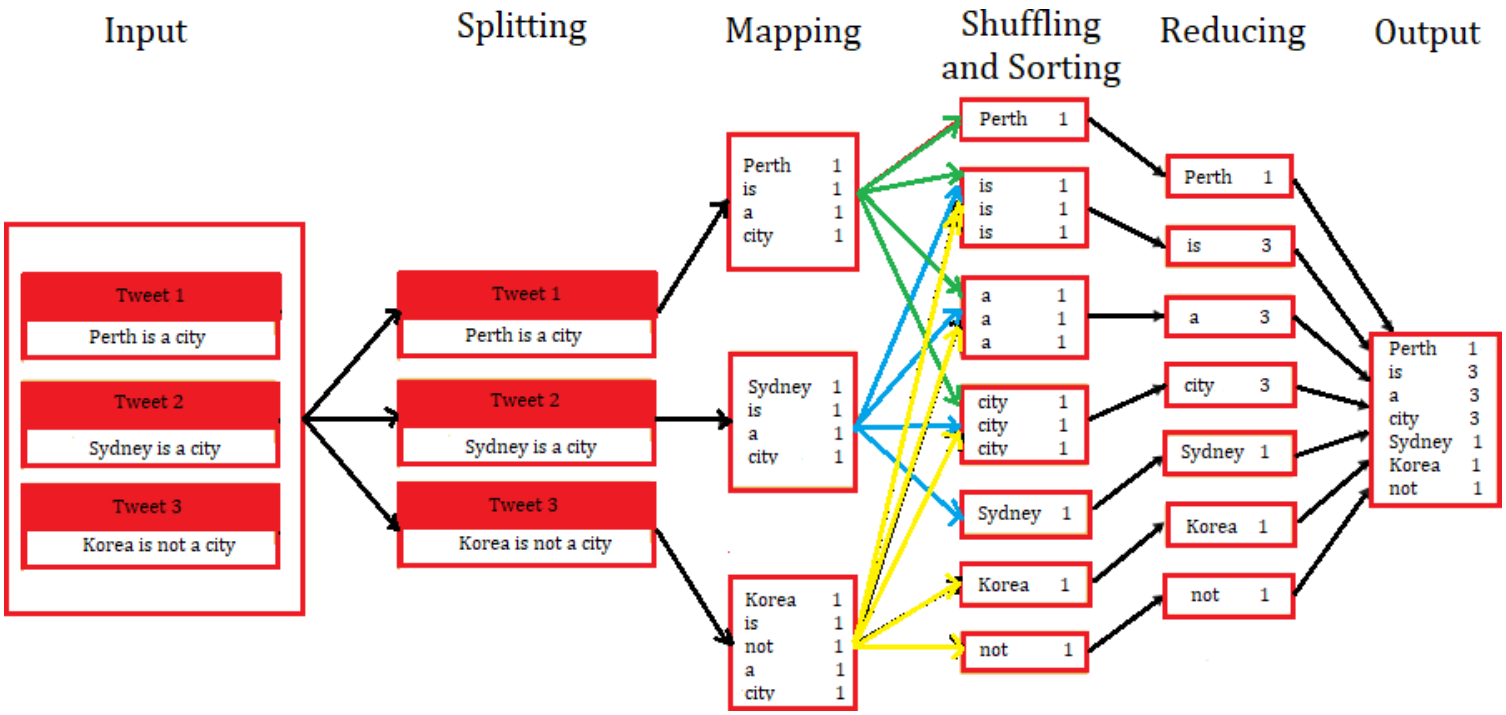
Task 2

The mapper takes in a line of the input file (i.e. curated_body_tweet.txt) as input and splits it into words. It then emits an intermediate key/value pair for each word observed, with the word itself as the key and a value of 1 (i.e. (word, 1)).

The reducer totals the partial counts for each word to derive the final count and emits a single key/value with the word and final count.

