CHAPTER 4

METADATA MANAGEMENT MODEL

4.1 INTRODUCTION

Owing to the fast and unpredictable increase in the number of files stored in the cloud servers, metadata also grows considerably and hence finding the metadata of user’s requirement with reduced latency becomes challenging. This gives rise to the need to provide an effective and efficient solution to meet the user’s requirement. Metadata provides the information about the data which describes the layout and attributes of the data stored. This includes attributes such as timestamps, access control information, file size, and also information on how to locate and store the metadata in the MDS. As metadata file holds the information about a file stored in the data servers, the issue of mapping the query to the corresponding metadata comes to the force. Several mapping techniques like hash-based mapping, table-based mapping, tree-based mapping and bloom filter mapping have been amply used efficiently in various flavours. However as the data storage in the cloud era progress towards zetabytes, suitability of the existing mapping techniques becomes a major issue. The suitability of existing mapping functions in various distributed databases have not always been useful due to higher look-up cost, migration cost, memory overhead, directory operations and scalability. While metadata usage becomes crucial, concurrently it becomes challenging due to the new factors. In order to overcome these negative effects while still working with metadata, a novel metadata model along with a special mapping data structures are the need of the hour.
Fluctuating load in cloud, renders metadata itself dynamic and hence handling metadata operations efficiently is an important aspect of the file system. Thus managing the dynamic metadata using a suitable data structure is critical in scaling the overall performance. This chapter explores a novel method of metadata management mechanism using a specially designed bloom filter called CBF, a probabilistic data structure. Use of CBF offers efficient retrieval of data stored across various data storage servers located in geographically dispersed locations. CBF comprises of GBF and LBF. This chapter also proposes scalable and decentralized metadata placement schemes using GBF which scales up the metadata performance by judiciously distributing heavy management workloads among multiple metadata servers. The proposed model efficiently handles the dynamic metadata with effective update mechanism using LBF.

4.2 BLOOM FILTER

Bloom filter is a space-efficient probabilistic data structure that was conceived by Burton H. Bloom in 1970 (B. Bloom, 1970). The structure of a bloom filter offers a compact probabilistic way to represent a set that can result in false positives, i.e., claiming non-existent element to be part of the set. However, bloom filter never results in false negatives, i.e., reporting an existing element to be not present in the set. This makes bloom filters useful for different kinds of applications and tasks that involve lists and sets.

4.2.1 Existing Bloom Filter

Bloom filter is a data structure that supports set membership queries (Sasu et al 2009). The basic operations involve adding elements to the set and querying for element membership in the probabilistic set representation. The basic bloom filter does not support the removal of elements. However, recently the enhanced bloom filter has been introduced that supports deletion
operation as well (Broder & Mitzenmacher2004). The accuracy of a bloom filter depends on three factors, viz, the size of the filter, the number of hash functions used in the filter, and the number of elements added to the filter. The more the elements added to a bloom filter, the higher the probability of the query operation reporting false positives.

4.2.1.1 Structure of bloom filter

A Bloom filter is an array of \( m \) bits for representing a set \( S = \{x_1, x_2, \ldots, x_n\} \) of \( n \) elements. Initially all the bits in the bloom filter are set to zero. The key idea is to use \( k \) hash functions, \( h_i(x) \), \( 1 \leq i \leq k \) to map items \( x \in S \) to random numbers uniform in the range \( 1, \ldots, m \). An element \( x \in S \) is inserted into the filter by setting the bits \( h_i(x) \) to one for \( 1 \leq i \leq k \) and guaranteed not to be a member if any bit \( h_i(x) \) is not set. The weak point of bloom filter is the possibility for a false positive. False positives are elements that are not a part of \( S \) but are reported as a belonging in the set by the filter.

Figure 4.1 represents the structure of the bloom filter where \( X \) is an input to the bloom filter. The input is processed by the hash function for various files \( x, y, z \), where \( k \) represents the number of hash functions used.

![Figure 4.1 Structure of a standard bloom filter](image-url)
The bloom filter consists of a bit string of length $m$. The outputs of the hash functions are noted and corresponding bit positions are set to 1. When the bloom filter is used for checking, the input is processed by the hash functions and the positions are verified for the corresponding bit value as 1. If any one of the positions is zero, the bloom filter reports that the element is not in the set. Sometimes it may also end up with erroneous results where the bits are set which belongs to other inputs. Dillinger & Manolios (2004), the false positive is represented by $f_p = (1 - e^{-\frac{kn}{m}})powered$. The number of hash functions $k$ is derived based on the function $k = (\frac{m}{n}) ln 2$ where $'m'$ represents the length of the bloom filter and $'n'$ is the number of elements to be inserted in the bloom filter. The false positives can efficiently be handled by increasing the number length of the bloom filter and also by increasing the number of hash functions in the bloom filter. As the bloom filters are represented using bit-vector, it can be successfully used in various bitwise operations and can be implemented easily for various applications in various domains like networking, natural language processing, database applications etc. In the year 2011 adaptive and scalable metadata management scheme is introduced, using bloom filter (Hua et al 2008, 2011). Bloom filter plays a critical role in the proposed ‘MaaS’ in cloud computing. Essentially the task of using bloom filter in ‘MaaS’ is to reduce the latency while satisfying the user need efficiently.

