**Summer Internship Project**

**1.PROBLEM STATEMENT**

**2.HLL Algorithm**

**a. Working**

**b. Results & Use Cases**

**3.UDAF & Accumulator**

**4.SOLUTION OVERVIEW**

**a. SOLUTION Approach 1**

**b. SOLUTION Approach 2**

**Problem Statement**

The **cardinality estimation problem** is the problem of finding the number of distinct elements in chunk of data with the same elements being repeated multiple times (multiset). This helps solve the problem of determining the unique entities in the datasets. For instance, to efficiently fetch unique users in the dataset.

The naive solution or brute force solution would be saving every element in a set and then compute its cardinality. It is O(n) in terms of both space and time.

**HLL-Hyper Loglog Algorithm**

It is a probabilistic algorithm used to estimate the cardinality of a multiset. Traditional methods of counting would calculate the exact number of unique items in the set but would need O(n) of space and time. So HLL Algorithm instead of calculating, estimates the number of distinct items in the set.

**WORKING**

The assumption that HLL makes is that the set is composed of uniformly distributed random numbers. So HLL hashes the input such that it becomes uniform and randomly distributed and then take their binary representations and counts the number of 0's at the end.

Reason---Assume that you have a string of length m which consists of {0, 1} with equal probability. What is the probability that it will start with 0, with 2 zeros, with k zeros? It is 1/2, 1/4 and 1/2^k. This means that if you have encountered a string with k zeros, you have looked through 2^k elements.

So, we could also say that since the probability is 2^k then the total number of elements in the set would be close to 2^k. If the algorithm worked in this state, then there would be a lot of variances and not particularly good.

So, to reduce the error multiple hashing was one of the solutions but since hashing is an expensive operation so some other work around was needed to be done. To reduce the error, it was proposed that the algorithm should divide the input into **buckets/nodes/sketches** and performs operations on each of them and then takes the harmonic average of these results to compute the result. To make this division into sketches/buckets what the algorithm does is that it takes first m bits of the hashed binary form of the input and based on these values assign them to a bucket of values and then the remaining bits to calculate the 0's and estimating the cardinality of the bucket.

By having m buckets, we are simulating a situation in which we had m different hash functions. This costs us nothing in terms of accuracy but saves us from having to compute many independent hash functions. This procedure is called stochastic averaging.



**RESULTS & USE CASES**

**Set operations on HLL.**

Set union operations are straightforward to compute in HLL and are lossless. To combine two HLL data structures, simply take the maximum corresponding bucket values of the two and assign that to the corresponding bucket in the union HLL. If we think about it, it is exactly as if we had fed in the union of the two data sets to one HLL to begin with. Such a simple set union operation allows us to easily parallelize operations among multiple machines independently using the same hash function and the same number of buckets.

The accuracy is above 98% for a dataset of size 1 billion while occupying 1.5Kb storage space.

The main application of this algorithm could be found in situations such as where a nearly accurate result is good enough such as estimating the traffic in the city through the car count or the spread of disease through patient count. The algorithm is used by Google to monitor the search results and by Facebook to estimate the active users.

Thus, HLL uses less time and resources but compromises a bit on the accuracy. There is an in-built function provided by Spark **approx\_distinct\_count ()** which uses this algorithm to give the results and is found to be over 60 times faster than aggregators such as count or distinct count.

**UDAF-User Defined Aggregator Function**

As part of any data analysis workflow, doing some sort of aggregation across groups, or columns is common.

These work exactly like aggregators except for the fact that these are not optimized and might run for a longer duration.

To define them we need to extend the UDAF class provided by spark and then override the functions such as initialize, update, merge, evaluate.

**Accumulator**

Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only “added” to, such as counters and sums. Thus, they are provided to the executor nodes where the operations are performed, and they store the values and in the driver node their values are merged, and final operations could be done. Then there accumulatorV2 API which is an abstraction of accumulators.

The **AccumulatorV2** API in spark enables you to define clean custom accumulators for stats for your job, which you would otherwise end up computing by running similar jobs/SQL queries on your output data.

Diagram

Description automatically generated

**Solution Approach**

So, there were two main parameters to keep in mind while thinking of the solution for the efficient cardinality estimation which were privacy and performance.

The existing solution of using Map Accumulators provides 100 percent accurate results but Spark also displays the content of accumulators with their count in the Spark UI. This was challenging as Users/entities are sensitive information and should not be leaked. The performance on using the HLL would not be 100% and there could possibly be an error of around 1% when the data entries are in the excess of 1 billion.

Another major point to keep in mind is the persistence of data. Every I/O operation is very costly so adding a new stage in the pipeline would not be very suitable, so we need to take care of this as well.

So as part of the solution we would need to be able to generate Random data for unique user count (meaningful dataset generation)

- Well distributed users in the dataset

- Skewed user dataset

- another dataset that can be useful

**Solution 1 Approach**

To use the in-built functionality provided by spark to approximately compute the distinct items in a set integrated with either the UDAFs or as standalone.

**Solution 2 Approach**

Define Map Accumulator for unique user counting. Define the internal data structure to store the data and expose only the count to the user with the HLL algorithm used in backdrop either from spark or from some other open-source libraries.

There are two open-source implementations found for the HLL algorithm **sql-alchemy** and **stream-lib**. Both implementations are open source and could be attempted to integrate with UDAF's or accumulators. The main difference in the implementation is in the hash function that these two algorithms use. The implementation provided by spark approx\_distinct\_count is an extension of the imperative aggregate class and only intakes SQL type expressions.

Then have the main class defined which wraps everything and then outputs the results to the user. Write well established test suite having unit tests covering all the scenarios.

The following code snippet for an Accumulator class using one of the libraries.



The following is the UDAF implementation using the library

