# 目标成绩： （填写ABCD）D为合格

# 大数据实验报告（实验3）

 班级 21计算机 班  学号 姓名

实验时间： 2024.04.30

一、实验名称：MapReduce

Introduction导言

在正式看Lab材料之前，你应该已经阅读过[MapReduce论文](https://link.zhihu.com/?target=https%3A//pdos.csail.mit.edu/6.824/papers/mapreduce.pdf)，如果没有，最好先阅读。

在这个实验中，你将构建一个MapReduce系统。你将实现一个调用应用程序的Map和Reduce函数，并处理读和写文件的**worker**进程，以及一个将任务分发给worker进程并处理失败的worker进程的**coordinator**进程。 你将构建类似于MapReduce论文描述的系统。 （注：本实验"coordinator"对应论文中的"master"。)

In this lab you'll build a MapReduce system. You'll implement a worker process that calls application Map and Reduce functions and handles reading and writing files, and a coordinator process that hands out tasks to workers and copes with failed workers. You'll be building something similar to the [MapReduce paper](http://research.google.com/archive/mapreduce-osdi04.pdf). (Note: this lab uses "coordinator" instead of the paper's "master".)

MapReduce框架

通过论文中MapReduce的流程图，首先了解其框架。

**输入**数据以文件形式进入系统。一些进程运行map任务，拆分了原任务，产生了一些**中间体**，这些中间体可能以**键值对**形式存在。一些进程运行了reduce任务，利用中间体产生**最终输出**。master进程用于分配任务，调整各个worker进程。

输入数据能够产生中间体，这说明原任务是**可拆的**，也就才有了写成分布式的可能性。若原问题不是可拆的，MapReduce也就无从谈起。

**中间体应均匀地分配给各个reduce任务**，每个reduce任务**整合**这些中间体，令中间体个数减少，直至无法再减少，从中整合出最终结果。

**输入**数据以什么形式进入系统，原任务应如何**拆分**，**中间体**如何保存和传输，master和worker之间如何通信和调度，**中间体**如何转化为最终输出。这些都是设计的考量，没有一定之规。

Getting started

Lab使用Golang为主要编程语言，若你不熟悉，可以快速看看[Golang官方入门](https://tour.go-zh.org/welcome/1)。



You need to [setup Go](https://pdos.csail.mit.edu/6.824/labs/go.html) to do the labs.

Fetch the initial lab software with [git](https://git-scm.com/) (a version control system). To learn more about git, look at the [Pro Git book](https://git-scm.com/book/en/v2) or the [git user's manual](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html).

使用git（一个版本控制系统）获取初始实验软件。要了解有关git的更多信息，请参阅Pro Git book或git用户手册。

$ git clone git://g.csail.mit.edu/6.5840-golabs-2024 6.5840

$ cd 6.5840

$ ls

Makefile src

$

## 任务总览和说明

Lab 3要求我们实现一个和MapReduce论文类似的机制，也就是数单词个数Word Count。在正式开始写分布式代码之前，我们先理解一下任务和已有的代码。

用于测试的文件在src/main目录下，以pg-\*.txt形式命名。每个pg-\*.txt文件都是一本电子书，非常长。我们的任务是统计出所有输入文件中出现过的单词，以及它们的出现次数。

* 非分布式实现

非分布式实现这个任务非常简单，如果不写成分布式的，一个实现在src/main/mrsequential.go中。

将所有文章中的单词分出，保存到一个类似数组的结构中。将这些单词排序，从而相同单词在数组中连续地出现在一起。排序完成后，遍历这个数组，由于相同的单词相邻地出现，统计单词个数就很简单了。

尝试运行mrsequential.go，看看最终的输出是什么样子的。

输出文件在src/main/mr-out-0，文件中每一行标明了单词和出现次数。

go run mrsequential.go之后的两项是传给mrsequential的命令行参数，分别是一个动态库和所有电子书。

We supply you with a simple sequential mapreduce implementation in src/main/mrsequential.go. It runs the maps and reduces one at a time, in a single process.

