Efficient way of information transportation in the distributed joining problem

Sixiao ZHU

1 Introduction

Distributed Join is an important topic in distributed database. One of the method to realize distributed join is Bloom Join, in this work we propose an original method to improve the false positive rate of Bloom Join under fixed network transmission cost, we provide a rigorous probabilistic analysis of our proposed method, we use experiment to show the effectiveness of our approach.

2 Distibuted Join

2.1 Join operation

We think of a table T of arity n as a collection of n-ary records, for each record e, $e[i]$ denotes its ith attribute. A join operation, denoted $join(T_1, T_2, i, j)$ of two tables on ith attribute of T_1 and jth attribute of T_2 returns a new table, constructed as concatenation of e_1 and e_2 for all the couple $(e_1, e_2) \in T_1 \times T_2$ such that $e_1[i] = e_2[j]$.

(Assumption 1) In a typical application, the entries that actually participate the join operation is much smaller than the whole table.

Modern database system make use of optimized approach such as Hash Join (DeWitt) and Sort Merge Join (Graefe, 1994), essentially these methods build an index on the join column of one of the table, thus omitting the need to scan the whole Cartesian product of the two tables.

2.2 Join operation in the distributed setting

When the two tables are located on separated servers S_1 and S_2, we must transfer information from one server to another to complete the join operation. One straight forward approach is to transfer the whole table T_1 from S_1 to S_2, then do the joining operation locally on S_2. However it imposes heavy network transportation cost.

3 Bloom filter and its application in our problem

3.1 Bloom Filter introduction

A Bloom Filter (Bloom Filter, n.d.) is a space-efficient probabilistic data structure used to test whether an element is in a set. Here we introduce a mathematical formalization of Bloom Filter for later discussion. We abstract the Bloom Filter as a sequence of bits $B = \{B_i\}$, We denote Σ an alphabet, $h = \{h_j\}$ a set of hash functions, in which $h_j: \Sigma^* \to [\![1, \dots, m]\!]$. Let E a set of entries inserted to B, we denote the hashing process $B = h(E) = \{B_i, i \in [\![1, \dots, m]\!] \mid B_i = 1 \; if \; \exists j \; and \; \exists \; e \in E, h_j(e) =$

$i, else\ 0\}$, we denote $\mathbb{1}_B = \{i|B_i = 1\}$, with **an abuse of notion**, we denote by $h^{-1}(B) = \cup \{E|\mathbb{1}_{h(E)} \subset \mathbb{1}_B\}$, that is the set of all the possible entries that would be recognized by B as member of E.

We denote $\gamma = m/|E|$, as the number of bits used by B for each value inserted. Normally this will be a pre-defined value as we would like to limit the cost. Let f be the false positive probability of B, that is the probability to miss identify an entry as a member of E

Under the assumptions 1. the k hash functions are independent. 2. All hash functions generate uniform distribution among m bits, we can proof:

$$f = \left(1 - \left(\frac{1}{m}\right)^{-k|E|}\right)^k =_{m\to\infty} \left(1 - e^{-k/\gamma}\right)^k$$

The optimal choice of k given fixed γ is $(\ln 2)\gamma$, which will lead to half of bits in B set to 1, and will bring the relation of f and γ:

$$f = \left(\frac{1}{2}\right)^{\gamma \ln 2} \tag{1}$$

Later we will show that in our specific hash join problem, some methods to improve this bould.

3.2 Bloom Filter's application in distributed join

Consider problem $join(T_1, T_2, i, j)$, we denote $A_1 = \{e[i]|\ \forall e \in T_1\}$, and A_2 accordingly.

The idea of Bloom Join is studied in (Ramesh S., 2008), Bloom Filter gives a way to approximately express the set $T_1[i]$ in a compact form. We generate a Bloom Filter $B = h(A_1)$. As shown in Figure 2, We transfer the B from S_1 to S_2, we find by querying B, a candidate set $E = h^{-1}(B) \cap A_2$, we have $T \subset E$ because obviously

$$A_1 \subset h^{-1}\big(h(A_1)\big)$$

Thus

$$T = A_1 \cap A_2 \subset h^{-1}\big(h(A_1)\big) \cap A_2 = E$$

How we actually conduct the join is a choice upon the specific application, for example, S_2 select its records $\{e \in T_2|e[j] \in E\}$ and sends them to a 3ed server (typically the query client side). We do the same process reversely from S_2 to S_1, once data both servers have arrived, we conduct the actually join from the 3ed server.

