

A Bloom Filter-Based Approach for Supporting the Representation and Membership Query of Multidimensional Dataset

Zhu Wang and Tiejian Luo

Abstract Bloom filter has been utilized in set representation and membership query. However, the algorithm is not quite suitable for representing multidimensional dataset. The paper presents a novel data structure based on Bloom filter for the multidimensional data representation. We further give the theoretical analysis and experimental evaluations of the algorithm. Results show that the algorithm can achieve the same false positive rate when dealing with exact membership queries. It can provide extra support of by-attribute membership query.

1 Introduction

Bloom filter [2] is widely used in Internet applications [3]. It has also been adopted in multidimensional indexing because of its space-efficient and time-efficient characteristics in supporting approximate membership queries. Standard Bloom filter takes the entire multidimensional data as a whole and generates the indices. Therefore, it cannot support by-attribute queries, in which only a subset of attributes in the queried item are provided. Multidimensional dynamic Bloom filter [4] and parallel Bloom filter [5, 6] store each dimensional data in a separate Bloom filter to answer by-attribute queries. However, when adopted with high-correlated queries, the performance of the algorithm degrades a lot. In this paper, we try to maintain the advantages mentioned in the algorithms above and avoid their shortages by using a Cartesian matrix of Bloom filters to store the dataset.

An index for a dataset is a data structure that is used to record the items of the set. In a multidimensional data search, the membership query is a request with several

This work gives real world based experiments of our paper [1] published previously.

Z. Wang (✉)

Data Communication Technology Research Institute (DCTRI), Beijing, China

e-mail: wangzhu09@mails.ucas.ac.cn

T. Luo

University of Chinese Academy of Sciences, Beijing, China

e-mail: tjluo@ucas.ac.cn

attributes keyed into the database system. It asks for the response of whether there is an item (data object) in the set that has all the wanted attributes. Exact membership query refers to a request which can provide all wanted attributes simultaneously, whereas by-attribute memberships query requests for the search result in response to queries containing a subset of all attributes that compose a data object. Let us look at an example to see the scenario of these queries. There are two items in a set, $S = \{(a, b), (x, y)\}$, each item has two dimensions. An exact membership query is a question like “Is there an item in S whose two attributes are (a, b) ?” In contrast, a by-attribute query only contains partial dimension information, such as “Is there an item in S whose first attribute is x ?” In addition, a correlative query contains the shared attributes with the set. For example, “Is (a, m) in the set?” is a correlative query because it shares “ a ” with the set. The more likely the query shares attributes with the set, the higher correlative it is. To fulfill these queries, various indexing structures are designed in applications. For example, one indexing technique [4] is to store each dimension in one container separately. For the set S , we maintain the first dimensional container $D_1 = \{a, x\}$ and the second $D_2 = \{b, y\}$. When a query arrives, the system just examines the containers dimension by dimension to locate the requested result and merges the result. However, this processing will cause a false positive mistake. E. g., query (a, y) will be judged as inside S because $a \in D_1$ and $y \in D_2$, but actually it is not. Thus in this technique like [4], any mix-up of existing items’ attributes will cause such mistakes. Obviously, the high-correlative queries will boost this error.

Bloom filter is a data structure that can give fast response to queries but have a low false positive rate. In typical applications like [7], the false positive occurrence may lead to a waste of time. However, the short response delay characteristic overweighs the previous cost and the overall system performance can be improved by using Bloom filter as an index.

The key for multidimensional data management is to establish a reliable data structure for data indexing, which is capable of supporting exact and by-attribute membership queries. The time and space overhead should be very low and the mix-up false rate caused by correlative queries must be reduced.

The remainder of the paper is organized as follows. In Sect. 2 we survey the existing Bloom filter based algorithms for multidimensional data indexing. Section 3 gives the theoretical analysis of false rate and optimal hash number of the Cartesian-join of Bloom Filters. The time and space complexity is also demonstrated. Experimental evaluation is shown in Sect. 4. Finally, Sect. 5 concludes the research and suggests future directions.

