33085625_FIT3182_Assignment_1

April 9, 2024

1 FIT3182 - Assignment 1 - Questions Workbook

This assignment consists of **four questions** that are, taken together, worth 10% of the final marks. **Please treat this Assignment as a take-home test. This is an individual assignment and you should complete the questions on your own**. You should not attempt to post questions on EdForum seeking solutions to the answers. If you require clarification on the Assignment questions, you can post a private post on EdForum or seek consultation from the tutors.

1. The first question is related to **Parallel Search Algorithms (2 Mark)**.
2. The second question is related to **Parallel Join Algorithms (4 Marks)**.
3. The third question is related to **Parallel Sort Algorithms (2 Marks)**.
4. The fourth question is related to **Parallel GroupBy Algorithms (2 Marks)**.

Instructions: - You will be using Python 3. Answer all questions inside this Jupyter Notebook. Please use the provided Docker to load the Jupyter Notebook. - Read the instructions, skeleton code, and comments carefully. - There are **code blocks that you need to complete** yourself as a part of the assignment. - You are also required to **answer the questions below**. - **Comment each line of code properly such that the tutor can easily understand what you are trying to do in the code. Marks may be deducted for insufficient or unclear comments.** - Once completed, please rename the Jupyter Notebook to include your Student ID at the beginning of the filename (e.g., 12345678_FIT3182_Assignment_1.ipynb). Submit this Jupyter Notebook into the Assignment 1 submission link in Moodle. Please refer to the Assignment 1 submission link (or page) in Moodle for information about the submission due date.

Please include your details as follows: - Student ID: 33085625 - Last Name: Foo - First (or Preferred) Name: Kai Yan - Monash Student Email: kfoo0012@student.monash.edu

[1]:
```python
from pprint import pprint
```

1.1 Dataset

For this test, we will use the following tables R and S to write solutions to three parallel algorithms.

[2]:
```python
# R consists of 15 pairs, each comprising two attributes (nominal and numeric)
R = [('Adele',8),('Bob',22),('Clement',16),('Dave',23),('Ed',11),
     ('Fung',25),('Goel',3),('Harry',17),('Irene',14),('Joanna',2),
     ('Kelly',6),('Lim',20),('Meng',1),('Noor',5),('Omar',19)]

# S consists of 8 pairs, each comprising two attributes (nominal and numeric)
```

```
S = [('Arts',8),('Business',15),('CompSc',2),('Dance',12),('Engineering',7),
     ('Finance',21),('Geology',10),('Health',11),('IT',18)]
```

1.2 1. Parallel Searching Algorithm (2 marks)

This section consist of 4 sub-questions.

In this task, you will build a **parallel search algorithm for range selection (continuous)** for a given query.

Implement a parallel search algorithm that uses local linear search (i.e., `linear_search()`) and is able to work with the hash partitioning method (i.e., `h_partition()`). **Complete the code block between `### START CODE HERE ###` and `### END CODE HERE ###`.**

1.2.1 1.1 Local Search

[3]:
```
# Linear search function
def linear_search(data, key):
    """
    Perform linear search on data for the given key.

    Arguments:
    data -- an input dataset (list OR numpy array)
    key -- an query attribute (number)

    Return:
    result --  list of matched records such as [('Adele', 8)]
               or the position of searched record
               whichever is more appropriate for your code
    """
    matches = []
    match_record = None
    position = -1 # not found position

    ### START CODE HERE ###

    for record in data:
        # Because data is in tuple form so we extract data from tuple with this
        name, number = record
        if number == key:  # If number is matched with key
            match_record = (name, number)
            position = data.index(record)
            # Add the matched record as a tuple in a list
            matches.append(match_record)
            break

    ### END CODE HERE ###
    return matches
```

[4]:
```
results = linear_search(R, 20)
print(results)
```

```
[('Lim', 20)]
```

1.2.2 1.2 Hash Partition

[5]:
```
# Define a simple hash function.
def s_hash(x, n):
    """
    Define a simple hash function for demonstration

    Arguments:
    x -- an input record attribute (int)
    n -- the number of processors (int)

    Return:
    result -- the hash value of x
    """
    return x % n
```

[6]:
```
# Hash data partitioning function.
# We will use the s_hash function defined above
def h_partition(data, n):
    """
    Perform hash data partitioning on data

    Arguments:
    data -- an input dataset (list)
    n -- the number of processors (int)

    Return:
    result -- the partitioned subsets of D
    """
    partitions = {}

    ### START CODE HERE ###

    for record in data:
        # Because data is in tuple form so we extract data from tuple with this
        name, number = record
        h = s_hash(number, n)  # Get the hash key of the input
        if h in partitions:
            partitions[h].append((name, number))
        else:
            partitions[h] = [(name, number)]

    ### END CODE HERE ###

    return partitions
```

[7]:
```
partitions = h_partition(R, 5)
pprint(partitions)
```

```
{0: [('Fung', 25), ('Lim', 20), ('Noor', 5)],
 1: [('Clement', 16), ('Ed', 11), ('Kelly', 6), ('Meng', 1)],
 2: [('Bob', 22), ('Harry', 17), ('Joanna', 2)],
 3: [('Adele', 8), ('Dave', 23), ('Goel', 3)],
 4: [('Irene', 14), ('Omar', 19)]}
```

