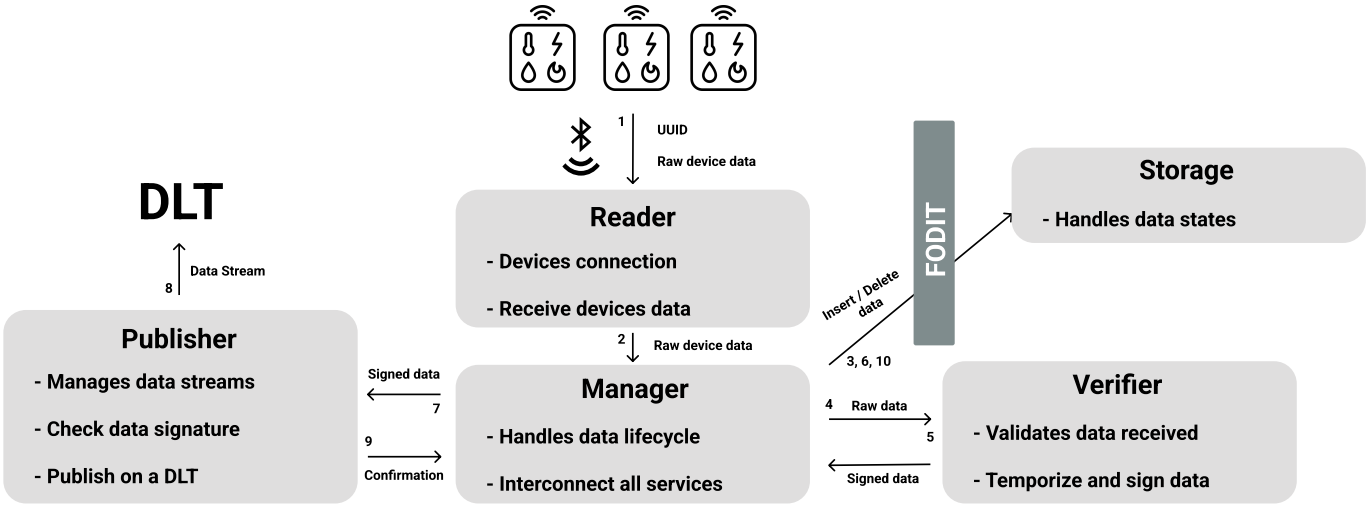

Figure 4. Integration of FODIT in Phonendo [25].

The Phonendo architecture is composed of the following components:

- *Reader*: Responsible for establishing connections with various IoT devices and collecting their data.
- *Manager*: Acts as the orchestration layer of the framework, coordinating all services and managing the data flow.
- *Verifier*: Verifies the integrity of incoming data and appends a digital signature to ensure its authenticity before storage.
- *Storage*: Handles the persistence of the system state, which can be managed either locally, DB, or via a cloud-based storage service, DaaS.
- *Publisher*: Oversees the management of data streams, verifying their digital signatures and publishing validated data to a DLT when required.

As shown in Figure 4, the FODIT module is integrated between the *Manager* and the *Storage* components. This placement allows FODIT to be consulted before any insertion, query, or deletion operation is performed on the selected storage system (DB, DaaS, or DLT), effectively preventing the storage of redundant data and thereby improving overall system efficiency.

As previously described, Phonendo's *Storage* module supports integration with different types of storage backends, including DB and DaaS. In the case of DLT, data is first stored locally as a backup before being published to the distributed ledger. This is why FODIT does not need to be placed between the *Manager* and the *Publisher*.

6. Evaluation

In this section, we will detail the case studies conducted to evaluate our hypothesis: "*the utilization of filters optimizes the storage of data generated in IoT environments, avoiding the storage of duplicate data and improving query time.*" The obtained results are also discussed, highlighting the impact of FODIT compared to traditional B5G IoT systems.

6.1. Case Studies

To evaluate our hypothesis, we conducted two case studies to assess the improvements achieved by avoiding redundant data storage and the time saved when querying non-stored data.

For these studies, we collected: timestamp, temperature, humidity, and luminosity data from an Arduino-based device designed to monitor environmental conditions during the transportation and storage of sensitive goods (e.g., food, vaccines). We then performed

three simulations—storing 10,000, 100,000, and 1,000,000 records, respectively—using different IoT storage options: a DB, DaaS, and DLT (see Section 4). Each simulation was executed both with and without the FODIT module to compare performance. It is important to highlight that FODIT stores a label for each data entry, which in our case is a hash of the data. This hash also serves as the identifier for the corresponding data stored in the different storage options.

Additionally, to evaluate the handling of redundant data, each simulation included a percentage (4%, 5%, 7%, 8%) of repeated data out of the total stored data. Regarding the query case study, we used the previous simulations and attempted to retrieve (4%, 5%, 7%, 8%) of data that was not stored (e.g., environmental conditions of a device at a specific timestamp).

6.2. Experiments and Results

According to case studies, two experiments were designed: Experiment 1 to minimize redundant data storage and Experiment 2 to optimize data retrieval efficiency. Both were executed on a computer with the following specifications:

- *CPU:* Core™ i7-7500U Intel® processor 2.70 GHz × 4
- *OS:* Ubuntu 22.04.2 LTS
- *Compiler:* gcc 7.4.0
- *Local Databases Manager:* MongoDB 6.0 LTS
- *Cloud Computing Service:* AWS RDS-PostgreSQL 10.9
- *DLT*: IOTA Tangle [47]

6.2.1. Experiment 1

The experiment involved inserting 10,000, 100,000, and 1,000,000 records into three different storage systems: a DB, a DaaS, and a DLT platform called IOTA [48]. The insertion process was performed both with and without the FODIT module. When conducted without the FODIT module, the process is referred to as classical insertion.

