2019 级

《大数据存储与管理》课程

课 程 报 告

姓　名　马世拓

学　号　U201914900

班　号　CS1901 班

日　期　2022.04.19

基于 Bloom Filter 的多维数据表示与检索研究

华中科技大学 马世拓 CS1901
2793055528@qq.com

摘要

Bloom Filter 模型自 1970 年提出到现在由来已久，能够解决某些元素是否为集合中元素的判断问题，突破了传统哈希函数的映射和存储元素的方式，在数据库、分布式系统、计算机网络等多个领域扮演着重要角色。而多维数据的表示与检索是存储与信息检索领域的难题，如何设计相应的模型和数据结构是解决此类问题的关键。本文在总结前人工作的基础上进行了利用 Bloom Filter 进行多维数据表示和检索的最优误报率等初步理论推导和实验，得出它与一般集合类型的异同与优势。

关键词: Bloom Filter，高维数据，误报率，存储

Abstract

The Bloom Filter model has been around for a long time since it was proposed in 1970. It can solve the problem of judging whether certain elements are elements in a set, breaking through the traditional hash function mapping and storing elements, and is widely used in databases, distributed systems, and computer networks. plays an important role in many fields. The representation and retrieval of multi-dimensional data is a difficult problem in the field of storage and information retrieval. How to design the corresponding model and data structure is the key to solving such problems. On the basis of summarizing the previous work, this paper conducts preliminary theoretical derivation and experiments such as the optimal false alarm rate for multi-dimensional data representation and retrieval using Bloom Filter, and concludes its similarities, differences and advantages with general collection types.

Key words: Bloom Filter, high-dimensional data, false positive rate, storage

1. 研究背景

高维数据的表示与检索是计算机存储领域的一项热门话题。而这两个话题是密切相关的，在不同的表示方式下应选用不同的查找策略，如线性遍历、二分查找、哈希查找等。这一过程与数据结构的设计密不可分。但高维数据在计算机当中具有容量大、难压缩、难检索的特性，容易造成维数灾难。无论是基于机器学习的高维数据压缩表示还是基于特征矢量的相似度索引结构都难以对高维数据进行高效率表示[1]，于是 Bloom, Burton.H.等提出 Bloom Filter 模型以期对高维数据在一定容错率下进行高效表示[2]。

2. 国内外研究综述

2.1 模型的发展

1970 年 Bloom 和 Burton.H.等人提出 Bloom Filter 模型用以解决某些元素是否为集合中元素的判断问题，突破了传统哈希函数的映射和存储元素的方式，通过一定错误率换取存储空间和查找时间，大大提升了性能[2]。随后，Mitzenmacher, Michael.等人又在原始 Bloom Filter 的基础上基于香农信息熵理论提出压缩型 Bloom Filter，以减小在分布式系统中不同节点进行检索时的交互与信息量[4]。而为了处理集合元素的动态变化问题，即元数据衰减与删除问题，F.Li.和 Cao.P.等人设计了计数型 Bloom Filter 进行修正[5]。但计数型 CBF 存在易溢出的问题，于是 Bonomi F.等人设计了 dlCBF 模型[6]，通过引入一种叫做 d-Left hash 的更均衡的哈希函数来降低 CBF 向量中 counter 的位数有效解决了溢出问题。与此同时，传统 CBF 的存储空间也可进一步优化，Ficara D, Giordano S, Procissi G 等人提出多层压缩式 CBF 有效提升了存储空间冗余[7]。

对于高维数据表示方面的研究，文献[8]认为 Bloom Filter 并不完全适合许多支持网络的新网络应用程序服务，例如具有多个属性而不是单个属性的项目的表示和查询，并提出了三种 BF 的变体结构：并行布隆过滤器、带有哈希表的 PBF（PBF-HT）和带有 Bloom Filter 的 PBF (PBF-BF)，具有多个属性，促进了更快的查询服务，并且大大降低误报概率，从而节省存储空间。而在文献[9]中提出了一种适用于超大规模文件系统的可扩展和自适应分散元数据查找方案，将元数据服务器 (MDS) 组织成多层查询层次结构并利用分组的 Bloom Filter 分层查询模式，显著提高了元数据管理的可扩展性和超大规模存储系统中的查询效率。