1.2.3 1.3 Parallel Search

[8]:
```
from multiprocessing import Pool

def collect_result(results, result):
    # Callback for when linear_search returns a result.
    # The results list is modified only by the main process, not the pool␣
 ↪workers.
    results.append(result)

# Parallel searching algorithm for range selection
def parallel_search_range(data, query_range, n_processor):
    """
    Perform parallel search for range selection on data for the given key.

    Arguments:
    data -- the input dataset (list)
    query_range -- a tuple with inclusive range end-points (e.g. [30, 50] means␣
 ↪the interval 30-50)
    n_processor -- the number of parallel processors (int)

    Return:
    results -- the matched record information
    """

    pool = Pool(processes=n_processor)

    ### START CODE HERE ###

    results = []

    # Perform data partitioning first
    data_partitioning = h_partition(data, n_processor)

    for query in range(query_range[0], query_range[1], 1):
        # Each element in data_partitioning has a pair (hash key: records)
        query_hash = s_hash(query, n_processor)
        partitioned_data = list(data_partitioning[query_hash])
        result = pool.apply(linear_search, [partitioned_data, query])
        collect_result(results, result)

    ### END CODE HERE ###

    return results
```

[9]:
```
n_processor = 3
results = parallel_search_range(R, [5, 15], n_processor)
pprint(results)
```

```
[[('Noor', 5)],
 [('Kelly', 6)],
 [],
 [('Adele', 8)],
 [],
 [],
 [('Ed', 11)],
 [],
 [],
 [('Irene', 14)]]
```

1.2.4 1.4 Parallel Search Variants

Briefly answer the following.

1. How would the parallel search you implemented need to be changed if we switched to round-robin partition for the data? Provide a small snippet of code with the modified search loop.

Your Answers: First I would have specifically implement the round-robin partition in another function and perform the round-robin partition for the data then save it in data_partitioning variable then only perform parallel search.

[10]:
```
def round_robin_partition(data, n_processor):
    result = []

    for i in range(n_processor):
        result.append([])

    # For each bin, perform the following
    for name, number in enumerate(data):
        # Calculate the index of the bin that the current data point will be␣
 ↪assigned
```

```
        index_bin = (int) (name % n_processor)
        result[index_bin].append(number)

    return result

def round_robin_partition_parallel_search(data, query_range, n_processor):
    pool = Pool(processes=n_processor)
    results = []

    # Perform data partitioning first
    data_partitioning = round_robin_partition(data, n_processor)

    for query in range(query_range[0], query_range[1], 1):
        # Each element in data_partitioning has a pair (hash key: records)
        query_hash = s_hash(query, n_processor)
        partitioned_data = list(data_partitioning[query_hash])
        result = pool.apply(linear_search, [partitioned_data, query])
        collect_result(results, result)

    return result
```

[11]:
```
n_processor = 3
results = round_robin_partition_parallel_search(R, [12, 15], n_processor)
pprint(results)
```

```
[('Irene', 14)]
```

1.3 2. Parallel Join Algorithm (4 marks)

This section consist of 5 sub-questions.

In this task, you will implement a **disjoint-partitioning based parallel join algorithm**. This algorithm consist of two stages. 1. Data disjoint-partitioning across processors. 1. Local join in each processor.

As a data partitioning method, use the range partitioning algorithm (i.e. `range_partition( )`). Assume that we have **3 parallel processors**, processor 1 will get records with join attribute value between 1 and 9, processor 2 between 10 and 19, and processor 3 between 20 and 29. Note that both tables R and S need to be partitioned using the same ranges described earlier.

As a joining technique, use the hash based join algorithm (i.e., `HB_join( )`). **Complete the code block between `### START CODE HERE ###` and `### END CODE HERE ###`.**

1.3.1 2.1 Range Partition

[12]:
```
# Range data partitionining function (Need to modify as instructed above)
def range_partition(data, range_indices):
    """
    Perform range data partitioning on data based on the join attribute
    Arguments:
    data -- an input dataset (list)
    range_indices -- the range end-points (e.g., [5, 10, 15] means
                        4 intervals: x < 5, 5 <= x < 10, 10 <= x < 15, 15 <= x)

    Return:
    result -- the partitioned subsets of D
    """
    result = []
    ### START CODE HERE ###

    # First, we sort the dataset according their values
    sort_by_value = sorted(data, key=lambda x: x[1])

    # Calculate the number of bins
    bins_number = len(range_indices)

    # For each bin, perform the following
    for i in range(bins_number):
        # Find elements to be belonging to each range
        partitioned_list = [x for x in sort_by_value if x[1] < range_indices[i]]
        # Add the partitioned list to the result
        result.append(partitioned_list)
        # Find the last element in the previous partition
        last_element = partitioned_list[len(partitioned_list)-1]
        # Find the index of of the last element
        last = sort_by_value.index(last_element)
        # Remove the partitioned list from the dataset
        sort_by_value = sort_by_value[int(last)+1:]

        # Append the last remaining data list
    result.append([x for x in sort_by_value if x[1] >=␣
↪range_indices[bins_number-1]])

    ### END CODE HERE ###

    return result
```

[13]:
```
pprint(range_partition(R, [10, 20]))
```

```
[[('Meng', 1),
  ('Joanna', 2),
  ('Goel', 3),
  ('Noor', 5),
  ('Kelly', 6),
  ('Adele', 8)],
 [('Ed', 11), ('Irene', 14), ('Clement', 16), ('Harry', 17), ('Omar', 19)],
 [('Lim', 20), ('Bob', 22), ('Dave', 23), ('Fung', 25)]]
```