Flow chart



Pseudocode

Calculate the count of number of occurrences of each word in the text of Tweets.
The mapper emits an intermediate key/value pair for each word in the text of a tweet.
1: **function** MAPPER(collection of tweets *a*, tweet *t*)
2: **for all** word *w* ∈ tweet *t* **do**
3: EMIT_INTERMEDIATE(word *w*, count *1*)

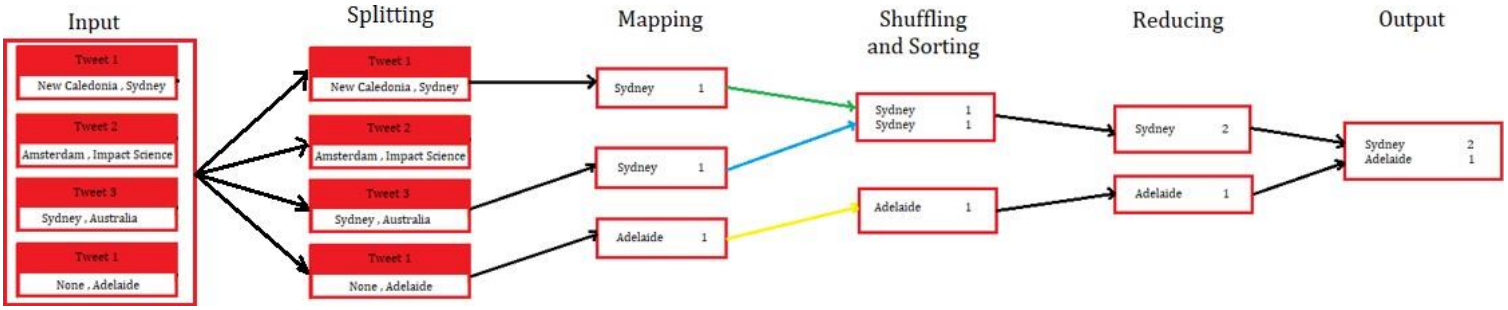
Calculate the count of number of occurrences of each word in the text of Tweets.
The reducer totals all counts for each word.
1: **function** REDUCER(word *w*, counts [c_1, c_2, . . .])
2: *sum* ← 0
3: **for all** count *c* ∈ counts [c_1, c_2, . . .] **do**
4: *sum* ← *sum* + *c*
5: EMIT(word *w*, count *sum*)

Task 3

The mapper takes in the input file (i.e. tweet_city.txt) as input and splits it into location texts. With a helper function to extract Australian location names from the text, the mapper then detects any word that is the Australian city name and emits an intermediate key/value pair for the location word observed, with the Australian location name itself as the key and a value of 1 (i.e. (location, 1)).

The reducer totals the partial counts for each Australian location name to derive the final count and emits a single key/value with the location as the key and final count as the value.

Flow chart



Pseudocode

Calculate the count of number of tweets for a list of different cities in Australia.
The mapper detects any word that is the Australian city name with a helper function (for example, *contains(Australian location name)* and emits an intermediate key/value pair for the location word observed. 1: **method** MAPPER(all location groups *a*, location group *g*) 2: **for all** location *l* ∈ location group *g* **do** 3: **if** *l*.contains*(*Australian location name) **do** 4: EMIT_INTERMEDIATE(location *l*, count *1*)

Calculate the count of number of tweets for a list of different cities in Australia.
The reducer totals all counts for each word identified as an Australian city. 1: **method** REDUCER(location *l*, counts [c_1, c_2, . . .]) 2: *sum* ← 0 3: **for all** count *c* ∈ counts [c_1, c_2, . . .] **do** 4: *sum* ← *sum* + *c* 5: EMIT(location *l*, count *sum*)

Task 4

Merge Sort

- *How many MapReduce Jobs? Why?*

There is one MapReduce job that partitions the input dataset into individual blocks of single ids that are then processed by the map task in parallel. The mapper assigns each object id to a partition for downstream comparison, and generates the output in the form of intermediate key-value pairs (partition, object id) which are for the reducer1 to group into independent pairs for comparison. Reducer2 aggregates the results by merging two sorted sub-partitions in the sorted ascending order that the MapReduce application requests.

