Extending PostgreSQL: Shifting Bloom Filters

Progress Report 2

Jacob Anderson - jacob.w.anderson-1@ou.edu
Joseph McGill - joseph.a.mcgill-1@ou.edu
Stephen Smart - smart@ou.edu

Objectives

- Become familiar with PostgreSQL.
    - This is important because without a foundational understanding of PostgreSQL, implementation issues will arise in the future.
- Implement the shifting bloom filter framework for set membership, association, and multiplicity queries (examples below).
    - This is important because the shifting bloom filter approach is currently not implemented in PostgreSQL and our implementation is the primary deliverable of this project.
- Compare shifting bloom filter framework to alternative methods such as standard bloom filters [Bloom, 1970], the iBF approach [Yang, 2016], and the count-min sketch [Cormode, 2005].
    - This is necessary to verify that our implementations are valid and useful.
- Analyze experimental results.
    - This is important because it will allow us to quantify the value of our implementations.

Set Query Examples

Based on feedback from Dr. Gruenwald, here we will provide concrete examples to illustrate how each type of set query functions. Each of the three types of queries below were described in [Yang, 2016].

Set membership queries are queries that ask whether a given data item exists in a set of data. For example, consider a data set consisting of IP addresses corresponding to visitors of a particular website. We may want to know whether a given IP address has visited the website before or if the user is visiting the website for the first time. We can perform a set membership query using the given IP address and the data set to determine the answer.

Set association queries are queries that ask whether a given data item exists in two sets, one set but not the other, or neither set. Consider the same data set from the set membership example, but now we are interested in whether a user has visited two different websites, just one of the two, or neither of them. The result of a set association query would answer this question.

Set multiplicity queries are queries that ask the frequency at which a given data item exists in a set. Consider the same data set from the set membership example, but now we are interested in how many times the user has visited a particular website. Maybe if

the user has visited the website 1000 times, they earn a special reward. This question of frequency is a set multiplicity query.

Literature Review

The idea for bloom filters was originally introduced by Burton H. Bloom in [Bloom, 1970]. This paper is relevant to our project as we will be comparing our shifting bloom filter implementation to a standard bloom filter implementation in PostgreSQL. As discussed in this paper, a bloom filter is a probabilistic data structure that can be used to process set membership queries with some allowed false positive rate, but with vastly improved space complexity. The bloom filter approach involves the use of hash functions and an array of bits. To build the bloom filter, all elements are hashed k times each, where the hash functions produce a random number in the range of the bit array indices. When a bit array index is produced in the hashing process, the corresponding bit is set to 1 if it is 0 and left at 1 otherwise. After the bloom filter is built, set membership queries can be processed by hashing the given element and checking if the resulting bits are set to 1. If any of the bits are 0, the element is not in the original dataset. If all bits are 1, the bloom filter reports that the element is in the original dataset, which may or may not be true. When this inaccuracy in the form of false positives is tolerable, bloom filters allow for the efficient processing of set membership queries without having to store or search through the dataset.

The count-min sketch was introduced by Graham Cormode and Shan Muthukrishnan in [Cormode, 2005]. The idea behind this probabilistic data structure is to extend the standard bloom filter to handle multiplicity set queries. As described above, standard bloom filters deal with sets of data, meaning data points cannot occur more than once. The count-min sketch is designed to work with multi-sets, where data points can occur more than once. With multi-sets, it is a common query to determine the frequency at which a given data point exists in the multi-set. The count-min sketch extends the bloom filter to answer this query by using counters instead of single bits within each cell of the sketch. When a new element is inserted into the sketch, the element is hashed by the various hash functions just like the standard bloom filter, but instead of flipping a bit from 0 to 1, a counter at each index is incremented. Similarly, for deleting an element in the sketch, the counter is decremented. This enables the use of multiplicity set queries. To determine the frequency at which a particular item exists in the sketch, the element is hashed to the corresponding indexes and the minimum value of the counters is selected. Hash collisions do occur which result in inaccuracies, but the estimated frequency is bounded by the following equation: $a \leq \widehat{a} \leq a + \varepsilon n$. This bound can be tuned by the developer to ensure the estimation is never that far off of the actual frequency. We will use the count-min sketch to compare our implementation of the shifting bloom filter framework with respect to set multiplicity queries.