在进入Go程序之后，动态库由代码主动加载进来。在src/main目录下命名为mr\*.go的几个代码文件中，都有loadPlugin函数。**如果你使用Goland作为主要IDE，编辑器会提示重复函数声明。**在这里，我们给mrsequential加载的是在src/mrapps目录下的wc.go编译得到的动态库。

文件wc.go以及mrapps目录下的其它几个文件，都定义了名为map, reduce的函数，这两个函数在mrsequential.go中加载并调用。给mrsequential绑定不同的\*.so文件，也就会加载不同的map, reduce函数。如此实现某种程度上的**动态绑定**。

mrsequential实现的是**非分布式**的Word Count，采用的算法就是上面描述的。这个文件的输出将作为之后测试的**标准**，分布式版本应给出和这个输出完全相同的输出。

We also provide you with a couple of MapReduce applications: word-count in mrapps/wc.go, and a text indexer in mrapps/indexer.go. You can run word count sequentially as follows:

$ cd ~/6.5840

$ cd src/main

$ go build -buildmode=plugin ../mrapps/wc.go

$ rm mr-out\*

$ go run mrsequential.go wc.so pg\*.txt

$ more mr-out-0

A 509

ABOUT 2

ACT 8

...

mrsequential.go leaves its output in the file mr-out-0. The input is from the text files named pg-xxx.txt.

Feel free to borrow code from mrsequential.go. You should also have a look at mrapps/wc.go to see what MapReduce application code looks like.

For this lab and all the others, we might issue updates to the code we provide you. To ensure that you can fetch those updates and easily merge them using git pull, it's best to leave the code we provide in the original files. You can add to the code we provide as directed in the lab write-ups; just don't move it. It's OK to put your own new functions in new files.

* **执行代码**

我们的代码主要写在src/mr目录下的几个文件，这几个文件由src/main目录下两个文件mrcoordinator.go, mrworker.go调用。这两个文件的作用是启动进程、加载map, reduce动态库，并进入定义在src/mr目录下的主流程。

上面展示了执行mrsequential的办法。要执行我们自己写的代码，需要执行mrcoordinator.go, mrworker.go。其中，要给mrcoordinator.go输入电子书文件列表pg-\*.txt，给mrworker.go指定动态库wc.so。由于master不需要动态库，worker不需要电子书文件名，两者接受的命令行参数是不一样的。

go run mrcoordinator.go pg-\*.txt

go run mrworker.go wc.so

现在还什么也没写，所以什么也运行不出来。每次这样的运行，都启动了一个新的进程，进程之间不能直接相互访问对方的变量，必须通过一定的**进程间通信机制**才能实现。我们使用的**进程间通信是rpc**。

## 我们的任务

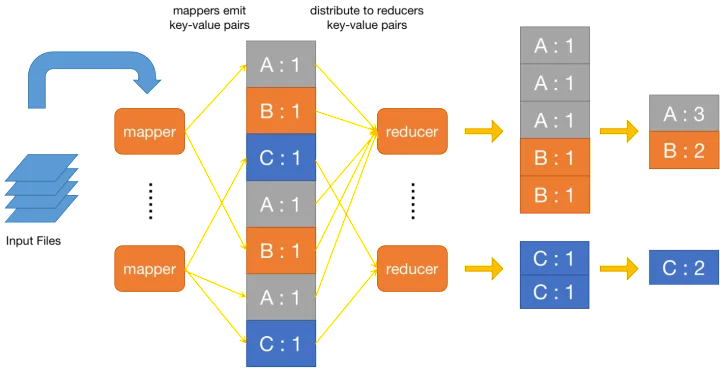
我们的工作是实现分布式MapReduce，它由两个程序组成，即coordinator和worker。将只有一个coordinator进程和一个或多个并行执行的worker进程。在一个真实的系统中，worker将在一堆不同的机器上运行，但对于这个实验，你将在一台机器上运行所有的进程。worker将通过RPC与coordinator交互。每个worker进程将向coordinator请求任务，从一个或多个文件中读取任务的输入，执行任务，并将任务的输出写入一个或更多个文件。worker应该注意到，如果一个worker没有在合理的时间内完成任务（对于这个实验，是10秒），并将相同的任务交给另一名worker。

测试时，启动一个coordinator和多个worker，也就是运行一次mrcoordinator.go、运行多次mrworker.go。

master进程启动一个rpc服务器，每个worker进程通过rpc机制向coordinator要任务。任务可能包括map和reduce过程，具体如何给worker分配取决于coordinator。