2 Related Work

There have been quite a few previous attempts to apply Bloom filters in multidimensional data indexing. Standard Bloom Filter (SBF) [2] offers an efficient way to represent multidimensional data. It uses a bit vector as an index of items of a set S . When an item is inserted, the algorithm uses a group of hash functions to map the item onto several locations in the bit vector and sets the corresponding bits to

one. When a membership query is submitted, the algorithm uses the same hash functions to calculate the hash positions of the queried item and checks if all the corresponding bits are one. If the answer is yes, the Bloom filter concludes that the queried item belongs to the set. Otherwise it reports that the query does not belong to the set. It needs to be mentioned that for each item belonging to S , since all the bits of the hash locations are already set to 1, the lookup procedure for the very item will definitely have the positive answer. So there will be no false negatives. However, there is a probability that items do not belong to the set be judged as inside S by the Bloom filter because its hash locations might have been set to 1 by some other items' hashing. That is to say, Bloom filter has a false positive rate. Research [8] shows that the false positive rate can be represented as follows:

$$f_{SBF} = \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (1)$$

Here m is the bit vector length. n is the item number. k is the hash number and f_{SBF} is false positive rate of SBF. Study [8] also shows that f_{SBF} reaches the minimal value when k values

$$k_{opt_SBF} = \frac{m}{n} \ln 2$$

k_{opt_SBF} is the optimal hash number. The time complexity is $O(k)$.

In multidimensional indexing, the standard Bloom filter first joins the attributes of an item and then operates with the joint result. In that way, SBF can easily avoid the false rate caused by the cross-dimensional mix-up. However, the method requires the entire attributes of a query and therefore is not capable of handling by-attribute search.

Guo et al. propose an algorithm called multidimensional dynamic Bloom filter (MDDBF) to support data indexing [4]. The algorithm builds one attribute Bloom filter as the index for each dimension of the multidimensional set. In item insertion it simply inserts each of dimensional attribute into the corresponding Bloom filter. When a query is submitted, the algorithm checks whether all of its attributes exist in the corresponding Bloom filter. If all the answers are yes, then it reports a positive response. It can be seen that while the method is capable of handling exact and by-attribute queries, it introduces a new type of false positive occurrence: the mix-up of attributes in different dimensions. For example, if (a,b) and (x,y) are in the set S , then the query (a,y) will be judged as inside S because both attributes of the query exists in each dimensional Bloom filter. The reason for that is the idea of storing the item's attributes of different dimensions separately splits the entire item and the relationship between the attributes of one item is lost. In the example, one cannot tell that attribute a and b belong to one item due to the separate storage.

Parallel Bloom filter (PBF) [5, 6] tries to solve the problem by adding a verification value of each attribute insertion for the item. Then it stores the verification value in a summary Bloom filter. The new Bloom filter's hash functions

are actually a composite function of attribute hashes, verification function and summary hash functions. So the approach is equivalent to adding a SBF for the integrated item besides the attribute Bloom filters for the separate dimensions. Therefore, PBF cannot support by-attribute query because it requires all attribution information, just like SBF.

3 CBF Design and Analysis

Table-based index can support flexible queries but takes too much time and space. SBF is an efficient indexing structure but do not allow by-attribute query. MDDBF provides that functionality but performs poorly with correlative queries. Designing a fast, accurate and space-efficient indexing mechanism for multidimensional dataset to support both exact and by-attribute membership queries which can achieve a high performance even with correlative queries is indeed a challenging task in multidimensional data management. Our proposal, the CBF Bloom filter is capable of satisfying those requirements. In this section we present the design of the CBF. We then analyze the false positive rate and the optimal hash function number of the algorithm. The time and space complexity is also given.

3.1 CBF Algorithm and Structure

The CBF is designed to support both exact and by-attribute membership queries. In order to avoid the mix-up mistake with the correlative queries (which is the main source of the false positive rate in MDDBF), we try to maintain the inner relationship between different attributes of a same item.

3.1.1 CBF Structure

We use one Bloom filter to represent one dimension's attributes of the d -dimensional dataset. The Bloom filter is called an attribute Bloom filter. All attribute Bloom filters have the same hash number k (yet the hash functions are independent). To avoid the loss of relationship between one item's attributes, we establish a d -dimensional CBF matrix which is the Cartesian product of empty attribute Bloom filters. That matrix is used to store the membership information of the dataset. To answer membership queries, the items should be inserted into the matrix: we store the hash result in the matrix by combining the corresponding results into a d -dimensional point in the CBF matrix.

CBF indexing works in the following procedure: in order to establish the index of the multidimensional database, we insert all the items into the CBF matrix using item insertion algorithm. When a membership query of a certain item is submitted

to the system, we look up the query in the CBF matrix using the same hash functions in the insertion period. Then we give the yes/no answer of the query.