The cuckoo filter is a modified version of the bloom filter introduced in [Fan, 2014]. The bloom filter cannot handle the removal of elements. Several attempts to change the

implementation of the bloom filter have been done in order to handle the removal of elements such as the counting bloom filter. These previous modifications have all added some amount of space, time, or false positive rate overhead. The cuckoo filter is a new modified bloom filter that performs even better than the standard bloom filter in several aspects such as space cost, cache misses per lookup and deletion support. These improvements are achieved by taking a unique approach to hashing the elements and adding a new algorithm for “relocating” existing hash values to a new index in the sketch. The novel algorithms presented in the paper are quite complicated, so in order to be concise we will avoid an overly detailed analysis of the behavior of these algorithms. We are particularly interested in the cuckoo filter implementation for the experimentation that was done. Extensive experiments were performed that compared the cuckoo filter with other modified bloom filter approaches. We will be doing similar experiments that compare the shifting bloom filter framework with other probabilistic data structures, so it will be beneficial for us to analyze how Fan et al. approached the experimentation section and what data was used therein.

PostgreSQL is an open source object-relational DBMS that was introduced in 1996 [Momjian, 2001]. One of the major purposes of PostgreSQL is to allow for the extension of the system through the use of custom data types and functions. PostgreSQL extensions can be written and compiled using C code. This is directly applicable to our project, as we will be extending PostgreSQL by implementing the shifting bloom filter framework.

Databases are often thought of as tables where information is already available for querying. However, there are cases (such as data streams) where the data is continuously pouring in and must be processed quickly. An example of this can be seen in a typical web application where requests and information is constantly streaming in from multiple users at once. Anand Rajaraman discusses methods for handling and processing data streams in [Rajaraman, 2012]. Specifically, Rajaraman describes methods to query, sample, filter, and count distinct elements in a data stream. This is relevant to our project as we are evaluating the shifting bloom filter framework using a large dataset. Data streams are perfect for this since the data is continuous, and should allow us to determine the effectiveness of our implementation.

In [Yang, 2016], modified bloom filters are proposed to process set membership, set association, and set multiplicity queries. The shifting bloom filter framework consists of three different modified bloom filters that share the concept of encoding auxiliary information in offset values when computing the bit indexes. For set membership queries, one hash function is used to compute an offset value for each element. Then, the indices produced by the other hash functions are each incremented by this offset and the final produced indices are the original indices together with those incremented by the offset value. This has the effect of reducing the necessary hash functions by a factor of two and, with some minor optimizations based on the word size of the machine used, reducing the number of memory accesses by a factor of two as well. For

association queries, the approach is similar to membership in that additional hash functions are used to compute an offset to add to the bit indices produced by hashing each element. When an element is in the first set but not the other, no offset is used. When an element is in the other set but not the first, a hash function is used. Lastly, when an element is in both sets, a second hash function is used. When querying the shifting bloom filter for association queries, the output is given in a way such that it is never incorrect, but it is sometimes not very descriptive (i.e. "The element is in one of the sets, or maybe both"). These cases are proven to be unlikely, showing the viability of the shifting bloom filter approach. Lastly, for set multiplicity queries, the offset used is the number of times the element appears in the multiset. When querying the shifting bloom filter for multiplicity, the values following the hashed indices for an element are checked. For cases where index + x = 1 for all indices, x is a potential frequency for the element. The largest potential frequency is returned as the final result to avoid false negatives.

MurmurHash3 is a non-cryptographic hash function developed by Austin Appleby [Appleby 2008]. MurmurHash3 hashes a given element twice and returns the two hash values. These two hash values can be combined to produce an arbitrary number of hash functions using the following equation: $h_1(e) + i * h_2(e)$ where $h_1(e)$ is the first hash value, $h_2(e)$ is the second hash value, and i is an arbitrary integer to produce a new unique hash function. The primary benefit of using MurmurHash3 in our project is that

