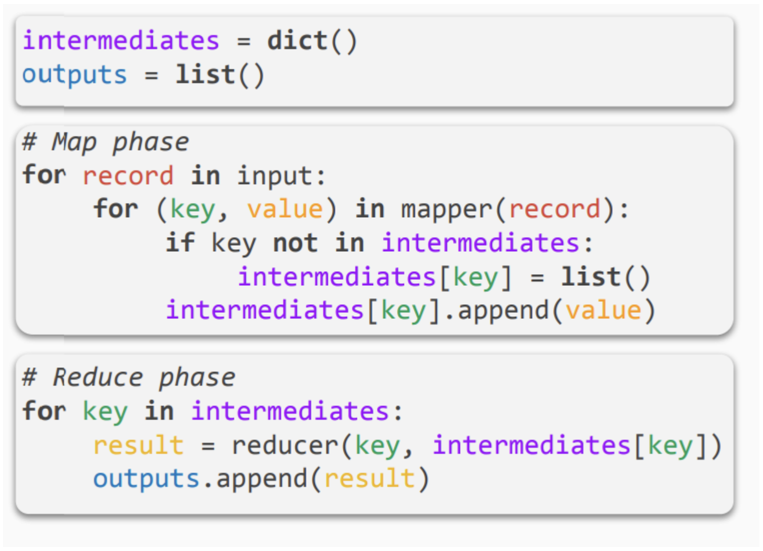
lecture 4: HDFS

wbg231

January 2023

1 Introduction

- 

```
intermediates = dict()
outputs = list()
```

```
# Map phase
for record in input:
    for (key, value) in mapper(record):
        if key not in intermediates:
            intermediates[key] = list()
        intermediates[key].append(value)
```

```
# Reduce phase
for key in intermediates:
    result = reducer(key, intermediates[key])
    outputs.append(result)
```

- suppose we have the above task we are running on one machine
- our steps are as follows
  1. run mapper on each input records
  2. collect the results for each intermediate key
  3. run reducer on each intermediate key
- how can we make this work on multiple machines in parallel?

map reduce details

- how is data shared over the cluster?
- how do we handel node failure?
- how can we optimize for map reduce as a framework?

start simple

- imagine that we were building map reduce from the ground up
- in map reduce the head node sends mapper and reducer code to each worker as well as a block of data to work with
- worker then send output back to head node
- the mappers produce intermediate results
- the reducers produce final results
- this will work but there are some issues
  1. each job moves hte entire data set over the network which is slow
  2. this is what we call a failure of locality that is though it would be more efficient to store data locally we are transferring all of it

localize all the data

- what if all data is replicated on all worker nodes?
- the head node would send the mapper and reduce code to each worker as well as the id of a data block where the data it will work with is stores in system memory
- workers then send output back to head
- this will work but it is expensive each worker needs a large amount of storage to hold all data, while most workers do not use most data

design considerations

- communication costs (it takes a lot of time to move data over the network)
- fault tolerance when we have a lot of machines things fails
- Redundancy vs communication (that is if we store more redundant information it is more likely to be near where it is used, but we need more storage)
- granularity of access that is can we let users write and read all files
- locality how much data do we store on each host machine
- common access patterns in map reduce programs are small (like map reduce has two simple functions) but data is large in these use cases

the hadoop distributed file system HDFS

- HDFS is the storage component of hadoop (used for more than just map reduce)
- provides distributed redundant storage
- it is optimized for single write (ie we only write to a file once) multi read (we can read that file many times) patterns

networked filesystem (NFS) VS HDFS

- NFS stores data on one machine but provide access from multiple
- distributed file system spear each file across multiple machines
- if a disk fails you need to take hte machine offline. so fault tolerance is a at the level of the machine not the disk (so having more machines with redundent info means more fault tolerance)
- so this means we can work around machine failure in our network not just disk failure

using HDFS

- HDFS is a file system but we use it differently
- HDFS sits on top of the operating systems built in file system unix on most host machines
- think of it kinda like an application that stores files for us, like good drive or what ever
- access data using the hadoop fs -command command
- so if you are working on dataproc at nyu, you have a host machine there that will store your files and work like a normal file system and can access HDFS

