MapReduce project

Introduction

In this assignment, you will implement a basic MapReduce infrastructure.

- http://research.google.com/archive/mapreduce.html

Your implementation can make several simplifying assumptions:

- All processes in the cluster are started using a central resource manager (you), and they all know the location of the master at startup time. The master's network address is passed in as a command-line parameter.
- You may assume that all data is string data. Keys are strings, and data items are strings, both for the input and for the intermediate/final data sets.
- You can support a single file format, namely SQLite database files. Given an input database file, the set of keys fed to a Map worker will be the location of the database file plus a range of keys (e.g., from "apple" to "kiwi" for task 1, from "lemon" to "orange" for task 2, from "pineapple" to "watermelon" for task 2). The code supporting the Map task will read them from the database to get the actual list of keys and the value associated with each one.

  The intermediate keys and values should be of the same format, and can be stored in new SQLite files; likewise for the final data.

  You may support other data formats if you wish, but you do not need to.

  Here is a suitable SQLite driver for Go:

  - https://github.com/mattn/go-sqlite3

  See the example program to get an idea of how to use it, and see the standard Go `database/sql` package docs for a reference.
- You do not need to handle worker failures. To simplify your implementation, it is okay for the entire MapReduce task to fail if one of the workers fails.

However, you should still plan for realistic data sets:

- You should not assume that the entire data set will fit in memory. You should process elements as you load them, and write intermediate and final results as you gather them, rather than buffering them all in memory.
- You should not assume that the subset of data assigned to a single Map or Reduce task worker can fit in memory.

Here is a small data file that you can use for testing. It contains the novels of Jane Austen. The key for each record is the name of a file and the offset into the file where the line began, and the value is a single line of text:

- austen.sqlite3

In addition, this Python script will generate input files using the same format:

- loadsource.py

Give it the name of a database to create and the names of one or more input text files, and it will process them into a sqlite3 file for you.

The assignment is in three parts:

Part 1

In this part, you will write support code to make it easier to work with the data files in sqlite3 format. You

should include the following functions in a file called `database.go`.

openDatabase

This is a simple helper function that opens a sqlite file and prepares it for access:

```
func openDatabase(path string) (*sql.DB, error)
```

It should open the database file and turn off journaling by issuing the following two commands to the database:

```
pragma synchronous = off;
pragma journal_mode = off;
```

These commands turn off some important safety features of the database that prevent data loss in the event of a crash or power failure. However, they also come at a significant performance cost. Your MapReduce implementation will have to start from scratch in the event of a crash anyway, so there is no reason to incur the cost involved in making these guarantees at the database level.

If anything goes wrong, this function should return the error. Note that if the database was opened successfully but an error occurred later, `openDatabase` must close the database before returning. When it returns, a non-nil `error` return value indicates a failure (in which case the `*sql.DB` must be nil and there should not be an open database) and a nil `error` return value indicates success (in which case the `*sql.DB` must be non-nil and ready for use).

createDatabase

This function creates a new database and prepares it for use:

```
func createDatabase(path string) (*sql.DB, error)
```

If the database file already exists, delete it before continuing. Otherwise, create it and issue the following commands to initialize it:

```
pragma synchronous = off;
pragma journal_mode = off;
create table pairs (key text, value text);
```

The return value semantics are the same as for `openDatabase`: if the function succeeds, it returns a non-nil `*sql.DB` and a nil `error`. If it fails, it returns a nil `*sql.DB` and a non-nil `error`, and ensures that the database has been closed (if it was opened successfully).

splitDatabase

When the master first starts up, it will need to partition the input data set for the individual mapper tasks to access. This function is responsible for splitting a single sqlite database into a fixed number of smaller databases.

```
func splitDatabase(source, outputPattern string, m int) ([]string, error)
```

It is given the pathname of the input database, and a pattern for creating pathnames for the output files. The pattern is a format string for `fmt.Sprintf` to use, so you might call it using:

```
paths, err := splitDatabase("input.sqlite3", "output-%d.sqlite3", 50)
```

In this instance, it would create output files called

- output-0.sqlite3
- output-1.sqlite3
- output-2.sqlite3
- …
- output-49.sqlite3

`splitDatabase` first opens the input database (using `openDatabase`) and then queries it to find out how many pairs it has to work with. The following query will work:

```
select count(1) from pairs
```

This number must not be smaller than `m` or it should return an error.