每个单词和它出现的次数以key-value**键值对**形式出现。map进程将每个出现的单词机械地分离出来，并给每一次出现标记为1次。很多单词在电子书中重复出现，也就产生了很多相同**键值对**。还没有对键值对进行合并，故此时产生的键值对的**值**都是1。此过程在下图中mapper伸出箭头表示。

已经分离出的单词以键值对形式分配给特定reduce进程，reduce进程个数远小于单词个数，每个reduce进程都处理一定量单词。相同的单词应由相同的reduce进程处理。处理的方式和上面描述的算法类似，对单词排序，令单词在数组中处在相邻位置，再统计单词个数。最终，每个reduce进程都有一个输出，合并这些输出，就是Word Count结果。此过程在下图中箭头进入reducer、以及后面的合并表示。



图中，相同的单词由相同reducer处理。如第一个reducer接受单词A, B，最后一个reducer接受单词C。

测试流程要求，输出的文件个数和参数nReduce相同，即每个输出文件对应一个reduce任务，格式和mrsequential的输出格式相同，命名为mr-out\*。我们的代码应保留这些文件，不做进一步合并，测试脚本将进行这一合并。合并之后的最终完整输出，必须和mrsequential的输出完全相同。

查看测试脚本test-mr.sh，可以看到合并每个输出mr-out\*的指令如下，将每个输出文件的每一行按行首单词排序，输出到最终文件mr-wc-all中。

sort mr-out\* | grep . > mr-wc-all

故每个reduce任务不能操作相同的单词，在map流程中分离出的相同单词**键值对**应由同一个reduce流程处理。

Your Job ([moderate/hard](https://pdos.csail.mit.edu/6.824/labs/guidance.html))

Your job is to implement a distributed MapReduce, consisting of two programs, the coordinator and the worker. There will be just one coordinator process, and one or more worker processes executing in parallel. In a real system the workers would run on a bunch of different machines, but for this lab you'll run them all on a single machine. The workers will talk to the coordinator via RPC. Each worker process will ask the coordinator for a task, read the task's input from one or more files, execute the task, and write the task's output to one or more files. The coordinator should notice if a worker hasn't completed its task in a reasonable amount of time (for this lab, use ten seconds), and give the same task to a different worker.

We have given you a little code to start you off. The "main" routines for the coordinator and worker are in main/mrcoordinator.go and main/mrworker.go; don't change these files. You should put your implementation in mr/coordinator.go, mr/worker.go, and mr/rpc.go.

Here's how to run your code on the word-count MapReduce application. First, make sure the word-count plugin is freshly built:

$ go build -buildmode=plugin ../mrapps/wc.go

In the main directory, run the coordinator.

$ rm mr-out\*

$ go run mrcoordinator.go pg-\*.txt

The pg-\*.txt arguments to mrcoordinator.go are the input files; each file corresponds to one "split", and is the input to one Map task.

In one or more other windows, run some workers:

$ go run mrworker.go wc.so

When the workers and coordinator have finished, look at the output in mr-out-\*. When you've completed the lab, the sorted union of the output files should match the sequential output, like this:

$ cat mr-out-\* | sort | more

A 509

ABOUT 2

ACT 8

...

We supply you with a test script in main/test-mr.sh. The tests check that the wc and indexer MapReduce applications produce the correct output when given the pg-xxx.txt files as input. The tests also check that your implementation runs the Map and Reduce tasks in parallel, and that your implementation recovers from workers that crash while running tasks.

If you run the test script now, it will hang because the coordinator never finishes:

$ cd ~/6.5840/src/main

$ bash test-mr.sh

\*\*\* Starting wc test.

You can change ret := false to true in the Done function in mr/coordinator.go so that the coordinator exits immediately. Then:

$ bash test-mr.sh

\*\*\* Starting wc test.

sort: No such file or directory

cmp: EOF on mr-wc-all

--- wc output is not the same as mr-correct-wc.txt

--- wc test: FAIL

$

The test script expects to see output in files named mr-out-X, one for each reduce task. The empty implementations of mr/coordinator.go and mr/worker.go don't produce those files (or do much of anything else), so the test fails.

When you've finished, the test script output should look like this:

$ bash test-mr.sh

\*\*\* Starting wc test.

--- wc test: PASS

\*\*\* Starting indexer test.

--- indexer test: PASS

\*\*\* Starting map parallelism test.