In brief, benefit by Bloom Filter, we run a preselection on T_2 set according to a fussy description T_1 set, we transfer via network a much smaller candidate set of records (by Assumption 1), thus the transmission efficiency.

4 First improvement

When dealing with huge amounts of data, a slight fluctuation of false positive rate will lead to significant performance discrepancy, we seek to reduce "False positive per transmission price we pay" f/γ as much as possible.

(Naïve Improvement) As indicated by (1), we could always use bigger γ as its will bring exponential drop of f, by increasing γ to αγ we pay α times of transmission cost, it will reduce f to f^{α}, as f is typically smaller than 0.1, it is a considerable refinement.

However, the following improvement will always lead to bigger drop given the same expansion of transmission cost. Without loss of generality, we assume $|A_2| = |A_1|$.

(Improvement 1) We notice that if we generate another Bloom Filter $B' = h(E)$ from E using the same hash setting h, send it back to S_1, and generate $E' = h^{-1}(B') \cap A_1$ from S_1 side, then E' also will contain T, as

$$T \subset E \subset h^{-1}\big(h(E)\big) = h^{-1}(B')$$

Thus

$$T \subset h^{-1}(B') \cap A_1 = E'$$

We denote $\rho = |T|/|A_2|$ which will be for example around 1/10 under Assumption 1, then $|E| \approx \rho|A_2| + f(1-\rho)|A_2|$ if we still get to use the same number of bits as B for B', that is$m' = m$, we can have B' much sparser, formally $\gamma' = \frac{m'}{|E|} = \frac{\gamma}{(\rho+(1-\rho)f)} \approx \frac{\gamma}{\rho}$. The approximation comes from the fact f is typically controlled in the order of magnitude of 0.01. The situation is illustrated in Figure 3.

For we use the same hash function for B and B', distribution of 1's of B' has statically correlation with that of B, by denoting $\mathbb{1}_B = \{i|B_i = 1\}$, it is not hard to see $\mathbb{1}_{B'} \subset \mathbb{1}_B$, as shown in Figure 1

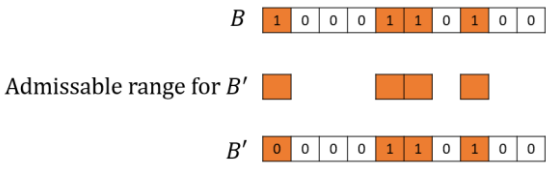


Figure 1

As range of $\mathbb{1}_{B'}$ is *a priori* limited to $\mathbb{1}_B$, from S_2 side we can view B' as a Bloom Filter operating on a bit set of length $|\mathbb{1}_B|$, for a random entry $e \in A_1$, and jth hash function of h, we denote

$$P'_{1,j} = Probability(h_j(e) \in \mathbb{1}_{B'} \mid h_j(e) \in \mathbb{1}_B)$$

Under uniform distribution assumption of $h_j(e)$ in $\mathbb{1}_B$, and the fact that $\mathbb{1}_{B'}$ distribute uniformly in $\mathbb{1}_B$we thus get:

$$P'_{1,j} = 1 - P'_{0,j} = 1 - \left(1 - \frac{1}{|\mathbb{1}_B|}\right)^{k|E|}$$

We denote f' the false positive rate of B', as we use $k' = k$, we have

$$f' = \prod_{h_j \in h} P'_1, j = \left(1 - \left(1 - \frac{1}{|\mathbb{1}_B|}\right)^{k|E|}\right)^k$$

Under the assumption that B adopts optimal configuration of k, which leads to $|\mathbb{1}_B| = \frac{1}{2}|B|$, we can estimate f' as

$$f' = \left(1 - e^{-\frac{2k|E|}{|B|}}\right)^k = \left(1 - e^{-\frac{2\rho k}{\gamma}}\right)^k = \left(1 - \left(\frac{1}{2}\right)^{2\rho}\right)^{\gamma \ln 2}$$