There are several ways to partition the database. Here is one method:

- Figure out how many pairs should be in each partition, computed as a `float64` figure.
- Iterate through all rows of the database using a single query:

  ```
  select key, value from pairs
  ```

- As you process each pair, figure out which of the `m` output files it should be in. Use `float64` arithmetic throughout and truncate to an `int` as the last step. Be careful that rounding errors do not push you to create too many output files.
- Any time you calculate that you should be in a new output file from where the previous pair went, close the old output file and create a new one (using `createDatabase`).
- Keep track of the pathnames of all of the files you create, and return that slice of names at the end.

If any error occurs, return the error code. You should ensure that all databases you opened or created are closed when you return, regardless of whether it was a successful return or an erroneous one.

mergeDatabases

There are two places where your MapReduce system will need to merge sqlite files. The first is when a reducer is gathering its intermediate input files and combining them into a single file. The second is when everything is finished and the master gathers all of the reducer outputs and merges them into a single output file.

```
func mergeDatabases(urls []string, path string, temp string) (*sql.DB, error)
```

You will transfer all data between MapReduce nodes using HTTP, so the input to `mergeDatabases` is a list of URLs for the input files. It does the following:

- Create a new output database using `createDatabase` with `path` as the file name.
- For each URL in the `urls` list:
  - Download the file at the given URL and store it in the local file system at the path given by `temp`.
  - Merge the contents of the file into the main output file (see below).
  - Delete the temporary file.

To make this easier, write a couple helper functions:

download

This function takes a URL and a pathname, and stores the file it downloads from the URL at the given location:

```
func download(url, path string) error
```

As always, it should return an error code. It should ensure that anything that needs closing (notably the `Body` of the response object) is closed whether the request succeeds or not.

See the Go documentation for the `net/http` package for examples of how to issue HTTP GET requests.

gatherInto

This function takes an open database (the output file) and the pathname of another database (the input file) and merges the input file into the output file:

```
func gatherInto(db *sql.DB, path string) error
```

sqlite can do most of the work for you. Issue the following sequence of commands to the output database:

```
attach ? as merge;
insert into pairs select * from merge.pairs;
detach merge;
```

Where `?` is the name of the input file. When this is complete, you should delete the input file as it is no longer needed.

Once again, check for every possible error condition and return any error found.

Testing Part 1

MapReduce will be a library, but for now it will be easier to test it if you put it in package main and write a `main` function. Use the Jane Austen example file and run your `splitDatabase` function to break it into several pieces. Count the number of rows in each piece from the command line using:

```
sqlite3 austen-0.sqlite3 'select count(1) from pairs'
```

Add the numbers up for each of the pieces and make sure the total matches the total number of rows in the input file.

Then have your main function merge the pieces back into a single file. Make sure it has the correct number of rows as well.

Since `mergeDatabases` expects to download data from the network, you need to run a simple web server. The following code will serve all files in the given directory:

```
go func() {
    http.Handle("/data/", http.StripPrefix("/data", http.FileServer(http.Dir(tempdir))))
    if err := http.ListenAndServe(address, nil); err != nil {
        log.Printf("Error in HTTP server for %s: %v", myaddress, err)
    }
}()
```

In this code `tempdir` is the directory containing the files to be served, and `/data/` is the prefix path that each file should have in its URL. `address` should be the network address to listen on. You can use `localhost:8080` to run it on a single machine.

For example, if `tempdir` is `/tmp/data` then `/tmp/data/austen.sqlite3` would be accessible as `http://localhost:8080/data/austen.sqlite3`

Part 2

In this part you will write the code to handle map and reduce tasks, but everything will still run on a single machine. Write your code in a file called `worker.go`.