two types of nodes

- we have the name node and data node

name node

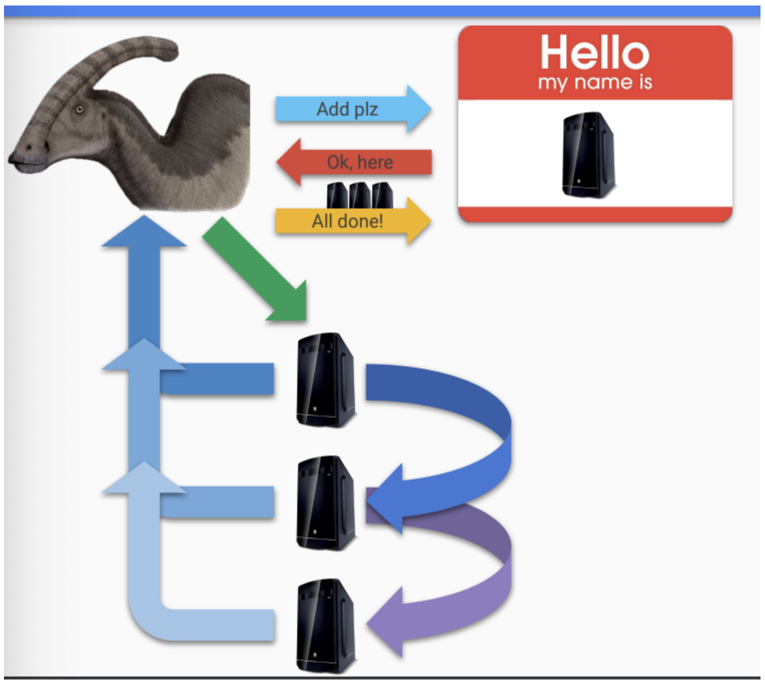
- clients talk to the name node to locate data (think of it like a file system)
- name node knows what files map to what blocks, and what blocks map to what data nodes.
- so if a client asks for information from some file the data node knows where the file is, what data nodes map to it and thus how to get that info
- ot also keeps a journal of transactions, which is backed up remotely for durability (ie not everything is lost if the name node goes down)

data node

- stores each block as two files in the local file system
    1. a data block of variable size that has the data
    2. and a file with meta data. this meta data includes a checksum as well as a generation stamp
- a checksum is used to detect storage errors, each piece of data is associated with a checksum so if the checksum is not correct for a data node it is likely that node has the wrong data
- a generation stamp is a piece of metadata used to check for updates

example writing to HDFS

- here is a diagram



  1. the client wants to add a block, the node name responds by giving a list of data nodes that could store the block
  2. client sends block of data to data node 1
  3. data node 1 stores the data, sends acknowledgment to the user that the data has been received and sends the block to data node 2
  4. data node 2 stores the data, sends acknowledgment to the user that the data has been received and sends the block to data node 3
  5. data node 3 stores the data, and acknowledgment to the user that the data has been received
  6. the add is done, the file is closed and the user tells the name node that the write is complete, the name node

name and data node communication

- data nodes send a periodic signal to the name node called a heart beat
- the name node always knowns which data nodes are alive, name node infers a data node is dead if it does not get a heart beat from a data node within a certain amount of time

- name nodes may respond with update messages, ie after each heart beat a name node can give a data node a command like replicate this block from this data node

recovering from failure: checkpoints

- checkpoints are snapshots of the current name node's state
- checkpoints hold directory structure, what blocks map to what data nodes, and journal (ie list of transactions history)
- name node keeps all this information in it's ram so it is quickly accessible
- these are created periodically to ensure the network can recover quickly from name node failures
- checkpoints cannot be updated only replaced

HDFS is not POSIX-compliant

- POSIX is more or less a standard for how unix operating systems work
- updates to files are append only (within applications a user can delete a file though)
- this means old data on hdfs can not be changed
- this makes replication logic much more simple
- also means that workers can read most of the same file while another worker is writing to it
- does not support all file formats like executable code
- POSIX is portable operating system interface that are standards around os and file system compatibility has standard operations like read write and seek
- HDFS updates are append only can not change old data making replication a lot easier

why does HDFS not work like a personal computer file system?

