Discussions to Improve Nearest Neighbor Search for LSH

In Spark 2.1, we designed and implemented Locality Sensitive Hashing (LSH) as well as Nearest Neighbor Search (NNS) using LSH. However, there was a lot of controversy around current implementation of multi-probe nearest neighbor search, which is supposed to implement methods in this paper. We use this doc to discuss what changes we should make for SPARK-18454.

Current Implementation of Multi-Probe NNS (as after SPARK-18409):

In LSH with *L* hash tables and *M* hash functions, we do the following to get *k* nearest neighbor of the querying object.

1. If the dataset does not contain hash column, transform the dataset to build the hash column.
2. Build a column to store the hash distance between each row and the querying object.
3. Calculate approxQuantile of (numNearestNeighbors / totalNumElements) on hash distance column to be a threshold t. We will probe all buckets within the threshold.
4. Filter all rows with hash distance <= t to get all rows we probe.
5. Build a column to store the key distances on the filtered dataset.
6. Sort all rows by the key distance and get top k nearest neighbors.

What hash distance (or Probing Sequence) should we use?

The current hash distance implements step-wise probing in the original paper. The paper described step-wise probing as “first probes all the 1-step buckets, then all the 2-step buckets, and so on”, where the number of steps is mentioned as “ (i.e., only one hash value is different from the M hash values of the query object) ”. For MinHash with L hash tables, there are *L * M* 1-step buckets, and *L * (M 2)* 2-step buckets. In the paper, there is another *2^n* term when counting this number because it assumes the difference of each hash value can be either −1 or +1.

The paper also mentioned query-directed probing. The query-directed probing is based on analysis of BucketedRandomProjection. It defined the score of a hash bucket as the distance between the projection of querying object in Euclidean space and boundary of the hash bucket. Everytime it probes the hash bucket with min score, and do shift/expand operation to get 2 new candidate hash buckets. One thing probably we can do is to implement this probing sequence for BucketedRandomProjectionLSH?

How shall we decide the number of probes?

The current implementation minimizes the number of probes we make. The number of probes is dynamically chosen by computing a threshold t so that all buckets within t-steps are probed. The threshold t is determined by approxQuantile of (numNearestNeighbors / totalNumElements) on hash distance. This avoids redundant probes as long as we get enough nearest neighbors. In

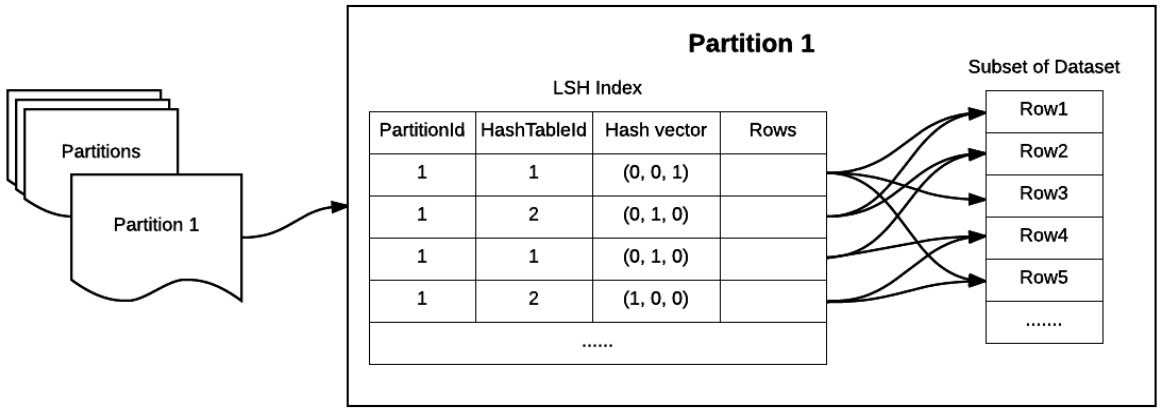
extreme cases, we always do L probes (the same as single probe) as long as there are enough elements in the buckets.

We have the following options to decide the number of probes:

(1) Dynamically minimizes the number of probes (Current Implementation).
(2) Dynamically choose the number of probes with user specified rate of redundancy.
(3) User specifies the number of probes for each hash table.

How shall we improve the performance of NNS?

The current nearest neighbor search is not optimized as we still need to compare hash values of all element with the querying object. One way to improve is build an index table within each partition, like the following:



The idea here is to implement a LSHIndexedDataFrame that extends DataFrame. LSHIndexedDataFrame contains an LSH index table where we can look up all rows given a hash vector in constant time. Besides the APIs in DataFrame, LSHIndexedDataFrame also provides two other APIs that help us perform algorithms using LSH:

```
// Get all rows in a list of hash buckets.
def getLSHBuckets(hashTableId: List[(Int, Vector)]): Dataset[Row]
```

With `getLSHBuckets`, we can do NN Search in a much more efficient way: we simply pass in the hashTableId's and hash vectors, and the LSH index will help us find all rows in the hash buckets without comparing with any other rows.