COMP5349 Assignment2

Submitted by

Student ID	Names of Students	Email
440609416	Ramanathan Sivagurunathan	rsiv5112@uni.sydney.edu.au
310250811	Woo Hyun Jung	ajun6567@uni.sydney.edu.au

Contents

Chapter 1

Introduction

This report describes the implementation and compare and contrast the performance of HIVE Query and Spark Program on a given task.

Our given task is to find out how many times a user has visited a particular country, as well as the maximum, minimum, average, and total time he/she spent in this country.

In both the implementation, our goal was to implement a scalable and efficient solution. We also took into account the code readability as well as one of the reason for using hive and spark is ability to understand code easily without much boiler plate codes.

Chapter 2

Dataset

We have been provided with the dataset of places and photos. Some of the characteristics of the data are

- 308,133 Unique Places
- Size of Places file on disk is 33 MB
- 80 Million Photos
- 70,000 Users
- Size of Photos file on disk is 11 G

2.1 Assumptions

We have made some reasonable assumptions about the scalability. They are

1. Photos can grow significantly large. In range of Billion Photos.
2. Users can grow significantly large as well, in the range of Billion Users. Each could be taking thousands of photos in various countries.

Chapter 3

Hive Query

Query to find the max, min, total, average stay per user per country by using only Hive Query turned out to be inefficient. This is because we need to use triple join, which will be highly inefficient for big data queries.

To avoid this we have used Hive Query and a map reduce job to achieve our objective