2.2 模型的应用场景

Bloom Filter 作为一种高维数据的高效压缩表示与存储方式，它在信息检索、存储、数字图像处理等领域有着重要应用。在数据库的基本操作、字典查询等领域 Bloom Filter 有着重要应用[10-13]，包括数据的查询与插入等。而在网络领域中，Bloom Filter 也被广泛应用于 P2P 网络[14]、网络测量管理[15]、入侵检测[16]等领域。文献[17]认为，随着 Bloom Filter 有关研究逐渐深入，Bloom Filter 在分布式系统和网络领域将逐渐得到更深的应用。

3. 我们的工作

我们在总结前人工作的基础上，进行了如下分析和实验：

1. 总结 Bloom Filter 的有关研究历程
2. 对 Bloom Filter 模型进行建立与调优
3. 进行实验观察不同数据体量和参数配置下的 Bloom Filter 性能

4. Bloom Filter 理论模型构建

4.1 模型建立

一个标准 Bloom Filter 模型的架构图如图 1 所示：

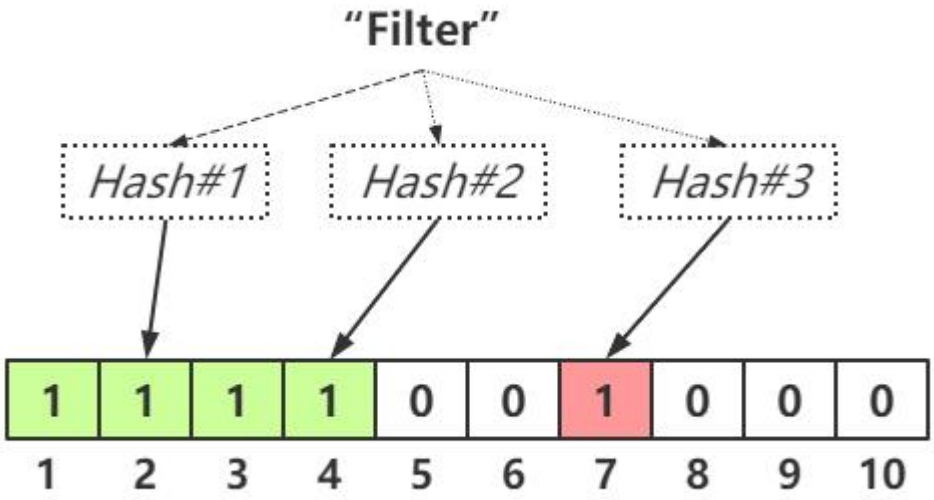


图 1. 标准 Bloom Filter 架构图

为了表达含有 n 个元素的集合 x，Bloom Filter 使用 k 个不同的哈希函数将其散列到 m 位位数组 BFV 的其中 k 个位。对于其中任意一个元素，第 i 个哈希函数对应的位数就会被置为 1。当查询某个元素 xi 时使用所有的哈希函数进行查找，若对应的所有位置全为 1 则表示属于集合，否则则不属于集合中。分析这一特性我们发现，假设集合元素为 50 个 ASCII 码字符的 URL，采用直接存储需要 400(n+1)个 bit 的空间，而 Bloom Filter 仅需 m 位就够了[2]，并且 m<<400n。但这一模型也容易出现误报情况，即将一小部分不属于集合中的元素判定为属于，这一误差水平我们将在 4.2 中讨论。

与经典的哈希函数相比，Bloom filter 最大的优势是它的空间效率。 另外，由于 Bloom filter 不用处理碰撞，无论集合中元素有多少，也无论多少集合元素已经加入到了位向量中，Bloom filter 在增加或查找集合元素时所用的时间都为哈希函数的计算时间。 由于 Bloom filter 对集合中的元素进行了编码，因此想从 Bloom filter 的位向量中恢复集合元素并不容易，如果不想让别人直接看到集合元素，这样的编码处理相当于一种加密，从而有利于保护隐私。

4.2 指标分析

对于经典 Bloom Filter 的错误率，对于使用 k 个哈希函数，向 m 位长的 Bloom filter 中装入 n 个元素后，位向量中某一位仍然为 0 的概率 p 为：

$$p=\left(1-\frac{1}{m}\right)^{kn}\approx e^{-\frac{kn}{m}} \tag{1}$$

对(1)进行变形易得：

$$k=-\frac{m}{n}\ln p \tag{2}$$

则对于 k 位而言，整体错误率为：

$$f_p=\left[1-\left(1-\frac{1}{m}\right)^{kn}\right]^k=(1-p)^k=e^{k\ln(1-p)} \tag{3}$$