--- map parallelism test: PASS

\*\*\* Starting reduce parallelism test.

--- reduce parallelism test: PASS

\*\*\* Starting job count test.

--- job count test: PASS

\*\*\* Starting early exit test.

--- early exit test: PASS

\*\*\* Starting crash test.

--- crash test: PASS

\*\*\* PASSED ALL TESTS

$

You may see some errors from the Go RPC package that look like

2019/12/16 13:27:09 rpc.Register: method "Done" has 1 input parameters; needs exactly three

Ignore these messages; registering the coordinator as an [RPC server](https://golang.org/src/net/rpc/server.go) checks if all its methods are suitable for RPCs (have 3 inputs); we know that Done is not called via RPC.

## 一些规则：

* map阶段应将**中间体**的**键**划分为nReduce个reduce任务的**存储桶**（桶的概念参考hash），其中nReduce是reduce任务的数量——main/mrcordinator.go传递给MakeCoordinator（）的参数。每个mapper都应该创建nReduce个中间文件，以供reduce任务使用。
* worker的实现应将第X个reduce任务的输出放在mr-out-X文件中。
* Reduce函数每做一次输出，应在mr-out-X文件存放一行。该行应使用Go“%v%v”格式生成，输出**键**和**值**。在main/mrsequential.go中查看注释为“this is the correct format”的行。如果您的实现与此格式偏离太多，那么测试脚本将失败。
* 你仅可以修改mr/worker.go、mr/coordinator.go和mr/rpc.go。你也可以**临时**修改其他文件进行测试，但要确保最终你的代码仅修改前述3个文件；我们将使用原始版本进行测试。
* worker应将Map输出**中间体**存放在当前目录中，以便worker稍后可以将其作为Reduce任务的输入读取。
* main/mrcoordinator.go要求mr/coordinator.go实现一个Done（）方法，该方法在MapReduce作业完全完成时返回true；此时，mrcoordinator.go将退出。
* 当worker完全完成时，应退出worker进程。实现这一点的一个简单方法是使用call（）的返回值：如果worker未能联系到coordinator，则可以假设coordinator已退出，因为作业已完成，因此worker也可以终止。根据你的设计，你可能还会发现coordinator可以给worker一个“请退出”的伪任务很有帮助。

A few rules:

* The map phase should divide the intermediate keys into buckets for nReduce reduce tasks, where nReduce is the number of reduce tasks -- the argument that main/mrcoordinator.go passes to MakeCoordinator(). Each mapper should create nReduce intermediate files for consumption by the reduce tasks.
* The worker implementation should put the output of the X'th reduce task in the file mr-out-X.
* A mr-out-X file should contain one line per Reduce function output. The line should be generated with the Go "%v %v" format, called with the key and value. Have a look in main/mrsequential.go for the line commented "this is the correct format". The test script will fail if your implementation deviates too much from this format.
* You can modify mr/worker.go, mr/coordinator.go, and mr/rpc.go. You can temporarily modify other files for testing, but make sure your code works with the original versions; we'll test with the original versions.
* The worker should put intermediate Map output in files in the current directory, where your worker can later read them as input to Reduce tasks.
* main/mrcoordinator.go expects mr/coordinator.go to implement a Done() method that returns true when the MapReduce job is completely finished; at that point, mrcoordinator.go will exit.
* When the job is completely finished, the worker processes should exit. A simple way to implement this is to use the return value from call(): if the worker fails to contact the coordinator, it can assume that the coordinator has exited because the job is done, so the worker can terminate too. Depending on your design, you might also find it helpful to have a "please exit" pseudo-task that the coordinator can give to workers.

## 提示

* 一种启动系统的方法是：修改mr/worker.go的worker（），向coordinator发送RPC请求来获得任务。然后修改coordinator回复文件名进行响应，代表尚未启动的map任务。然后修改worker以读取该文件并调用应用程序Map函数，如mrsequential.go中所示。
* 应用程序的Map和Reduce函数，是以Go插件的形式，以名称为.so结尾的文件加载。
* 如果你更改了mr/目录中的任何内容，你可能需要重新构建你使用的任何MapReduce插件，比如go build-buildmode=plugin/mrapps/wc.go文件
* 该实验要求worker共享**同一个**文件系统。当所有worker都在同一台机器上运行时，这不是问题，但如果worker在不同的机器上运行，则需要像GFS这样的全局文件系统。
* 中间文件的合理命名约定是mr-X-Y，其中X是Map任务编号，Y是reduce任务编号。
* worker的map任务代码需要一种方法来将中间键/值对存储在文件中，这种方法需要保证在reduce任务期间正确读取。一种可能性是使用Go的encoding/json包。要将JSON格式的键/值对写入打开的文件，请执行以下操作：