To start out, you will need some data types:

```
type MapTask struct {
    M, R       int    // total number of map and reduce tasks
    N          int    // map task number, 0-based
    SourceHost string // address of host with map input file
}

type ReduceTask struct {
    M, R        int      // total number of map and reduce tasks
    N           int      // reduce task number, 0-based
    SourceHosts []string // addresses of map workers
}

type Pair struct {
    Key   string
    Value string
}

type Interface interface {
```

```
    Map(key, value string, output chan<- Pair) error
    Reduce(key string, values <-chan string, output chan<- Pair) error
}
```

`MapTask` and `ReduceTask` describe individual worker tasks. For now, you will fill these in with test data, but in part 3 these will be generated by the master and used to track a task on the master side and the worker side. Some fields will be filled in by the master and others will be filled in by the worker when it reports that a task has been completed.

A `Pair` groups a single key/value pair. Individual client jobs (wordcount, grep, etc.) will use this type when feeding results back to your library code.

The `Interface` type represents the interface that a client job must implement to interface with the library. In Go, it is customary to consider the full name of a value when selecting its name. In this case, the client code will refer to this type as `mapreduce.Interface`, so its apparently vague name is appropriate.

Since the master and workers will be exchanging data across HTTP using well-known filenames, it makes sense to apply consistent naming to data files. Here are some functions that will generate suitable names. Use these whenever you need to name a file or generate a URL:

```
func mapSourceFile(m int) string       { return fmt.Sprintf("map_%d_source.sqlite3", m) }
func mapInputFile(m int) string        { return fmt.Sprintf("map_%d_input.sqlite3", m) }
func mapOutputFile(m, r int) string    { return fmt.Sprintf("map_%d_output_%d.sqlite3", m, r) }
func reduceInputFile(r int) string     { return fmt.Sprintf("reduce_%d_input.sqlite3", r) }
func reduceOutputFile(r int) string    { return fmt.Sprintf("reduce_%d_output.sqlite3", r) }
func reducePartialFile(r int) string   { return fmt.Sprintf("reduce_%d_partial.sqlite3", r) }
func reduceTempFile(r int) string      { return fmt.Sprintf("reduce_%d_temp.sqlite3", r) }
func makeURL(host, file string) string { return fmt.Sprintf("http://%s/data/%s", host, file) }
```

You must implement a method to process a single map task, and a method to process a single reduce task.

MapTask.Process

Implement the following method:

```
func (task *MapTask) Process(tempdir string, client Interface) error
```

Use the database functions you wrote in part 1 as appropriate. This method processes a single map task. It must:

- Download and open the input file
- Create the output files
- Run a database query to select all pairs from the source file. For each pair:
    - Call `client.Map` with the data pair
    - Gather all Pair objects the client feeds back through the output channel and insert each pair into the appropriate output database. This process stops when the client closes the channel.

To select which output file a given pair should go to, use a hash of the output key:

```
hash := fnv.New32() // from the stdlib package hash/fnv
hash.Write([]byte(pair.Key))
r := int(hash.Sum32()) % task.R
```

Be vigilant about error checking, and make sure that you close all databases before returning.

Carefully plan how you will use manage the necessary channels and goroutines. Be careful to ensure that you have processed all output pairs from a single call to `client.Map` before you start processing the next input pair.

ReduceTask.Process

Implement the following method:

```
func (task *ReduceTask) Process(tempdir string, client Interface) error
```

This method processes a single reduce task. It must:

- Create the input database by merging all of the appropriate output databases from the map phase
- Create the output database
- Process all pairs in the correct order. This is trickier than in the map phase. Use this query:

  ```
  select key, value from pairs order by key, value
  ```

  It will sort all of the data and return it in the proper order. As you loop over the key/value pairs, take note whether the key for a new row is the same or different from the key of the previous row.