最终经过化简，可以得到整体错误率为

$$f_p=\exp\left(-\frac{m}{n}\ln(p)\ln(1-p)\right) \tag{4}$$

图 2(a)探究了整体错误率随单个位错误率的变化曲线，对 f 求极值易得其极小值在 p=0.5 处。于是我们得到最优解为：

$$k=\left[\ln 2\left(\frac{m}{n}\right)\right] \tag{5}$$

我们还探究了其他参数对问题的影响，例如，我们将(2)式带入(4)中可以得到以下代换形式：

$$\ln f_p=k\ln\left(1-e^{-\frac{kn}{m}}\right) \tag{6}$$

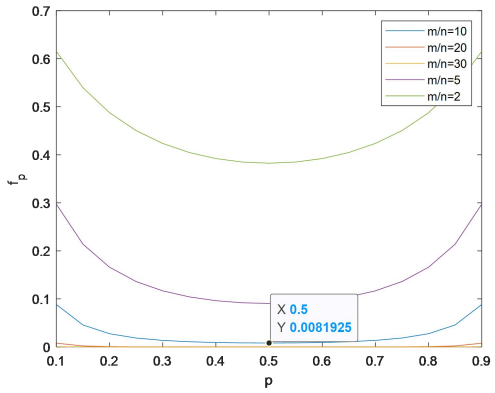
图 2(b)和图 2(c)探究了参数大小对错误率的影响。从图 2(b)中可以看出若 k 取值越大则总体错误率初始水平越低但变化也越迅速，具体表现为曲线斜率。图 2(c)中可以得到 m 与 n 的比值和 K 呈现反比例关系，并且总体错误率越高则曲线越贴近坐标轴，变化越快。

图 2(d)描述了最优情况下总错误率和 mn 比值的关系，可以看到它是一条指数衰减曲线。事实上，将(5)带回(6)易得曲线的方程为：

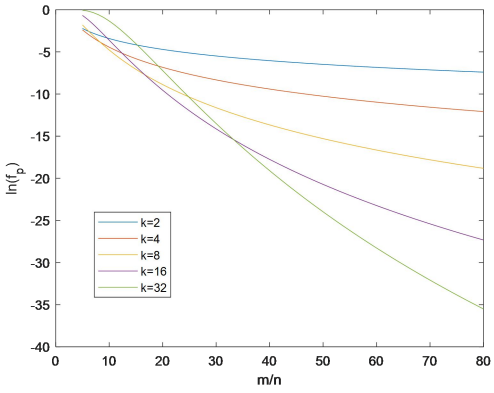
$$f_p=(0.6185)^{\frac{m}{n}} \tag{7}$$

由上述推导易知，当 m 和 n 的比值越大则要求哈希函数的个数 k 越大，并且其错误率越小。

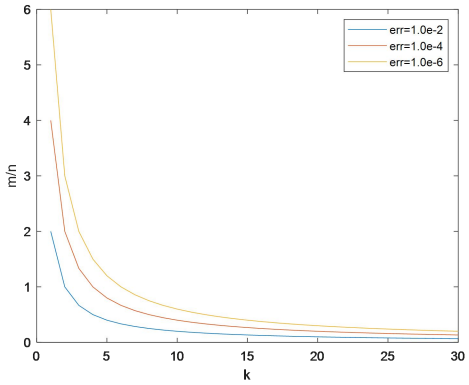
除最基本的模型探究以外，如文献[17]也对分布式系统中的 Bloom Filter 做出了误差分析和建模，对模型误差进行了进一步修正使其能够适应分布式系统。