3.1 Design

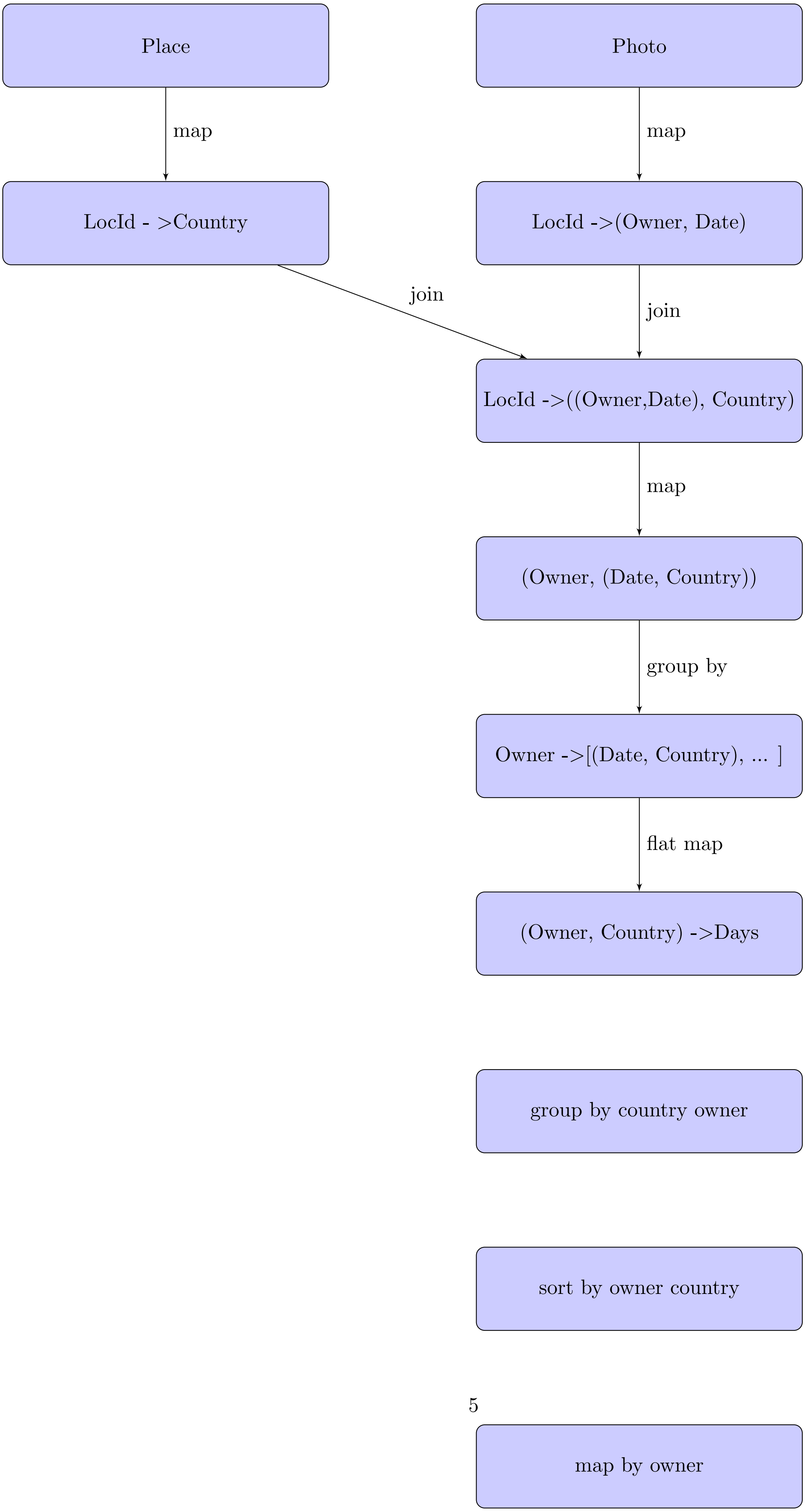
So we split the task into three parts

Part1: Join Place Table and Photos Table to produce table containing $Owner, DateTaken, Country$.
We have use Hive Query for this job because writing query for this task is simple as a SQL Query. This makes code readable for everyone.

Part2: Map Job to return the content as it is but distribute by Owner and Sort it by date. For distributing and sorting we have leveraged Map reduce framework.

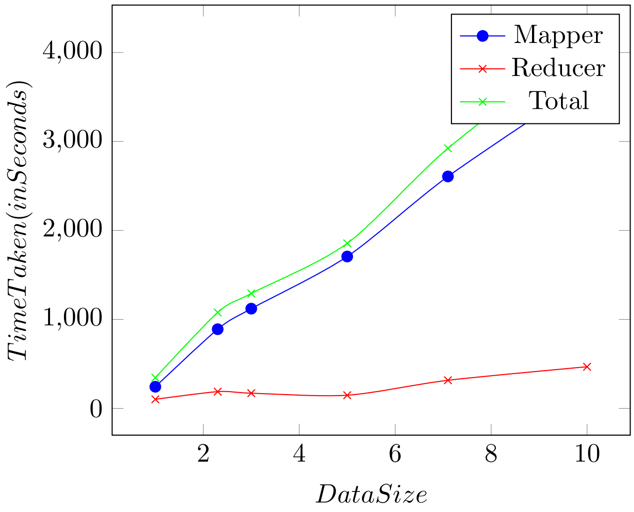
Part3: Reduce Job to Find the Max, Min, Average, Total stay per User Per Country. We used python script to do the reduce as this is simple to sequentially go through the time sorted list and find out the time a user stayed in each country.

3.2 Lineage



Place
Photo
map
map
LocId - >Country
LocId ->(Owner, Date)
join
join
LocId ->((Owner,Date), Country)
map
(Owner, (Date, Country))
group by
Owner ->[(Date, Country), ...]
flat map
(Owner, Country) ->Days
group by country owner
sort by owner country
map by owner

3.3 Performance



Chapter 4

Spark Program

4.1 Design

We have created 2 Jobs for solving the problem chained one after another

Job1: Unique Users Per Locality

Job2: Top 10 Location For each country

Design Choice 1: Use of Distributed Cache
Since the current number of places has about 300,000 Entries and the size is about 33 MB, we have decided to use it as a distributed cache rather than running a Mapreduce job for joining the places and photos. If each Hash entry takes about 128 Bytes (Hash, String, Value) then for a million entry it would be 128 MB. Given the RAM capacity of modern machine, this approach can scale easily upto 8 Million Entries for 1G of RAM and even more.
Design Choice 2: Use of Composite Keys
We have used composite keys for letting framework to sort and group so that we can find the unique users efficiently without the use of any hashmap which will make the solution not scalable.