With ρ set to 1/10, we have $f' = 0.13^{\gamma \ln 2}$, develop the equation above we get

$$f' =_{\rho \to 0} (1.386\rho)^{r ln 2} \tag{2}$$

For we have sent two Bloom Filters (back and forth) of the same size, when comparing, for the same amount of transmission, Naïve Improvement would be allowed double its γ, that is

$$f = \left(\frac{1}{2}\right)^{2\gamma \ln 2} = 0.25^{\gamma \ln 2} \tag{3}$$

We can see for most case our approach will outperform the Naïve Improvement.

5 Second Improvement

If we substitute the hash functions h used for generating B' and E' by another set of hash functions h' *independent* from h, we will receive a significant improvement of f' without paying any performance price, as h' removes the correlation between B and B'.

For ease of analysis, we assume we still use the same m' as m, and we adjust k' the to the optimal with respect to γ'. The f' under this setting would be:

$$f' = \left(\frac{1}{2}\right)^{\gamma' \ln 2} = (0.5^{\rho})^{\gamma \ln 2} \tag{4}$$

It improves exponentially result of (3).

6 Rule of compression in our work

6.1 Compressing the sparce Bloom Filter

By Shannon's Source Coding Theorem (Shannon, 1948), the Information Entropy (denoted H) of a Bloom Filter of signifies the limit of lossless compression. In the optimal configuration of k, the probability that a bit of B being set to 1 is 0.5, we cannot gain anything from compression. However, in the first improvement, we has used $k' = k$, which is much lesser than the optimal configuration $\gamma' \ln 2$ of k', making B' very sparse, thus renders possibility of compression. Using Arithmatique Encoder, denote z the size of bits of B' after compression, after averaging large number of experiments, we would have

$$z < m' H(p) \ll m'$$

Where we denote p the probability a bit of B' is set to zero, we have

$$p = \left(1 - \frac{1}{m'}\right)^{k|E|} \approx e^{\frac{\rho k}{\gamma}} = \left(\frac{1}{2}\right)^{\frac{1}{\rho}} \gg \frac{1}{2}$$

The first sign of inequality reflects the fact that $\mathbb{1}_B$, do not uniformly distribute amount all the positions of B', but concentrate in the range of $\mathbb{1}_B$, thus further lowers the complexity of B'. With the setting $\rho = 1/10$, we have a compressing rate of 0.24.

6.2 Compressed Bloom Filter

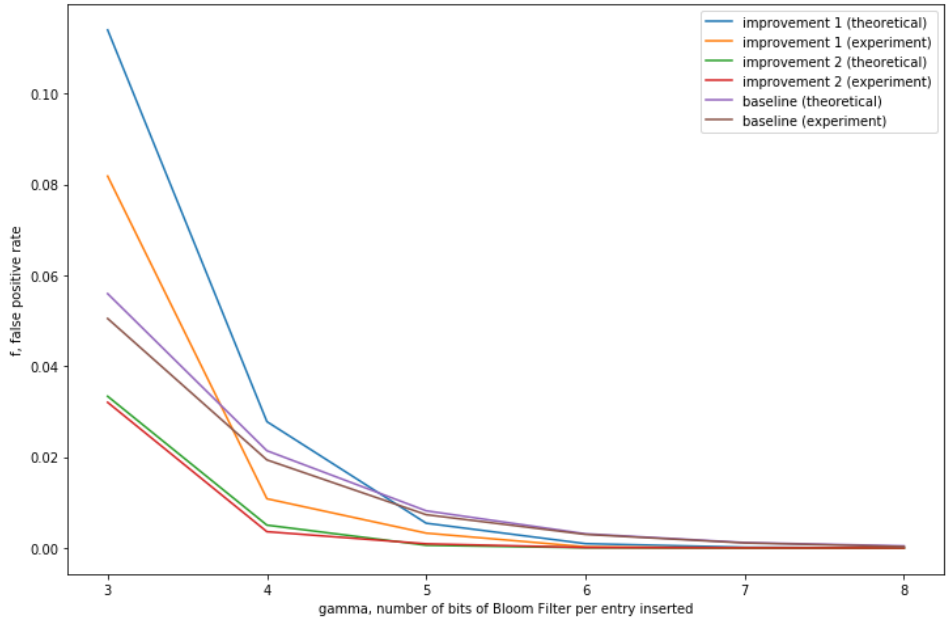
The compression allow us to use bigger γ for the Bloom Filter (Though we would not choose k as optimized for this big γ). In (Michael, 2002), the author discovered the following astonishing result: Under a fixed cost of transmission γ, that is size of Bloom Filter after compression in our case, the false positive of compressed Bloom Filter will always outperform regular Bloom Filter. Given