First, we analyzed classical insertion in the DB. The initial simulation focused on inserting 10,000 records plus 4%, 5%, 7%, and 8% redundant data. Subsequently, using the FODIT module, only 10,000 records were inserted since it prevents duplicate data insertion. However, the FODIT module's filtering process for detecting duplicates still required computation. For the second and third simulations involving 100,000 and 1,000,000 records, respectively, we followed the same procedure as the first simulation.

The process described above for classical insertion in a local DB was also applied to classical insertions in DaaS. Regarding the classical insertion in DLT, the process is a little bit different because it is necessary to incorporate an extra cost associated with the structure used to manage scenarios in which data are sent to the DLT but, in the end, are not recorded in the DLT. Table 3 presents the time required for both classical insertion and insertion with the FODIT module. The table's first column indicates the storage option name, followed by the corresponding data volume. The "Classical" column shows the percentage of duplicate data, with the associated insertion time reported for each percentage. The "FODIT" column consists of three components: (1) the "No Duplicates" sub-column shows the insertion time without duplicate data, (2) the "Filter" sub-column displays the time needed for FODIT to build the filter, (3) the "AuxS" sub-column presents the time required by the structure used in DLT, and (4) the "Total" sub-column represents the sum of these two times. It is important to mention that the time shown pertains to DLT processing.

Table 3. This table displays the times required for classical insertion and insertion using the FODIT module.

Storage	Data	Classical				FODIT			
		Duplicates (4%)	Duplicates (5%)	Duplicates (7%)	Duplicates (8%)	Not Duplicates	Filter	AuxS	Total
DB	10,000	2.9277	2.9367	3.0359	3.0776	2.8172	0.0492	0	2.8664
	100,000	29.2806	29.6053	29.8442	30.1098	27.9589	0.4877	0	28.4466
	1,000,000	288.6968	291.0532	298.5853	303.6195	274.8504	4.9137	0	279.7637
DaaS	10,000	8.9053	8.9604	9.1657	9.2461	8.3792	0.0492	0	8.4284
	100,000	89.7634	90.3721	92.5781	93.7923	84.2795	0.4877	0	84.7672
	1,000,000	878.5982	881.9156	887.2513	891.7634	841.7258	4.9137	0	846.6395
DLT	10,000	345.0824	350.1425	360.5115	378.1140	331.81	0.0492	0.0151	331.8743
	100,000	3485.3322	3536.2650	3605.1156	3656.8780	3367.8715	0.4877	0.3327	3368.6919
	1,000,000	37,268.8992	37,731.7741	38,372.4992	39,054.037	35,835.48	4.9137	2.1324	35,842.5261

To better compare classical insertion with insertion using the FODIT module, we plotted the time results from Table 3. Figure 5 shows comparative graphs for the local DB, where the classical insertion is represented by a green bar chart, while the insertion with the FODIT module is shown in the blue bar chart.

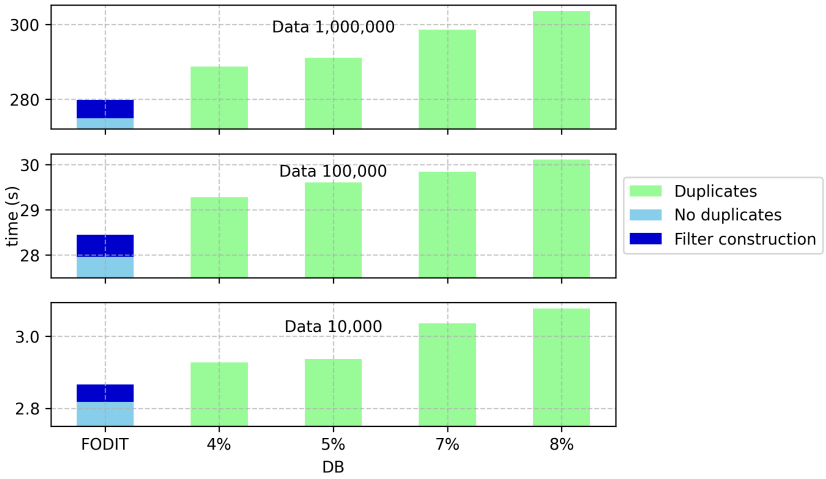

Figure 5. The figure displays classical insertion times compared with insertion times with the FODIT module for the local DB using 4%, 5%, 7%, and 8% of duplicate data.

Figure 6 illustrates comparative graphs for the DaaS cloud service, where the classical insertion is represented by pink bar charts, while the insertion with the FODIT module is shown in purple bar charts.

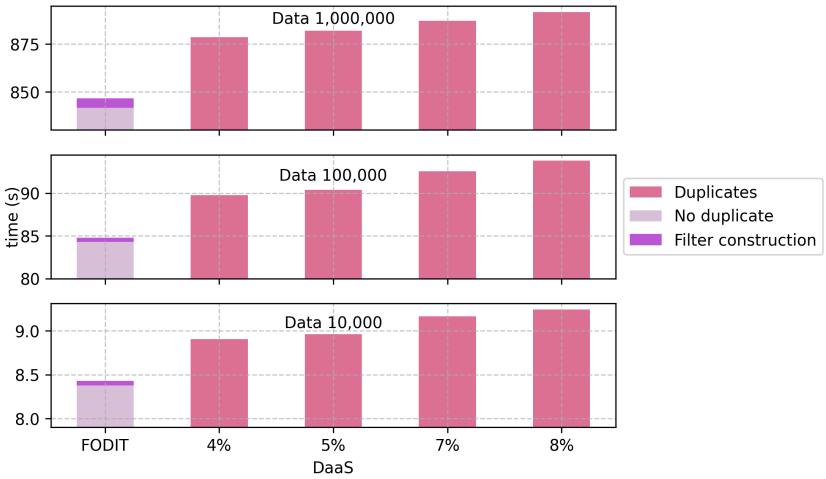

Figure 6. The figure displays classical insertion times compared with insertion times with the FODIT module for the DaaS cloud service using 4%, 5%, 7%, and 8% of duplicate data.