4.2.2 Proposed Cloud Bloom Filter

Proposed CBF specially designed for cloud data storage and retrieval, plays a major role in the proposed metadata model. CBF proposed in the current research comprises of two levels of bloom filters viz, 1.GBF 2. LBF.
4.2.2.1 Structure of cloud bloom filter

CBF comprises two levels of bloom filters. The first level is the global bloom filter which takes care of location of metadata file in the metadata server. The second level is the local bloom filter with “d” layers where “d” indicates the value of number of metadata attributes. Each layer in the LBF holds the information about the attributes of the metadata. GBF is effectively used at metadata placement and lookup layer and LBF at metadata update layer. GBF is designed in such a way that the query is mapped to the exact metadata server, which in turn maps the query to the target data server, leading to the fast retrieval of data from the large scale data servers. The design of GBF results in avoiding costly disk lookups for non-existent metadata files in databases. Avoiding costly disk lookups considerably increases the performance of a database. Hence, GBF along with the proposed model provides a very high accuracy during the uploading and downloading process.

LBF, a multi layered bloom filter in the metadata server effectively reduces the time taken to update the respective file with limited updating overhead. LBF takes care of supply of the updated data instead of stale data which leads to erroneous results. LBF along with the proposed model functions effectively so that the time and bandwidth involved in the update process is reduced.

4.3 METADATA MANAGEMENT ARCHITECTURE

The metadata management architecture is a novel scheme, responsible for handling three different issues viz. (i) a protocol for creating the metadata (ii) a mechanism for placing metadata using GBF. (iii) a protocol for handling the update mechanism of dynamic metadata using LBF. The metadata management architecture is as shown in Figure 4.2.
Metadata management model involves in solving issues dealing with mapping of query to the data via metadata and consistency of metadata. These issues are effectively resolved using GBF and LBF in the proposed architecture. The metadata management model consists of three different sub layers. 1. Metadata Creation layer 2. Metadata Placement and Lookup layer 3. Metadata Update layer.

4.4 METADATA CREATION

The metadata creation layer deals with the creation of metadata file and the attributes are selected using Dublin Core Metadata Initiative (DCMI) standard, a general metadata element set for describing all types of resources. When a user uploads the file, the file is analyzed and based on DCMI design...
the relevant attributes are extracted and stored as a metadata file. The proposed metadata architecture supports file name and keyword based data retrieval. When a user logs in and uploads a file the metadata attributes like Filename, File Owner, File size, File type, Date of modification, Time of modification and Keywords are extracted.

4.4.1 Protocol for Selecting Metadata Attributes and Creation of Metadata

The protocol uses DCMI standard, which defines a set of elements that describes resources. The original DCMI has 13 core elements like Title, Creator, Subject, Description, Publisher, Contributor, Date, Type, Format, Identifier, Source, Language, Relation, Coverage, and Rights. The DCMI standard used in this proposed model describes the document in a simple and concise manner. The elements in the Dublin core are optional and can be presented in any order. As DCMI can be used for variety of purposes, there are hundreds of projects worldwide that use the Dublin core either for cataloging or to collect data from the internet. In this chapter the DCMI is refined by narrowing the scope of an element by identifying the schema used in representing an element value. Hence the metadata schema has expanded beyond by simply maintaining the Dublin core metadata element set that describes itself as dedicated to promoting the widespread adoption of metadata standards and developing specialized metadata vocabularies like keyword extraction for discovery systems. The special attribute keyword is extracted using tfidf algorithm, which plays an important role in reducing the search space and the usage is further carried out in next chapter. The purpose of keyword extraction is to identify a set of words representative of a document. To achieve this, a simple statistical approach is used. In order to exploit the properties of a document and of a repository, there comes a need to find the comparable measures. One of the simple weighting method is
TF * IDF. The TF part imparts a higher score to a document that has more occurrences of a term, while the IDF part is to penalize words that are popular in the whole collection. The keyword extraction is conducted exploiting the TF * IDF weight of the term. It is calculated according to the formula:

\[ TF * IDF = W_i * \log \left( \frac{N}{N_i} \right) \]

where,

- \( W_i \) - Frequency of a term in the given document
- \( N \) - Total number of documents in the collection
- \( N_i \) - Number of documents containing that word.