Now we illustrate the insertion method of a d -dimensional item $A = (a_1, a_2, \dots, a_d)$. The i th attribute Bloom filter is ABF_i . The hash function group for the i th attribute Bloom filter is $H_i = \{h_{i1}, h_{i2}, \dots, h_{ik}\}$. The length of the i th ABF is m_i . For the first dimension a_1 , we use the hash functions group H_1 of the ABF_1 to calculate the first hash results group $H_1(a_1) = \{h_{11}(a_1), h_{12}(a_1), \dots, h_{1k}(a_1)\}$, which contains k hash results. We do the same with the second dimension using second hash function group until the last dimension. Now we obtain the hash result groups $\{H_1(a_1), H_2(a_2), \dots, H_d(a_d)\}$, each contains k hash results. In each group, we select the first hash results in each hash result groups and join the selections to form a d -dimensional data point $(h_{11}(a_1), h_{21}(a_2), \dots, h_{d1}(a_d))$. We set the corresponding point in CBF matrix to one. We do the same for the rest of hash result groups until all k hash result points are set to one. Figure 1 shows the data structure of CBF matrix. We take $d = 2$ as an example.

Here the CBF matrix is the Cartesian product of empty ABFs.

$$\begin{aligned} CBF &= ABF_1 \times ABF_2 \times \dots \times ABF_d \\ &= \{X | X = (x_1, x_2, \dots, x_d), x_i \in [0, m_i]\} \end{aligned}$$

Unlike the MDDBF and the PBF algorithm which put the multidimensional items in several parallel Bloom filters, CBF makes Cartesian-join of Bloom filters to form a matrix. That indicates the major difference of our algorithm. Recall the example in the related work, the query (a, y) has a large possibility to be judged as outside S by CBF because the hash points of items (a, b) , (x, y) , and (a, y) in CBF can be different.

3.1.2 Item Insertion

The item insertion procedure has already been described in the section above. Now we give the detailed algorithm in Fig. 2.

The major advantage of the indexing algorithm is that unlike MDDBF, the CBF manages to preserve the inner relationship between attributes of the same item. That goal is achieved by combining the same-order hash results of each dimension belonging to one item into one data point. Each dimension value of the data point comes from one attribution Bloom filter's hash results. When the corresponding position is set to 1 in the CBF matrix, the inner relationship of the item's different attributes is maintained.