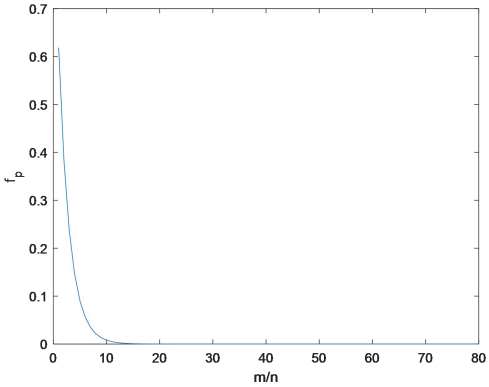
(a) P 的极值点对总错误率影响



(b) 不同 k 取值下 mn 比值与错误率曲线



(c). k 与 mn 比值之间的关系



(d) 最优解曲线

图 2. 不同参数设置对总错误率的影响探究

5. 实验设计与结果

为了对 Bloom Filter 的性能进行观测和探究，我们设计了有关实验对 Bloom Filter 进行了简单实现。

5.1 数据结构设计

我们基于 Bloom Filter 算法设计了如下数据结构：

Bloom Filter 类具有三种基本操作，增加，清除和判定。Bloom Filter 类可以设置 k、m、n 三个参数值，当进行增加操作时对所有哈希进行一次操作，若全部满足则将对应位置变为 1；当进行清除操作时将整个比特数组变为 0；当进行判定操作时同样进行哈希散列以后对对应位置进行判定，若不全为 1 则判定不在集合中。代码见附录。

5.2 基准测试

我们选用包含 466550 个单词的词汇数据集进行测试，对比了 Bloom Filter 和集合类数据结构之间的性能差异。分别添加 100、1000、10000、100000、200000、

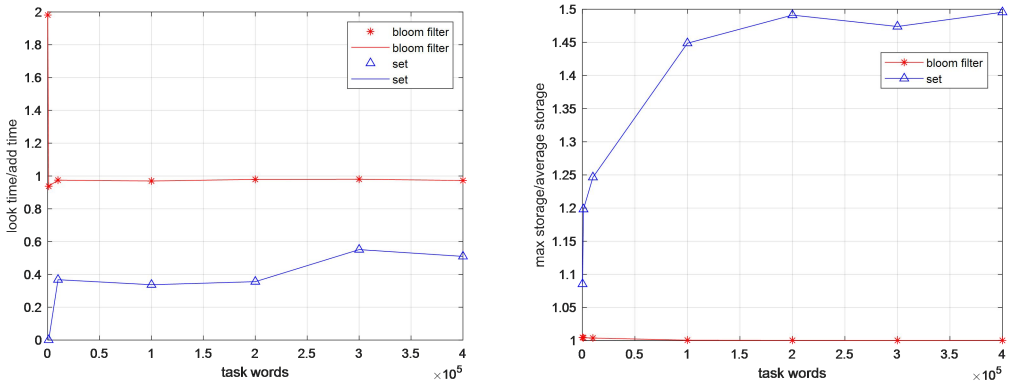
300000、400000 个单词到数据结构中，其性能如表 1 所示：

表 1. Bloom Filter 和 Set 对比实验数据

数据量	Bloom Filter				Set		
	错误率	添加时间	查询时间	高峰存储	添加时间	查询时间	高峰存储
100	0.0000	0.0010069	0.0019946	0.15197	0	0	0.012128
1000	0.0020	0.015957	0.01496	0.01493	0.0007117	0	0.042849
10000	0.0016	0.1585332	0.1545094	0.25104	0.0012071	0.0004439	0.65725
100000	0.0017	1.6188392	1.5695068	1.19315	0.0177745	0.0059899	6.293347
200000	0.0018	3.1624144	3.0966309	2.26601	0.0506821	0.0180317	12.73152
300000	0.0016	4.7718511	4.6775587	3.16250	0.0688171	0.0379361	12.58480
400000	0.0018	6.3443497	6.1698713	4.44070	0.103723	0.0528723	25.31443

(时间单位：s，存储单位：MB)

从表 1 中我们很容易看到，对于集合类数据类型的添加时间和查询时间都远小于 Bloom Filter，但查询时间与添加时间的比值而言 Bloom Filter 是趋近于 1 的而集合类数据类型的这一比值会随着数据量增大而稳定在 0.5 左右。就占用空间容量来看，对于小体量数据集合数据展现出较强的容量优势，而当数据量增大以后集合的容量在指数增长而 Bloom Filter 的存储容量则是线性增长，适合于大数据场景。除此以外，图 3 中绘制了查询时间与存储时间比值随数据量变化的曲线图以及高峰存储与平均存储空间比值随数据量变化的曲线图，能够更清晰地看到，随着存储容量的增长，无论是查询时间与添加时间的比值还是存储容量峰值与均值之比，Bloom Filter 总是维持在 1 左右，这是一项重要性质，能够在大数据的存储中加快速度的同时降低更小内存。