Several preprocessing steps such as stopword removal and stemming are carried out before calculating the TF*IDF. The words which hold the maximum TF-IDF value is taken as keyword and is considered as a keyword attribute in the metadata file.

### 4.4.1.1 Stopword removal

The uploaded file is preprocessed by removing the stop words like dot, comma etc., and the special characters are also removed. After removing the stopwords, the file is written into the temporary file and is processed for stemming.

### 4.4.1.2 Stemming

Temporary file is given as input to the stemming procedure which process the temporary file word by word and removes the tenses, adverbs for each word. For example the word ‘marked’ will be stemmed as ‘mark’ and ‘classification’ will be stemmed as ‘classify’. The process uses porter-stemmer (Porter et al 2011) algorithm in an efficient way.
4.5 METADATA PLACEMENT AND LOOKUP LAYER

GBF helps in searching metadata file efficiently by having a specific protocol for placing and searching the metadata file. This section explains about the protocol for metadata placement and lookup using global bloom filter.

4.5.1 Protocol for Metadata Placement and Lookup using GBF

A specific protocol is designed for placing the metadata file in the MDS which includes GBF as the main component. GBF is a single layered bloom filter, representing the whole of the metadata servers. The protocol for placing the metadata files in the original location and its replication locations are defined by GBF. The global bloom filter works at the time of uploading the file to the data server as well as while downloading the file from the data server.

At the time of uploading the file, the name of the file is given as input to the global bloom filter. The output of the GBF provides the location of metadata file. If the file exists it will update the information else it will create the metadata and store it in the respective locations based on the output of GBF. While downloading a file, the request from the user may take either (1) File name as input (2) Keyword as input. When the user requests a data the input is given to GBF in order to find the location of the metadata server. The hashed output from the global bloom filter is used to find out the location of the metadata server, and the input is forwarded to the respective metadata server which in turn forwards the query to the respective data server.

4.5.2 Design Structure of GBF

Global Bloom Filter defines the protocol for placing the metadata file in the metadata server and its replication locations. The GBF maintains the
information about where the replica is available and from which location the metadata file can be accessed. The probabilistic structure of the global bloom filter is shown in Figure 4.3.

![Figure 4.3 Structure of global bloom filter](image)

GBF consists of a bit string of varying length based on the number of hashing functions. In this work, three hash functions are used ($h_1$, $h_2$ & $h_3$) i.e., the value of $k$ is 3. The file name is given as an input to the 3 independent hash functions of the GBF. Let the output of the hash values be $x$, $y$ and $z$. Based on the hashing output, the corresponding bits are set to 1. For example, let the bloom filter be a bit string of length 16. The bit positions are numbered 0 to 15, from right to left. Three hash functions, $h_1$, $h_2$, and $h_3$ are MD5, SHA-1 and SHA-2 respectively. The bloom filter starts with empty bit string with all bits set to zero. When adding an element the values of $h_1$ through $h_3$ are calculated for the element, and corresponding bit positions are set to one. In practice, hash functions yielding sufficiently uniformly distributed outputs, such as MD5, SHA-1 and SHA-2 are useful for most probabilistic filter purposes where the major characteristic are irreversible and hence the collision range is reduced. A collision is a state where the bits are repeated in the hashed output and hence all bits to be looked up were previously set. This results in collision. One reason behind the probability of false positives in the bloom filter is due to the effect of collision.
4.5.2.1 Justification of hashing algorithms

The proposed model uses MD5, SHA-1 and SHA-2 for hashing schemes. The algorithms MD5 hashing is 128 bits wide, and SHA-1 is 160 and SHA-2 is 160 bits wide. With the above bit values the probability of colliding if all three hashes collides is

\[ P(\text{Collision}) = P(\text{MD5 collides}) \times P(\text{SHA-1 collides}) \times P(\text{SHA-2 collides}) \]

where,

\[ P(\text{MD5 collides}) = \frac{1}{2^{128}}, \quad P(\text{SHA-1 collides}) = \frac{1}{2^{160}} \quad \text{and} \quad P(\text{SHA-2 collides}) = \frac{1}{2^{160}}. \]

Hence the probability that the independent hashing collides for two inputs is negligible. Thus the collisions in independent hashing are independent for same input. That is, that the hashing algorithms behave differently enough that collision in MD5 are no more likely than usual to collide in SHA-1 and SHA-2.