Figure 7 shows comparative graphs for the DLT, where the classical insertion is represented by orange bar charts, while the insertion with the FODIT module is shown in brown bar charts.

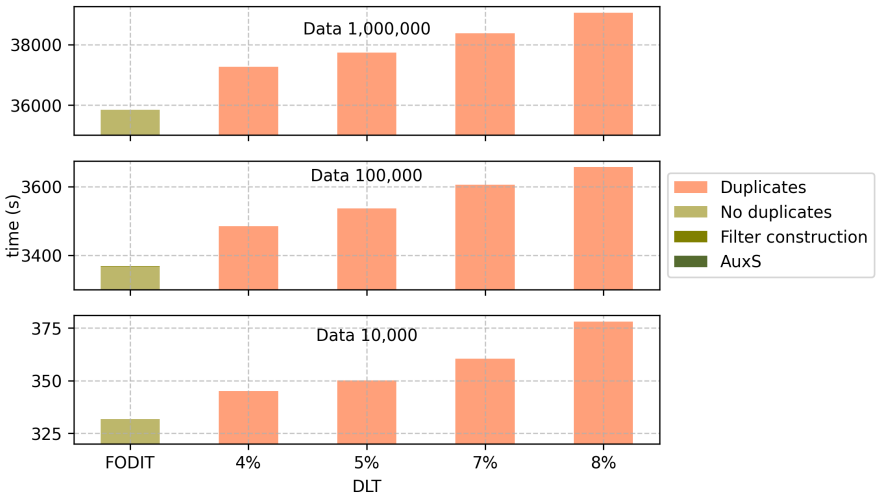

Figure 7. The figure displays classical insertion times compared with insertion times with the FODIT module for the DLT using 4%, 5%, 7%, and 8% of duplicate data. Please note that the AuxS time (olive) and Filter construction time (dark olive) is not visible in the picture due to its very small numerical value compared with the No duplicates time (light green).

As shown in Figures 5–7, the FODIT module's insertion process is faster than classical insertion across all three scenarios (DB, DaaS, and DLT). This time difference becomes more significant as the volume of inserted data increases. Although the FODIT module employs filters and AuxS as alternative data structures to prevent duplicate data insertion, the time needed to build these filters is still shorter than the time required to insert duplicate data using the classical approach.

6.2.2. Experiment 2

Following the query case study methodology, we developed a query process to retrieve data from the datasets inserted in Experiment 1. The experiment involved executing selection queries across three storage systems, as follows: a local DB, the DaaS cloud service, and the DLT. We performed the query process both with and without the FODIT module. Queries executed without the FODIT module are referred to as classical queries.

For the classical query process in a local DB, we conducted three simulations. The first simulation involved executing selection queries on a local DB containing 10,000 records plus 4%, 5%, 7%, and 8% duplicate data. We then executed the same selection queries on the local DB that had been populated using the FODIT module. The second and third simulations followed the same methodology, but with 100,000 and 1,000,000 records, respectively, each including the same 4%-8% range of redundant data. We applied this identical experimental approach to the classical query processes in both the DaaS cloud service and DLT.

Table 4 displays the response times for both classical queries and queries using the FODIT module. The table structure is organized as follows: the first column identifies the storage option, the second column indicates the data volume stored, the "Classical" section shows: duplicate record percentages (4%, 5%, 7%, 8%), corresponding query response times for each percentage in the local DB. The "FODIT" column presents the total query response time when using the FODIT module.

Table 4. This table presents the times required for the classical query process and the query process with the FODIT module to respond to the queries.

	Data	Classical								FODIT
		Duplicates (4%)	Time	Duplicates (5%)	Time	Duplicates (7%)	Time	Duplicates (8%)	Time	Time
DB	10,000	10,400	3.05175×10^{-5}	10,500	3.3140×10^{-5}	10,700	4.0292×10^{-5}	10,800	4.12695×10^{-5}	3.0279×10^{-5}
	100,000	104,000	4.2676×10^{-5}	105,000	4.3869×10^{-5}	107,000	4.4562×10^{-5}	108,000	4.5776×10^{-5}	4.1246×10^{-5}
	1,000,000	1,040,000	5.3153×10^{-5}	1,050,000	5.4527×10^{-5}	1,070,000	5.5936×10^{-5}	1,080,000	5.6523×10^{-5}	5.2193×10^{-5}
DaaS	10,000	10,400	5.8275×10^{-5}	10,500	5.9173×10^{-5}	10,700	6.1293×10^{-5}	10,800	6.2371×10^{-5}	5.4682×10^{-5}
	100,000	104,000	6.2454×10^{-5}	105,000	6.3587×10^{-5}	107,000	6.5729×10^{-5}	108,000	6.6192×10^{-5}	5.8935×10^{-5}
	1,000,000	1,040,000	8.0145×10^{-5}	1,050,000	8.1937×10^{-5}	1,070,000	8.3475×10^{-5}	1,080,000	8.3876×10^{-5}	6.8942×10^{-5}
DLT	10,000	10,400	7.9766×10^{-5}	10,500	7.9802×10^{-5}	10,700	7.9826×10^{-5}	10,800	7.9866×10^{-5}	7.866×10^{-5}
	100,000	104,000	8.8083×10^{-5}	105,000	8.8100×10^{-5}	107,000	8.8257×10^{-5}	108,000	8.8376×10^{-5}	8.4714×10^{-5}
	1,000,000	1,040,000	9.5363×10^{-5}	1,050,000	9.6423×10^{-5}	1,070,000	9.7891×10^{-5}	1,080,000	9.9452×10^{-5}	9.208×10^{-5}

Figure 8 presents comparative graphs of query processing times between the classical approach and the FODIT module. The X-axis represents the data stored, including duplicate data, and the Y-axis represents the time required to perform the query process. The query process with the FODIT module is presented with the first three bar charts, while the classical query process is depicted with bar charts using 4%, 5%, 7%, and 8% of duplicate data.

