Implementation of Bloom Filter and Count-Min Sketch in Apache Spark

Author: Logothetis N. Fragkoulis M. Sc. Candidate

Sort Description of Bloom Filters

An empty Bloom filter is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions, generating a uniform random distribution. Typically, k is a constant, much smaller than m, which is proportional to the number of elements to be added; the precise choice of k and the constant of proportionality of m are determined by the intended false positive rate of the filter. To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.

To query for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is definitely not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem.

The requirement of designing k different independent hash functions can be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields. Alternatively, one can pass k different initial values (such as 0, 1, ..., $k - 1$) to a hash function that takes an initial value; or add (or append) these values to the key. For larger m and/or k, independence among the hash functions can be relaxed with negligible increase in false positive rate.

Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to k bits, and although setting any one of those k bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach re-adding a previously removed item is not possible, as one would have to remove it from the "removed" filter.

It is often the case that all the keys are available but are expensive to enumerate (for example, requiring many disk reads). When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.

Implementation of Bloom Filter in Spark

In this project the streaming parallel edition of the Bloom Filter is implemented to answer queries about previous existence of a specific IP in the stream. The Spark Streaming Apache was utilized as it is a powerful Big Data tool and a great number of scientists keep it up-to-date. Although, the Spark Streaming cannot handle tuple-per-tuple processing the micro-batching approach is more than enough to develop such a simple Sketch as Bloom Filter is. The main advantage of the Bloom Filters is that they need reasonable size of memory, therefore, Spark is the ideal candidate as it always caches the part of the data that is commonly retrieved. Hence, the Bloom Filter will be with high odd in-memory, something that results in high performance.

Our implementation was based on key-values pairs for the reason that Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. For example, pair RDDs have a reduceByKey() method that can aggregate data separately for each key, and a join() method that can merge two RDDs together by grouping elements with the same key. It is common to extract fields from an RDD (representing, for instance, an event time or other identifier) and use those fields as keys in pair RDD operations.

Firstly, we developed a python code for generating IP addresses. That addresses will feed the streaming Bloom Filter in order to be trained or evaluated. The generated IP could be any possible combination of IPV4. Note that IP are written in the file without dots (i.e 147563845321 is the IP 147.563.854.321). The streaming algorithm takes as input two files; one for training purpose and the other for queries answering. Since our code is running locally (only for validation) the files divided in micro-batches which are stored in a Queue Stream instance. Queue Stream is an internal library of Apache Spark that is used commonly to simulate the streaming process. Each of the micro windows could have special duration and a pre-fixed number of data.

Secondly, we designed a graph with the operations that should be executed to keep the Bloom Filter up-to-date for new arrivals. The main execution plan is illustrated in the following figure.