Let us consider the number of collisions in cloud bloom filter by \(C_f\). The probability that there exists no false positive due to collision at all, is given by \(P(f_p) = 1 - C_f\). The probability that there exists false positives due to increase in number of collisions is \(P(f_p > 0) = 1 - P(f_p)\). Probability computation for having no false positives due to filter collisions at all for \(i\) elements is approximately the product of these given by \(P(f_p = 0) = \prod_{i=0}^{n-1} (1 - C_f(i))\) where “\(i\)” is a factor of collision and ranges from 0 to \(n-1\) where ‘n’ is the number of elements in the bloom filter. Hence by adding all the values of \(i\), the expected number of false positives due to filter collision is
\[ E(f_p) = \sum_{i=0}^{n-1} C_{f(i)} \]. Thus the equation states that as “i” the number of collision elements increases, the false positive also increases. The proposed model uses independent multiple hashing methodologies, and also as the hash functions used are perfect hashing whose values do not repeat for different inputs hence the collision element “i” is less in this research.

Thus the proposed model is effective in reducing the false positives due to the effect of collision in hashing methodologies.

4.5.3 Metadata Placement using GBF

The metadata placement layer is responsible for placing the metadata file in respective locations using GBF. The layer proposes a novel mechanism for placing the metadata file. The metadata file is placed in primary location and also replicated in two other locations (Rahmana et al 2008). Replicating metadata files deals with two issues viz. 1. nearby lookup which reduces the time taken to reach the destination fast. 2. fault tolerance. The location of the replication is determined by the global bloom filter. The output of each hashing in GBF represents the independent locations of replica placing. The mapping of hash values to integer value is done using CRC32 polynomial and is used for providing the locations of the metadata server in order to store the file during uploading and to check the availability of the file during lookup process. The output of MD5 gives the primary location and output from SHA-1 & SHA-2 gives the replica location.

Figure 4.4 illustrate the placement mechanism of a created metadata file using GBF.
The metadata placement layer takes filename as input. When an input is forwarded to the GBF, the GBF process the input by making use of the independent hashing functions. The output of these hash functions are converted into integers using CRC32 algorithm which is explained in function Maploc. Using the output of GBF the metadata file is placed in the respective locations. Thus making use of these protocols the placement and lookup becomes easier. Algorithm 1 explains the working process of metadata placement using GBF.
Algorithm 1 : Metadata Placement using Global Bloom Filter

Input: File name of file “F” is given as input to GBF
Output: Primary location and Replica location

Insert (filename)

int B[m][i] = 0 /* i ranges from 1 to m*/
/* Initialize the bits in GBF=0 of length m bits*/

for j : 1 . . . k do /* hashing the value of filename “M”, k-times */
    
    B[m][x] = h_j(filename) /* B[m][x] array of bloom filter & ‘x’ is the bit to be set using hash functions*/

Maploc (B[m][x]);

for k : 1 . . . t do /* Hash function of length “t” bits */
    
    if (B[m][x] == 0)
    {
        B[m][x]=1;
        count_m++; /* Increment the GBF counter */
        Maploc(B[m][x]);
    } else
    {
        count_m++;
        Maploc(B[m][x])
    }

Maploc(B[m][x])

{ 
L ← MakeCRC[B[m][x]] ; /* Built in function “MakeCRC” */
MDS_L ← F ;
}
4.5.4 Metadata Lookup using GBF

The metadata lookup layer takes filename or keyword as input. If a keyword is provided as input, the request is forwarded to the metadata lookup table (MLT) which consists of the filename and the keyword of a respective file. The file name is then directed as an input to the global bloom filter. The global bloom filter will in turn forward the query to the respective metadata server. An MLT is represented as $F : k \rightarrow f$ where ‘F’ is a search function and ‘$k$’ is the keyword which maps the file ‘$f$’. When querying the file the data structure with $x \in X$ is checked first, whether the function $f(x) \in h(x)$ is satisfied where $f(x)$ represents the input file and $h(x)$ is the hash function of the respective input file. The working model of GBF is represented as

$$\forall(x \in X) = \begin{cases} f(x) \in h(x), & \text{FileExists} \\ f(x) \notin h(x), & \text{FilenotExists} \end{cases}$$