  When you encounter a key for the first time:

  - Close out the previous call to `client.Reduce` (unless this is the first key, of course). This includes closing the input channel (so `Reduce` will know it has processed all values for the given key) and waiting for it to finish.
  - Start a new call to `client.Reduce`. Carefully plan how you will manage the necessary goroutines and channels. This includes receiving output pairs and inserting them into the output database.

Be vigilant about watching for and handling errors in all code. Make sure that you close all databases before returning.

Also, when you finish processing all rows, do not forget to close out the final call to `client.Reduce`.

Testing part 2

In both of your `Process` methods, count the number of input keys and values, and the number of output pairs. At the end of the method, log a single summary line. For example:

```
map task processed 8423 pairs, generated 85153 pairs
reduce task processed 6460 keys and 234249 values, generated 6460 pairs
```

Write a main function to do the following:

- Pick some number of map tasks (say, m=9) and reduce tasks (say, r=3)
- Create a temporary directory. I suggest something like:

  ```
  tempdir := filepath.Join(os.TempDir(), fmt.Sprintf("mapreduce.%d", os.Getpid()))
  ```

- Delete the temp directory if it exists (use `os.RemoveAll`) and create an empty directory. You may also want to automatically delete it when your program finishes:

  ```
  defer os.RemoveAll(tempdir)
  ```

  Although you may wish to disable this while testing so you can examine the temporary data.

- Start an HTTP server that serves all files in your temporary directory:

  ```
  go func() {
          http.Handle("/data/", http.StripPrefix("/data",
  http.FileServer(http.Dir(tempdir))))
          if err := http.ListenAndServe(address, nil); err != nil {
                  log.Printf("Error in HTTP server for %s: %v", address, err)
          }
  }()
  ```

- Split your input file into `m` pieces in your temporary directory. Give them the map source name (not the map input name).
- Create a wordcount client (see below) and then create and process all map tasks.
- Create and process all reduce tasks.

- Merge the reduce outputs into a single output file.

Here is suitable code for a sample wordcount client:

```
type Client struct{}

func (c Client) Map(key, value string, output chan<- Pair) error {
    defer close(output)
    lst := strings.Fields(value)
    for _, elt := range lst {
        word := strings.Map(func(r rune) rune {
            if unicode.IsLetter(r) || unicode.IsDigit(r) {
                    return unicode.ToLower(r)
            }
            return -1
        }, elt)
        if len(word) > 0 {
            output <- Pair{Key: word, Value: "1"}
        }
    }
    return nil
}

func (c Client) Reduce(key string, values <-chan string, output chan<- Pair) error {
    defer close(output)
    count := 0
    for v := range values {
        i, err := strconv.Atoi(v)
        if err != nil {
            return err
        }
        count += i
    }
    p := Pair{Key: key, Value: strconv.Itoa(count)}
    output <- p
    return nil
}
```

If you use the Jane Austen input file, your output file should match mine: Jane Austen wordcounts. Try querying the top 20 word counts to compare them:

```
select key, value from pairs order by value+0 desc limit 20
```

My solution takes roughly 15 seconds to run on oxygen. If yours takes significantly longer, you should try to find out why. A few tips:

- Use prepared statements for all of your `insert` operations. Since there are many of these operations for each output file, this can yield a significant savings.
- Use buffered channels. Since the various processing tasks (scanning the database, inserting into the database, client data processing) are happening concurrently and at different speeds, a buffered channel will let more work happen in parallel. A buffer of size 100 will give a good boost over unbuffered channels.
- Call `runtime.GOMAXPROCS(1)` at the beginning of your `main` function so Go will only try to use a single operating system thread. This will avoid unnecessary context switching overhead in the operating system. There is little benefit in this project to using multiple threads, and there can be significant overhead.

Part 3

In this part you will write the code to handle the master and the workers, enabling a MapReduce job to be split across multiple machines. Write your code in a file called `master.go`.

