HyperLogLog based approximation framework for very big datasets

("*If something appears to be a duck, moves like a duck, and sounds like a duck, chances are, it is indeed a duck.*")

The HyperLogLog (HLL)[2] algorithm is based on the observation that the cardinality of a multiset of uniformly distributed random numbers can be estimated by calculating the maximum number of trailing zeros in the binary representation of each number in the set. If the maximum number of trailing zeros observed is denoted as n, then an estimate for the number of distinct elements in the set is given by 2^n. [1]

However, the variance of this estimation can be significant. To address this issue, the HLL algorithm divides the multiset into multiple subsets. It calculates the maximum number of trailing zeros in the numbers within each subset and then combines these estimates using a harmonic mean to provide an overall estimate of the cardinality of the entire set.

The HLL data structure is represented as a k-tuple $\mathbf{t = (n_1, n_2, \ldots n_i \ldots n_k)}$, where n_i represents the maximum number of trailing zeros calculated for the i-th subset in the multiset. This structure allows for the lossless merging of two or more HLLs, where the resulting HLL is equivalent to the calculation performed on the union of the original datasets and yields the same cardinality estimation.

While the HLL structure does not support other set operations, such as intersection, it is possible to enhance it by replacing the maximum number of zeros in the tuple t with bit-vectors. By incorporating bit-vectors to store all numbers of trailing zeros for each subset, this upgraded structure, we call it as HllSets (HyperLogLog Sets), enables the implementation of all set operations.

This article will delve into the implementation details of HllSets, provide experimental evidence of its adherence to major set properties, and explore its potential applications in metadata, statistics, and graphs.

HyperLogLog as an approximation of datasets

The key component of the algorithm discussed in the Introduction is the set of subsets we utilize to determine the number of leading zeros. Let's delve deeper into the process of gathering these numbers.

In many HLL implementations ([4], [5]), this set is represented as an array (vector) known as registers. The size of the registers vector is typically a power of 2. For instance, in the Redis implementation [4], 2^{13} = 8192 elements are used.

Here is a Python pseudo-code algorithm for constructing the registers (regvector):

```python
# Fill registers with zeros
# k is the power of 2 in the formula for the registers size
registers = = [0] * 2**k

for entry in dataset:
  # Calculate hash for each dataset entry
  h_entry = hash_fn(entry)

  # Calculate index of the registry as the first k bits from the h_entry
  # As mentioned Redis takes 13 leading bits
  reg_idx = get_reg_idx(h_entry, k)

  # Count longest sequence of running zeros
  z_num = zero_count(h_entry)

  # Update registers, if calculated z_num greater than
  # current value in the registers, replace it with z_num
  if z_num > registers[reg_idx]:
    registers[reg_idx] = z_num
  # Or without using if statement
  #  registers[reg_idx] = max(register[reg_idx], z_num)
```

After running this algorithm, each element of the registers holds the maximum number of consecutive zeros for each subset of the dataset. Each subset is associated with the k leading bits of the hash value for the elements belonging to that subset.

The algorithm for building the registers has several useful properties:

1. It is idempotent, meaning that running the algorithm on the same dataset will not change the result.
2. It creates a unique representation for each dataset. If two registers vectors are different, then the corresponding datasets are different. However, it is important to note that having the same registers for two datasets does not guarantee that the datasets are equivalent. This may lead to false positives, but there are no false negatives when comparing datasets using HLL. Datasets with different registers are not considered equal.
3. The relationship between registers and the cardinality of a dataset is not straightforward. The calculation formula needs to be adjusted based on the dataset size, and it may vary depending on the type of data and its distribution within the dataset. These discrepancies are challenging to address in a generic way and have not been corrected in current implementations of HLL.

Operations in current implementations of HyperLogLog

The majority of HLL implementations offer support for the following operations:
1. **ADD**: This operation involves adding an element from a dataset to the registers, and is an integral part of the algorithm.
2. **MERGE**: This operation utilizes a similar algorithm to ADD, with a minor modification.

```python
# dataset_1 - registers for dataset 1
# dataset_2 - registers for dataset 2
# out - registers for resulting dataset

def merge(dataset_1, dataset_2):
  # Initialize resulting registers
  out = [0] * 2**k
  for i in range(dataset_1.size() - 1):
    # Set out[i] to the max of elements in dataset_1[i] and dataset_2[i]
    out[i] = max(dataset_1[i], dataset_2[i])
```

It is clear that the sizes of all three register vectors must match in order to perform a lossless MERGE operation.

COUNT is the core function of HyperLogLog (HLL), as it determines the cardinality of the dataset based on the values stored in the registers. The **union** operation in HLL is used to **merge** datasets without altering the original counters, instead producing a new HllSet as the output.

Interestingly, the **intersect** command is notably absent in all known implementations of HLL, and there are valid reasons for this omission.

Simply replacing the maximum value with the minimum value in the **union** operation does not suffice for **intersection**. Applying the cardinality command to the **intersection** obtained using this approach consistently results in an overestimate of the cardinality, typically around twice the actual value.

In the case of intersecting two HLLs, it is not sufficient to only determine the minimum value between corresponding registers; the resulting regvector must also be adjusted accordingly. This highlights the complexity of handling intersections, as a one-size-fits-all solution is not applicable.

Each register in the registers vector represents data items with hash values sharing the same prefix bits, serving as an identifier for that register. Therefore, any hash with the same prefix bits and fewer trailing zeros than the current register value could potentially exist.

Despite the challenges of intersecting HLLs, the resulting registers still exhibit key properties of HLLs:

1. They are idempotent.
2. They provide a unique representation for each dataset.
3. The relationship between registers and dataset cardinality remains intricate and not straightforward.

Addressing the shortcomings in the current implementations of HyperLogLog (HLL)

Even without any modifications, the current implementations of HLL are already effective. The only issue arises with intersections, but this can be easily resolved by using an inclusion/exclusion formula.

```python
card(intersect(A, B)) = (card(A) + card(B)) - card(union(A, B))
```

By using this method, we can determine the cardinality of the intersection between datasets A and B. However, we are unable to obtain the approximate HyperLogLog (HLL) set on its own. The solution to this issue is surprisingly straightforward. By making slight enhancements to the data structures of the registers, we can achieve the desired outcome.

Moving forward, we will be transitioning to Julia for our data analysis needs. This decision is primarily motivated by Julia's speed and its compatibility with the convenient features we rely on, similar to Python.

Introducing a new data structure known as **counters**, we will be utilizing these enhanced registers in our analysis going forward.

The most significant change being made is the replacement of the integer (representing the maximum number of consecutive zeros) with a BitVector. This update will enable us to retain all counts of consecutive zeros for every register, which we refer to as bins.

```python
struct HllSet{P}
    counts::Vector{BitVector}

    function HllSet{P}() where {P}
        validate_P(P)
        n = calculate_n(P)
        counts = create_bitsets(n)
        return new(counts)
    end
```

```
    function validate_P(P)
        isa(P, Integer) || throw(ArgumentError("P must be integer"))
        (P < 4 || P > 18) && throw(ArgumentError("P must be between 4 and 18"))
    end

    function calculate_n(P)
        return 1 << P
    end

    function create_bitsets(n)
        return [falses(64) for _ in 1:n]
    end
end
```

In BitVector, the maximum number of consecutive zeros is determined by the highest index in the BitVector where a "1" is present.