1.3.2 2.2 Hash Join

[14]:
```
def H(r):
    """
    We define a hash function 'H' that is used in the hashing process works
    by summing the first and second digits of the hashed attribute, which
    in this case is the join attribute.

    Arguments:
    r -- a record where hashing will be applied on its join attribute

    Return:
    result -- the hash index of the record r
    """

    # Convert the value of the join attribute into the digits
    digits = [int(d) for d in str(r[1])]

    # Calulate the sum of elemenets in the digits
    return sum(digits)
```

[15]:
```
def HB_join(T1, T2):
    """
    Perform the hash-based join algorithm.
    The join attribute is the numeric attribute (in second position) in the␣
 ↪input tables T1 & T2.

    Arguments:
    T1 & T2 -- Tables to be joined

    Return:
    result -- the joined table
    """
    result = []
    dic = {} # We will use a dictionary

    # For each record in table T2
    for s in T2:
        # Hash the record based on join attribute value using hash function H␣
 ↪into hash table
        s_key = H(s)
        if s_key in dic:
            dic[s_key].add(s) # If there is an entry
        else:
            dic[s_key] = {s}
    print(dic)
    # For each record in table T1 (probing)
    for r in T1:
        # Hash the record based on join attribute value using H
        r_key = H(r)
        # If an index entry is found Then
        if r_key in dic:
            # Compare each record on this index entry with the record of table␣
 ↪T1
            for item in dic[r_key]:
                if item[1] == r[1]:
                    # Put the rsult
                    result.append( (r[0], r[1], item[0]) )

    return result
```

[16]:
```
# print the partitioned result
HB_join(R,S)
```

```
{8: {('Arts', 8)}, 6: {('Business', 15)}, 2: {('Health', 11), ('CompSc', 2)}, 3:
{('Finance', 21), ('Dance', 12)}, 7: {('Engineering', 7)}, 1: {('Geology', 10)},
9: {('IT', 18)}}
```

[16]:
```
[('Adele', 8, 'Arts'), ('Ed', 11, 'Health'), ('Joanna', 2, 'CompSc')]
```

1.3.3 2.3 Parallel Join

[17]:
```
# Include this package for parallel processing
import multiprocessing as mp

def consolidate_result(results, result):
    # This is called whenever HB_Join(T1, T2) returns a result.
    # The results list is modified only by the main process, not the pool␣
 ↪workers.
    results.append(result)

def DPBP_join(T1, T2, n_processor):
    """
    Perform a disjoint partitioning-based parallel join algorithm.
    The join attribute is the numeric attribute in the input tables T1 & T2

    Arguments:
    T1 & T2 -- Tables to be joined
    n_processor -- the number of parallel processors

    Return:
```

```
    result -- the joined table
    """

    ### START CODE HERE ###

    # Partition T1 & T2 into sub-tables using range_partition().
    # The number of the sub-tables must be the equal to the n_processor
    T1_subsets = range_partition(T1, [10, 20])
    T2_subsets = range_partition(T2, [10, 20])

    # Apply local join for each processor
    pool = mp.Pool(processes = n_processor)
    for i in range(len(T1_subsets)):
        result = pool.apply_async(HB_join, [T1_subsets[i], T2_subsets[i]])
        output = result.get()
        results.append(output)

    ### END CODE HERE ###
    return results
```

[18]:
```
n_processor = 3
DPBP_join(R, S, n_processor)
```

```
{2: {('CompSc', 2)}, 7: {('Engineering', 7)}, 8: {('Arts', 8)}}
{1: {('Geology', 10)}, 2: {('Health', 11)}, 3: {('Dance', 12)}, 6: {('Business',
15)}, 9: {('IT', 18)}}
{3: {('Finance', 21)}}
```

[18]:
```
[('Irene', 14),
 [('Joanna', 2, 'CompSc'), ('Adele', 8, 'Arts')],
 [('Ed', 11, 'Health')],
 []]
```

1.3.4 2.4 Duplication Outer Join Algorithm (DOJA)

In this task, you will implement a **Duplication Outer Join Algorithm (DOJA)**. This algorithm consist of four main steps. 1. (Replication): Duplicate the small table 2. (Local Inner Join): Perform inner join in each processor 3. (Distribution): Reshuffle the inner join result based on R.x 4. (Local Outer Join): Perform outer join between the initial table R and the inner join result

Assume **n_processor=3** and the dataset is initially partition using the **hash** method. For joining technique, please use the provided **outer join** function for the inner join and outer join operation. **Complete the code block between `### START CODE HERE ###` and `### END CODE HERE ###`.**

[19]:
```
# R consists of 15 pairs, each comprising two attributes (nominal and numeric)
R = [('Adele',8),('Bob',22),('Clement',16),('Dave',23),('Ed',11),
     ('Fung',25),('Goel',3),('Harry',17),('Irene',14),('Joanna',2),
     ('Kelly',6),('Lim',20),('Meng',1),('Noor',5),('Omar',19)]
```

```
# S consists of 8 pairs, each comprising two attributes (nominal and numeric)
S = [('Arts',8),('Business',15),('CompSc',2),('Dance',12),('Engineering',7),
     ('Finance',21),('Geology',10),('Health',11),('IT',18)]
```