- *What to write in the mapper(s)?*

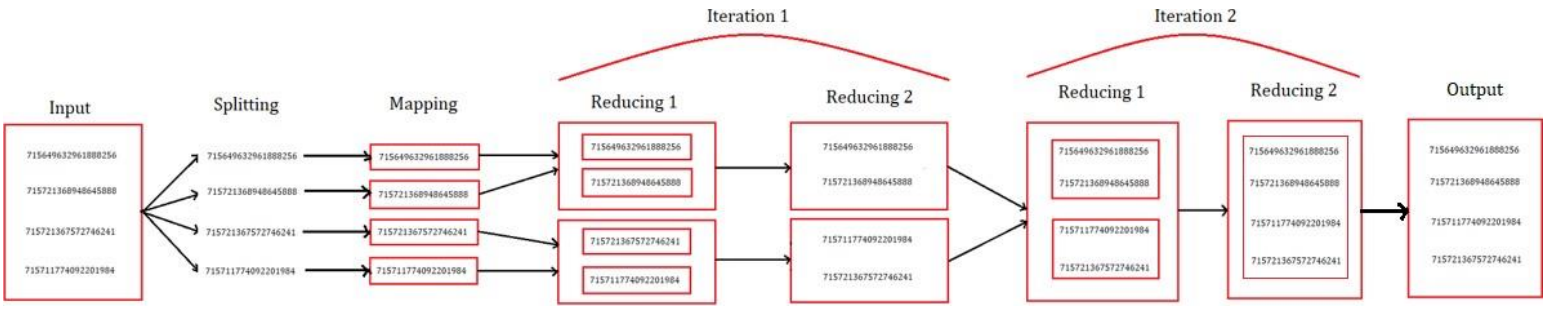
The mapper takes in the input file (i.e. curated_tweet_object_id.json) as input and splits it into different ids. It then assigns each object id to a partition for downstream comparison and emits an intermediate key/value pair for each id observed, with a partition (i.e. the index of the observed id within the list of different ids after splitting) as the key and the object id as the value (i.e. (partition, object id)).

- *What to write in the reducer(s)?*

There are two reducers for Merge Sort. The reducer1 merges two successive partitions for the sake of comparison by dividing their index by 2 so that both partitions achieve the same index, and emits a single key/value, with the merged partition as the key and a pair of lists of sorted object ids as the value. The reducer2 merges two sorted sub-partitions in the sorted ascending order, and emits a single key/value, with the merged partition as the key and the sorted list of all ids of two sub-partitions as the value. As the Merge Sort algorithm requires that these two processes are repeated until the two ordered halves are merged together to form a single ordered list, the reducer1 and reducer2 will be recursively executed until the single

ordered list is obtained. In this final step, the reducer2 emits a single key/value, with the merged partition asthe key and a sorted list of all object ids from the input file.

Flow chart



Pseudocode

Implement the Merge Sort algorithm using Map-Reduce.
The mapper assigns each object id to a partition for downstream comparison and emits an intermediate key/value pair for each id observed, with a partition (i.e. the index of the observed id within the list of different ids after splitting) as the key and the object id as the value (i.e. (partition, object id)). 1: **function** MAPPER(collection of tweets *a*, tweet *t*) 2: **for all** (partition *p*, object id *id)* ∈ enumerate(tweet *t)* **do** 3: EMIT_INTERMEDIATE(partition *i*, object id *id*)

Implement the Merge Sort algorithm using Map-Reduce.
The reducer1 merges two successive partitions for the sake of comparison by dividing their index by 2 so that both partitions achieve the same index, and emits a single key/value, with the merged partition as the key and a pair of lists of sorted object ids as the value. 1: **function** REDUCER1(partition *i*, list of object ids *lst*) 2: *i* ← *i* // 2 3: EMIT1(partition *i*, list of object ids *lst*)

Implement the Merge Sort algorithm using Map-Reduce.
The reducer2 merges two sorted sub-partitions in the sorted ascending order, and emits a single key/value, with the merged partition as the key and the sorted list of all ids of two sub-partitions as the value. 1: **function** REDUCER2(partition *i*, pair of lists *p*) 2. *sorted_ob_ids* ← list() 3. **if** len(*p*) = 2 **do** 4. *left* ← *p[0]* 5. *right* ← *p[1]* 6. *x* ← 0 7. *y* ← 0 8. **while** (*x* < len(*left*) and *y* < len(*right*)) **do** 9. **if** (*left[x]* < *right[y]*) **do** 10. *sorted_ob_ids*.append*(left[x])* 11. *x* ← *x* + 1 12. **else do** 13. *sorted_ob_ids.append(right[y])* 14. *y* ← *y* + 1 15. **while** (*x* < len(*left*)) **do** 16. *sorted_ob_ids*.append*(left[x])* 17. *x* ← *x* + 1 18. **while** (*y* < len(*right*)) **do** 19. *sorted_ob_ids*.append*(right[y])* 20. *y* ← *y* + 1

```
21.    else do
22.        sorted_ob_ids.extend(pair_of_lists[0])
23.    EMIT2(partition i, list of sorted ids sorted_ob_ids)
```