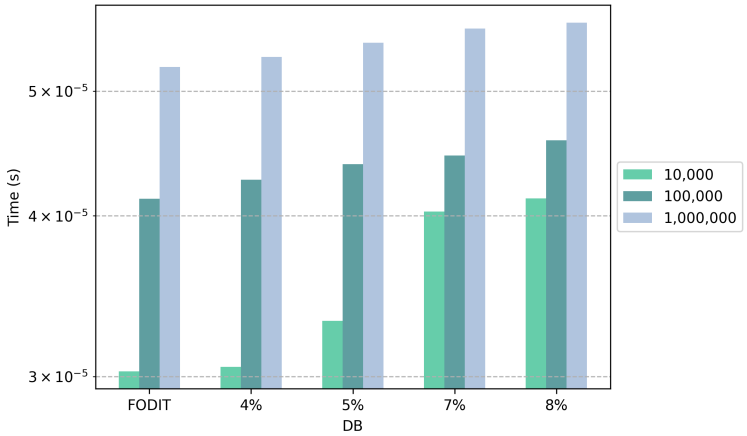

Figure 8. Figure presents comparative graphs between the query process with the FODIT module and the classical query process executed on the local DB using 4%, 5%, 7%, and 8% of duplicate data.

The graphs in Figure 9 show the query process on the DaaS cloud service. The first bar charts illustrate the query process with the FODIT module, and the rest of the bar charts present the classical query process using 4%, 5%, 7%, and 8% of duplicate data.

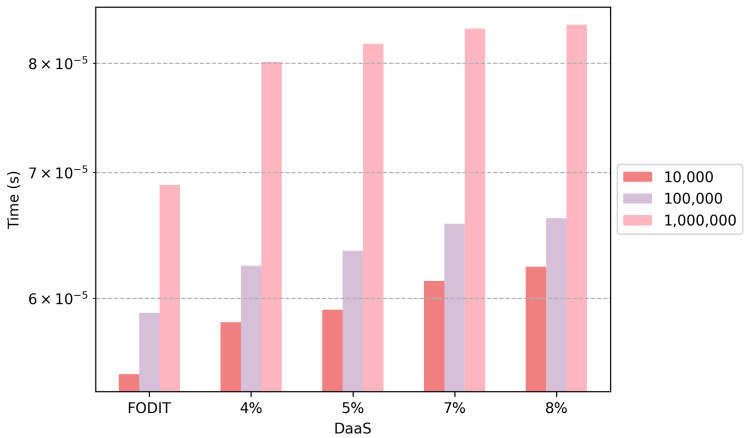

Figure 9. Figure presents comparative graphs between the query process with the FODIT module and the classical query process executed on DaaS cloud service using 4%, 5%, 7%, and 8% of duplicate data.

Figure 10 illustrates the query process in DLT. The bar charts show the query process with the FODIT module, and the next bar charts illustrate the query process using 4%, 5%, 7%, and 8% of duplicate data.

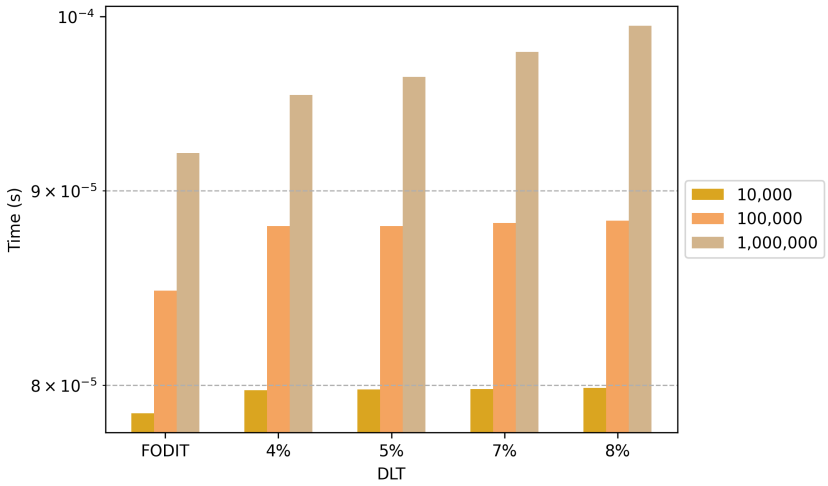

Figure 10. Figure presents comparative graphs between the query process with the FODIT module and the classical query process executed on DLT using 4%, 5%, 7%, and 8% of duplicate data.

The graphs in Figures 8–10 compare query performance across the three storage systems (DB, DaaS, and DLT), demonstrating that the FODIT module consistently outperforms classical query processing in terms of speed. The visualization clearly shows that data access via the FODIT module is significantly faster than traditional query methods. The key advantage of the FODIT module lies in its ability to eliminate unnecessary database searches. By using the filter, clients can immediately determine whether a record exists in the database without performing a full search. This pre-filtering mechanism avoids the computational overhead of querying for non-existent records.