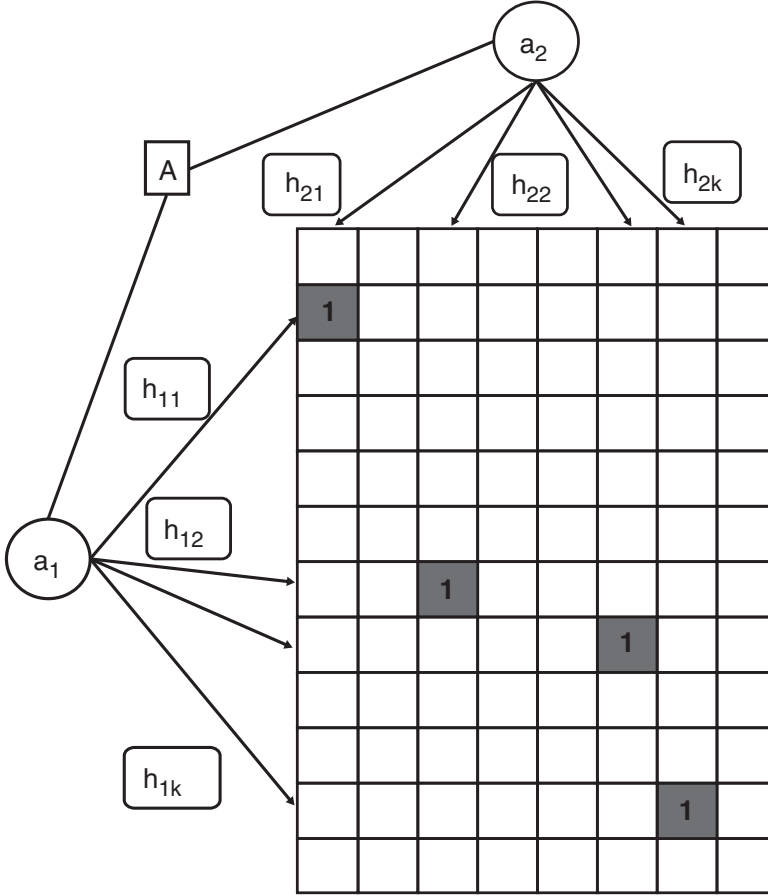


Fig. 1 CBF structure: inserting item A

3.1.3 Exact Membership Query

In an exact membership query, all attributes of queried item are provided. The processing of such query follows the similar way as the item insertion procedure. Let a query $Q = (q_1, q_2, \dots, q_d)$ where no attribute q_i is empty. The algorithm first uses the same attribute Bloom filters' hash function groups to find the hash results of the item's attributes $H_i(q_i) = \{h_{i1}(q_i), h_{i2}(q_i), \dots, h_{ik}(q_i)\}$. Then it combines the same-order hash results into query points and checks if all the query points in the CBF matrix are one. If the answer is yes, it concludes that the queried item exists in the set. Otherwise it returns a negative answer. The algorithm is given in Fig. 3.

The algorithm handles the exact membership query by using the same hash function groups to calculate the hash results. If the queried item does exist in the set, then it must have been inserted into the CBF matrix through the item insertion method. So the corresponding data points in CBF must have been set to 1. The look

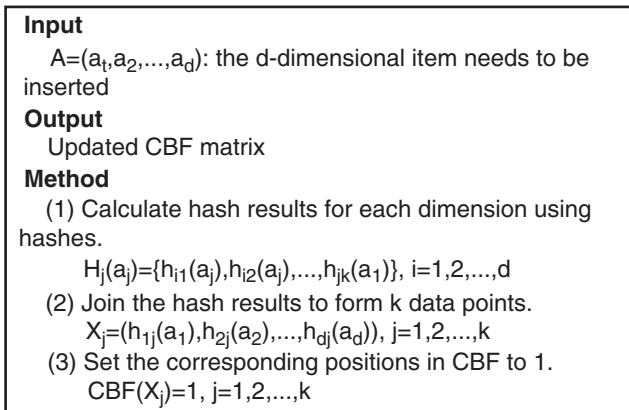


Fig. 2 The item insertion algorithm

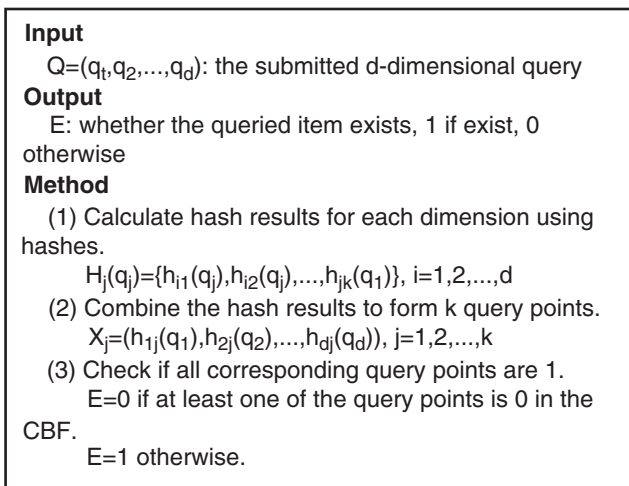


Fig. 3 The exact membership query algorithm

up procedure will surely get a positive response. That is to say, the algorithm will have no false negative mistakes. Yet for a queried item that does not belong to the set, it is possible that it be judged as inside the set because its corresponding data points in the CBF have been set to 1 by some other items, so the algorithm has a false positive rate. We will analyze that in Sect. 3.2.

It needs to mention that since the inner relationship between one item's attributes is preserved, the algorithm prevents the occurrence of mix-up mistake between attributes. For instance, the set $S = \{(a,b), (x,y)\}$. Hash number $k = 1$. $h_{11}(a) = 1$, $h_{11}(x) = 2$, $h_{21}(b) = 3$, $h_{21}(y) = 4$. In the insertion of (a,b) and (x,y) , we set the

corresponding data points in the CBF to 1. $CBF(1,3) = 1, CBF(2,4) = 1$. Now let's see whether the mix-up (a,y) will be judged as inside S . Using the hash results, the query point will be $X_1 = (1,4)$. The corresponding position $CBF(1,4)$ is 0. So the mix-up query (a,y) does not belong to S . In the example, provided that there are no hash collisions, the hash points of different items will never coincide. More broadly, when storing a multidimensional item, the CBF algorithm maintains the inner relationship between its attributes by combining the hash results of each attributes Bloom filter. In that way, the algorithm avoids the mix-up false positive mistake. Later we will see in experiment Sect. 4.2 that false rate is a major part of false positive mistakes with correlative queries.