- desktop computers need to support all kinds of users, like many small configuration files or files that update frequently like a browser cache

- hdfs is for large data analysis jobs which have different needs, generally in this context we have a few large files, that are frequently read, and infrequently updated
- so hdfs restricts these functionalities to optimize for what it is actually trying to do

division of responsibilities

- the name node does not store data
- the data node does not store meta data
- name node failure is really bad
- data node failure can be tolerated up to a point (this varies depending on how much our data is replicated)
- **slido question**: Why do you think the HDFS designers parallelized storage at the level of blocks instead of files?
- files can be to large to store all in one place and each data node only needs a portion of each file.
- further having a uniform max block size as opposed to files of different lengths makes allocating data across the cluster and making replications of the data easier

HDFS and map reduce

a typical map reduce work flow

1. upload data from your unix file system to HDFS done with HDFS -put $\text{file}_{\text{name}}$
2. next we run the map reduce program
   - each mapper get a portion of $\text{file}_{\text{name}}$
   - each mapper produces intermediate outputs that are saved HDFS
   - shuffle stage collects intermediate outs to give to reducers
   - reducers operate on intermediate keys to produce final output as multiple blocks (saved in HDFS)
3. we get the output from HDFS using command hadoop fs -getmerge $\text{file}_{\text{name}}$ (this is required because the outputs from reducers are stored in multiple blocks so we are merging them back to one file)

how does hdsfs help map reduce

- hdfs shares blocks over data nodes
- map reduce shares jobs over compute nodes
- in big data applications it is cheaper to organize our computation around where we put our data, instead of organize our data around our computation
- so for instance if we were going to do a map reduce call and were considering sorting our data to make computation easier it is likely that the cost of moving all the data after it is sorted and before the program is run would outweigh the computational gains from sorting

job scheduling and input splits

- most map reduce programs run over 1 large file (broken into blocks in the HDFS file system)
- map map reduce devices divides the input data into splits. (where a split is a unit of work assigned to each mapper)
- each split maps onto one or more block in HDFS
- so we try to assign wok such that work for a split is done on a machine within its blocks.
- HDFS exposes block layout to the application layer to make this possible
- so to clarify what happens
  1. user uploads data to HDFS, HDFS splits data into blocks
  2. the user gets the location of these blocks from the name node, and calls the map reduce function
  3. the map reduce function breaks the input data into splits.
  4. the mapper nodes in map reduce must then get all the data from there split to run, so it is best if try to map splits to blocks that are stored in that map workers file system
- **slido** what do you think will happen if a split is spread across multiple HDFS blocks?
- the program will be slow because there needs to be communication

splits and blocks

- a split is one logical division of the input data for a map process. each split typically has multiple rows
- the machine running the mapper must have access to the entire split so if some of the data for its split is not stored in that mappers file system data blocks have to move
- by default split size = block size but some fragmentation will happen as the input file does not always break evenly into splits of that block size

where should the program execute

- HDFS machines are kept in wracks. it is faster for a program to communicate with other machines within the same wrack
- best case we run the program on a node that stores the block data we need
- middle case we execute the program on a different node in the same rack. this is okay since within wrack communication is relatively fast
- worst case run the program on a node in a different wrack than where the block is held. between wrack communication is very slow

replication factors

- replicating block data on multiple nodes makes scheduling jobs more easy since it is more likely that a worker with our data will be available
- HDFS lets you set the replication factor for each file. (it is not free however since it takes us more storage space)
- typical replication factors is 3, keep 2 replicas in the same rack and 1 replica in a different wrack
- this set up means that pur data is protected against both node and wrack failure

CAP and HDFS

the CAP theorem for distributed storage

- the CAP theorem says a distributed file system can at most have two of the following

1. Consistency ie reading any file always produces the most recent value of that file
2. Availability ie request can not be ignored (the system can not go down)
3. Partition-tolerance ie the system maintains correctness during network failure

assume that we could have all three CAP traits

- suppose we have to machines that are both initialized with the value $x = 0$
- the network then fails
- a user then updates the value of x in machine 2 to $x = 1$
- a user than reads the value of x from machine 2
- what happens
- either the data is wrong ie the system does not have Consistency
- the system does not allow for the update of x to be made while down in which case Availability is lost
- or the system lets $x = 1$ get updated and then once the system is back up sets the value of $x = 1$ in machine 1, but in this case we do not have Partition tolerance
-
- so no matter what we violate one

slido

- textbfquestion which cap property is lost in HDFS and why
- HDFS has Consistency, there is a centralized name node that always has a consistent view of the file system data cna be appended but not modified
- it does not have Availability if our name node goes out of line the system shuts down
- Partition-tolerance kind of depends on how the specific HDFS network is set up and the replication factors

hdfs summary

- files divide into blocks and are replicated across the cluster
- checksums are replicated within each block
- name nodes allocate blocks and direct clients
- blocks are append only which means they are well optimized from write once read many tasks
- next we move onto Spark and spark-sql