Job1

This jobs outputs the unique users per locality. Each locality includes all the neighbourhood as well. Since the place.txt is small, i have used it as a distributed cache rather than adding another Map Reduce job.

Mapper task takes the distributed cache and creates the photos and outputs a Composite key (Place, PlaceType, User) and value as (user). If the place type is Neighbourhood, there will be two outputs. One for the neighbourhood and other for the Locality.

I have used user in key as well as value because if i sort the key based on entire key and group based on Place, i will get a sorted list of values (user). So it will be easier to find the unique as same users will be successive. So i have just traverse through the list and taking the first value. If the next is equal to previous i will skip that.

Combiner task is takes the key: (Place, PlaceType, User) Value: (users) and outputs only one (User). This avoids lot of bytes being shuffled across.

Partition is based on the Place. As we need to find the unique users per locality. This partition will allow us to find that.

Sorting is based on the entire key. (Place, PlaceType, User). So that the same users of the place will be successive in the resulting list.

Grouping is based on the place. Now we will have all the Users of the place available to reducer as sorted list.

Reducer task takes the (Place, PlaceType, User) as a key and list of sorted users as value. and returns (Place, PlaceType) as output key and count of unique users as Value.

Segments	Input	Output	Comments
Mapper	Cache: Place.txt Photos	Key: (Place, PlaceType, User) Value: (user)	**Setup:** 1. create Locality HashMap(locId, Placename) 2. create Nbrhood HashMap(nbrid, NbrhoodName) 2. For each line in Place.txt Locality HashMap[locId] = Placename Nbrhood HashMap[nbrid] = NbrhoodName **Map:** 1. For each Line if PlaceType = Locality then Output((PlaceName, 7, user), user) else if PlaceType = Nbrhood then Output((PlaceName, 7, user), user) Output((NbrhoodName, 22, user), user)
Combiner	Key: (Place, PlaceType, User) Value: (user)	Key: (Place, PlaceType, User) Value: (user)	for each key, list(value) Output(key, value)
Partitioner	Partition based on the Place		
Sorter	Sort based on Place, PlaceType and last by User.		Now we have all the same users for the place consecutively. This will help us to find unique users easily.
Grouper	Group based on Place.		This will get all the users in the iterator, sorted. So we can count unique users easily
Reducer	Key: (Place Name , Place Type, User) Value: (users)	Key: (Place Name, Place Type) Value: (unique User Count)	for each key, list(Value) Count the unique users in the list Output((PlaceName, Place Type) , Count)

Job2

This jobs outputs the top 10 Locality for each country and for each locality a top Neighbourhood based on number of unique users. Each locality includes all the neighbourhood as well. Since the place.txt is small, i have used it as a distributed cache rather than adding another Map Reduce job.

Mapper task takes the output of previous job with key (Place, PlaceType) and Value: (count) and returns key: (Country, Locality, Count, Name) and Value as Text with format ”PlaceType:Count:Place”. Locality is in output for Neighbourhood but for actual locality the output will be ””.

For Example. For Locality Paris the output will be (France, ””, 1000) (”7:1000:Paris”) For Neighbourhood in Paris, the output will be (France, Paris, 100) (”22:100:Nbr1”)

Partition is based on the Country. As we need to find the Top 10 Locality per country. This partition will allow us to find that.

Sorting is based on the Country, Locality and reverse (count).

Grouping is based on the Country. Now we will have all the Descending ordered Locality for each country, followed by Neighbourhood.