Bucket Sort

- *How many MapReduce Jobs? Why?*

There is one MapReduce job that partitions the input dataset into individual blocks of single ids that are then processed by the map task in parallel. The mapper assigns each object id to its relevant bucket in the form of intermediate key-value pairs (bucket, object id). Reducer1 sorts the object ids within a bucket ascendingly and outputs key-value pairs as (bucket, list of sorted object ids). Reducer2 aggregates the results by obtaining the total ordering as the buckets are ordered in their own rights.

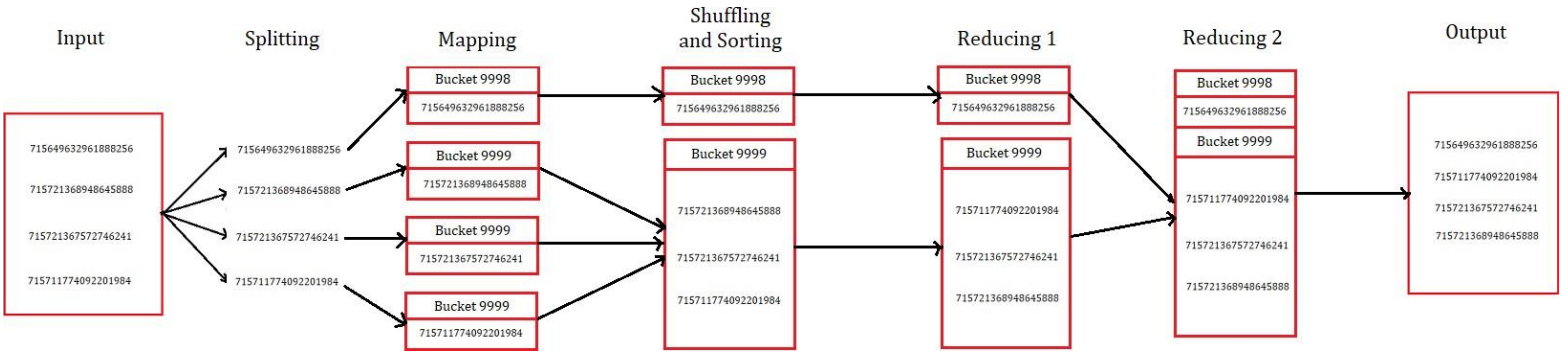
- *What to write in the mapper(s)?*

The mapper takes in the input file (i.e. curated_tweet_object_id.json) as input and splits it into different ids. It then assigns each object id to its relevant bucket and emits an intermediate key/value pair for each id observed, with a bucket (i.e. bucket = total number of object ids * (object id // max object id)) as the key and the object id as the value (i.e. (bucket, object id)).

- *What to write in the reducer(s)?*

There are two reducers for Bucket Sort. The reducer1 sorts the object ids within a bucket ascendingly, and emits a single key/value, with the bucket as the key and a list of sorted object ids as the value. The reducer2 aggregates the results to obtain the total ordering as the buckets are ascendingly ordered in their own rights,and emits the value as the sorted list of all ids (the key is discarded and is merely None).

Flow chart



Pseudocode

Implement the Bucket Sort algorithm using Map-Reduce.

The mapper assigns each object id to its relevant bucket and emits an intermediate key/value pair for each id observed, with a bucket (i.e. bucket = total number of object ids * (object id // max object id)) as the key and the object id as the value (i.e. (bucket, object id)).

```
1: function MAPPER(collection of ids a, max object id mx)
2:     for all object id id ∈ collection of ids a do
3:         EMIT_INTERMEDIATE(len(a) * (id // mx), id)
```

Implement the Bucket Sort algorithm using Map-Reduce.