3.1.4 By-Attribute Membership Query

In by-attribute query, only some of the attributes in the queried item is provided. Some attributes are empty. When the query is submitted, the user intends to ask whether there is an item in the set which has the same attributes as the non-empty attribute of the query. For example, the dataset is composed of students of a class, with two dimensional attributes: age and gender. An exact membership query is "Is there a 12-years-old boy in the class?" A by-attribute membership query is "Is there a girl in the class?" Without loss of generality, we assume that the last p attributes are not provided in the query.

The query $Q = (q_1, q_2, \dots, q_d)$, $q_{d-p+1} = q_{d-p+2} = \dots = q_d = \text{NULL}$. Now we want to find whether there is an item in the set whose first $d-p$ attributes are q_1, q_2, \dots, q_{d-p} respectively. Assume that there is an item A that satisfies the query,

$$A = (a_1, a_2, \dots, a_d) = (q_1, q_2, \dots, q_{d-p}, a_{d-p+1}, \dots, a_d), \\ a_{d-p+1}, \dots, a_d \text{ can be any value,}$$

Since all the items have been inserted into the CBF, using the insertion algorithm, the corresponding hash data points of item A must have been set to 1. That is,

$$CBF(h_{1j}(q_1), \dots, h_{(d-p),j}(q_{d-p}), h_{(d-p+1),j}(a_{d-p+1}), \dots, h_{dj}(a_d)) \\ = 1, j = 1, 2, \dots, k$$

As well as there exist a_{d-p+1}, \dots, a_d that satisfy the conditions above, we conclude the queried item exists in set S .

In the look up procedure of query Q , (j values from 1 to k), for each j , we first calculate $h_{1j}(q_1), h_{2j}(q_2), \dots, h_{(d-p),j}(q_{d-p})$ and search all points whose first $d-p$ dimensions values $h_{1j}(q_1), h_{2j}(q_2), \dots, h_{(d-p),j}(q_{d-p})$, correspondingly. We call these points a query bunch. If all these point in the bunch are zero in CBF, it means that no such points exist, and therefore the query doesn't exist in the set. Otherwise it means it is possible that the query exists, so we continue the $j+1$ hashes and repeat the same procedure. If all k procedures receives the positive answer (Fig. 4).

Input
$Q=(q_1, q_2, \dots, q_{d-p}, \text{NULL}, \dots, \text{NULL})$: the submitted d-dimensional query
Output
E: whether the queried item exists, 1 if exist, 0 otherwise
Method
(1) Calculate hash results for each dimension using hashes. $H_j(q_j)=\{h_{1j}(q_j), h_{2j}(q_j), \dots, h_{kj}(q_j)\}, i=1, 2, \dots, d-p$
(2) Combine the hash results to form k query bunches. $X_j=(h_{1j}(q_1), h_{2j}(q_2), \dots, h_{dj}(q_d), *), j=1, 2, \dots, k$
(3) Check if all query bunches have a 1 in CBF $E=1$ if exists X_1, \dots, X_k s.t. $\text{CBF}(X_1)=1, \dots, \text{CBF}(X_k)=1$ $E=0$ otherwise.

Fig. 4 The by-attribute membership query algorithm

3.2 False Rate and Optimal Hash Number

As stated above, the CBF algorithm has zero false negative rate because all queries for an existing item will receive a positive response. However, it is possible that a query outside the set S is judged as inside S because its corresponding data points in the CBF have been set to 1 by some other items. Now we analyze the false positive rate. Since in the latter experiments, we will compare the false positive rate between different Bloom filter-based algorithms, we don't use other algorithms as a comparison in this experiment.

Given the attribute Bloom filters ABF_i with size m_i and hash number k , the size of CBF is $m_1 m_2 \dots m_d$. For one item, the possibility that the first hash functions of each dimension hit a certain position in CBF is $p_{\text{hit}} = 1/(m_1 m_2 \dots m_d)$ because each hash has a uniform distribution in the ABF and the hash functions are independent from each other. After inserting n items, each using k hash function groups, the probability that a certain place in CBF is still zero is

$$p = (1 - p_{\text{hit}})^{kn}$$

Here $1-p_{\text{hit}}$ is the probability that the position is not hit by one hash. After inserting n items, each with k hash groups, the probability that the position remain zero is p , as conducted.