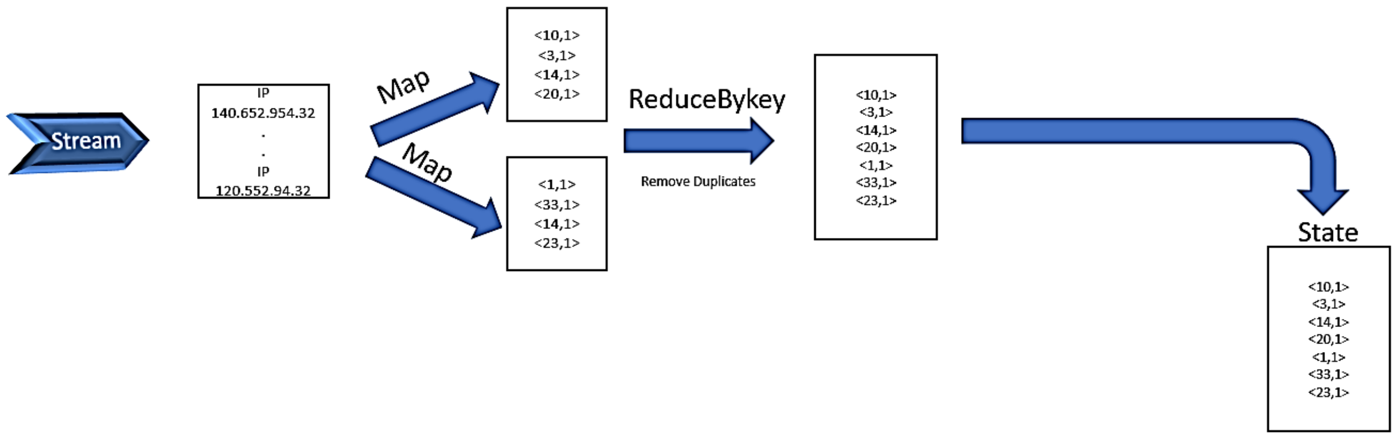

Figure 1 : Training of Bloom Filter

As the above figure shows, the procedure starts with the mapping of each IP to array positions of the Bloom Filter by leveraging k hash functions. The produced key/value pairs are corresponding to indices of the Boom Filter (key) that must be set to '1' (value). Since the mappers are running in parallel, the results are united using "union" operator and the reducebykey() operator is called to remove the duplicates (minimize overload). Since the current RDD will be comprised for new arrivals from the input stream, the state of the bloom filter must be updated. The array positions that are not existed in the state are inserted in the current state, with the limitation that the already appeared cannot be changed. By the end of the stream the state memory will include all the array positions that are set to '1', therefore, the positions that are '0' will not be existed in the state.