unlimited use of hash functions, we can approach an optimal false positive rate $f = \left(\frac{1}{2}\right)^{\gamma}$ compared to $\left(\frac{1}{2}\right)^{\gamma \ln 2}$ for regular Bloom Filter. Applying this technique in our framework, the transmission cost will drop to the next level.

7 Experiment

We generate randomly two sets of strings A_1 and A_2, making $\rho = 1/10$, we construct our Bloom Filter using Murmurhash3 (MurmurHash, n.d.) hash function, different hashing functions are construct by using the same `mmr` function but appending different characters to the end of the input string. We conduct 3 experiments, representing the Naïve Improvement (Considered as baseline in our experiments), the Improvement 1, and Improvement 2. We have also compared the false positive rate of our theoretical analysis and the experiments.

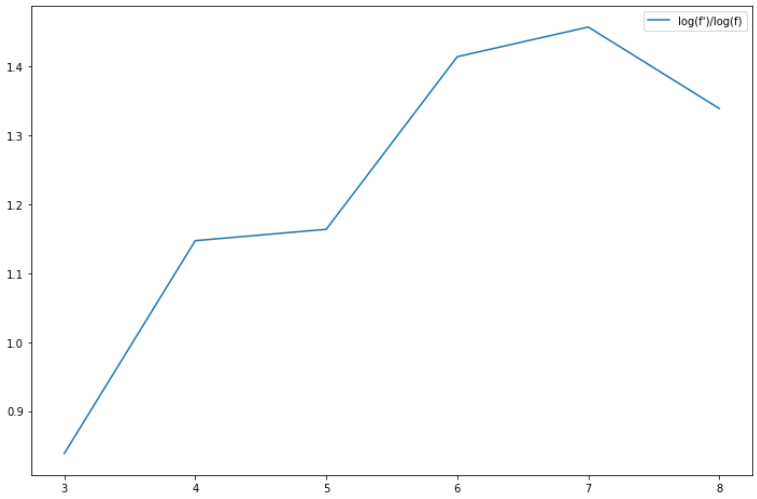
The following figure plots the trending of f for different γ configuration:



We can conclude:

- Two of our improvements indeed improve f/γ ratiao
- Using the independent hashing function for B and B' indeed make Improvement 2 better then Improvement 1

The following figure plots $\ln(f_{improvement\ 1}) / \ln(f_{naive\ improvement})$ for different γ value.



We can see the value falls in the range of 1.0 – 1.5, this fits our theoretical analysis of equation (2) and (3), which indicates that this ratio should be $\frac{\ln(0.138)}{\ln(0.25)} = 1.42$.

8 Further improvement

Bloom filter itself has many variations and improvements (Luo, 2018) in regards of choice of hashing function, mechanism of storing; and so on. Besides Bloom filter; there are other data structure for member ship tests, for example, Cuckoo Filter (Fan, 2014) guarantees $\gamma = (\log_2 \frac{1}{f} + 2)/\alpha$, where α is the hash table load factor and f the false probability, comparing to $\gamma = 1.44 \log_2 \frac{1}{f}$ for a regular Bloom Filter. Future work can be done in exploring the adoption of these tools in our frameworks.

9 Bibliography

Bloom Filter. (s.d.). Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/Bloom_filter

DeWitt, D. J. (s.d.). Multiprocessor hash-based join algorithms . *University of Wisconsin-Madison, Computer Sciences Department*.

Fan, B. e. (2014). Cuckoo filter: Practically better than bloom. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*.

Graefe, G. (1994). Sort-merge-join: An idea whose time has (h) passed? *Proceedings of 1994 IEEE 10th International Conference on Data Engineering. IEEE*.