The case in which a non-existent item is judged as inside S happens when all k hashed data points are set to 1 by other items. So the false positive rate of CBF is

$$\begin{aligned}
 f_{CBF} &= (1-p)^k = \left(1 - \left(1 - \frac{1}{m_1 m_2 \dots m_d}\right)^{kn}\right)^k \\
 &\approx \left(1 - e^{-\frac{kn}{m_1 m_2 \dots m_d}}\right)^k
 \end{aligned} \tag{2}$$

The corresponding optimal hash number can be conducted using similar method as that in [8].

$$k_{opt-CBF} = \frac{m_1 m_2 \dots m_d}{n} \ln 2$$

When the length of ABFs and the item number is fixed, the CBF can reach its lowest false positive rate when k reaches that optimal number. That is similar to the optimal hash number of SBF.

It needs to mention that Bloom filter does not contain the actual data. It can just support membership queries of certain sets and give a yes/no response. The applications have to access the data eventually and eliminate the false rate.

3.3 Space Efficiency and Time Complexity

We further look into the false positive rate of CBF, which is the main description of the algorithm's performance. Comparing Eqs. (2) and (1), we find that the only difference between the false rate of CBF and SBF is the product $m_1 m_2 \dots m_d$, which happens to be the total size of CBF. Replace $m_1 m_2 \dots m_d$ with data structure size M , the two expressions are the same. So we have

$$f_{CBF} = F_{SBF} = \left(1 - e^{-\frac{kn}{M}}\right)^k \tag{3}$$

It means that if the total sizes of the two data structures are the same, they will have the same false positive rate. That is to say, the space efficiency of CBF and SBF are the same. As we all know from [3, 9] that, standard Bloom filter has a very high space efficiency, so does CBF.

For every d -dimensional query, the algorithm will have to calculate k hash functions to find the result. So the time complexity for a query is $O(dk)$, higher than that of SBF ($O(k)$). However, in actual use case, d and k are fixed. So the time complexity of CBF is $O(C)$, which is independent of n (number of items in the set). That is lower than the $O(\log n)$ of B-trees.

3.4 *Growth of Dimensions*

Now we analyze what will happen when the dimension number grows. From Eq. (3) we can see that if we want to maintain the false rate, we need to only keep the total size of CBF unchanged. When the dimension grows, we can keep the false rate without enlarging the CBF size. Actually, we can reduce the average ABF length to achieve that. The only limitation is that the length of each Bloom filter should be no less than 2. So long as the dimension number is less than $\log_2 M$, it has no impact on the false rate of CBF.

The time complexity is $O(dk)$, which has a linear relationship with dimension. However, most applications' dimension size is not very large, say $2 \sim 100$. In that case, the time complexity can be constant, as stated in Sect. 3.3.

4 Experimental Evaluation

In this section we use three experiments to validate our design objectives. We first show the correctness of the false rate and hash number derivation. In the second and third experiments, we present the CBF performance in exact query and by-attribute search respectively, in comparison with baselines.

4.1 *False Positive Rate and Optimal Hash Number*

In this experiment we want to find the relationship between the false positive rate and the hash number. Then we can identify the optimal hash number from the experiment and compare it with that derived from theoretical analysis.

The role of CBF is to find if a certain query exist in a dataset. It verifies if there is a same item in the dataset, but pays no attention on what the attributes of the query is. That reminds us to use datasets and queries with different items, but to ignore what the attributes of the queries really are. To verify our assumption, we use real world dataset in this experiment.

We use real-world dataset—the Truck dataset from the ChoroChronos Website [10]. The dataset is composed of 112,203 rows. We use the fourth, fifth, sixth and seventh column of the dataset and get rid of the rows that shares the same coordinates to have 29,018 different four dimensional items. The size of CBF is $2^{18} = 262144$. The hash number k ranges from 1 to 16. The result is given in Fig. 5.

The experimental false rate is in accordance with the theoretical one. Here we can see that our analysis of false rate and optimal number is correct and the conclusions drawn from that have a convincible base.