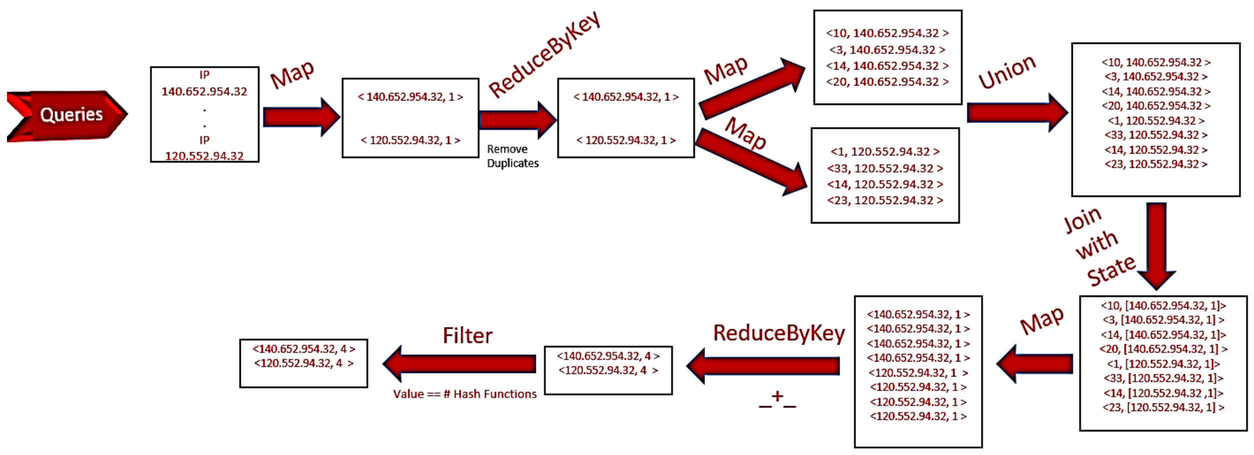

Figure 2 : Queries on Bloom Filter

Query Stream is separate and independent from the training stream (suppose two different sockets, one for training and the other for queries). Specifically, the file with queries is comprised of IP addresses. If the ith IP is in the set, the algorithm returns (print or save to HDFS) that address, otherwise the i-th address is not written in the output file. The second figure depicts the way that a query is answered. Firstly, the IPs are mapped to key/value pairs and the reducebykey() operator take place to remove the duplicates. Secondly, the tuples are translated into array positions using k hash functions and after the uniting of the hash function's results, the joining operation between the current state of the stream and the query stream returns new key/value pairs, which are formulated as <Key=Array Positions, Value= List of [IP, 1]>. Last but not least, from the aforementioned tuples we keep only the list of values. The first elements of the list will be the new keys and the second the new values. Finally, the number of the different values with the same key is counted. If the number of the ith IP is equal to the number of the hash functions, all the array positions that the hash functions returned are existed and their value is equal to '1'. On the other hand, if the value of the ith IP is less than the number of the hash functions, means that one of the array positions has value '0'. In that case, the ith IP has not appeared in the previous events.

Sort Description of Count-Min Sketch

In computing, the count–min sketch (CM sketch) is a probabilistic data structure that serves as a frequency table of events in a stream of data. It uses hash functions to map events to frequencies, but unlike a hash table uses only sub-linear space, at the expense of overcounting some events due to collisions. Count–min sketches are essentially the same data structure as the counting Bloom filters. However, they are used differently and therefore sized differently; a count-min sketch typically has a sublinear number of cells, related to the desired approximation quality of the sketch, while a counting Bloom filter is more typically sized to match the number of elements in the set. The goal of the basic version of the count–min sketch is to consume a stream of events, one at a time, and count the frequency of the different types of events in the stream. At any time, the sketch can be queried for the frequency of a particular event type i ($0 \leq i \leq n$ for some n), and will return an estimate of this frequency that is within a certain distance of the true frequency, with a certain probability P(error).

The actual sketch data structure is a two-dimensional array of w columns and d rows. The parameters w and d are fixed when the sketch is created, and determine the time and space needs and the probability of error when the sketch is queried for a frequency or inner product. Associated with each of the d rows is a separate hash function; the hash functions must be pairwise independent. The parameters w and d can be chosen by setting $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln 1/\delta \rceil$, where the error in answering a query is within an additive factor of ε with probability $1 - \delta$ (see below), and e is Euler's number.

When a new event of type i arrives, we update as follows; for each row j of the table, apply the corresponding hash function to obtain a column index $k = h_j(i)$. Then increment the value in row j, column k by one.

Several types of queries are possible on the stream.

- Point query
- Inner product

Implementation of Count-Min Sketch in Spark

The implementation scenario will be quite same as Bloom Filters. Specifically, once can convert the w x d Count-Min Sketch to a single dimension array with size 1 x (w x d). Therefore, the operations that was utilized in Bloom Filters could be reused in that problem with slight variation. During training, the Bloom Filter saves only a bit (0/1), which is necessary to make decisions about the previous events. On the other hand, Count-Min sketch have to keep update w x d counters, hence, even though the mapping to key/value pairs using the appropriate hash functions remains, the state that the stream should save are counters instead of a bit (i.e similar with Word Count). In that case, the reducebykey() operator count all the appearances of an event during a certain micro-batch and by using the updatebykey() operator the updated sketch is saved, as it is illustrated in Figure 3. This process is repeatedly executed until the end of the stream.