[20]:
```
import numpy as np

def H(r):
    """
    We define a hash function 'H' that is used in the hashing process works
    by summing the first and second digits of the hashed attribute, which
    in this case is the join attribute.

    Arguments:
    r -- a record where hashing will be applied on its join attribute

    Return:
    result -- the hash index of the record r
    """

    # Convert the value of the join attribute into the digits
    digits = [int(d) for d in str(r[1])]

    # Calulate the sum of elemenets in the digits
    return sum(digits)

def outer_join(L, R, join="left"):
    """outer join using Hash-based join algorithm"""

    if join == "right":
        L, R = R, L

    # inner join
    if join == "inner":
        h_dic = {}
        for r in R:
            h_r = H(r)
            if h_r in h_dic.keys():
                h_dic[h_r].add(r)
            else:
                h_dic[h_r] = {r}

        result = []
        for l in L:
            h_l = H(l)
            if h_l in h_dic.keys():
                for item in h_dic[h_l]:
                    if item[1] == l[1]:
                        result.append([l[0], item[1], item[0]])
        return result

    elif join in ["left", "right"]:
        h_dic = {}
        for r in R:
            print(r)
            h_r = H(r)
            if h_r in h_dic.keys():
                h_dic[h_r].add(r)
            else:
                h_dic[h_r] = {r}
        print(h_dic)

        result = []
        for l in L:
            isFound = False # to check whether there is a match found.
            h_l = H(l)

            if h_l in h_dic.keys():
                for item in h_dic[h_l]:
                    if item[1] == l[1]:
                        result.append([l[0], item[1], item[2]])
                        isFound = True
                        break

            if not isFound:
                result.append([l[0], l[1],str(np.nan)])
        return result


    else:
        raise AttributeError('join should be in {left, right, inner}.')
```

[21]:
```
def hash_distribution(T, n):
    """distribute data using hash partitioning"""

    # Define a simple hash function for demonstration
    def s_hash(x, n):
        h = x%n
        return h

    result = {}
    for t in T:
        h_key = s_hash(t[1], n)
        if h_key in result.keys():
            result[h_key].add(tuple(t))

        else:
            result[h_key] = {tuple(t)}

    return result
```

[22]:
```
import multiprocessing as mp

def DOJA(L, R, n):
    """left outer join using DOJA"""

    ### Start CODE

    output = []

    # Compare to see which is the shorter and longer list between R and S
    # Temporary place inputted list then may change later
    longer_list = R # S list
    shorter_list = L  # R list

    if len(L) < len(R):
        longer_list = R
        shorter_list = L
        print("List L is shorter than List R")
    elif len(L) > len(R):
        longer_list = L
        shorter_list = R
        print("List R is shorter than List L")
    else:
        print("Both lists are equal length") # Can randomly assign either of␣
 ↪list to be the 'shorter' side

    # Distribute longer list to n number of processors using hash distribution
    left_distribution = hash_distribution(longer_list, n)
    print("left distribution", left_distribution)

    # Create a new distribution with the shorter list added to each processor␣
 ↪like in a form of [duplicated short list][left distribution]
    duplicated_list = []
    for dup in range(n):
        duplicated_list.append(shorter_list)

    # INNER JOIN
    pool = mp.Pool(n)
    results = []
    for i in left_distribution.keys():
        result = pool.apply_async(outer_join, [left_distribution[i],␣
↪duplicated_list[i], "inner"])
        # use async cus you want them to perform outer join at the same time
        results.append(result)

    # REDISTRIBUTE
    inner_peace = []
    for x in results:
        inner_peace.extend(x.get())

    # Reshuffle the inner join result
    resuffled = hash_distribution(inner_peace, n)
    print("Resuffled: ", resuffled)

    # Get the key
    the_key = 0
    for key in resuffled:
        the_key = key

    # local outer join and combine
    outputed = []
    for i in left_distribution.keys():
        outputs = pool.apply_async(outer_join, [left_distribution[i],␣
↪resuffled[the_key], "left"])
        print("left_distribution", left_distribution)
        print("resuffled", resuffled)
        # use async cus you want them to perform outer join at the same time in␣
↪parallel
        outputed.append(outputs)

    for y in outputed:
        output.extend(y.get())

    ### End CODE

    return output
```