6.3. False Positives Analysis

False positives in the cuckoo filter occur when two elements, x and y, share the same fingerprint and the same buckets. The probability of encountering a repeated fingerprint depends on the number of entries per bucket b and the fingerprint length f and is given by the following equation:

$$1 - \left(1 - \frac{1}{2^f}\right)^{2b} \approx \frac{2b}{2^f} \tag{5}$$

Equation (5) is derived through the following analysis. The probability of obtaining an identical fingerprint is as follows:

$$\frac{1}{2^f}$$

Consequently, the probability of obtaining a different fingerprint is as follows:

$$1 - \frac{1}{2^f}$$

Given that the cuckoo filter permits b entries per bucket, the probability that none of the $2b$ entries (across two buckets) match a specific fingerprint is as follows:

$$\left(1 - \frac{1}{2^f}\right)^{2b}$$

Assuming a desired false positive rate ϵ, if $\frac{2b}{2^f} \leq \epsilon$, then the minimum required fingerprint length is approximately as follows:

$$f \geq \left\lceil \log_2\left(\frac{2b}{\epsilon}\right) \right\rceil = \left\lceil \log_2\left(\frac{1}{\epsilon}\right) + \log_2(2b) \right\rceil \tag{6}$$

Currently, there is no theoretical framework for determining the optimal number of entries per bucket. Nonetheless, based on empirical studies [49], it has been observed that the optimal configuration corresponds to $b = 4$.

For the experiments in this article, the number of entries per bucket is $b = 4$, and the length of the fingerprint is $f = 256$ because we are using the hash function SHA-256 for computing the fingerprint f.

Given a target false positive rate of $\epsilon = 0.01$ (i.e., 0.001%) and a number of entries per bucket $b = 4$, we can compute the minimum required fingerprint length using the formula in Equation (6):

$$f \geq \left\lceil \log_2\left(\frac{2b}{\epsilon}\right) \right\rceil$$

Substituting the following values:

$$f \geq \left\lceil \log_2\left(\frac{2 \cdot 4}{0.01}\right) \right\rceil = \lceil \log_2(800) \rceil \approx \lceil 9.64 \rceil = 10$$

A fingerprint of at least $f = 10$ bits is required to achieve a 1% false positive rate when $b = 4$. Table 5 summarizes multiple false positive rates, as well as the corresponding percentages and the fingerprint lengths.

Table 5. This table presents an analysis of the false positive rate (FPR) regarding the cuckoo filter.

FPR	Percentage	Length f
$\epsilon = 0.01$	1%	10 bits
$\epsilon = 0.001$	0.1%	13 bits
$\epsilon = 0.0001$	0.01%	17 bits
$\epsilon = 0.00001$	0.001%	20 bits
$\epsilon = 0.000001$	0.0001%	23 bits
$\epsilon = 0.0000001$	0.00001%	27 bits
$\epsilon = 0.00000001$	0.000001%	30 bits
$\epsilon = 0.000000001$	0.0000001%	33 bits
$\epsilon = 0.0000000001$	0.00000001%	37 bits
⋮	⋮	⋮
$\epsilon = 6.9 \times 10^{-77}$	1×10^{-75}%	256 bits

On the other hand, if we are interested in computing the FPR, then given a fingerprint length of $f = 256$ bits and $b = 4$ entries per bucket, the false positive rate is approximately as follows:

$$\epsilon \approx \frac{2b}{2^f} = \frac{8}{2^{256}}$$

Since

$$2^{256} \approx 1.16 \times 10^{77}$$

Then

$$\epsilon \approx \frac{8}{1.16 \times 10^{77}} \approx 6.9 \times 10^{-77}$$

The analysis of false positives in the cuckoo filter demonstrates that the false positive rate is tightly governed by the number of entries per bucket b and the fingerprint length

f. Specifically, the false positive rate can be approximated by $\epsilon \approx \frac{2b}{2^f}$, indicating that even modest increases in fingerprint length yield exponential improvements in accuracy. As shown in Table 5, a fingerprint length of $f = 10$ bits is sufficient to ensure a false positive rate of 1% when $b = 4$. However, for applications requiring much lower false positive rates, such as 10^{-6} or lower, fingerprint lengths must increase accordingly, as shown in Figure 11. This practically eliminates the occurrence of false positives in the system, far exceeding the needs of most real-world applications. While there is no theoretical consensus on the optimal number of entries per bucket, empirical studies have shown that $b = 4$ offers a good trade-off between space efficiency and accuracy [49]. Therefore, the chosen parameters $b = 4$ and $f = 256$ provide a robust configuration for high-integrity applications, offering both excellent performance and minimal risk of false positives.

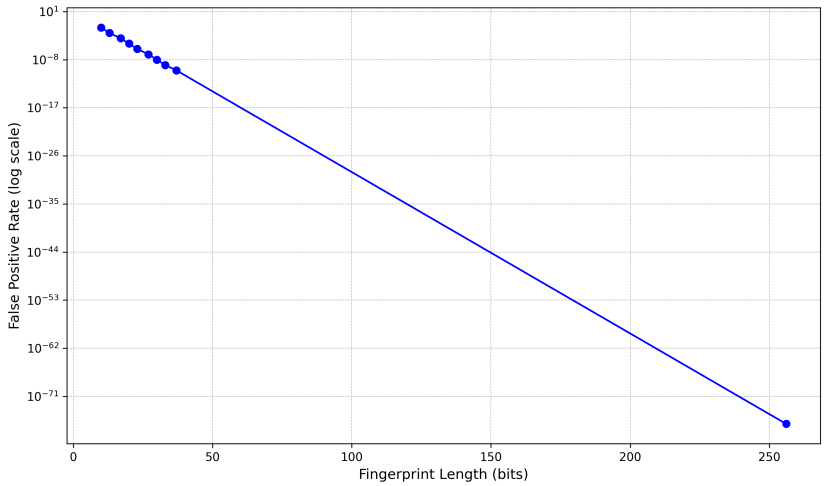

Figure 11. Figure presents comparative graphs of fingerprint length in bits vs. false positive rate on a logarithmic scale.

