Proceedings of the 2023 International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/31/20230171

Research on the fast geolocation algorithm based on Bloom filter

Weitong Wang

Weihai International College, Beijing Jiaotong University, Weihai, Shandong, 100044, China

20722064@bjtu.edu.cn

Abstract. With the rapid development of geolocation information services, which involves large-scale location data querying, including map data and user location data. Traditional data storage and query methods face challenges of storage space and query efficiency, and the ability to process real-time geolocation data becomes critical in Geographic Information Services positioning systems that require servers to achieve millisecond-level response speeds. In addition to this, in geolocation services, for a given latitude and longitude information, it is necessary to quickly determine whether it exists or matches some specific location in the map service. The common traditional information finding algorithm in geographic location service is B-tree, and this paper proposes a geographic location information finding algorithm based on the Bloom filter, which mainly solves two problems. One is the storage problem of large amounts of latitude and longitude information; the other is how to convert the data stored in bytes level to bits level, which greatly reduces the space complexity and the inefficient information finding. The Bloom filter reduces the query time to linear time complexity O(n). The fastest query time for 5 million items of latitude and longitude information can reach 3.12 seconds. This article will illustrate the efficiency of the proposed method through a real-time traffic navigation scenario.

Keywords: Geographic Information Services(GIS), space compression, bloom filter, bitmap.

1. Introduction

With the increasing volume of geospatial data, the storage and management of GIS data information has become a major challenge for scientists and Geographic Information Services (GIS) professionals [1]. Although traditional data storage and query methods are very simple and easy to use, they encounter bottlenecks of storage space and query efficiency when the data volume is large. Consider a data collection with a large amount of latitude and longitude information: $L=\{a_1, a_2, a_3, \ldots a_n\}$, Assuming that the storage size of each data $a_i, i \in n$ is 64ytes and there are 10 billion data in total, the total storage space is about 600GB, which is a huge load for the data server and makes it even more difficult to query the data.

A popular solution to server overload is to use distributed data storage, which stores massive amounts of data on different data servers to avoid single points of failure while increasing the concurrent queries on the server. For example, in Google, we run Bigtable, on GFS, which improved file storage and read/write efficiency [2, 3]. But this will inevitably lead to a waste of resources: the data that originally only needed to be stored on one machine is now stored on multiple servers. Or can we further improve

DOI: 10.54254/2755-2721/31/20230171

storage efficiency on the basis of a distributed system? Another high-performance lookup algorithm is B-tree, which is suitable for handling large amounts of data and can efficiently store and index large data sets. Nodes of the B-tree can hold multiple keywords and corresponding pointers, so the height of the tree is relatively low, which can reduce the number of disk accesses and improve query performance.

In this paper, we will propose a Bloom filter-based data lookup algorithm that uses a bitmap data structure to transform byte storage level data into bit storage level, which greatly improves the rate of space utilization by using multiple hash functions to map data into different bit slots and the mapping speed between blocks of data. The algorithm performs well for fast data de-duplication, data existence judgment, improving data query, storage efficiency, etc., and has a broad development prospect in the future.

2. Related concepts

In this section, some basic concepts are briefly introduced, including binary search tree and B tree[4-5].

2.1. Binary search tree

2.1.1. Properties. A binary search tree has two main properties. Each node has at most two children: a left child node and a right child node. The other is that for any node, the values of all nodes in its left subtree are less than the value of that node,, while the values of all nodes in its right subtree are greater than the value of that node.

2.1.2. Implementation process.

```
1    function binarySearch(root, target):
   2     if root is null or root.value equals target:
   3     return root
   4     if target < root.value:
   5          return search(root.left, target)
   6     if target > root.value:
   7          return search(root.right, target)
```

2.1.3. Time complexity. On average, the time complexity of a Binary Search Tree (BST) lookup, insertion and deletion operation is O (log n), where n is the number of nodes in the tree. This is because in a balanced binary search tree, each operation reduces the size of the tree to about half.

However, in the worst case, i.e., when the BST is not balanced, the time complexity may reach O(n), where n is the number of nodes in the tree. For example, when data are inserted in an ordered or inverted order, the BST may degenerate into a chain table, resulting in the need to traverse the entire tree for each operation, and the time complexity becomes linear.

If the lookup algorithm is applied to the geolocation service problem proposed in this paper, the space complexity of the latitude and longitude information is O(n), the time complexity of transforming the stored latitude and longitude information into a binary search tree structure is O(nlogn), and the time complexity of querying the result once is O(logn). The querying efficiency is significantly improved compared with the previous one, but when the amount of data is increased to a certain degree, the querying efficiency is minimal. In addition, due to the limitations of the binary search tree itself, if the tree structure cannot be balanced, then the query efficiency will be reduced to O(n), which cannot solve the problem proposed in this paper.