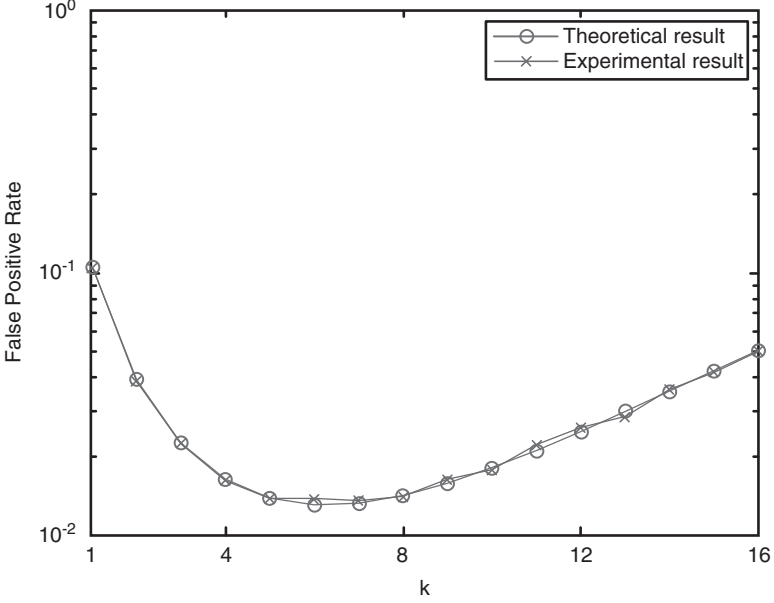


Fig. 5 Effect of hash number

4.2 Exact Membership Search

In this experiment we compare the performance of CBF with other indexing algorithms. The baseline approaches are SBF, PBF and MDDBF. We further want to see the effect of relationship between the query and the dataset. Let α be the possibility that one attribute of the query can find a match in the set. So $\alpha = 0$ means all attributes of the query are new. For the set $S = \{(a,b),(x,y)\}$, the query may be (c,d) . $\alpha = 1$ means all attributes of the query can find a copy in the corresponding dimensions of the dataset, the query is just a mix-up of the existing items of the dataset, but it is different from any of the existing items in the dataset. For the set $S = \{(a,b),(x,y)\}$, the query may be (a,y) .

In the experiment, the dataset has 29,018 four dimensional items. The size of data structures is all the same, $2^{18} = 262144$ bits. The hash number reaches the optimal value for all algorithms. The query number is 29,018. We choose α from 0 to 1 and calculate the false positive rate. The result is shown in Fig. 6.

When $\alpha = 0$, the algorithms have the similar performance. However, the false rates of MDDBF and PBF grow linearly with α , MDDBF even reaches 100 %. That is because of the mix-up mistake. The more similar the query is with the dataset, the more likely its attributes find a match in the set, i.e., the higher false rate.

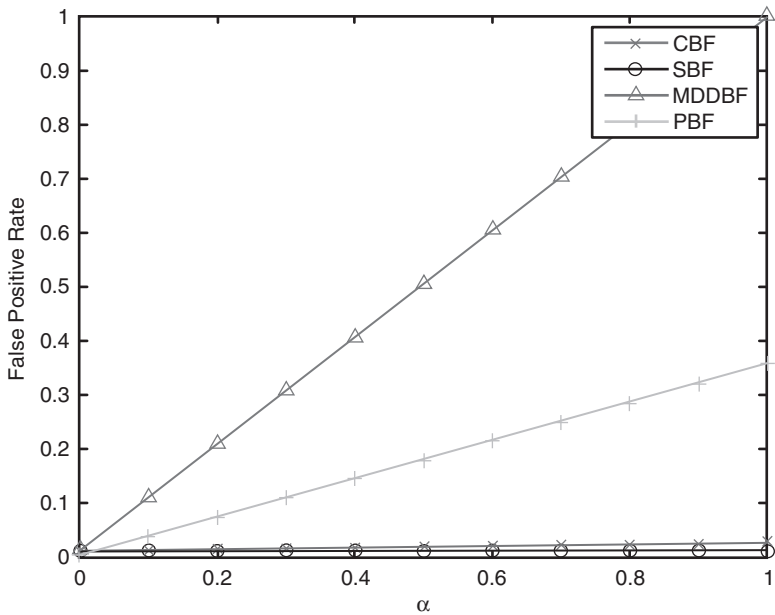


Fig. 6 Effect of query similarity

The false rate of SBF remains stable as α increases, while that of CBF increases very slowly with α . The two algorithms’ performance is both acceptable but CBF can support the by-attribute search, while SBF cannot.

It needs to mention that the average space per query is about 10 bits and it remains stable when dimension increases, while in a table only one attribute can use up more than ten bytes of storage. The false positive rate is about 3 % for CBF.

4.3 By-Attribute Membership Search

Now let’s look at the by-attribute search performance of CBF. Since only MDDBF can support by-attribute search besides CBF, we use MDDBF as a comparison.