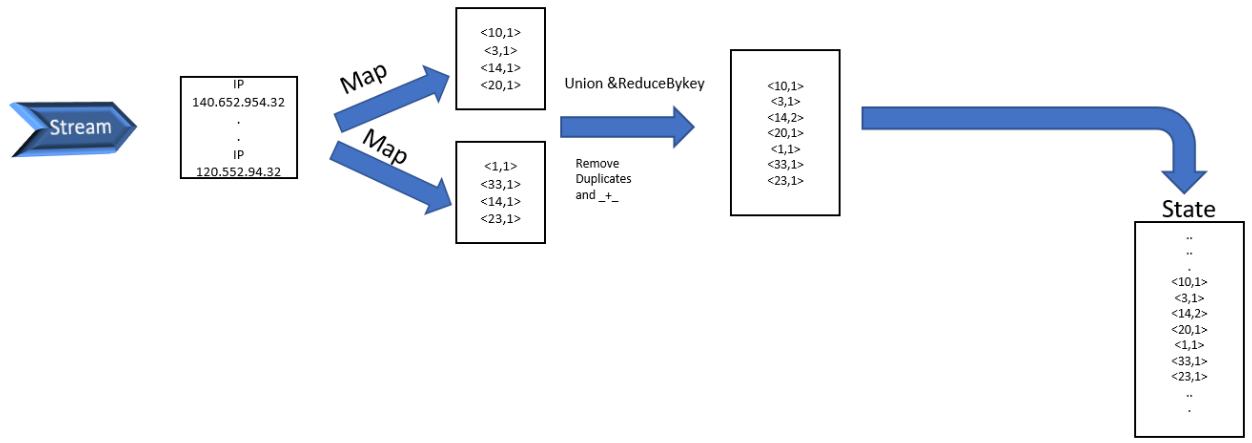

Figure 3 : Training of Count-Min Sketch

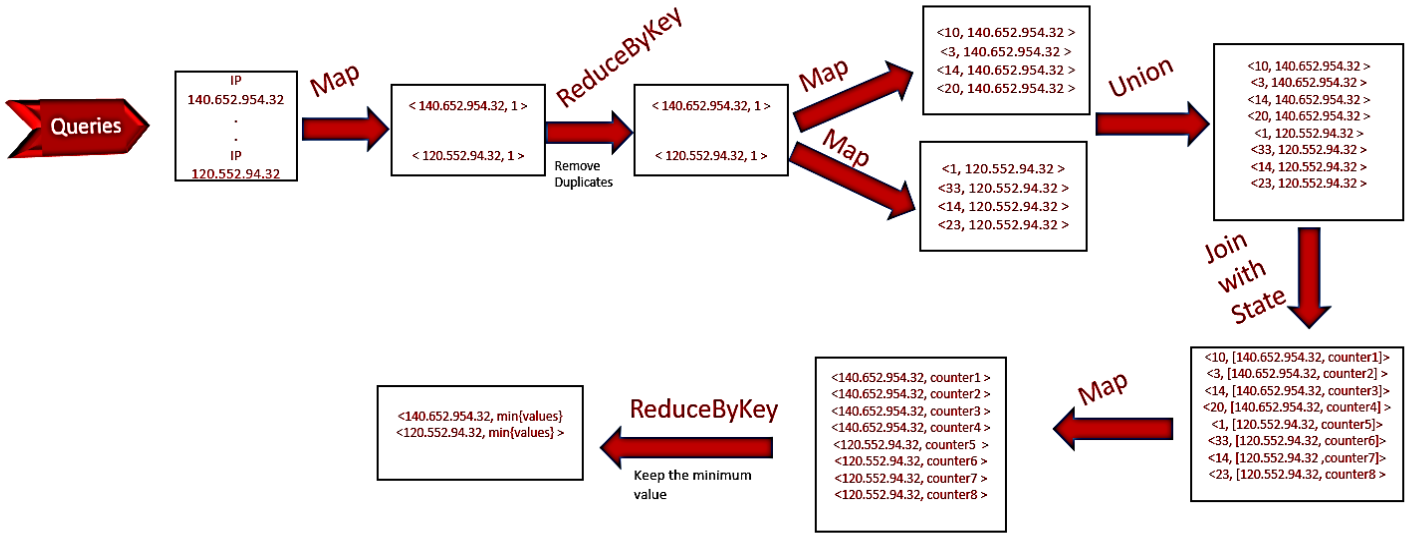

Figure 4 : Queries on Count-Min Sketch

Answering point queries is similar to answering point queries in Bloom Filters. The main difference stems from the fact that in the final stage the minimum value among the elements of the d hash functions should be calculated. The reducebykey() operation in the final transformation, as it is depicted in Figure 4, exposes that minimum value for each IP that is queried instead of extracting only an IP address as Bloom Filters does.