2.1.4. Properties. B-tree is a multiplexed search tree in which each node can have more than one child node. It can accommodate more child nodes, thus improving storage and search efficiency. In addition, it can maintain self-balancing, all leaf nodes of the tree are in the same hierarchy. By adjusting the insertion and deletion operations of the nodes, the tree is automatically balanced and reorganised to maintain the balance of the tree. Finally, the B-tree ensures that the data is ordered, and the child nodes

DOI: 10.54254/2755-2721/31/20230171

of the B-tree are arranged in a certain order. In each node, the keys of the child nodes are smaller and larger than the key of the node, respectively, which makes it more efficient to perform lookup operations in the B-tree.

2.1.5. Implementation process.

```
function search(node, key):
    if node is null:
        return null
    else:
        i=0
        while i < node.keyCount and key > node.keys[i]:
            i++
            if i < node.keyCount and key == node.keys[i]:
                return node
            else if node.isLeaf:
                return null
            else:
                return search(node.children[i], key)
```

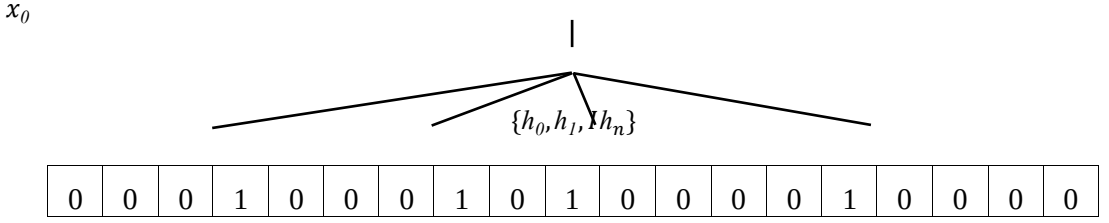
2.1.6. Time complexity. Compared with binary search tree, because of its self-balancing property, the best and worst time complexity of B tree is O(logn), and there is no need to traverse all nodes of the tree. The time complexity of transforming data into B tree data structure is O(nlogn), which is the same as that of binary search tree. In view of the problem proposed in this paper, the query efficiency is better than that of binary search tree, but the performance is still reduced in the case of a large amount of data. Next, we will propose an algorithm with better performance to solve this problem.

3. Proposed method

The Bloom filter was introduced by Bloom Burton H. in 1970 [6]. In solving the practical problem of eliminating duplicate geolocation information in the proposed paper, HashSet is a better choice because its time complexity is only O (1), but with the explosive growth of stored information, the corresponding space complexity rises dramatically, so we use Bloom filter.

3.1. Bloom filter

The basic idea of Bloom filter is that a hash table data structure is used to map the input data to a point in a bitmap by a fixed hash function, and when another input data is mapped by the same function, it is judged whether the current point is already occupied and thus whether it exists. The basic framework is as follows:



If another element x_1 is mapped to the same location in the bitmap by multiple hash functions as above, then both elements are the same and do not need to be added again.

However, the hash function has a certain error rate and may filter out elements that have not yet been added to the database. We will give a rigorous mathematical proof to determine which parameters can be used to minimize the error rate.

DOI: 10.54254/2755-2721/31/20230171

3.1.1. Misjudgment. We assume that S is the size of the bitmap, N is the number of hash functions, and D is the number of input elements. According to the nature of the hash function, the probability of selecting and setting the bit bit to 1 with equal probability is $\frac{1}{S}$, so the probability that a bit is not set to 1 is $1-\frac{1}{S}$, and the probability that it is not set to 1 after N hash functions is $(1-\frac{1}{S})^{N}$. If n elements are inserted, the probability that the position is not set to 1 is $(1-\frac{1}{S})^{DN}$, so the probability that the position is set to 1 Is $1-(1-\frac{1}{S})^{DN}$.

Misjudgments occur when a new element is mapped to position 1 through each hash function, so the misjudgments rate is:

$$\left[1-\left(1-\frac{1}{S}\right)^{DN}\right]^{N} \approx (1-e^{-\frac{DN}{S}})^{N} \tag{1}$$

3.1.2. Number of hash functions. According to the above formula, we find that the false positive rate decreases when the bitmap size S increases, and increases when the input data amount D increases. We need to find the relationship between the number of hash functions N and S, D.

The misjudgment rate function is:

$$f(N)=\left(1-e^{-\frac{DN}{S}}\right)^{N} \tag{2}$$

Let $a=e^{\frac{D}{S}}$,

We can get $f(N)=\left(1-a^{-N}\right)^{N}$.

