University of Zürich

Distributed Database

Communication Cost Comparison in 3-Site Semi-Join using Composite Bloom Filters

Andreas Schaufelbühl, Mirko Richter

December 18, 2016

1 Introduction

The basis for this project is a paper that gives extensions of bloom filters from basic two-sites database joins to settings with multiple sites (Michael 2007). Generally, in a distributed database setting bloom filters can be used to reduce the communication costs. Bloom filters can represent a dataset in a compressed way by hashing the data into a bit array, which we call the bloom filter. The result of hashing a data point gives the index or indices in the bit array which corresponds to this data point. This leads to two important properties of bloom filters. First, bloom filters can not be used to send data from one site to another. It rather represents a set of data points such that at a different site one can verify whether a given data point is included in the bloom filter or not. Secondly, when used for verifying whether a given data point is in a bloom filter, the bloom filter can actually give a false positive. Meaning, it claims that this data point is included in the set while it actually is not. As a result, it is not always advisable to use bloom filters. Michael (2007) shows us that there are three variables which contribute to the probability that a bloom filter return a false positive. First, the amount of data to be represented matters as well as the size or length of the bloom filter and the number of hash functions used. The paper already suggests an optimal relation between those variables for a simple bloom filter join between two sites and it goes on an shows how the bloom filters can be combined when there are multiple sites such that one can reduce communication cost even further. This leads us to our project where we set up an experiment to verify some claims made in Michael *et al* [1].

2 Problem Description

A join in a distributed database setting can lead to a lot of communication between the different sites. To reduce the communication cost, one can use bloom filters instead of sending the data points plainly. But since bloom filters do not contain data points but only a representation of them and can return false positives, it is not obvious that a join with bloom filters will perform better.

Michael (2007) provides an optimal relation

$$k \approx \frac{m}{n} * ln(2) \tag{1}$$

between k the number of hash functions, m the size of the bloom filter and n the number of data points. This is for a bloom filter based join over two sites, this implies that there is only one bloom filter to construct. If we now consider a setting in which we want to join over data points distributed over more than 2 sites, Michael (2007) suggests building composed bloom filters. In our project we only consider the intersection[1] of two bloom filters. First of, to be able to intersect two bloom filters, they must be constructed the same way, in other words, they have to have the same length and use the identical hash

[1]Micheal (2007) also covers the union of two bloom filter.

functions.
Then, to build a composed bloom filter bf_c representing the intersection of two bloom filter bf_1 and bf_2 we set a index i in bf_c as follows

$$bf_c[i] = \begin{cases} 1, & bf_1[i] == 1 \wedge bf_2[i] == 1 \\ 0, & otherwise \end{cases} \tag{2}$$

Let us consider a master $site_m$ which wants to join data from two other sites, $site_1$ and $site_2$. Now instead of performing a bloom filter based join with both sites to get the data, the master site can demand the bloom filter from both sites, build the intersection bloom filter and send it back, so that both sites use the composed bloom filter to find join matches. The main focus of our project is this composed bloom filter. Since in the distributed setting the number of data points needed in the composed bloom filter is not available when $site_1$ and $site_2$ are building the bloom filters based on their data, it is not clear how $site_1$ and $site_2$ should choose the number of hash functions and the size of the bloom filter to minimize the communication cost. Michael (2007) provides theory to pre-compute error probabilities for composed bloom filters. For the intersection of two bloom filters, as we have established above, a bit at some index is set to 1 if and only if at the same index both bloom filters is set to 1. If we assume the distribution of our data points to be independent, Michael (2007) suggests that we can combine the probabilities as follows

$$\begin{aligned} P[bf_c[i] \text{ is set to } 1] &= P[bf_1[i] \text{ is set to } 1 \wedge bf_2[i] \text{ is set to } 1] \\ &= P[bf_1[i] \text{ is set to } 1] * P[bf_2[i] \text{ is set to } 1] \end{aligned} \tag{3}$$

and the probability of a single bloom filter to set 1 at a index is

$$P[bf[i] \text{ is set to } 1] = 1 - (1 - \frac{1}{m})^{k*n} \tag{4}$$

This leads to the probability of a false-positive in the composed bloom filter

$$\begin{aligned} P[\text{false-positive in } bf_c] &= (P[bf_1[i] \text{ is set to } 1] * P[bf_2[i] \text{ is set to } 1])^k \\ &= [(1 - (1 - \frac{1}{m})^{k*n_1}) * (1 - (1 - \frac{1}{m})^{k*n_2})]^k \end{aligned} \tag{5}$$

We then developed a tool, which we will present in detail in the Implementation section, that allows us to build an intersection of two bloom filters containing real data an measure how many ones are set in each of the bloom filters as well as how many false positives the composed bloom filter produces when used to perform a join. We are able to select the size of the bloom filters and the number of hash functions they use prior to performing the join. This allows us to experiment with those two variables and collect data for different inputs on how many ones each bloom filter has set as well as the number of false positives. We will compare the collected data to the expected number of ones set and false positives according to equation 4 and 5. We expect the that the data we collect and the expected values correlate with each other.

Hash functions requirements and solution? Data independency?