(a) 查询时间与添加时间的比值　　(b) 存储容量峰值与存储容量均值比

图 3. 两种数据结构的对比

6. 总结

本文是我在大数据存储与管理这门课的结课报告，选择了选题一。在本文中我们总结了 Bloom Filter 模型的发展历程，对 Bloom Filter 的误报率进行了初步推导与探索，并设计相关实验进行了初步描述。由于是第一次大胆尝试将自己的课程报告写成论文的形式，可能学术规范性等很多地方没太注意，还请批评指正！

参考文献

[1] 董道国. 高维数据索引结构研究[D].复旦大学,2005.

[2] Bloom, Burton H . Space/time trade-offs in hash coding with allowable errors[J]. Communications of ACM, 1970.

[3] Mitzenmacher, Michael. Compressed bloom filters. 2001.

[4] Coulouris, George F . Distributed systems : concepts and design[J]. Computer Communications, 2011, 12(4):240-240.

[5] F Li, Cao P . Summary cache: a scalable wide-area Web cache sharing protocol[J]. IEEE/ACM Transactions on Networking, 2000, 8(3):P.281-293.

[6] Bonomi F, Mitzenmacher M, Panigrahy R, et al. An improved construction for counting bloom filters. In: Lecture Notes in Computer Science, Zurich, Switzerland, September 2006

[7] Ficara D, Giordano S, Procissi G. MultiLayer compressed counting bloom filters. In: Proc of the Infocom, Phoenix, AZ, USA, April 2008

[8] B. Xiao and Y. Hua, “Using Parallel Bloom Filters for Multi-Attribute Representation on Network Services,” IEEE Trans. Parallel and Distributed Systems, vol. 21, no. 1, pp. 20-32, Jan. 2010

[9] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, “Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems,” Proc. 28th Int’l Conf. Distributed Computing Systems (ICDCS ’08), pp. 403-410, 2008

[10] Mullin J K. Optimal semijoins for distributed database systems. IEEE Transactions on Software Engineering,1990,16(5):558～560

[11] Udi M, Sun W. An algorithm for approximate membership checking with application to password security. Information Processing Letters, 1994, 50(4):191～197

[12] Gremillion L L. Designing a bloom filter for differential file access. Communications of the ACM,1982,25(9):600～604

[13] James K M. A second look at bloom filters. Communications of the ACM,1983,26(8):570～571

[14] Reynolds P, Vahdat A. Efficient peer-to-peer keyword searching. In: Proc of Middleware, Riode Janeiro, Brazil, June 2003

[15] Heeyeol Y, Mahapatra R. A memory-efficient hashing by multi-predicate bloom filters for packet classification. In: Proc of the Infocom, Phoenix, AZ, USA, April 2008

[16] Locasto M E, Parekh J J, Keromytis A D, et al. Towards collaborative security and P2P intrusion detection. In: Proc of SMC 2005, NY, USA, June 2005

[17]Zhu Y， Hong J . False Rate Analysis of Bloom Filter Replicas in Distributed Systems[C]// International Conference on Parallel Processing. IEEE, 2006.

附录

Bloom filter.py

```python
# murmur3 hashing
import mmh3
import math


def optimal_hash_count(size, element_count):
    return int(math.ceil((size / element_count) * math.log(2.0)))
def optimal_bit_size(falsePositiveProbability, element_count):
    return int(math.ceil((-1.0 * element_count * math.log(falsePositiveProbability)) /
math.pow(math.log(2.0), 2.0)))
class BloomFilter:
    def __init__(self, bit_count=8, hash_count=4):
        self.bit_count = bit_count
        self.hash_count = hash_count
        self.seeds = range(hash_count)
        self.bit_field = [0 for x in range(math.ceil(bit_count/8))]
    def add(self, value):
        # Hash value hash_count times and set the correct bit to 1
        for idx in range(self.hash_count):
            hashed_result = mmh3.hash(value, self.seeds[idx])
            position = hashed_result % self.bit_count
            byte_position = int(position/8)
            bit_mask = 1 << (position % 8)
            self.bit_field[byte_position] |= bit_mask
    def exists(self, value):
        for idx in range(self.hash_count):
            hashed_result = mmh3.hash(value, self.seeds[idx])
            position = hashed_result % self.bit_count
            byte_position = int(position/8)
            bit_mask = 1 << (position % 8)
            # If there there is one bit that doesn't match, we are sure that the value isn't present
            # This is where we can get a false positive (if two value share the same bit signature)
            if self.bit_field[byte_position] & bit_mask == 0:
                return False
        return True
    def clear(self):
        self.bit_field = [0 for x in range(self.bit_count)]
```