Take the logarithm of both sides:

$$\ln f(N)=N\ln\left(1-a^{-N}\right) \tag{3}$$

The derivative of both sides with respect to N:

$$\frac{1}{f(N)}\cdot f'(N)=\ln\left(1-a^{-N}\right)+\frac{Na^{-N}\ln a}{1-a^{-N}} \tag{4}$$

When f(N) takes its maximum value, then

$$f'(N)=0:$$

$$\ln\left(1-a^{-N}\right)+\frac{Na^{-N}\ln a}{1-a^{-N}}=0$$

We can get the results:

$$N=\ln 2\cdot\frac{S}{D}=0.7\cdot\frac{S}{D}$$

When the number of hash functions N = $0.7\cdot\frac{S}{D}$, the misjudgment rate $=2^{-\ln 2\cdot\frac{S}{D}}$.

Table1. Real isjudgment rate under various S/D and N combinations.

S/D	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8
2	0.393	0.400						
3	0.283	0.237	0.253					
4	0.221	0.155	0.147	0.160				
5	0.181	0.109	0.092	0.092	0.101			

DOI: 10.54254/2755-2721/31/20230171

Table1. (continued).

6	0.154	0.080	0.061	0.056	0.058	0.064		
7	0.133	0.062	0.042	0.036	0.035	0.036		
8	0.118	0.049	0.031	0.024	0.0217	0.0216	0.0229	
9	0.105	0.040	0.023	0.016	0.014	0.0133	0.0135	0.0145
10	0.095	0.033	0.017	0.012	0.009	0.0084	0.0081	0.0084
11	0.086	0.028	0.014	0.008	0.006	0.0055	0.0051	0.0050
12	0.080	0.024	0.011	0.006	0.005	0.0037	0.0032	0.0031
13	0.074	0.020	0.009	0.005	0.003	0.0025	0.0021	0.0019
14	0.069	0.018	0.007	0.004	0.002	0.0017	0.0014	0.0013
15	0.064	0.016	0.006	0.003	0.0018	0.0012	0.001	0.0008

From the above table 1, we can see that choosing the right number of hash functions can minimize the misjudgment rate and make the Bloom filter as efficient as expected in terms of storage and query.

3.1.3. Implementation process.

```
Bitmap=[0,0,...,0]   // Initialize an array of bits of length m, with all bits set to 0
function insert(elements[]):
    for item in elements[]:
    index=hash_function(element[item])%S   //S is the size of the bitm ap
    bitmap[index]=1   //Set the bit in the corresponding position to 1
end.
function query(elements[]):
    for item in elements[]:
    index = hash_function(element[i]) % S
    if bitmap[index] == 0:
        return false
    else:
        return true
end.
```

4. Conclusion

In this paper, we propose a new geolocation service solution, the Bloom Filter, which solves several problems.

Using Bloom filter, we can quickly judge whether a geographic location already exists or not, avoid saving duplicate geographic information, reduce storage space and query overhead, and reduce the time complexity O(n), where n refers to the amount of data in the hash function, which is much smaller than the amount of stored data, so the time complexity is greatly reduced.

In addition, Bloom filters have the ability to quickly determine the existence of a geographic location in a constant time to determine whether a geographic location already exists in the existing data set. In addition, Bloom filters can be used to pre-process and optimise query operations on location data. By adding frequently visited geographic locations to the Bloom filter, it is possible to quickly determine whether matching location data is likely to exist prior to querying. Finally, Bloom filters can be used to anonymise geolocation information to protect user privacy.

However, the Bloom filter also has some areas for improvement. We can't get the elements stored in the database directly through Bloom filters because the actual elements have been converted into bit symbols; in addition, Bloom filters have a certain false positive rate. Although the false positive rate can be minimised or even ignored by choosing the appropriate hash function and bitmap size, we still need to explore whether we can avoid the false positive rate completely.

DOI: 10.54254/2755-2721/31/20230171

References

[1] Tsou, MH. (2002). An Operational Metadata Framework for Searching, Indexing, and Retrieving Distributed Geographic Information Services on the Internet. In: Egenhofer, M.J., Mark, D.M. (eds) Geographic Information Science. GIScience 2002. Lecture Notes in Computer Science, vol 2478. Springer, Berlin, Heidelberg.

[2] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (Nov. 2006), pp. 205–218.

[3] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles (Oct. 2003), pp. 29–43.

[4] Hu, T. C., Tan, K. C.: Least upper bound on the cost of optimum binary search trees. Acta Informatica 1, 307–310 (1972).

[5] Bayer, R. Symmetric binary B-Trees: Data structure and maintenance algorithms. Acta Informatica 1, 290–306 (1972). https://doi.org/10.1007/BF00289509

[6] Bloom Burton H. 1970. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13, 7 (July 1970), 422–426.