In the first two parts of this project, everything was part of the main package. In this part, you will split the library code into a package called `mapreduce`, and the client code into a separate package that imports `mapreduce`. Here is the complete source for a suitable client that implements word count:

```
package main

import (
```

```
    "log"
    "mapreduce"
    "strconv"
    "strings"
    "unicode"
)

func main() {
    var c Client
    if err := mapreduce.Start(c); err != nil {
        log.Fatalf("%v", err)
    }
}

// Map and Reduce functions for a basic wordcount client

type Client struct{}

func (c Client) Map(key, value string, output chan<- mapreduce.Pair) error {
    defer close(output)
    lst := strings.Fields(value)
    for _, elt := range lst {
        word := strings.Map(func(r rune) rune {
            if unicode.IsLetter(r) || unicode.IsDigit(r) {
                return unicode.ToLower(r)
            }
            return -1
        }, elt)
        if len(word) > 0 {
            output <- mapreduce.Pair{Key: word, Value: "1"}
        }
    }
    return nil
}

func (c Client) Reduce(key string, values <-chan string, output chan<- mapreduce.Pair) error {
    defer close(output)
    count := 0
    for v := range values {
        i, err := strconv.Atoi(v)
        if err != nil {
            return err
        }
        count += i
    }
    p := mapreduce.Pair{Key: key, Value: strconv.Itoa(count)}
    output <- p
    return nil
}
```

In this example, the library is imported as `mapreduce`, implying that it is located at `$GOPATH/src/mapreduce`. You may put it at a different location and update the import path accordingly.

There are a few points worth noting:

1. The `Pair` type is implemented in the `mapreduce` package, so the client code must name it as `mapreduce.Pair`.
2. The `Client` type can be anything, and can be named anything. All that matters is that it implements the `mapreduce.Interface` interface so that it can be used in the call to `mapreduce.Start`. Note that there is no mention of `mapreduce.Interface` in the code—the compiler verifies that the `Client` type implements the correct interface, so no declaration is necessary.
3. The code is the same for the master or a worker. All command-line arguments are handled by `mapreduce.Start`, and it decides which role this instance will assume.
4. In our implementaiton, the `Client` instance should not hold any important information that needs to be shared across nodes. Each instance (master and workers) will create its own fresh instance of `Client`, so workers will not be able to use `Client` as a way to share data.

The `mapreduce.Start` function starts by processing command-line arguments. You can use the `flag` package from the standard library to make this easier, or you can use a different mechanism. You should choose a reasonable set of command-line options that will give the user sufficient control over the process. Here are a few issues to consider when you design your interface:

- The user must be able to specify whether an instance is the master or a worker.
- Most instances will be workers, so the interface should be streamlined for creating workers.
- Workers can get most of the basic parameters from the master (including the values of M and R). Workers should just need to know where the master is and any configuration that is necessary for the local machine (like where to locate the temporary storage directory, which network port to listen on, etc.)
- You may wish to allow the user to (optionally) separate the actions of splitting the input data and actually acting as the master. When working on a large data set, you may need to re-start the computation many times as you test your code, and incurring the time to split the input on each run will slow the process down a lot.

The master

The master node needs to do the following:

1. Split the input file and start an HTTP server to serve source chunks to map workers.
2. Generate the full set of map tasks and reduce tasks. Note that reduce tasks will be incomplete initially, because they require a list of the hosts that handled each map task.
3. Create and start an RPC server to handle incoming client requests. Note that it can use the same HTTP server that shares static files.
4. Serve tasks to workers. The master should issue map tasks to workers that request jobs, then wait until all map tasks are complete before it begins issuing reduce jobs.
5. Wait until all jobs are complete.
6. Gather the reduce outputs and join them into a single output file.
7. Tell all workers to shut down, then shut down the master.

Workers

A worker node needs to do the following in a loop:

1. Start an HTTP server to serve intermediate data files to other workers and back to the master.
2. Request a job from the master.
3. Process each job using the code from part 2.
4. Shut down and clean up when finished.

RPC server

You are free to design and implement your RPC calls in a way that makes sense to you. The master and workers all share a single code base, so they can easily share data structures.