Figure 4.5 illustrates the metadata lookup mechanism of a given file. Algorithm 2 explains the working model of metadata lookup using GBF.
Algorithm 2: Metadata lookup process using Global Bloom Filter:
Input: File name of file “F” is given as input to GBF
Output: MDS location and Replica Location
Insert (F)
\[
\begin{align*}
\text{int } B_m[i] &= 0 \quad /* i ranges from 1 to m */ \\
\text{for } j : 1 \ldots k /* hashing the value of filename, k-times */ \\
& \quad \{ \\
& \quad \quad B_m[x] = h_j(\text{filename}) \quad /* x is the bit to be set using hash functions*/ \\
& \quad \} \\
\text{for } k: 1 \ldots t \text{ do } /* Hash function of length “t” bits */ \\
& \quad \{ \\
& \quad \quad \text{if } (B_m[x] == 1) \\
& \quad \quad \{ \\
& \quad \quad \quad \text{output} = \text{“file exists”} \\
& \quad \quad \quad \text{Maploc}(B_m[x]); \\
& \quad \quad \quad \text{count}_m--; /* Decrement the GBF counter */ \\
& \quad \quad \} \text{ else} \\
& \quad \quad \quad \text{output} = \text{“file doesn’t exists”} \\
& \quad \} \\
\text{Maploc}(B_m[x]) \\
& \{ \\
& \quad L \leftarrow \text{MakeCRC}[B_m[x]]; /* Built in function “MakeCRC” */ \\
& \quad \text{MDS}_L \leftarrow F; \\
\}
\]

4.6 METADATA UPDATE

The metadata update layer is used for maintaining the data consistency across the MDS using LBF. The space occupied by local bloom
filter is less in the whole of the metadata server (MDS). The local bloom filter is grouped based on the suffix value.

4.6.1 Protocol for Metadata Update using LBF

This research proposes a specific protocol designed to update the metadata file across the MDS. The protocol includes LBF as main component. The protocol for updating the metadata files in the original location and its replication locations are defined by LBF. The LBF is activated whenever the critical point is reached. Critical point is a point where the XOR values of existing and new LBF values converges to zero.

4.6.2 Structure of Local Bloom Filter

Local Bloom Filter is a multi-layered bloom filter which is responsible for update mechanism of metadata. The number of layers in the LBF represents the number of attributes in the metadata file. Every metadata server has its own LBF which is a part of cloud bloom filter. Each layer comprises of suffix, header and body. The suffix information provides information about the replica locations of the respective file. The header information is used to identify the attribute. The body of the frame is the standard bloom filter.

Frame of LBF

| Header | Suffix | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 4.6 Structure of local bloom filter

In LBF, each layer has an independent modified bloom filters. The attributes assigned to these layers are independently hashed using MD5,
SHA-1 and SHA-2 algorithms. Initially all the bit values are set to zero. When a metadata file is stored in the metadata server, the respective attributes are immediately hashed and the information is stored into the respective layers of local bloom filter. The schemas are identified by their header values. Any changes in the metadata attributes are updated in the LBF and in turn to other replica. LBF provides updated data to the user, as the consistency of the metadata is maintained which paves the way for efficient and perfect file access. The quality of metadata is identified by means of the percentage of error rate. The overhead due to false alarm also reduces by updating the metadata at proper critical point.

4.6.3 Metadata Update using LBF

An efficient update mechanism deals with two issues. The first issue is when and what kind of update is required and second issue is the amount of bandwidth consumed in maintaining consistency. In order to make the proposed system work properly with consistent data, the update has to be carried out in the following two different sequences 1. Whatever changes occurred in the original file i.e. any modifications to the existing file has to be updated to the metadata file 2. The changes in the metadata file have to be updated to its replica location.

In the proposed model the efficiency is introduced in two perspectives. First is the update based on LBF, since updating the metadata file continuously at runtime is expensive thus reduces the number of updates while still ensuring that update occurs at regular interval.

The second optimization occurs by the amount of data transfer. Since normally changes in data happens locally, the corresponding updates in metadata would be only on a few attributes. Consequentially, one or two of the LBF layers need to be updated. Hence the amount of update of corresponding
replica files is also restricted to the data in the respective layers. Thus inspite of updating the whole of the metadata file only the corresponding attribute gets updated which reduces the network traffic and also reduces bandwidth consumption. Algorithm 3 explains about the update of metadata using local bloom filter.

Algorithm 3: Metadata update mechanism using LBF

Input: Any change occur in data file.
Output: Updated metadata

if fileloc=1 /* fileloc is the location of the file */ /* if Flag=1 change in location */
{
    Update(A_l) /* A_l represents the location attribute */
}

for i: 1 to t /* t is the time period */
{
    X = B_i[x] (E_X - O_R) B_j[x] /* B_i[x] bloom filter value of attribute
        if (X = 0) then x*/
        
    Old (B_i[x]) ← New (B_i[x]) /* Bi[x] represents the bloom filter value of X */
}

Update(A_l) /* Updating the LBF of each metadata server */
{
    Old (B_i[x]) ← New (B_i[x]) /* B_i[x] represents the bloom filter value of X */
}