[23]:
```
DOJA(R,S,3)
```

```
List R is shorter than List L
left distribution {2: {('Dave', 23), ('Adele', 8), ('Ed', 11), ('Noor', 5),
('Harry', 17), ('Joanna', 2), ('Irene', 14), ('Lim', 20)}, 1: {('Omar', 19),
('Fung', 25), ('Meng', 1), ('Clement', 16), ('Bob', 22)}, 0: {('Kelly', 6),
('Goel', 3)}}
```

```
('Adele', 8, 'Arts')('Adele', 8, 'Arts')

('Joanna', 2, 'CompSc')('Joanna', 2, 'CompSc')

('Ed', 11, 'Health')('Ed', 11, 'Health')

{8: {('Adele', 8, 'Arts')}, 2: {('Joanna', 2, 'CompSc'), ('Ed', 11,
'Health')}}{8: {('Adele', 8, 'Arts')}, 2: {('Joanna', 2, 'CompSc'), ('Ed', 11,
'Health')}}

('Adele', 8, 'Arts')
('Joanna', 2, 'CompSc')
('Ed', 11, 'Health')
{8: {('Adele', 8, 'Arts')}, 2: {('Joanna', 2, 'CompSc'), ('Ed', 11, 'Health')}}
Resuffled:  {2: {('Adele', 8, 'Arts'), ('Joanna', 2, 'CompSc'), ('Ed', 11,
'Health')}}
left_distribution {2: {('Dave', 23), ('Adele', 8), ('Ed', 11), ('Noor', 5),
('Harry', 17), ('Joanna', 2), ('Irene', 14), ('Lim', 20)}, 1: {('Omar', 19),
('Fung', 25), ('Meng', 1), ('Clement', 16), ('Bob', 22)}, 0: {('Kelly', 6),
('Goel', 3)}}
resuffled {2: {('Adele', 8, 'Arts'), ('Joanna', 2, 'CompSc'), ('Ed', 11,
'Health')}}
left_distribution {2: {('Dave', 23), ('Adele', 8), ('Ed', 11), ('Noor', 5),
('Harry', 17), ('Joanna', 2), ('Irene', 14), ('Lim', 20)}, 1: {('Omar', 19),
('Fung', 25), ('Meng', 1), ('Clement', 16), ('Bob', 22)}, 0: {('Kelly', 6),
('Goel', 3)}}
resuffled {2: {('Adele', 8, 'Arts'), ('Joanna', 2, 'CompSc'), ('Ed', 11,
'Health')}}
left_distribution {2: {('Dave', 23), ('Adele', 8), ('Ed', 11), ('Noor', 5),
('Harry', 17), ('Joanna', 2), ('Irene', 14), ('Lim', 20)}, 1: {('Omar', 19),
('Fung', 25), ('Meng', 1), ('Clement', 16), ('Bob', 22)}, 0: {('Kelly', 6),
('Goel', 3)}}
resuffled {2: {('Adele', 8, 'Arts'), ('Joanna', 2, 'CompSc'), ('Ed', 11,
'Health')}}
```

```
[23]: [['Dave', 23, 'nan'],
       ['Adele', 8, 'Arts'],
       ['Ed', 11, 'Health'],
       ['Noor', 5, 'nan'],
       ['Harry', 17, 'nan'],
       ['Joanna', 2, 'CompSc'],
       ['Irene', 14, 'nan'],
       ['Lim', 20, 'nan'],
       ['Omar', 19, 'nan'],
       ['Fung', 25, 'nan'],
       ['Meng', 1, 'nan'],
       ['Clement', 16, 'nan'],
       ['Bob', 22, 'nan'],
       ['Kelly', 6, 'nan'],
       ['Goel', 3, 'nan']]
```

1.3.5 2.5 Parallel Join Variants

Briefly answer the following question.

1. For each partitioning algorithm besides range-partitioning, state and justify whether we can use it with the code for disjoint-partitioning based parallel join above.

Your answer: Yes. Other partitioning algorithm like hash-partitioning and round-robin partitioning can be used with disjoint-partitioning based parallel joins. There is another partitioning algorithm which is the random-unequal data partitionining method but this method should not be used with disjoint-partitioning based parallel joins.

For hash partitioning method, it distributes the tuples by implementing a hash function on the join key which is appropriate for parallel joins as it evenly spreads data among partitions, reducing the possibility of skewing. Disjoint-partitioning method can be used as long as the hash functions used in both hash partitioning and joining phases are compatible.

For round-robin partitiioning method, it evenly distributes tuples among partitions by distributing them in a sequential manner whereby there is an equal spread of tuples among partitions and which is good for balancing workloads in parallel joins. It can work with disjoint-partitioning method because it is not dependent on the data value, which reduced the possibility of skewing.

For random-unequal data partitionining method, it does not evenly distributes tuples among partitions into disjoint steps which is the main requirement for disjoint-partitioning based parallel joins. In addition to that, if random-unequal data partitionining method is used, there would be a high possibility of skewing as the data are randomly partitioned and the partitions are most likely unequal partitions.

1.4 3. Parallel Sorting Algorithm (2 marks)

In this task, you will implement **parallel binary-merge sort** method. It has two phases same as the parallel merge-all sort that you learnt in the labs. 1. Local sort 2. Final merge.

The first phase is similar to the parallel merge-all sort. The second phase, the merging phase, is pipelined instead of concentrating on one processor. In this phase, we take the results from two processors and then merge the two in one processor, called binary merging. The result of the merge between two processors is passed on to the next level until one processor (the host) is left.

Complete the code block between `### START CODE HERE ###` and `### END CODE HERE ###`. Assume that we use the round robin partitioning method (i.e. `rr_partition()`).

1.4.1 3.1 Round-robin Partition

[24]:
```python
# Round-robin data partitionining function
def rr_partition(data, n):
    """
    Perform data partitioning on data
    Arguments:
    data -- an input dataset which is a list
    n -- the number of processors

    Return:
    result -- the paritioned subsets of D
    """
    result = []
    for i in range(n):
        result.append([])

    ### START CODE HERE ###

    # For each bin,
    for name, number in enumerate(data):
        # Calculate the index of the bin that the current data point will be␣
↪assigned
        index_bin = (int) (name % n)
        result[index_bin].append(number)

    ### END CODE HERE ###

    return result
```

[25]:
```
# Test the round-robin partitioning function
result = rr_partition(R, 3)
pprint(result)
```

```
[[('Adele', 8), ('Dave', 23), ('Goel', 3), ('Joanna', 2), ('Meng', 1)],
 [('Bob', 22), ('Ed', 11), ('Harry', 17), ('Kelly', 6), ('Noor', 5)],
 [('Clement', 16), ('Fung', 25), ('Irene', 14), ('Lim', 20), ('Omar', 19)]]
```

1.4.2 3.2 Serial Sort

[26]:
```
def qsort(arr):
    """
    Quicksort a list

    Arguments:
    arr -- the input list to be sorted

    Return:
    result -- the sorted arr
    """
    if len(arr) <= 1:
        return arr
    else:
        # take the first element as the pivot
        pivot = arr[0]
        left_arr = [x for x in arr[1:] if x[1] < pivot[1]]
        right_arr = [x for x in arr[1:] if x[1] >= pivot[1]]
        # uncomment this to see what to print
        # print("Left:" + str(left_arr)+" Pivot : "+ str(pivot)+" Right: " +␣
↪str(right_arr))
        sorted_arr = qsort(left_arr) + [pivot] + qsort(right_arr)

        return sorted_arr
```