The reducer1 sorts the object ids within a bucket ascendingly, and emits a single key/value, with the bucket as the key and a list of sorted object ids as the value.

```
1: function REDUCER1(bucket b, list of unsorted object ids lst)
2.     for all index i ∈ range(1, len(lst)) do
3.         ahead ← lst[i]
4.         j ← i - 1
5.         while j >= 0 and lst[j] > ahead do
```

```
6.          lst[j + 1] ← lst[j]
7.          j ← j - 1
8.      lst[j + 1] ← ahead
9.  EMIT1(bucket b, list of sorted object ids lst)
```

Implement the Bucket Sort algorithm using Map-Reduce.

The reducer2 aggregates the results to obtain the total ordering as the buckets are ascendingly ordered in their own rights, and emits the value as the sorted list of all ids (the key is discarded and is merely None).

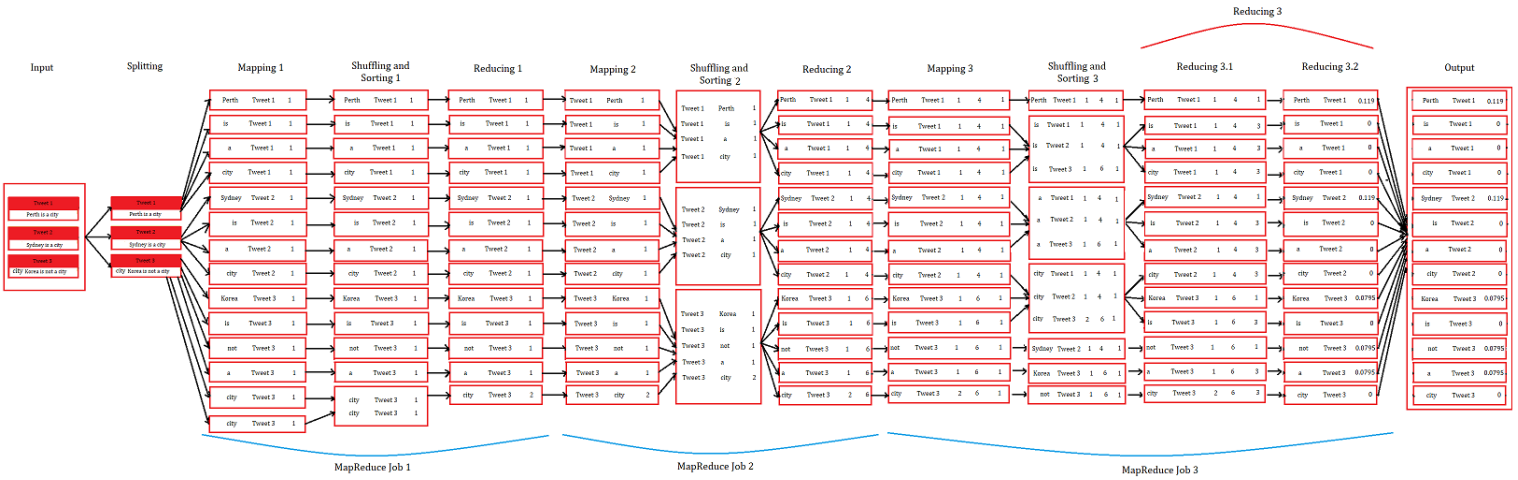
```
1: function REDUCER2 (_, lists of sorted object ids lsts)
2.  for all list of sorted object ids lst ∈ lists of sorted object ids lsts do
3.      EMIT2(None, list of sorted object ids lst)
```

TASK 5

As three unknown values of TF Numerator and Denominator and IDF need to be computed, there are three MapReduce jobs to implement the TF-IDF algorithm for the term “health” in the text of the Tweets.