Luo, L. e. (2018). Optimizing bloom filter: challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials.*

Michael, M. (2002). *Compressed bloom filters." IEEE/ACM Transactions on Networking (TON)*, 604-612.

MurmurHash. (s.d.). Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/MurmurHash

Ramesh S., P. O. (2008). Optimizing Distributed Joins with Bloom Filters. *Distributed Computing and Internet Technology. ICDCIT* .

Shannon, C. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal*, pp. 379–423, 623-656.

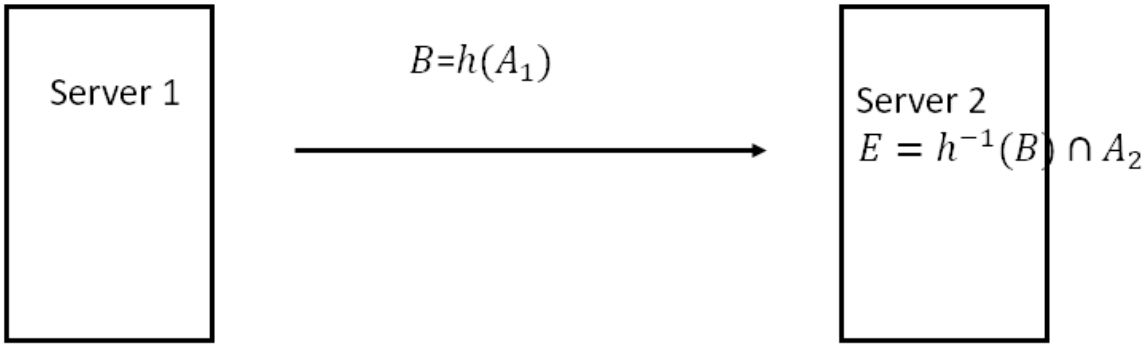


Figure 2

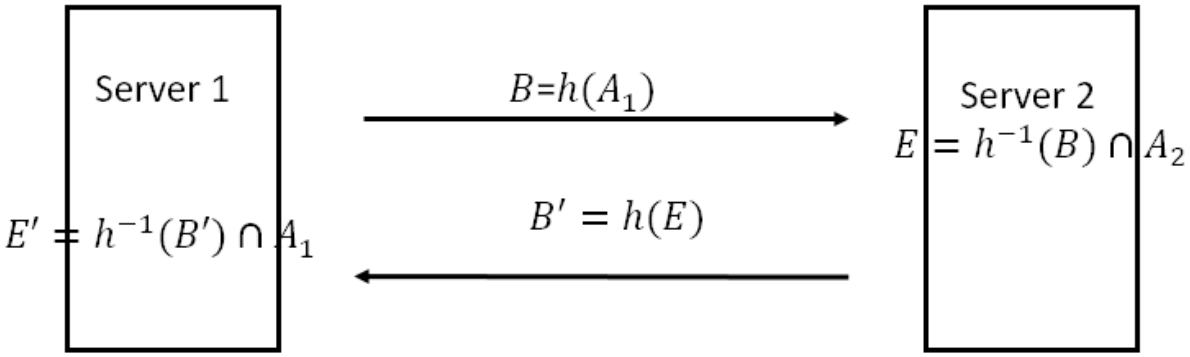


Figure 3

Code for experiment

```
import mmh3
import numpy as np


# generalize set
def id_generator(size=8, chars=string.ascii_uppercase + string.digits):
    return ''.join(random.choice(chars) for _ in range(size))
def gen_n(n):
    return [id_generator() for i in range(n)]


# contructing a bloom filter
def bf(l,k,m):
    b = np.zeros(m)
    for i in l:
        for j in range(k):
            h = mmh3.hash(i + str(j))
            if h < 0:
                h = -h
            b[h%m] = 1
    return b
# test function of bloom filter
def test(b, k, m, x):
    ret = True
    for i in range(k):
        h = mmh3.hash(x + str(i))
        if h < 0:
            h = -h
        if b[h%m] == 0:
            ret = False
    return ret
```

```
def bf2(l,k,m):
    b = np.zeros(m)
    for i in l:
        for j in range(k):
            h = mmh3.hash(i + str(j) + 'x')
            if h < 0:
                h = -h
            b[h%m] = 1
    return b


def test2(b, k, m, x):
    ret = True
    for i in range(k):
        h = mmh3.hash(x + str(i) + 'x')
        if h < 0:
            h = -h
        if b[h%m] == 0:
            ret = False
    return ret


def do_experiments(gamma, n):
    common = gen_n(int(n/10))
    A = gen_n(int(n-int(n/10))) + common
    B = gen_n(int(n-int(n/10))) + common


    common = list(set(A) & set(B))


    print("f ", str(f(gamma)))
    print('size common ' + str(len(common)))
    print("expected first candidate set ", str(len(common) + (n-
len(common))*f(gamma)))
```

```
    # experiment for improvement 2

    k = int(round(np.log(2) * gamma))

    m = int(gamma * n)

    b = bf(A,k,m)

    C = [i for i in B if test(b,k,m,i) == True]




    # experiment for improvement 1

    b2_ = bf(C,k,m)

    C2_ = [i for i in A if test(b2_,k,m,i) == True]

    print("First Candidate set " + str(len(C)))