[27]:
```
qsort(R)
```

[27]:
```
[('Meng', 1),
 ('Joanna', 2),
 ('Goel', 3),
 ('Noor', 5),
 ('Kelly', 6),
 ('Adele', 8),
 ('Ed', 11),
 ('Irene', 14),
 ('Clement', 16),
 ('Harry', 17),
 ('Omar', 19),
 ('Lim', 20),
 ('Bob', 22),
 ('Dave', 23),
 ('Fung', 25)]
```

[28]:
```
# Let's first look at 'k-way merging algorithm' that will be used
# to merge sub-record sets in our external sorting algorithm.
import sys

# Find the smallest record
def find_min(records):
    """
    Find the smallest record

    Arguments:
    records -- the input record set

    Return:
    result -- the smallest record's index
    """
    m = records[0]
    index = 0
    for i in range(len(records)):
        if(records[i][1] < m[1]):
            index = i
            m = records[i]
    return index

def k_way_merge(record_sets):
    """
    K-way merging algorithm

    Arguments:
    record_sets -- the set of multiple sorted sub-record sets

    Return:
    result -- the sorted and merged record set
    """
    # indexes will keep the indexes of sorted records in the given buffers
    indexes = []
    for _ in record_sets:
        indexes.append(0) # initialisation with 0

    # final result will be stored in this variable
    result = []
    while(True):
        merged_result = [] # the merging unit (i.e. # of the given buffers)

        # This loop gets the current position of every buffer
        for i in range(len(record_sets)):
            if(indexes[i] >= len(record_sets[i])):
                merged_result.append((None, sys.maxsize))
            else:
                merged_result.append(record_sets[i][indexes[i]])

        # find the smallest record
        smallest = find_min(merged_result)

        # if we only have sys.maxsize on the tuple, we reached the end of every␣
 ↪record set
        if(merged_result[smallest][1] == sys.maxsize):
            break

        # This record is the next on the merged list
        result.append(record_sets[smallest][indexes[smallest]])
        indexes[smallest] +=1

    return result
```

[29]:
```
# Test k-way merging method
buffers = rr_partition(R, 3)
print(buffers)
result = k_way_merge([qsort(b) for b in buffers])
pprint(result)
```

```
[[('Adele', 8), ('Dave', 23), ('Goel', 3), ('Joanna', 2), ('Meng', 1)], [('Bob',
22), ('Ed', 11), ('Harry', 17), ('Kelly', 6), ('Noor', 5)], [('Clement', 16),
('Fung', 25), ('Irene', 14), ('Lim', 20), ('Omar', 19)]]
[('Meng', 1),
 ('Joanna', 2),
 ('Goel', 3),
 ('Noor', 5),
 ('Kelly', 6),
 ('Adele', 8),
 ('Ed', 11),
 ('Irene', 14),
 ('Clement', 16),
 ('Harry', 17),
 ('Omar', 19),
 ('Lim', 20),
 ('Bob', 22),
 ('Dave', 23),
 ('Fung', 25)]
```

[30]:
```
def serial_sorting(dataset, buffer_size):
    """
    Perform a serial external sorting method based on sort-merge
    The buffer size determines the size of each sub-record set

    Arguments:
    dataset -- the entire record set to be sorted
    buffer_size -- the buffer size determining the size of each sub-record set

    Return:
    result -- the sorted record set
    """

    if (buffer_size <= 2):
        print("Error: buffer size should be greater than 2")
        return

    result = []

    ### START CODE HERE ###

    # Length of dataset
    n = len(dataset)

    # --- Sort Phase ---
    # Sort and divide the dataset into subsets
    start_position = 0
    sorted_set = []
    while True:
        if (n - start_position) > buffer_size:
            subset = dataset[start_position: start_position + buffer_size]
            sorted_subset = qsort(subset)
            sorted_set.append(sorted_subset)
            start_position += buffer_size
        else:
            subset = dataset[start_position:]
            sorted_subset = qsort(subset)
            sorted_set.append(sorted_subset)
            break

    # --- Merge Phase ---
    merge_buffer_size = buffer_size - 1
    while True:
        merged_set = []
        n = len(sorted_set)
        start_position = 0
        while True:
            if (n - start_position) > merge_buffer_size:
                subset = sorted_set[start_position: start_position +␣
↪merge_buffer_size]
                merged_set.append(k_way_merge(subset))
                start_position += merge_buffer_size
            else:
                subset = sorted_set[start_position:]
                merged_set.append(k_way_merge(subset))
                break

        sorted_set = merged_set
        if len(sorted_set) <= 1:
            result = merged_set
            break

    ### END CODE HERE ###

    return result
```

[31]:
```
print(R)
```

```
[('Adele', 8), ('Bob', 22), ('Clement', 16), ('Dave', 23), ('Ed', 11), ('Fung',
25), ('Goel', 3), ('Harry', 17), ('Irene', 14), ('Joanna', 2), ('Kelly', 6),
('Lim', 20), ('Meng', 1), ('Noor', 5), ('Omar', 19)]
```