The update functionality is determined by B_i[x] (E_X - O_R) B_j[x] = 0.
4.6.3.1 Analysis of LBF based on critical point

In cloud environment when the update probability increases, the probability that there is a read operation following a write operation within refresh duration (say every 10 minutes) increases therefore, the error rate increases accordingly. By contrast, the false alarms decrease with the update probability. This is because when update probability increases, the probability that there is a write operation between any two refreshes increases, thus lowering the false alarms. Hence it is observed that the error rates decrease as the threshold value increases. The threshold value is the value of number of times the update takes place. This is because when the threshold value increases, the refresh time of a database item increases accordingly. By contrast, the false alarms decrease when the threshold increases. This is because the refresh time increases when threshold value increases. The probability that there is a write operation between any two refreshes increases, thus lowering the false alarms. The hit ratios decrease as the update probability increases. This is because when the update probability increases, there happens more write operations on each database item. This will decrease the refresh time of a database item, thus, shortening the duration. The hit ratios also increase as threshold increases. By contrast, the response times decrease as threshold increases due to increased hit ratios. Hence the LBF update plays a better option which reduces the error rate and false alarm to a compromised level.

4.7 DERIVATION OF FALSE POSITIVE PROBABILITY OF CBF

Let us consider \( m_h \) being the number of message bits to be set in the CBF and \( k \) being the number of hash functions involved in the bloom filter and \( n \) is the number of files entered into the bloom filter. The total
probability with respect to the bit set and reset is \( P(i) + P(j) = 1 \), where \( P(i) \) is the probability that the bit is set to one and \( P(j) \), the probability that the bit is set to zero. With respect to \( m_h' \), the probability that certain number of bits set is given as \( P(i) = \frac{1}{m_h} \) and \( P(j) = 1 - \frac{1}{m_h} \). Consider the number of elements entered is “\( n \)” and number of hash function keys is ‘\( k \)’ then the final probability that any of the bit not being set to one is \( P(j) = (1 - \frac{1}{m_h})^{kn} \). Hence the probability that certain number of bits set to one is \( P(i) = 1 - (1 - \frac{1}{m_h})^{kn} \).

The total number of bits to be set depends on the output of the hash algorithms used. The analysis that an element is hashed by all “\( k \)” hash functions is \( P(i) = \left(1 - \left(1 - \frac{1}{m_h}\right)^{kn}\right)^{\frac{k}{m}} \) and from this equation the false positive of the model is found as \( f_p = \left(1 - e^{-\frac{kn}{m_h}}\right)^k \) where, \( e^{-\frac{kn}{m_h}} = (1 - \frac{1}{m_h})^{kn} \).

Thus, the probability of false positive remains as the standard bloom filter but it has a trade-off between the numbers of bits used to set per element with respect to the hashing function. As the proposed model has increased key value ‘\( m_h \)’ due to MD5, SHA-1&SHA-2 the above expression clearly states that whenever there is an increase in the value of \( m_h' \), the value of \( f_p \) decreases. A false positive decrease is due to decrease the collision of hash functions. For example SHA-1 produces output of 160 bits, the values of \( k = 1 \) and \( m_h = 2^{160} \) would produce no false positives at all, since each bit in the CBF would correspond to a unique hash value.
Table 4.1 Effect of “m<sub>h</sub>” in probability of false positive

<table>
<thead>
<tr>
<th>n</th>
<th>K</th>
<th>m&lt;sub&gt;h&lt;/sub&gt;</th>
<th>False Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>3</td>
<td>2&lt;sup&gt;16&lt;/sup&gt;</td>
<td>8.9*10&lt;sup&gt;-5&lt;/sup&gt;</td>
</tr>
<tr>
<td>1000</td>
<td>3</td>
<td>2&lt;sup&gt;32&lt;/sup&gt;</td>
<td>3.4*10&lt;sup&gt;-19&lt;/sup&gt;</td>
</tr>
<tr>
<td>2000</td>
<td>3</td>
<td>2&lt;sup&gt;16&lt;/sup&gt;</td>
<td>6.6*10&lt;sup&gt;-4&lt;/sup&gt;</td>
</tr>
<tr>
<td>2000</td>
<td>3</td>
<td>2&lt;sup&gt;32&lt;/sup&gt;</td>
<td>2.7*10&lt;sup&gt;-18&lt;/sup&gt;</td>
</tr>
<tr>
<td>3000</td>
<td>3</td>
<td>2&lt;sup&gt;16&lt;/sup&gt;</td>
<td>2.1*10&lt;sup&gt;-3&lt;/sup&gt;</td>
</tr>
<tr>
<td>3000</td>
<td>3</td>
<td>2&lt;sup&gt;32&lt;/sup&gt;</td>
<td>9.2*10&lt;sup&gt;-18&lt;/sup&gt;</td>
</tr>
<tr>
<td>4000</td>
<td>3</td>
<td>2&lt;sup&gt;16&lt;/sup&gt;</td>
<td>4.6*10&lt;sup&gt;-3&lt;/sup&gt;</td>
</tr>
<tr>
<td>4000</td>
<td>3</td>
<td>2&lt;sup&gt;32&lt;/sup&gt;</td>
<td>2.1*10&lt;sup&gt;-17&lt;/sup&gt;</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
<td>2&lt;sup&gt;16&lt;/sup&gt;</td>
<td>8.5*10&lt;sup&gt;-3&lt;/sup&gt;</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
<td>2&lt;sup&gt;32&lt;/sup&gt;</td>
<td>4.2*10&lt;sup&gt;-17&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Thus the results from Table 4.1 explores that the increase in value of “m<sub>h</sub>” has a high impact on the false positive of the CBF. Hence even if “n” is greater than one million values, the proposed CBF performs well with reduced false positives due to the selection of hashing techniques. Thus the proposed CBF performs better compared to standard bloom filter.