6.4. Discussion

Based on Experiments 1 and 2, we can conclusively demonstrate that filter-based optimization significantly enhances data management in B5G IoT environments. The experimental results reveal a dual advantage, as follows: (1) storage efficiency improves through automatic duplicate detection and prevention, and (2) query response times accelerate compared to classical approaches. This performance comes from the FODIT module's pre-processing capability, which eliminates redundant storage operations and bypasses unnecessary database scans during queries. Particularly in bandwidth-constrained B5G scenarios where edge devices generate massive data streams, this optimization reduces both storage overhead and network transmission costs. The filter mechanism proves especially valuable for time-sensitive IoT applications like healthcare, where real-time data access is critical. However, these benefits come with a minor computational overhead for filter creation and maintenance, suggesting an optimal use-case balance between write and read operations.

6.5. Study Limitations

The following limitations of the proposed approach have not been addressed, as they fall outside the scope of this paper. However, they will be considered in future work.

The first limitation concerns security. The use of FODIT may, to some extent, centralize access to data that is persistently stored in local databases, cloud services, or a blockchain. This could introduce a vulnerability, as in the event of an attack, some data patterns could potentially be inferred. While this may not be critical in all scenarios—for instance, in public blockchains, data is accessible to any user—it could become a privacy concern in

cases where data confidentiality is required. One possible mitigation would be to encrypt the filter, thereby protecting both read and write operations. However, this is not a trivial solution, as encryption could significantly impact the system's performance, particularly in terms of processing speed.

Another important limitation is the lack of dynamic context awareness. FODIT currently does not adapt the filter size or the time window based on the scenario where it is implemented. Introducing context-driven optimization mechanisms would allow the FODIT to adapt to the needs of different smart environments, improving efficiency and responsiveness.

Finally, although we have proposed a solution to prevent inconsistencies between the filter and the data stored on the blockchain, it would be valuable to explore alternative approaches. This would allow for a comparative analysis and help identify the most suitable data structure for maintaining consistency and reliability.

7. Conclusions

In this work, we have presented FODIT, a module designed to optimize data storage and querying in B5G IoT environments using probabilistic filtering techniques. Implemented with a cuckoo filter, well-suited to the high data volume and performance demands of IoT systems, FODIT prevents the storage of duplicate data and enables efficient membership queries without directly accessing the underlying storage. Our evaluation shows that these features lead to significant improvements in both storage efficiency and query overhead reduction.

In future work, we aim to enhance the scalability of the filtering mechanism and extend FODIT with heuristics to improve the relevance and quality of stored information, further adapting the module to the evolving needs of B5G IoT environments. Additionally, we consider it essential to explore alternative data structures to ensure consistency with the DLT. In this context, incorporating context-aware mechanisms could help optimize performance by adapting dynamically to each scenario, particularly under high-density B5G IoT deployments. Finally, we also plan to investigate potential security challenges, ensuring that any improvements do not compromise system performance.

Author Contributions: Conceptualization, B.R.-C. and F.J.Q.-R.; methodology, B.R.-C., F.J.Q.-R., J.A.-P. and J.Z.-M.; software, B.R.-C., F.J.Q.-R. and J.Z.-M.; validation, B.R.-C. and F.J.Q.-R.; investigation, B.R.-C. and F.J.Q.-R.; resources, F.J.Q.-R. and J.A.-P.; writing—original draft preparation, B.R.-C. and F.J.Q.-R.; writing—review and editing, J.A.-P. and J.Z.-M.; visualization, F.J.Q.-R. and J.A.-P.; supervision, J.A.-P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by the Research Project TED2021-132073B-I00 PHADAS, funded by MCIN/AEI/10.13039/501100011033 and NextGenerationEU/PRTR, and by the University of Jaén through the research support operational plan via action 8a (to the first author).

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

B5G	Beyond 5G
IoT	Internet of Things
DB	Database

DaaS	Database as a Service
DLTs	Distributed Ledger Technologies
FFBF	Fuzzy-Folded Bloom Filter
FODIT	Filter-Based Optimization for Data Storage in IoT
mMTC	Massive Machine-Type Communications
PDS	Probabilistic Data Structure
CF	Cuckoo Filter
IF	Insertion Filter function
SF	Search Filter function
DF	Deletion Filter function