[32]:
```
result = serial_sorting(R, 4)
print("final sorting result:" + str(result))
```

```
final sorting result:[[('Meng', 1), ('Joanna', 2), ('Goel', 3), ('Noor', 5),
('Kelly', 6), ('Adele', 8), ('Ed', 11), ('Irene', 14), ('Clement', 16),
('Harry', 17), ('Omar', 19), ('Lim', 20), ('Bob', 22), ('Dave', 23), ('Fung',
25)]]
```

1.4.3 3.3 Parallel Binary-merge Sort

[33]:
```
# Include this package for parallel processing
import multiprocessing as mp

def parallel_binary_merge_sorting(dataset, n_processor, buffer_size):
    """
    Perform a parallel binary-merge sorting method

    Arguments:
    dataset -- entire record set to be sorted
    n_processor -- number of parallel processors
    buffer_size -- buffer size determining the size of each sub-record set

    Return:
    result -- the merged record set
    """

    if (buffer_size <= 2):
        print("Error: buffer size should be greater than 2")
        return

    result = []

    ### START CODE HERE ###

    # Call rr_partition to do round-robin partitioning
    round_robin = rr_partition(dataset, n_processor)

    # pool for parallel processing
    pool = mp.Pool(processes = n_processor)

    # Sort phase
    sorted_dataset = []
    for elem in round_robin:
        # call serial sorting method
        print("elem: ", elem)
        sorted_dataset.append(*pool.apply_async(serial_sorting, [elem,␣
↪buffer_size]).get())
        print("sorted_dataset: ", sorted_dataset)
    pool.close()

    # Merge phase
    # For loop for binary merge
    while len(sorted_dataset) > 1:
        updated = []
        for i in range(0, len(sorted_dataset), 2):
            if i + 1 < len(sorted_dataset):
                merged = k_way_merge([sorted_dataset[i], sorted_dataset[i + 1]])
                updated.append(merged)
            else:
                # If there's an odd number of sorted datasets, just append the␣
↪last one
                updated.append(sorted_dataset[i])
        sorted_dataset = updated

    # The final merged result
    result = sorted_dataset[0]

    ### END CODE HERE ###

    return result
```

[34]:
```
result = parallel_binary_merge_sorting(R, 10, 20)
print("final result:" + str(result))
```

```
elem:  [('Adele', 8), ('Kelly', 6)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)]]
elem:  [('Bob', 22), ('Lim', 20)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)]]
elem:  [('Clement', 16), ('Meng', 1)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)]]
elem:  [('Dave', 23), ('Noor', 5)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)], [('Noor', 5), ('Dave', 23)]]
elem:  [('Ed', 11), ('Omar', 19)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)], [('Noor', 5), ('Dave', 23)], [('Ed', 11),
('Omar', 19)]]
elem:  [('Fung', 25)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)], [('Noor', 5), ('Dave', 23)], [('Ed', 11),
('Omar', 19)], [('Fung', 25)]]
```

```
elem:  [('Goel', 3)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)], [('Noor', 5), ('Dave', 23)], [('Ed', 11),
('Omar', 19)], [('Fung', 25)], [('Goel', 3)]]
elem:  [('Harry', 17)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)], [('Noor', 5), ('Dave', 23)], [('Ed', 11),
('Omar', 19)], [('Fung', 25)], [('Goel', 3)], [('Harry', 17)]]
elem:  [('Irene', 14)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)], [('Noor', 5), ('Dave', 23)], [('Ed', 11),
('Omar', 19)], [('Fung', 25)], [('Goel', 3)], [('Harry', 17)], [('Irene', 14)]]
elem:  [('Joanna', 2)]
sorted_dataset:  [[('Kelly', 6), ('Adele', 8)], [('Lim', 20), ('Bob', 22)],
[('Meng', 1), ('Clement', 16)], [('Noor', 5), ('Dave', 23)], [('Ed', 11),
('Omar', 19)], [('Fung', 25)], [('Goel', 3)], [('Harry', 17)], [('Irene', 14)],
[('Joanna', 2)]]
final result:[('Meng', 1), ('Joanna', 2), ('Goel', 3), ('Noor', 5), ('Kelly',
6), ('Adele', 8), ('Ed', 11), ('Irene', 14), ('Clement', 16), ('Harry', 17),
('Omar', 19), ('Lim', 20), ('Bob', 22), ('Dave', 23), ('Fung', 25)]
```

1.4.4 3.4 Parallel Binary-merge Behavior

Briefly answer the following.

1. Does parallel binary-merge utilize all available processors? State and justify with respect to each phase: sort, merge.

Your answer: Yes, parallel binary-merge do utilize all available processors whereby all the available processors can be used to sort different data subsets in parallel during the sorting phase and during the merge phase, the binary-merge algorithm merges two sorted lists at a time but when merging, the number of lists to merge reduces so less and less processors will be used so in conclusion, although all processors can be utilized at the sorting and merging phase, the merging phase may not fully use all the processors as it goes on as the workload has decreased as it goes on.