The experimental settings are the same as that in Sect. 4.2. All queries have one missing attribute. The result is given below in Fig. 7.

We see that the false rate of CBF grow very slowly with the increase of α . In comparison, the performance of MDDBF is unacceptably high when α is large.

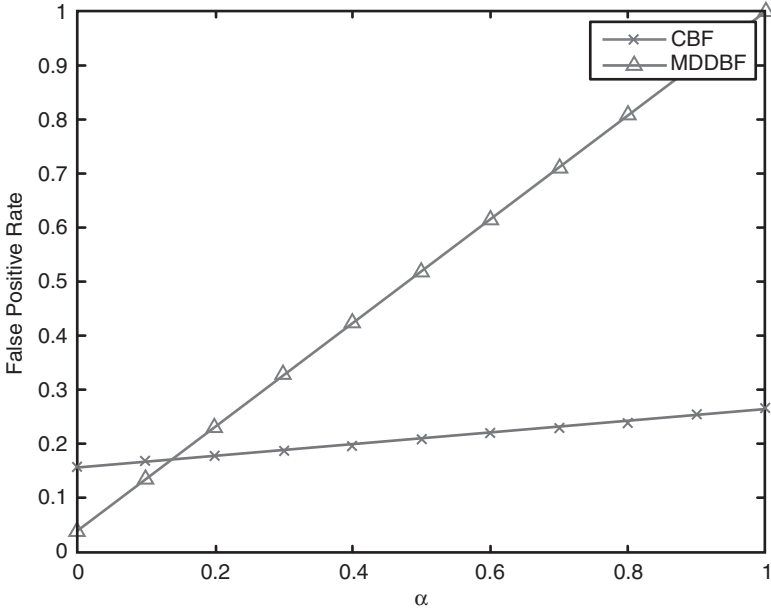


Fig. 7 By-attribute search

5 Conclusion and Future Work

In this paper, we have proposed the Cartesian-join of Bloom Filters to represent multidimensional dataset. The method has low time and space complexity. Our algorithm can support both exact membership query and by-attribute membership query of multidimensional data. In the paper we use theoretical method to analyze the false positive rate, space complexity, time complexity and dimensional scalability of CBF. We use real-world multidimensional dataset to test the performance of our algorithm. The experiments prove the correctness of our theoretical deduction.

The CBF algorithm uses many hash functions to calculate hash results. That causes the time complexity of CBF to be $O(dk)$, which grows linearly with dimension number. In future research, we plan to reduce the time complexity of CBF to achieve better performance. The analysis of the Bloom filter size's impact on system performance will also continue.

References

1. Z. Wang, T. Luo, G. Xu, X. Wang, The application of cartesian-join of bloom filters to supporting membership query of multidimensional data, in *2014 I.E. International Congress on Big Data (BigData Congress)*. IEEE, 2014, pp. 288–295

2. B.H. Bloom, Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **13**, 422–426 (1970)
3. S. Tarkoma, C.E. Rothenberg, E. Lagerspetz, Theory and practice of bloom filters for distributed systems. *Commun. Surv. Tut. IEEE* **14**(1), 131–155 (2012)
4. D. Guo, J. Wu, H. Chen, X. Luo et al., Theory and network applications of dynamic bloom filters, in *INFOCOM*, 2006, pp. 1–12
5. B. Xiao, Y. Hua, Using parallel bloom filters for multiattribute representation on network services. *IEEE Trans. Parallel Distrib. Syst.* **21**(1), 20–32 (2010)
6. Y. Hua, B. Xiao, A multi-attribute data structure with parallel bloom filters for network services, in *High Performance Computing-HiPC 2006*. Springer, 2006, pp. 277–288
7. Z. Wang, T. Luo, Optimizing hash function number for bf-based object locating algorithm, in *Advances in Swarm Intelligence*. Springer, 2012, pp. 543–552
8. J.K. Mullin, A second look at bloom filters. *Commun. ACM* **26**(8), 570–571 (1983)
9. A. Broder, M. Mitzenmacher, Network applications of bloom filters: a survey. *Internet Math.* **1** (4), 485–509 (2004)
10. Trucks—chorochronos.org, <http://www.chorochronos.org/?q=node/5>

Big Data Applications and Use Cases

Hung, P.C.K. (Ed.)

2016, VI, 214 p. 70 illus., 62 illus. in color., Hardcover

ISBN: 978-3-319-30144-0