3 Experiment Set Up

We used a open source auto-generated dataset from GitHub[2]. This is a relational dataset, with auto-generated fake data, which perfectly fits our needs for the experiment set-up. The following list shows two tables, on which we execute our experiment:

- employees (emp_no, birth_date, first_name, last_name, gender, hire_date)
- salaries (emp_no, salary, from_date, to_date)

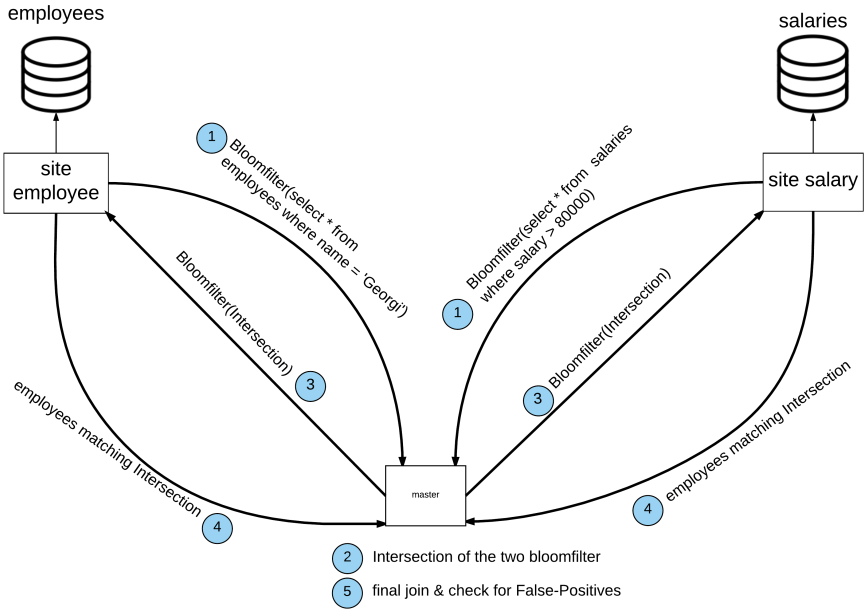

Figure 1: The architectual set-up of the experiment

One key aspect of any bloom filter implementation are the hash functions. They need to be independent and distribute uniformly. Both, equation 4 and 5, are only true for such hash functions. We decided to implement a universal class of hash functions. This allows us to define the number of hash functions to be used at the start and then we can choose those functions from our class. We used the same method as seen in Cormen (2009, p. 267), on designing a universal class of hash functions. As in the book described we chose a prime number p such that every key is smaller than p. Since we need to hash emp_no in the bloom filter and the biggest emp_no is smaller than 500'000, we chose

[2]https://github.com/datacharmer/test_db

$p = 500009$. We define two sets $\mathbb{Z}_p$ and $\mathbb{Z}_p^*$. $\mathbb{Z}_p$ denotes the set $\{0, 1, ..., p-1\}$ and $\mathbb{Z}_p^*$ denotes $\{1, 2, ..., p-1\}$. Then we can define a hash function with $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$

$$h_{ab}(k) = ((ak + b)\ mod\ p)\ mod\ m. \tag{6}$$

In our case, $p = 500009$ and m is the size of the bloom filter. This is one hash function of the family of hash functions defined by

$$\mathbb{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}. \tag{7}$$

This means, for every hash function we simply choose a and b randomly from their sets to create h_{ab}.

4 Implementation

DO WE NEED THIS SECTION? DO WE WANT TO SHOW SOME OF THE IMPLEMENTATIOS-TECHNIQUES WE USED. MAYBE DEPENDS ON LENGTH OF OTHER SECTIONS? (- TRICK HOW WE COUNT FALSE POSITIVE THAT WE DONT NEED TO COMPARE AGAIN) In this section we want to present some details of our implementation with focus on interesting parts.

4.1 Counting False Positive

When we perform the join as seen in figure REFERENCE TO FIGURE, in step INSERT NUMBER both sites return the join-matching tuples. Lets define these results as two sets S_{emp} from site employee and S_{sal} from site salary. We claim that if the emp_no of a tuple in S_{emp} is also in S_{sal}, then it cannot be a false positive. In other words, whenever a tuple in S_{emp} has a emp_no that does not exist in S_{sal} and vice versa, it must be a false positive.
To show this, we first consider what sort of emp_no these sets potentially can contain. Lets consider site employee. We check all emp_no returned from the query

```
SELECT emp_no FROM employees WHERE first_name = 'Georgi'
```

against the composed bloom filter. Hence, all tuples in S_{emp}, true matches and false positives, have a emp_no from a employee with $first_name = Georgi$. Same is true for S_{sal} just with emp_no from this query

```
SELECT emp_no FROM salaries WHERE salary > 80000.
```

Now, we assume there exists a emp_no which is in S_{emp} and in S_{sal} and *is* a false positive. By construction a emp_no in S_{emp} is from a tuple with $first\_name = Georgi$ and a emp_no from S_{sal} is form a tuple with $salary > 80'000$. Hence, a emp_no which is in S_{emp} and in S_{sal} corresponds to a employee with $first\_name = Georgi$ and $salary > 80'000$. This is a true join match and therefore cannot be a false positive.

5 Evaluation

6 Conclusion

References

[1] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. In *21st International Conference on Advanced Information Networking and Applications (AINA 2007), May 21-23, 2007, Niagara Falls, Canada*, pages 187–194, 2007.