enc := json.NewEncoder(file)

for \_, kv := ... {

err := enc.Encode(&kv)

and to read such a file back:

dec := json.NewDecoder(file)

for {

var kv KeyValue

if err := dec.Decode(&kv); err != nil {

break

}

kva = append(kva, kv)

}

* worker的map部分可以使用ihash（key）函数（在worker.go中）为给定的键选择reduce任务。
* 您可以从mrsequential.go中窃取一些代码，用于读取Map输入文件，用于在Map和Reduce之间排序相互关联的键/值对，以及用于将Reduce输出存储在文件中。
* 作为RPC服务器的cordinator将是**并发**的；不要忘记**锁定共享数据**。
* 使用Go的race detector，配合Go run-race。test-mr.sh在开头有一个注释，告诉如何使用-race。
* worker有时需要等待，例如，直到最后一个map完成才能开始reduce。一种可能性是，worker定期向coordinator请求任务，每次请求之间都执行time.Sleep()休眠。另一种可能性是，coordinator中的相关RPC处理程序有一个等待的循环，采用time.Sleep()或sync.Cond。Go在其自己的线程中为每个RPC运行处理程序，因此一个处理程序正在等待的事实不必阻止协调器处理其他RPC。
* coordinator无法可靠地区分崩溃worker、活着但因某种原因停滞的worker和执行速度太慢而无法发挥作用的worker。你能做的最好的事情就是让cordinator等待一段时间，然后放弃并将任务重新发布给另一个worker。对于这个实验，让coordinator等待10秒钟；在那之后，coordinator应假设worker已经死亡（当然，可能没有）。
* 如果您选择实现备份任务（第3.6节），请注意，我们测试了当worker在不崩溃的情况下执行任务时，您的代码不会安排其它无关的任务。备份任务只能在一段相对较长的时间（例如10秒）后被调度。
* 要测试崩溃恢复，你可以使用mrapps/crash.go插件。它能在Map和Reduce函数中随机退出。
* 为了确保没有人在出现崩溃时观察到部分写入的文件，MapReduce的论文提到了使用临时文件并在完全写入后自动重命名的技巧。您可以使用ioutil.Temp文件创建一个临时文件，并使用os.Rrename对其进行原子重命名。
* test-mr.sh在子目录mr-tmp中运行其所有进程，因此，如果出现问题，并且你想查看中间文件或输出文件，请查看那里。请随意临时修改test-mr.sh以在测试失败后退出，这样脚本就不会继续测试（并覆盖输出文件）。
* test-mr-many.sh连续多次运行test-mr.sh，你可能想这样做，以发现低概率的错误。它以运行测试的次数作为参数。您不应该并行运行多个test-mr.sh实例，因为协调器会重用同一个套接字，从而导致冲突。
* Go RPC只发送名称以大写字母开头的结构字段。子结构还必须具有大写的字段名称。
* 当调用RPC call（）函数时，应答结构应该包含所有默认值。RPC调用应如下所示：

reply := SomeType{}

call(..., &reply)

* 而不在呼叫之前设置任何应答字段。如果传递具有非默认字段的回复结构，RPC系统可能会静默地返回不正确的值。

Hints

* The [Guidance page](https://pdos.csail.mit.edu/6.824/labs/guidance.html) has some tips on developing and debugging.
* One way to get started is to modify mr/worker.go's Worker() to send an RPC to the coordinator asking for a task. Then modify the coordinator to respond with the file name of an as-yet-unstarted map task. Then modify the worker to read that file and call the application Map function, as in mrsequential.go.
* The application Map and Reduce functions are loaded at run-time using the Go plugin package, from files whose names end in .so.
* If you change anything in the mr/ directory, you will probably have to re-build any MapReduce plugins you use, with something like go build -buildmode=plugin ../mrapps/wc.go
* This lab relies on the workers sharing a file system. That's straightforward when all workers run on the same machine, but would require a global filesystem like GFS if the workers ran on different machines.
* A reasonable naming convention for intermediate files is mr-X-Y, where X is the Map task number, and Y is the reduce task number.
* The worker's map task code will need a way to store intermediate key/value pairs in files in a way that can be correctly read back during reduce tasks. One possibility is to use Go's encoding/json package. To write key/value pairs in JSON format to an open file:

enc := json.NewEncoder(file)

for \_, kv := ... {

err := enc.Encode(&kv)

and to read such a file back:

dec := json.NewDecoder(file)

for {

var kv KeyValue

if err := dec.Decode(&kv); err != nil {

break

}

kva = append(kva, kv)

}

* The map part of your worker can use the ihash(key) function (in worker.go) to pick the reduce task for a given key.
* You can steal some code from mrsequential.go for reading Map input files, for sorting intermedate key/value pairs between the Map and Reduce, and for storing Reduce output in files.
* The coordinator, as an RPC server, will be concurrent; don't forget to lock shared data.
* Use Go's race detector, with go run -race. test-mr.sh has a comment at the start that tells you how to run it with -race. When we grade your labs, we will **not** use the race detector. Nevertheless, if your code has races, there's a good chance it will fail when we test it even without the race detector.
* Workers will sometimes need to wait, e.g. reduces can't start until the last map has finished. One possibility is for workers to periodically ask the coordinator for work, sleeping with time.Sleep() between each request. Another possibility is for the relevant RPC handler in the coordinator to have a loop that waits, either with time.Sleep() or sync.Cond. Go runs the handler for each RPC in its own thread, so the fact that one handler is waiting needn't prevent the coordinator from processing other RPCs.
* The coordinator can't reliably distinguish between crashed workers, workers that are alive but have stalled for some reason, and workers that are executing but too slowly to be useful. The best you can do is have the coordinator wait for some amount of time, and then give up and re-issue the task to a different worker. For this lab, have the coordinator wait for ten seconds; after that the coordinator should assume the worker has died (of course, it might not have).
* If you choose to implement Backup Tasks (Section 3.6), note that we test that your code doesn't schedule extraneous tasks when workers execute tasks without crashing. Backup tasks should only be scheduled after some relatively long period of time (e.g., 10s).
* To test crash recovery, you can use the mrapps/crash.go application plugin. It randomly exits in the Map and Reduce functions.
* To ensure that nobody observes partially written files in the presence of crashes, the MapReduce paper mentions the trick of using a temporary file and atomically renaming it once it is completely written. You can use ioutil.TempFile to create a temporary file and os.Rename to atomically rename it.
* test-mr.sh runs all its processes in the sub-directory mr-tmp, so if something goes wrong and you want to look at intermediate or output files, look there. Feel free to temporarily modify test-mr.sh to exit after the failing test, so the script does not continue testing (and overwrite the output files).
* test-mr-many.sh runs test-mr.sh many times in a row, which you may want to do in order to spot low-probability bugs. It takes as an argument the number of times to run the tests. You should not run several test-mr.sh instances in parallel because the coordinator will reuse the same socket, causing conflicts.
* Go RPC sends only struct fields whose names start with capital letters. Sub-structures must also have capitalized field names.
* When calling the RPC call() function, the reply struct should contain all default values. RPC calls should look like this:

reply := SomeType{}

call(..., &reply)

without setting any fields of reply before the call. If you pass reply structures that have non-default fields, the RPC system may silently return incorrect values.

No-credit challenge exercises

Implement your own MapReduce application (see examples in mrapps/\*), e.g., Distributed Grep (Section 2.3 of the MapReduce paper).

Get your MapReduce coordinator and workers to run on separate machines, as they would in practice. You will need to set up your RPCs to communicate over TCP/IP instead of Unix sockets (see the commented out line in Coordinator.server()), and read/write files using a shared file system. For example, you can ssh into multiple [Athena cluster](http://kb.mit.edu/confluence/display/istcontrib/Getting+Started+with+Athena) machines at MIT, which use [AFS](http://kb.mit.edu/confluence/display/istcontrib/AFS+at+MIT+-+An+Introduction) to share files; or you could rent a couple AWS instances and use [S3](https://aws.amazon.com/s3/) for storage.