- The first job derives TF Numerator, which partitions the input dataset into individual blocks of wordsfor every single tweet (document) that are then processed by the map task in parallel. The mapper1 generates the output in the form of intermediate key-value pairs (<word, document name>, 1). The reducer1 totals all counts for each word within a document and outputs (<word, document name>, frequency of the word in the same document).
- The second job derives TF Denominator, which partitions the input dataset into individual blocks of words for every single document that are then processed by the map task in parallel. The mapper2 takes the output of the reducer1 and generates the output in the form of intermediate key-value pairs (document name , <word, frequency of the word in the same document>). The reducer2 then totals the frequency of each word within a document to obtain the total number of words in this document and outputs (<word, document name>, <frequency of the word in the same document, total number ofwords in this document>).
- The third job derives TF-IDF, which partitions the input dataset into individual blocks of words for every single document that are then processed by the map task in parallel. The mapper3 takes the output of the reducer2 and generates the output in the form of intermediate key-value pairs (word, <document name, frequency of the word in the same document, total number of words in this document, 1>). The reducer3 then totals all counts of documents that contains the word to obtain IDF value and outputs (<word, document name>, TF-IDF).

Flow chart



Pseudocode

Implement the TF-IDF algorithm using Map-Reduce for the term "health" in the text of the Tweets.

The mapper1 generates the output in the form of intermediate key-value pairs (<word, document name>, 1).

```
1: function MAPPER1(collection of tweets data)
2.     for all index, line ∈ enumerate(collection of tweets data) do
3.         text ← json.loads(line)[0]
4.         if len(text) != 0 do
5.             text ← text.lower()
6.             words ← text.split()
7.             for all word w ∈ words do
8.                 EMIT_INTERMEDIATE1(word w, tweet name f"Tweet {index + 1}", count 1)
9.         else do
10.            EMIT_INTERMEDIATE1(word "N/A", f"Tweet {index + 1}", count 0)
```

Implement the TF-IDF algorithm using Map-Reduce for the term "health" in the text of the Tweets.

The reducer1 totals all counts for each word within a document and outputs (<word, document name>, frequency of the word in the same document).

```
1: function REDUCER1(word w, document name doc_name, counts c)
2.     word_frequency ← sum(c)
3.     EMIT1(word w, document name doc_name, word frequency word_frequency)
```

Implement the TF-IDF algorithm using Map-Reduce for the term "health" in the text of the Tweets.

The mapper2 takes the output of the reducer1 and generates the output in the form of intermediate key-value pairs (document name, <word, frequency of the word in the same document>).

```
1: function MAPPER2(word w, document name doc_name, word frequency word_frequency)
2:     EMIT_INTERMEDIATE2(document name doc_name, word w, word frequency word_frequency)
```

Implement the TF-IDF algorithm using Map-Reduce for the term "health" in the text of the Tweets.

The reducer2 then totals the frequency of each word within a document to obtain the total number of words in this document and outputs (<word, document name>, <frequency of the word in the same document, total number of words in this document>).

1: **function** REDUCER2(word *w*, document name *doc_name,* word frequency *word_frequency,* frequency counts *frequency_counts*)
2. *total_count_within_document* ← sum(*frequency_counts*)
3. **if** total_count_within_document = 0 **do**
4. EMIT2(word *w,* document name *doc_name,* word frequency *word_frequency,* total count within document *total_count_within_document* + 1)
5. **else do**
6. EMIT2(word *w,* document name *doc_name,* word frequency *word_frequency,* total count within document *total_count_within_document*)

Implement the TF-IDF algorithm using Map-Reduce for the term "health" in the text of the Tweets.

The mapper3 takes the output of the reducer2 and generates the output in the form of intermediate key-value pairs (word, <document name, frequency of the word in the same document, total number of words in this document, 1>).

1: **function** MAPPER3(word *w,* document name *doc_name,* word frequency *word_frequency,* total count within document *total_count_within_document*):
2. EMIT_INTERMEDIATE3(word *w,* document name *doc_name,* word frequency *word_frequency,* total count within document *total_count_within_document,* count *1*)

Implement the TF-IDF algorithm using Map-Reduce for the term "health" in the text of the Tweets.

The reducer3 then totals all counts of documents that contains the word to obtain IDF value and outputs (<word, document name>, TF-IDF).

1: **function** REDUCER3(word *w,* document name *doc_name,* word frequency *word_frequency,* total count within document *total_count_within_document,* count *counts*)
2. *documents_containing_word* ← *sum(counts)*
3. *tf* ← *word_frequency / total_count_within_document*
4. *idf* ← math.log10*(10000 / documents_containing_word)*
5. *tf_idf* ← *tf * idf*
6. EMIT3*(*word *w,* document name *doc_name,* tfidf *tf_idf)*