    print("Candidate set of improvement 1 / True common set "+ str(len(C2_)) + "/"
+ str(len(common)))

    print("Theoritical for improvement 1: ", str(f1(gamma, 1/10)))

    fp1 = 1.0 * (len(C2_) - len(common)) / len(A)

    print("False positive rate for Improvement 1: " + str(fp1))

    print("p1 precentage ", str(1-np.e**(-k/gamma)))



    # experiment for improvement 2

    b2 = bf2(C,k,m)

    C2 = [i for i in A if test2(b2,k,m,i) == True]

    print("First Candidate set " + str(len(C)))

    print("Candidate set  of improvement 2/ True common set "+ str(len(C2)) + "/" +
str(len(common)))

    print("Theoritical for improvement 2: ", str(f2(gamma, 1/10)))

    fp2 = 1.0 * (len(C2) - len(common)) / len(A)

    print("False positive rate for Improvement 2: " + str(fp2))




    # experiment for baseline

    k2 = int(round(np.log(2) * 2 * gamma))
```

```
    m2 = int(2 * gamma * n)

    bb = bf(A, k2, m2)

    CC = [i for i in B if test(bb,k2,m2,i) == True]

    print("Candidate set / True common set "+ str(len(CC)) + "/" +
str(len(common)))

    print("Theoritical for baseline: ", str(f(2 * gamma)))

    fp_baseline = 1.0 * (len(CC) - len(common)) / len(B)

    print("False positive rate for baseline: " + str(fp_baseline))


    return [f1(gamma, 1/10), fp1, f2(gamma, 1/10), fp2, f(2 * gamma), fp_baseline]


# code for showing the plot figure 2

collect= []

for i in range(6):

    print(i)    x = do_experiments(3+i,100000)

    collect.append(x)


import matplotlib.pyplot as plt

fig= plt.figure(figsize=(12,8))

plt.plot([3,4,5,6,7,8],[d[0] for d in data], label='improvement 1 (theoretical)')

plt.plot([3,4,5,6,7,8],[d[1] for d in data], label='improvement 1 (experiment)')

plt.plot([3,4,5,6,7,8],[d[2] for d in data], label='improvement 2 (theoretical)')

plt.plot([3,4,5,6,7,8],[d[3] for d in data], label='improvement 2 (experiment)')

plt.plot([3,4,5,6,7,8],[d[4] for d in data], label='baseline (theoretical)')

plt.plot([3,4,5,6,7,8],[d[5] for d in data], label='baseline (experiment)')

plt.legend()

plt.xlabel("gamma, number of bits of Bloom Filter per entry inserted")

plt.ylabel("f, false positive rate")

plt.show()


# code for showing the plot figure 3
```

```
fig= plt.figure(figsize=(12,8))

print( (np.log(0.138) / np.log(0.25)))

plt.plot([3,4,5,6,7,8],[np.log(d[1])/np.log(d[5]) for d in data],
label='log(f\')/log(f)')

plt.legend()

plt.show()
```