4.8 EXPERIMENTS AND RESULTS

The experimental analysis is carried out in an identical environment described in chapter 3 using KDD Cup 2003 dataset.

4.8.1 Metrics Used

The tests used 1000 files of various sizes, uploaded into the storage and then downloaded based on the user’s requirement. The metrics considered in these experiments are 1. Response time 2. Number of bits transferred
3. Error rate 4. False alarm. Response time is the access time to obtain required data from a storage device. Error rate is found to be the percentage of erroneous values i.e. the availability of stale data. False alarm is a alarm where the update is triggered unnecessarily, even if there exists no change in original file. Error rate and false alarm is unit less measures which are expressed in percentage and response time in seconds.

4.8.2 Experimental Results and Analysis

Our first experiment compares the performance of file access with and without using metadata file stored in MDS. Figure 4.7 illustrates the comparison of response time of file access with metadata and without using metadata. X-axis represents the file identification number and y-axis represents the response time in milliseconds.

![Comparison of response time for file retrieval with and without using metadata](image)

**Figure 4.7** Comparison of response time for file retrieval with and without using metadata

From Figure 4.7 it is observed that the time taken to retrieve the file from the data server using metadata is less when compared to without using metadata as mapping of user query to the data server is efficiently handled by metadata files.
The second experiment shows the effect of LBF in the proposed model based on the impact of propagation threshold. The proposed update mechanism using LBF, analyses the threshold value with the error rates and false alarms caused by the change in data file. The duration for update is based on the convergence value of local bloom filter. The experiment uses the above mentioned dataset and investigates the effect of update performance of metadata using LBF schemes. The error rates, false alarms, hit rates, and response times are measured with respect to various update probabilities and threshold values. The results using LBF is compared with the periodic and triggered update. Periodic update is defined as the update happening at regular intervals (say every 10 minutes). Triggered update is defined as the update that is automatically executed in response to every change in a file. Update probability is defined as probability of rate of change that happens in the existing file. Table 4.2 compares the number of bits transferred during periodic update using LBF and periodic update.

Table 4.2 Comparison of update time and bit transfer for LBF update and periodic update

<table>
<thead>
<tr>
<th>File ID</th>
<th>Number of bits to be transferred for update</th>
<th>Time taken to Update in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LBF Update</td>
<td>Periodic Update</td>
</tr>
<tr>
<td>1</td>
<td>196</td>
<td>98304</td>
</tr>
<tr>
<td>2</td>
<td>294</td>
<td>122880</td>
</tr>
<tr>
<td>3</td>
<td>352</td>
<td>131072</td>
</tr>
<tr>
<td>4</td>
<td>490</td>
<td>139264</td>
</tr>
<tr>
<td>5</td>
<td>600</td>
<td>155648</td>
</tr>
<tr>
<td>6</td>
<td>700</td>
<td>142671</td>
</tr>
<tr>
<td>7</td>
<td>500</td>
<td>157867</td>
</tr>
<tr>
<td>8</td>
<td>298</td>
<td>186754</td>
</tr>
<tr>
<td>9</td>
<td>300</td>
<td>179654</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>189564</td>
</tr>
</tbody>
</table>
The results clearly depicts that the update time using LBF is less when compared to periodic update, as the time depends on the speed up factor with respect to number of bits transferred to bit transfer rate. By using proposed LBF method the number of bits to be transferred is less when compared with periodic update because in LBF update happens only for a specific attribute and hence unnecessary data transmission is minimized.