References

1. Ahmed, E.; Yaqoob, I.; Gani, A.; Imran, M.; Guizani, M. Internet-of-things-based smart environments: State of the art, taxonomy, and open research challenges. *IEEE Wirel. Commun.* **2016**, *23*, 10–16. [CrossRef]
2. Elmustafa, S.A.A.; Mujtaba, E.Y. Internet of things in smart environment: Concept, applications, challenges, and future directions. *World Sci. News* **2019**, *134*, 1–51.
3. Ben-Daya, M.; Hassini, E.; Bahroun, Z. Internet of things and supply chain management: A literature review. *Int. J. Prod. Res.* **2019**, *57*, 4719–4742. [CrossRef]
4. Babangida, L.; Perumal, T.; Mustapha, N.; Yaakob, R. Internet of Things (IoT) based activity recognition strategies in smart homes: A review. *IEEE Sens. J.* **2022**, *22*, 8327–8336. [CrossRef]
5. Babayigit, B.; Abubaker, M. Industrial internet of things: A review of improvements over traditional scada systems for industrial automation. *IEEE Syst. J.* **2023**, *18*, 120–133. [CrossRef]
6. Rejeb, A.; Rejeb, K.; Treiblmaier, H.; Appolloni, A.; Alghamdi, S.; Alhasawi, Y.; Iranmanesh, M. The Internet of Things (IoT) in healthcare: Taking stock and moving forward. *Internet Things* **2023**, *22*, 100721. [CrossRef]
7. Muñoz-Higueras, C.; Serradilla-Gil, A.M.; Moreno-Colmenero, P.; Quesada-Real, F.J. Integrating IoT and DLT to Enhance Patient Wait Time Traceability in Radiotherapy Oncology. In *Proceedings of the International Conference on Ubiquitous Computing and Ambient Intelligence, Belfast, UK, 2–5 December 2014*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 932–942.
8. Ramos-Cruz, B.; Quesada-Real, F.J.; Rodriguez-Garcia, M.; Andreu-Pérez, J.; Martínez, L. Combining Distributed Ledger Technologies and Differentially Private Sketching Techniques for Securing Health Monitoring. In *Proceedings of the International Conference on Ubiquitous Computing and Ambient Intelligence, Belfast, UK, 2–5 December 2014*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 920–931.
9. Stojkoska, B.L.R.; Trivodaliev, K.V. A review of Internet of Things for smart home: Challenges and solutions. *J. Clean. Prod.* **2017**, *140*, 1454–1464. [CrossRef]
10. Zanella, A.; Bui, N.; Castellani, A.; Vangelista, L.; Zorzi, M. Internet of things for smart cities. *IEEE Internet Things J.* **2014**, *1*, 22–32. [CrossRef]
11. Mehmood, Y.; Ahmad, F.; Yaqoob, I.; Adnane, A.; Imran, M.; Guizani, S. Internet-of-things-based smart cities: Recent advances and challenges. *IEEE Commun. Mag.* **2017**, *55*, 16–24. [CrossRef]
12. Attar, H.; Alghanim, M.; Ababneh, J.; Rezaee, K.; Alrosan, A.; Deif, M.A. B5g applications and emerging services in smart IoT environments. *Int. J. Crowd Sci.* **2025**, *9*, 79–95. [CrossRef]
13. Lessi, C.C.; Gavrielides, A.; Solina, V.; Qiu, R.; Nicoletti, L.; Li, D. 5G and beyond 5G technologies enabling industry 5.0: Network applications for robotics. *Procedia Comput. Sci.* **2024**, *232*, 675–687. [CrossRef]
14. Qi, Q.; Chen, X.; Zhong, C.; Zhang, Z. Integrated sensing, computation and communication in B5G cellular Internet of Things. *IEEE Trans. Wirel. Commun.* **2020**, *20*, 332–344. [CrossRef]
15. Uddin, H.; Gibson, M.; Safdar, G.A.; Kalsoom, T.; Ramzan, N.; Ur-Rehman, M.; Imran, M.A. IoT for 5G/B5G applications in smart homes, smart cities, wearables and connected cars. In Proceedings of the 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Limassol, Cyprus, 11–13 September 2019; IEEE: Piscataway, NY, USA, 2019; pp. 1–5.
16. Popovski, P.; Trillingsgaard, K.F.; Simeone, O.; Durisi, G. 5G wireless network slicing for eMBB, URLLC, and mMTC: A communication-theoretic view. *IEEE Access* **2018**, *6*, 55765–55779. [CrossRef]
17. Pokhrel, S.R.; Ding, J.; Park, J.; Park, O.; Choi, J. Towards enabling critical mMTC: A review of URLLC within mMTC. *IEEE Access* **2020**, *8*, 131796–131813. [CrossRef]
18. Khan, B.S.; Jangsher, S.; Ahmed, A.; Al-Dweik, A. URLLC and eMBB in 5G industrial IoT: A survey. *IEEE Open J. Commun. Soc.* **2022**, *3*, 1134–1163. [CrossRef]