Reducer task takes the key (Country, Locality, Count, Name) and the string with format ”PlaceType:Count:Place”. Takes only first 10 entries for Locality. And For each Locality it takes the top Neighbourhood.

Segments	Input	Output	Comments
Mapper	Text: placename PlaceType Count	Key: (Country, Locality, Count, Name) Value: Text of Format ”PlaceType:Count:Place”	
Partition	Partition based on Country		
Sorting	Sort Country, Locality, count (descending order).		
Grouping	Grouping is based on Country		
Reducer	Key: (Country, Locality, Count, Name) Value: ”PlaceType:Count:Place”	CountryName (placeName: NumOfUsers, neighbourhoodName: NumOfUsers)+	Gives out Top 50 Places with the number of photos count

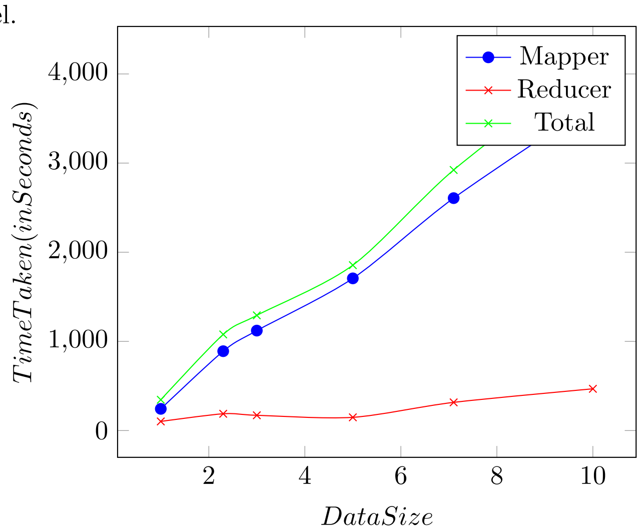
4.2 Performance

For a highly scalable Map Reduce Job we need to make sure the following parameters are minimized

1. Time Taken for a Map Task
2. Time Taken for Shuffling
3. Time Taken for a Reduce Task

Higher the number of shuffle bytes, more the time for reduce task. So knowing where the time is more gives us better idea to optimize the solutions. Below is our performance graph for our implementation. On X-Axis is the Input Size and on Y-Axis is Total Time taken by all mapper or Reducer. We have plotted both Mapper, Reducer, Mapper + Reducer. Due to our design the amount of work done in reducer is very minimal.

The overall running time is less than 3 Minutes, given we have enough Mapper task to run in parallel.



4.3 Alternate Design

It is possible to design a solution for the given problem with just one Map Reduce Job. Implementation for the design is also provided.

Since the number of places is small, we are using the places in the distributed cache.

Segments	Input	Output	Comments
Mapper	Text: Photos	Key: (Country, Locality, Neighbourhood, User) Value: 1	For each Neighbourhood there are two output one for (Country, Locality, "", User) and (Country, Locality, Nbrhood, User)
Combiner	Key: (Country, Locality, Neighbourhood, User) Value: 1	Key: (Country, Locality, Neighbourhood, User) Value: 1	For every key we just output only one entry. This is for counting unique users.
Partition	Partition based on Country		
Reducer	Key: (Country, Locality, Count, Name) Value: 1	CountryName (̂placeName: NumOfUsers, neighbourhoodName: NumOfUsers)+	We have a hash map of all the places per country. So for every entry we use hashmap to count and at the end of the Reducer (In Cleanup Function) we output the desired output taken from the hashmap.

4.3.1 Pros and Cons

Pros:

1. Number of Map Reduce Job is reduced
2. Better Performance if we have small sized data.

Cons:

1. Not scalable as few countries will have more places and more photos will be there. So a single reducer will be highly overloaded causing the delay

References

[1] Hadoop Map Reduce Framework http://hadoop.apache.org