In order to make the model work successfully a suitable point for triggering the update has to be found i.e. at what condition (at which point) update should be triggered. The threshold here refers to the update triggers after certain number of changes in the original file. Initially the experiments are carried out to exhibit the impact of threshold in the efficiency of the model with respect to error rate and false alarm. Figures 4.8 - 4.11 illustrates the percentage of error rate and false alarm for various threshold points ranging from 1 to 20 for an update probability, \( \beta \) ranging from 0.1 to 1.

**Figure 4.8** Comparison of percentage of error rate and false alarm for \( \beta = 0.1 

**Figure 4.9** Comparison of percentage of error rate and false alarm for \( \beta = 0.3 

From the results it is observed that for a smaller propagation threshold, updation are carried out more frequently so that the likelihood of having a hit is increased, hence results in higher percentage of false alarm and decrease in percentage of error rate. With an increase in propagation threshold decrease of percentage of false alarm and increase in error rate percentage is observed. From the result it is understood that the whole of the model works successfully with respect to a proper critical point and there is a clear trade-off between the error rate and false alarm. Hence in order to make the proposed model work efficiently a critical point has to be decided.

The performance comparison of proposed LBF update with existing update mechanisms with respect to variation in update probabilities is depicted in the Figures 4.12- 4.13.

The experiment is carried out and the readings are noted. The false alarm is calculated under the condition that the periodic update is considered for lesser time interval say 2ms and trigger is set for all changes that happen in the original file.
Figure 4.12 Comparison of percentage of false alarm for various updates with respect to increase in update probability

From the Figure 4.12 it is observed that the percentage of false alarm is less for LBF update and is high for Periodic and triggered updates. The percentage of false alarm is reduced for increase in update probability and reaches the lowest percentage which does not affect the efficacy of the overall metadata management model.

The next experiment is carried out to calculate change in the percentage error rate under the condition that the periodic update is considered for higher time (10ms) interval and trigger update is considered only for file name and location attributes.

Figure 4.13 Comparison of % of error rate for various updates with respect to increase in update probability
From the Figure 4.13 it is observed that the error rate is less for LBF update and is high for periodic and triggered updates. The percentage of error rate alarm is reduced using LBF for increase in update probability and is sustained which does not affect the efficacy of the overall metadata management model. The results 4.12 and 4.13 shows that the percentage rate of change in false alarm and error rate with respect to change in update probability using LBF update is varied and is less when compared to periodic and triggered update. In case of periodic and triggered update the false alarm is high. In case of triggered mode even when there exists a small change in the original file which has the least preference in the metadata, update happens and hence increases the false alarms. In case of periodic update due to its periodicity, alarm happens even when there is no change in the data file, hence increases the false alarm and thus reduces the efficiency of the network as unnecessary alarm increases the network traffic and affects the consecutive reads and writes.

Again the experiments are carried out by uploading the files to cloud storage and the results are verified under the conditions having update probability of 0.5. Figure 4.14 and 4.15 below illustrates the comparison results of various update mechanism with respect to an update probability of .5.

**Figure 4.14** Performance comparison of percentage of error rate for various update policies with respect to an update probability of 0.5
A comparison result of various update mechanisms with respect to increase in number of files are shown in Figures 4.14 - 4.15. From the results it is observed that the average variation in percentage of error rate is 6% and false alarm is 5% with respect to threshold fixed from LBF and for update probability of 0.5. The result shows that the update point fixed by the LBF reduces the error rate and false alarm and hence proves that the model works efficiently using LBF.

4.9 CONCLUSION

In this chapter a distributed scheme for file mapping and file lookup using GBF in cloud scenario is well proposed. The chapter also efficiently handles critical points in metadata management by making use of LBF within a group of metadata servers. The results of CBF based metadata shows that the time taken for retrieving the file from the cloud server is less reducing latency. The use of GBF reduces the complexity due to disk lookups in case of file existence. The proposed LBF based update mechanism enhances the consistency of the metadata file stored by means of reducing the error rate and
decreasing the false alarm, thereby helping to achieve better relevance with respect to tasks in hand. Thus by making use of GBF and LBF the proposed model is not bound by the performance of disk operations and hence increases the efficacy of the overall system. However GBF and LBF take care of only placing, mapping and look up operations of metadata file. These do not deal with other operations i.e. searching of metadata file inside MDS. In order to explore the search operation effectively, analysis of metadata file inside MDS has to be carried out effectively which would further reduce the latency in the data retrieval. Hence the next chapter analyzes the metadata file stored in MDS. The analysis model in next chapter proposes an efficient and accurate metadata pre-fetching algorithm so as to improve metadata operation performance and thereby increasing the overall efficacy of the MaaS.