1.5 4 Parallel GroupBy (2 marks)

1.5.1 4.1 Parallel GroupBy with Merge-All

In this task, you will implement **Parallel GroupBy with Merge-All** method. It invloves the following two steps: 1. Local aggregate in each processor 2. Global aggregation

Assume **n_processor=4** and the dataset has already been pre-partitioned. For local aggregation, please use the provided **local_groupby** function. **Complete the code block between `### START CODE HERE ###` and `### END CODE HERE ###`.**

[35]:
```
D1 = [('A', 1), ('B', 2), ('C', 3), ('A', 10), ('B', 20), ('C', 30)]
D2 = [('A', 2), ('B', 3), ('C', 4), ('A', 20), ('B', 30), ('C', 40)]
D3 = [('A', 3), ('B', 4), ('C', 5), ('A', 30), ('B', 40), ('C', 50)]
D4 = [('A', 4), ('B', 5), ('C', 6), ('A', 40), ('B', 50), ('C', 60)]
```

[36]:
```
# The first step in the merge-all groupby method
def local_groupby(dataset):
    """
    Perform a local groupby method

    Arguments:
    dataset -- entire record set to be merged

    Return:
    result -- the aggregated record set according to the group_by attribute␣
 ↪index
    """

    dict = {}
    for index, record in enumerate(dataset):
        key = record[0]
        val = record[1]
        if key not in dict:
            dict[key] = 0
        dict[key] += val
    return dict
```

[37]:
```
result = local_groupby (D1)
print(result)
```

```
{'A': 11, 'B': 22, 'C': 33}
```

[38]:
```
import multiprocessing as mp

def parallel_merge_all_groupby(dataset):
    """
    Perform a parallel merge_all groupby method

    Arguments:
    dataset -- entire record set to be merged

    Return:
    result -- the aggregated record dictionary according to the group_by␣
 ↪attribute index
    """

    result = {}

    ### START CODE HERE ###

    # Define the number of parallel processors: the number of sub-datasets.
    n_processor = len(dataset)
    pool = mp.Pool(processes=n_processor)

    # ----- Local aggregation step -----
    local_result = []
    for s in dataset:
        # call the local aggregation method
        local_result.append(pool.apply(local_groupby, [s]))
    pool.close()

    # ---- Global aggregation step ----
    # Assume that the global operator is sum
    for r in local_result:
        for key, val in r.items():
            if key not in result:
                result[key] = 0
            result[key] += val

    ### END CODE HERE ###

    return result
```

[39]:
```
E = [D1, D2, D3, D4]
result = parallel_merge_all_groupby (E)
print(result)
```

```
{'A': 110, 'B': 154, 'C': 198}
```

1.5.2 4.2 Redistribution Method

In this task, you will implement **Parallel GroupBy with Redistribution Method** method. It invloves the following two steps: 1. (Partitioning phase): Redistribute raw records to all processor 2. (Aggregation phase): Each processor performs a local aggregation

Assume **n_processor=4** and the dataset should be partitioned into 4 groups using **range partition** Where first group with A & B, second with C & D, third with E & F and fourth with G & H. For local aggregation, please use the provided **local_groupby** function. **Complete the code block between `### START CODE HERE ###` and `### END CODE HERE ###`.**

[40]:
```
D = [('A', 1), ('B', 2), ('C', 3), ('D', 10), ('E', 20), ('F', 30), ('G', 20),␣
 ↪('H', 30),
     ('A', 2), ('B', 3), ('C', 4), ('D', 20), ('E', 30), ('F', 40), ('G', 30),␣
 ↪('H', 40),
     ('A', 3), ('B', 4), ('C', 5), ('D', 30), ('E', 40), ('F', 50), ('G', 40),␣
 ↪('H', 50),
     ('A', 4), ('B', 5), ('C', 6), ('D', 40), ('E', 50), ('F', 60), ('G', 50),␣
 ↪('H', 60)]
```

Create range partition function based on GroupBy attribute

[41]:
```
# Range data partitionining function (Need to modify as instructed above)
def range_partition(data, range_indices):
    """
    Perform range data partitioning on data based on the join attribute

    Arguments:
    data -- an input dataset (list of tuple)
    range_indices -- the range end-points (e.g., ['C', 'E', 'G'] means
                        4 intervals: x < 'C', 'C' <= x < 'E', 'E' <= x < 'G',␣
 ↪'G' <= x)

    Return:
    result -- the partitioned subsets of D
    """

    result = []

    ### START CODE HERE ###

    # First, we sort the dataset according to their values
    new_data = sorted(data, key=lambda x: x[1])

    # Calculate the number of bins
    n_bin = len(range_indices) + 1

    # For each bin, perform the following
    for i in range(n_bin):
        # Find elements belonging to each range
        if i == 0:
            s = [x for x in new_data if x[0] in ('A', 'B')]
        elif i == n_bin - 1:
            s = [x for x in new_data if x[0] in ('G', 'H')]
        else:
            s = [x for x in new_data if range_indices[i-1] <= x[0] <␣
 ↪range_indices[i]]

        # Add the partitioned list to the result
        result.append(s)

    ### END CODE HERE ###

    return result
```

[42]:
```
import multiprocessing as mp

def parallel_redistributed_groupby(dataset, range_indices):
    """
    Perform a parallel redistributed groupby method

    Arguments:
    dataset -- entire record set to be merged

    Return:
    result -- the aggregated record dictionary according to the group_by␣
↪attribute index
    """

    result = []

    ### START CODE HERE ###

    # call range partition
    ranged = range_partition(dataset, range_indices)

    # Define the number of parallel processors: the number of sub-datasets.
    n_processor = len(dataset)

    pool = mp.Pool(processes=n_processor)

    # ----- Local aggregation step -----
    result = []
    for each in ranged:
        # call the local aggregation method
        result.append(pool.apply(local_groupby, [each]))
    pool.close()

    ### END CODE HERE ###

    return result
```

[43]:
```
result = parallel_redistributed_groupby (D, ['C', 'E', 'G'])
print(result)
```

```
[{'A': 10, 'B': 14}, {'C': 18, 'D': 100}, {'E': 140, 'F': 180}, {'G': 140, 'H':
180}]
```

[ ]: