Question 1)

1. It can improve the performance of Hadoop map reduce by making the solution load balanced properly. It will prevent some reducers to be overloaded and equally balances out the distribution of data. It will also send the relevant key value pairs to the same reducer and we won’t have to do another MapReduce job to do the final aggregation. It can also reduce the number of computations done on the reducer. It can also help with grouping data in the map phase so that only important information goes to the reducer.
2. The data structure we can use for both the RDD and data frame solution is set. The spark feature that we can use is ‘filter’ to ignore a given set of items in the mining task. We use the set data structure because it is simple and reliable. We use filter because it is efficient to use it for big data.
3. One of the reasons might be because of the lack of partitioning of nodes. This will allow the code to do parallel jobs increasing the speed of the performance. It can also be because of the approach of using RDD over data frame.

Question 4:

1. A = “the data is big the value is high” and B = “the value in the data is high”.

2-shingle for A: {the data, data is,,is big, big the, the value, value is, is high}

2-shingle for B: {the value, value in, in the, the data, data is, is high}

The intersection between 2-shingle for A and 2-shingle for B is

{the data, data is, the value, is high} -> length = 4

Union is

{the data, data is,is big, big the, the value, value is, is high, value in, in the} -> length = 9

So Jaccard similarity is 4/9 = 0.444

1. h1(n) = (2n + 1) mod M where M=7

h2(n) = (n + 2) mod M

h3(n) = (5n + 6) mod M

for row 1:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector 1 | Vector 2 |  |
| H1(1) | 3 | infinite |  |
| H2(1) | 3 | infinite |  |
| H3(1) | 4 | infinite |  |

For row 3:

|  |  |  |
| --- | --- | --- |
|  | Vector 1 | Vector 2 |
| H1(3) | 0 | 0 |
| H2(3) | 3 | 5 |
| H3(3) | 0 | 0 |

Wont change for row 4

For row 5:

|  |  |  |
| --- | --- | --- |
|  | Vector 1 | Vector 2 |
| H1(5) | 0 | 0 |
| H2(5) | 0 | 0 |
| H3(5) | 0 | 0 |

1. The min hash signatures for both vectors are zeros. And as a result, the bands will also be zeros and h(b) will be 0. The hash values in the vectors are equal to each other. So they are a candidate pair. 210100121

Question 5:

1. i) h1(str) = Σ(𝑖−′𝑎′)𝑖∈𝑠𝑡𝑟 mod 9

h2(str) = (str.length \* 2) mod 9

h3(str) = (str.length \* str.length) mod 9

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | big | graph | data | Bloom |
| H1 | (1+8+6)mod9=6 | (6+17+15+7)mod9=0 | 21mod9=3 | 100100100 |
| H2 | 6 | 1 | 8 | 110100101 |
| H3 | 0 | 7 | 7 | 110100111 |

ii)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | big | graph | data | Bloom |
| H1 | (1+8+6)mod9=6 | (6+17+15+7)mod9=0 | 21mod9=3 | 100100100 |
| H2 | 6 | 1 | 8 | 110100201 |
| H3 | 0 | 7 | 7 | 210100221 |

iii) Using the bloom filter 110100111, the word ‘axe’ would result in h1 = (0+23+4) mod 9 = 0, h2 = 6, and h3 = 0. This would result in a false positive.

1. With 𝜖=0.2, dividing the sequence into windows of 1/𝜖 = 5.

[1,1,2,3,4,5,1,1,1,5,3,3,1,1,2]

Window one -> [1,1,2,3,4]

Frequency during count (before decrement) = {1:2, 2:1,3:1,4:1}

Frequency after count after decrement = {1:1}

Window two -> [5,1,1,1,5]

Frequency during count (before decrement) = {1:4, 5:2}

Frequency after count after decrement = {1:3,5:1}

Window three -> [3,3,1,1,2]

Frequency during count (before decrement) = {1:5, 2:1,3:2,5:1}

Frequency after count after decrement = {1:4,3:1}

Question 6:

1. Iteration one:

Map:

Read n1 --> 0 | n2: 5, n6: 3

Emit: (n2, 5), (n6, 3), and the adjacency list (n1, n2: 5, n6: 3)

The other lists will also be read and emit, but they do not contribute, and

thus ignored.

Reduce:

Receives: (n2, 5), (n6, 3), (n1, <0, (n2: 5, n6: 3)>)

The adjacency list of each node will also be received, ignored in example

Emit:

N1 --> 0 | n2: 5, n6: 3

N2 --> 5 | n3: 8

N6 --> 3 | n2: 1, n4:3,n5:3

Iteration two:

Map:

Read: n2 --> 5 | n3:8

Emit: (n3, 13), (n2, <5, (n3: 8)>)Read: n6 --> 3 | n2: 1, n4:3, n5:3

Emit: (n2, 4), (n4, 6), (n5, 6)

Ignore the processing of the other lists

Reduce:

Receives: (n2, (4, <5, (n3: 8)>)), (n6, <3, n1: 3, n3:9，n4:2>), (n3, 13)), (n5, 6)

Emit:

N2 --> 4 | n3:8

N6 --> 3 | n1: 3, n3:9，n4:2

n3 --> 13 | n5:10

n5 --> 6 | n1:7, n3:3

Iteration three:

Map:

Read: N2 --> 4 | n3:8

Emit: (n3, 12), (n2, <4, (n3:8)>)

Read: n6 --> 3 | n2: 1, n4:3， n5:3

Emit: (n2, 4), (n4, 6), (n5, 6)，(n6, <3, (n2: 1, n4:3， n5:3)>)

Read: n3 --> 13 | n5:10

Emit: (n5, 23)，(n3, <13, (n5:10)>)

Read: n5 --> 6 | n1:7, n3:3

Emit: (n1, 13), (n3, 9), (n5, <6, (n1:7, n3:3)>)

Reduce:

Emit:

N2 --> 4 | n3:8

N6 --> 3 | n2: 1, n4:3, n5:3

n3 --> 9 | n5:10

n5 --> 6 | n1:7, n3:3

n4 --> 6 | n2:3, n3:3

1. rank value for each node is 1/6 initially

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | N1 | N2 | N3 | N4 | N5 | N6 |
| N1 | 0 | 0 | 0 | 0 | 1/2 | 0 |
| N2 | 1/2 | 0 | 0 | 1/2 | 0 | 1/3 |
| N3 | 0 | 1 | 0 | 1/3 | 1/2 | 0 |
| N4 | 0 | 0 | 0 | 0 | 0 | 1/3 |
| N5 | 0 | 0 | 1 | 0 | 0 | 1/3 |
| N6 | 1/2 | 0 | 0 | 0 | 0 | 0 |

R0 = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6]

R1 = [1/12, 4/18, 396/1296 , 1/18, 4/18,1/12]

R2=[1/9, 42/432, 4/18+1/54+4/36, 1/36, 396/1296+1/36, 1/24]

R2 = [0.111, 0.097, 0.351, 0.027, 0.333, 0.041]

Calculations were very tedious im sorry!