19. Kong, X.; Wu, Y.; Wang, H.; Xia, F. Edge computing for internet of everything: A survey. *IEEE Internet Things J.* **2022**, *9*, 23472–23485. [CrossRef]
20. Xu, W.; Yang, Z.; Ng, D.W.K.; Levorato, M.; Eldar, Y.C.; Debbah, M. Edge learning for B5G networks with distributed signal processing: Semantic communication, edge computing, and wireless sensing. *IEEE J. Sel. Top. Signal Process.* **2023**, *17*, 9–39. [CrossRef]
21. Rafique, W.; Barai, J.; Fapojuwo, A.O.; Krishnamurthy, D. A survey on beyond 5g network slicing for smart cities applications. *IEEE Commun. Surv. Tutor.* **2024**, *27*, 595–628. [CrossRef]
22. Morocho-Cayamcela, M.E.; Lee, H.; Lim, W. Machine learning for 5G/B5G mobile and wireless communications: Potential, limitations, and future directions. *IEEE Access* **2019**, *7*, 137184–137206. [CrossRef]
23. Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
24. Fan, B.; Andersen, D.G.; Kaminsky, M.; Mitzenmacher, M.D. Cuckoo filter: Practically better than bloom. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, Sydney, Australia, 2–5 December 2014; pp. 75–88.
25. Moya, F.; Quesada, F.J.; Martínez, L.; Estrella, F.J. Phonendo: A Platform for Publishing Wearable Data on Distributed Ledger Technologies. *Wirel. Netw.* **2024**, *30*, 6507–6521. [CrossRef]
26. Wu, W.; Zhou, C.; Li, M.; Wu, H.; Zhou, H.; Zhang, N.; Shen, X.S.; Zhuang, W. AI-native network slicing for 6G networks. *IEEE Wirel. Commun.* **2022**, *29*, 96–103. [CrossRef]
27. Alsenwi, M.; Tran, N.H.; Bennis, M.; Pandey, S.R.; Bairagi, A.K.; Hong, C.S. Intelligent resource slicing for eMBB and URLLC coexistence in 5G and beyond: A deep reinforcement learning based approach. *IEEE Trans. Wirel. Commun.* **2021**, *20*, 4585–4600. [CrossRef]
28. Singh, A.; Garg, S.; Kaur, R.; Batra, S.; Kumar, N.; Zomaya, A.Y. Probabilistic data structures for big data analytics: A comprehensive review. *Knowl.-Based Syst.* **2020**, *188*, 104987. [CrossRef]
29. Bonomi, F.; Mitzenmacher, M.; Panigrahy, R.; Singh, S.; Varghese, G. An improved construction for counting bloom filters. In Proceedings of the Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, 11–13 September 2006; Proceedings 14; Springer: Berlin/Heidelberg, Germany, 2006; pp. 684–695.
30. Naor, M.; Yogev, E. Sliding bloom filters. In Proceedings of the International Symposium on Algorithms and Computation, Hong Kong, China, 16–18 December 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 513–523.
31. Singh, A.; Garg, S.; Batra, S.; Kumar, N.; Rodrigues, J. Bloom filter based optimization scheme for massive data handling in IoT environment. *Future Gener. Comput. Syst.* **2017**, *82*, 440–449. [CrossRef]
32. Podnar Zarko, I.; Pripuzic, K.; Serrano, M.; Hauswirth, M. IoT data management methods and optimisation algorithms for mobile publish/subscribe services in cloud environments. In Proceedings of the 2014 European Conference on Networks and Communications (EuCNC), Bologna, Italy, 23–26 June 2014; pp. 1–5.
33. Jeong, J.; Joo, J.W.J.; Lee, Y.; Son, Y. Secure Cloud Storage Service Using Bloom Filters for the Internet of Things. *IEEE Access* **2019**, *7*, 60897–60907. [CrossRef]
34. Singh, A.; Garg, S.; Kaur, K.; Batra, S.; Kumar, N.; Raymond Choo, K. Fuzzy-Folded Bloom Filter-as-a-Service for Big Data Storage in the Cloud. *IEEE Trans. Ind. Inform.* **2019**, *15*, 2338–2348. [CrossRef]
35. Pintilei, M.A.; Schreiner, C.; Socotar, D. Exploring Data Compression: Solutions to Optimize Efficiency and Improve Performance. In Proceedings of the 2024 IEEE International Conference And Exposition On Electric And Power Engineering (EPEi), Iasi, Romania, 17–19 October 2024; pp. 280–286. [CrossRef]
36. Yu, J.; Shen, W.; Zhang, X. Cloud storage auditing and data sharing with data deduplication and private information protection for cloud-based EMR. *Comput. Secur.* **2024**, *144*, 103932. [CrossRef]
37. Aladiyan, A. Efficient Data Structures and Algorithms for Cloud Computing Platforms. In Proceedings of the 2024 4th International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE), Greater Noida, India, 14–15 May 2024; pp. 1717–1721. [CrossRef]
38. Idrees, S.K.; Azar, J.; Couturier, R.; Idrees, A.K.; Gechter, F. SZ4IoT: An adaptive lightweight lossy compression algorithm for diverse IoT devices and data types. *J. Supercomput.* **2025**, *81*, 392. [CrossRef]
39. Altowaijri, S.M. Efficient Data Aggregation and Duplicate Removal Using Grid-Based Hashing in Cloud-Assisted Industrial IoT. *IEEE Access* **2024**, *12*, 145350–145365. [CrossRef]
40. Kumar, M.; Singh, A. Bloom filter empowered smart storage/access in IoMT [edge-fog-cloud] hierarchy for health-care data ingestion. *Concurr. Comput. Pract. Exp.* **2024**, *36*, e8012. [CrossRef]
41. Cooper, J.; James, A. Challenges for database management in the internet of things. *IETE Tech. Rev.* **2009**, *26*, 320–329. [CrossRef]
42. Wu, J.; Ping, L.; Ge, X.; Wang, Y.; Fu, J. Cloud storage as the infrastructure of cloud computing. In Proceedings of the 2010 International Conference on Intelligent Computing and Cognitive Informatics, Kuala Lumpur, Malaysia, 22–23 June 2010; IEEE: Piscataway, NY, USA, 2010; pp. 380–383.
43. Nakamoto, S. Bitcoin Whitepaper. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 25 June 2025).

44. Farahani, B.; Firouzi, F.; Luecking, M. The convergence of IoT and distributed ledger technologies (DLT): Opportunities, challenges, and solutions. *J. Netw. Comput. Appl.* **2021**, *177*, 102936. [CrossRef]
45. Quesada-Real, F.J.; Moya-Pérez, F.; Rodriguez-Garcia, M.; Dutta, B. A Transparent and Ecologically Sustainable DLT-based Approach for Tendering Processes. *J. Univers. Comput. Sci. JUCS* **2025**, *31*, 277–297. [CrossRef]
46. Moya, F.; Quesada, F.J.; Martínez, L.; Estrella, F.J. CertifioT: An IoT and DLT-based solution for enhancing trust and transparency in data certification. In Proceedings of the International Conference on Ubiquitous Computing and Ambient Intelligence, Riviera Maya, Mexico, 25 November 2023; Springer: Berlin/Heidelberg, Germany, 2023; pp. 127–138.
47. Popov, S. The tangle. *White Pap.* **2018**, *1*, 30.
48. Popov, S.; Lu, Q. IOTA: Feeless and free. *IEEE Blockchain Tech. Briefs* **2019**, *6*, 964.
49. Bose, P.; Guo, H.; Kranakis, E.; Maheshwari, A.; Morin, P.; Morrison, J.; Smid, M.; Tang, Y. On the false-positive rate of Bloom filters. *Inf. Process. Lett.* **2008**, *108*, 210–213. [CrossRef]