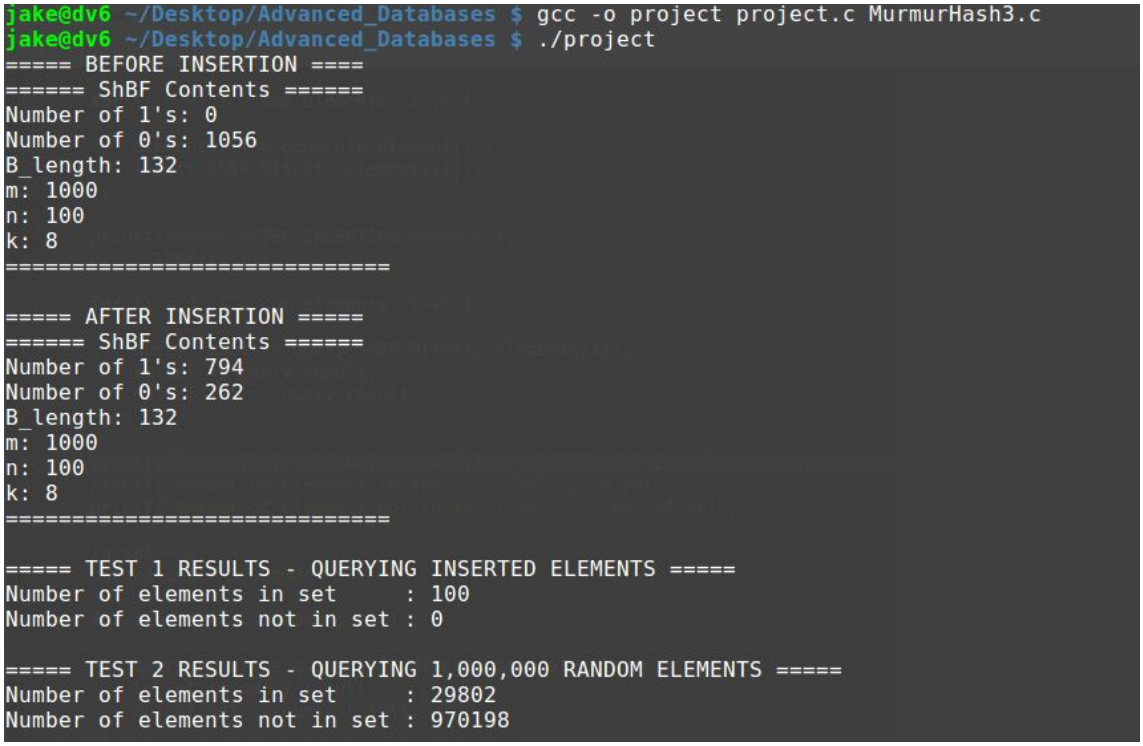
we need *k* independent hash functions, so using the method described above, we can call MurmurHash3 once and loop from 0 to *k-1* to produce k unique hash functions.

The chief architect of PipelineDB, Usman Masood, developed a PostgreSQL implementation of a standard bloom filter. In his blog post, he provides code examples and explanations that will help us learn how to implement the PostgreSQL types and functions that we need for our project.

Work Completed

Since the last progress report, we have implemented the general code base for the shifting bloom filter framework in C. Additionally, we have implemented the shifting bloom filter approaches for membership and multiplicity set queries. We have implemented most of the code for the association set query, but we have not yet fully implemented this approach. Implementing these approaches involved creating a new data structure in C and functions that handle the construction, update, and query functionality described in [Yang, 2016]. Furthermore, we have implemented functions that generate random data that we will use to build datasets for our experiments. This function generates a random 10 digit decimal number using C's standard rand() function. Currently we are slightly behind schedule since we have not yet completed the implementation of all three approaches mentioned above. It should be noted that this is not the fault of any individual group member, as association set queries are the most complex of the three.

The below screenshot shows the output of our test code. Our tests create a shifting bloom filter for set membership, insert data into the ShBF, and query the ShBF. The first test queries the ShBF for the elements that were inserted into the ShBF. The results show that each inserted element is correctly identified as a member of the original set. This test shows exhibits zero false negatives, which is a property of shifting bloom filters and bloom filters in general. The second test is meant to be a rough approximation of the false positive rate of our ShBF implementation. The test generates 1 million random elements and queries the ShBF for their membership. The results show a ~3% false positive rate, but it should be noted that this is not the actual false positive rate. The randomly generated elements could contain duplicates and/or members of the original set (true positives). When we thoroughly evaluate our implementation, we will account for these issues.



Work To Be Done

Other than finishing the implementation for the shifting bloom filter approach for association set queries, we need to write PostgreSQL types and functions that link to our implemented data structures and functions in C. Additionally, we need to perform experiments that evaluate the performance our implementation by comparing it to alternative approaches described above. The approaches will be compared using the following metrics: speed (algorithm running time and number of memory accesses), space, number of hash functions, and false positive rate. Also, we need to do analysis and draw conclusions on how our implementation performs, the issues we ran into, and where we could improve.

References

Appleby, A (2008). MurmurHash3 on Github https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM,13(7),* 422-426. doi:10.1145/362686.362692

Cormode, G., & Muthukrishnan, S. (2005). An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, *55*(1), 58-75.

Fan, B., Andersen, D. G., Kaminsky, M., & Mitzenmacher, M. D. (2014, December). Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (pp. 75 - 88). ACM.

Masood, U. (2015). *Making Postgres Bloom* https://www.pipelinedb.com/blog/making-postgres-bloom

Momjian, B. (2001). Extending PostgreSQL using C. *PostgreSQL: introduction and concepts* (pp. 372 - 380). Boston, MA: Addison-Wesley.

Rajaraman, A., Ullman, J. D., & Leskovec, J. (2012). Mining Data Streams. *Mining of Massive Datasets* (pp. 131 - 161). New York, NY: Cambridge University Press.

Yang, T., Liu, A. X., Shahzad, M., Zhong, Y., Fu, Q., Li, Z., . . . Li, X. (2016). A Shifting Bloom Filter Framework for Set Queries. *Proceedings of the VLDB Endowment*, *9(5)*, 408-419. doi:10.14778/2876473.2876476