Bloom filter bench.py

```python
import bloom_filter
import tracemalloc
import time

print("---------------- Bloom filters benchmark ----------------")
words = []
with open("words.txt") as f:
    words = f.read().splitlines()
print(f"Loaded: {len(words)} words")
desired_false_prob = 0.01
size = bloom_filter.optimal_bit_size(desired_false_prob, len(words))
hash_count = bloom_filter.optimal_hash_count(size, len(words))
print(f"{desired_false_prob} error => {size} bits and {hash_count} hash")
tracemalloc.start()
print("Creating the bloom filter data structure")
start_time = time.time_ns()
bf = bloom_filter.BloomFilter(size, hash_count)
print(f"Adding {len(words)} elements")
for word in words:
    bf.add(word)
end_time = time.time_ns()
elapsed_time = end_time - start_time
print(f"Time to add all elements: {elapsed_time}ns ({elapsed_time / 1000000000}s)")
print(f"Looking up every element")
start_time = time.time_ns()
for word in words:
    if bf.exists(word) == False:
        print(f"Error: {word} wasn't found")
end_time = time.time_ns()
elapsed_time = end_time - start_time
print(f"Time to know if the elements were present: {elapsed_time}ns ({elapsed_time / 1000000000}s)")
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(
    f"Current memory usage is {current / 10**6}MB; Peak was {peak / 10**6}MB")
print(f"Clearing the bloomfilter")
bf.clear()
print(f"Validating the error count")
errorCount = 0
for word in words:
    if bf.exists(word) == True:
        errorCount += 1
    bf.add(word)
print(f"There was {errorCount} error(s) [error rate: {errorCount / len(words)}]")
```

Set bench.py

```
import tracemalloc
import time

print("---------------- Set benchmark ----------------")
words = []
with open("words.txt") as f:
    words = f.read().splitlines()
words=words[0:400000]
print(f"Loaded: {len(words)} words")
tracemalloc.start()
print("Creating the set data structure")
start_time = time.time_ns()
set_struct = set([])
print(f"Adding {len(words)} elements")
for word in words:
    set_struct.add(word)
end_time = time.time_ns()
elapsed_time = end_time - start_time
print(f"Time to add all elements: {elapsed_time}ns ({elapsed_time / 1000000000}s)")
start_time = time.time_ns()
for word in words:
    if word not in set_struct:
        print(f"Error: {word} wasn't found")
end_time = time.time_ns()
elapsed_time = end_time - start_time
print(f"Time to know if the elements were present: {elapsed_time}ns ({elapsed_time / 1000000000}s)")
current, peak = tracemalloc.get_traced_memory()
print(
    f"Current memory usage is {current / 10**6}MB; Peak was {peak / 10**6}MB")
tracemalloc.stop()
```

Bloom filter test.py

```
import bloom_filter

print("---------------- Bloom filters test ----------------")
print("Creating the bloom filter data structure")
bf = bloom_filter.BloomFilter(8, 5)
val1 = "Monkey"
val2 = "Computer"
print(f"Making sure that the word '{val1}' isn't in the bloom filter yet")
print(f"'{val1}' exists? {bf.exists(val1)}")
print(f"\nAdding the word '{val1}' to the bloom filter")
bf.add(val1)
```

```
print(f"\nMaking sure the word '{val1}' is present in the bloom filter")
print(f"'{val1}' exists? {bf.exists(val1)}")
print(f"\nMaking sure the word'{val2}' isn't present in the bloom filter")
print(f"'{val2}' exists? {bf.exists(val2)}")
```