HllSet operations

1. Adding new item to HllSet

There are two versions of this function available:

1. The first version supports adding a single item at a time.
2. The second version supports adding multiple items as a set.

```
Python
function add!(hll::HllSet{P}, x::Any) where {P}
    h = hash(x)
    bin = getbin(hll, h)
    idx = getzeros(hll, h)
    hll.counts[bin][idx] = true
    return hll
end

function add!(hll::HllSet{P}, values::Set) where {P}
    for value in values
        add!(hll, value)
    end
    return hll
end
```

2. Union (union)

The operation **union** serves as an alternative to the **MERGE** operation in conventional HyperLogLog (HLL) implementations.

```python
function Base.union!(dest::HllSet{P}, src::HllSet{P}) where {P}
    length(dest.counts) == length(src.counts) || throw(ArgumentError("HllSet{P} must have same size"))
    for i in 1:length(dest.counts)
        dest.counts[i] = dest.counts[i] .| src.counts[i]
    end
    return dest
end
```

Please note that in the HLL merge operation, we select the larger of two integers by utilizing the max function.

```python
dest.counts[i] = max(dest.counts[i], src.counts[i])
```

By introducing a new struct that reflects the HllSet structure, we can seamlessly integrate the bitwise OR (.|) operation for BitVectors.

3. Intersection (intersect)

The introduction of the intersection operation became possible when we transitioned from using **Vector(Int)** to **Vector(BitVector)** in the struct definition of **counters**.

```python
function Base.intersect(x::HllSet{P}, y::HllSet{P}) where {P}
    length(x.counts) == length(y.counts) || throw(ArgumentError("HllSet{P} must have same size"))
    z = HllSet{P}()
    for i in 1:length(x.counts)
        z.counts[i] = x.counts[i] .& y.counts[i]
    end
    return z
```

```
end
```

The function returns a HllSet that can be utilized in various operations involving a collection of HllSets.
You can also recover the original values of this intersection if you've maintained all elements from every dataset in the system's inverted index. We will explore this aspect of the HllSet in our upcoming posts.

4. Difference (diff)

```Python
function Base.diff(x::HllSet{P}, y::HllSet{P}) where {P}
    length(x.counts) == length(y.counts) || throw(ArgumentError("HllSet{P} must
have same size"))
    z = HllSet{P}()
    for i in 1:length(x.counts)
        z.counts[i] = x.counts[i] .& .~(y.counts[i])
    end
    return z
end
```

Proving that HllSet is a Set

The operations on sets must adhere to the following properties [7]:

Fundamental Properties:
- Commutative Property:
1. (A ∪ B) = (B ∪ A)
2. (A ∩ B) = (B ∩ A)

- Associative Property:
3. (A ∪ B) ∪ C) = (A ∪ (B ∪ C))
4. (A ∩ B) ∩ C) = (A ∩ (B ∩ C))

- Distributive Property:
5. ((A ∪ B) ∩ C) = (A ∩ C) ∪ (B ∩ C)
6. ((A ∩ B) ∪ C) = (A ∪ C) ∩ (B ∪ C)

- Identity:
7. (A ∪ ∅) = A
8. (A ∩ U) = A

Additional Laws:
- Idempotent Laws:
9. (A ∪ A) = A
10. (A ∩ U) = A

Source code for proving that HllSet satisfies all of these requirements you can find in **lisa.ipynb**.[8]

Summary

This project aims to provide a fresh perspective on the widely recognized computing concept of HyperLogLog. We believe that HyperLogLog has a broader scope beyond simply calculating the cardinality of a set. By utilizing the HLL representation of data, we can perform various analytical tasks without directly accessing the original data. These tasks may include data searching, creating structural groupings, comparing distributions of values in sets, and more.

References

1. https://en.wikipedia.org/wiki/HyperLogLog
2. https://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
3. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf
4. https://redis.io/docs/data-types/probabilistic/hyperloglogs/
5. https://github.com/ascv/HyperLogLog/blob/master/README.md
6. https://en.wikipedia.org/wiki/Inclusion%E2%80%93exclusion_principle
7. https://en.wikipedia.org/wiki/Algebra_of_sets
8. https://github.com/alexmy21/lisa_meta/blob/main/lisa.ipynb