MapReduce Implementation with Sort/Merge

Team Members:

- Danh Nguyen (nguy1952)
- Wen Chuan Lee (leex7095)

For a quick overview on how to run the project and test, please go to the **How to Run the Project section**, as well reading the **Client** section regarding the UI commands.

Overall Design

We designed the entire system atop Apache Thrift to perform map-reduce in a non-blocking manner. After considering all the possible failure scenarios this was the key to the design of a robust system that is able to continually function as long as the server is alive and there is at least one compute node.

On a high level, the Client first makes a request to perform a computation on a file listed in `data` on the server after connecting to the server. The client will then block until it receives the output file name containing the result. This is the only blocking call.

The assumptions on this entire system set up are that:

a. The server will not crash b. The all distributed processes have a shared/common filesystem on NFS (Networked File System) c. ComputeNodes can crash at any time (before, during or after executing/queuing a job) d. Only one client runs a compute job on the cluster at a time.

After a client makes a request, the server will then first perform an analysis on the data file based on the chunk size provided by the client to compute how many sort tasks have to be carried out. The larger the chunk size, the smaller the number of sort tasks to be carried out. The server then assigns these sort tasks to all compute nodes in a FIFO manner, and wait for all sorting to be complete. This is done by having the compute node perform an RPC call back to the server. Once all tasks (including any tasks that have been failed and reassigned with a heartbeat algorithm) are complete. The server then calculates the number of merges to be done based on the number of intermediate files, and the number of files per merge (provided by the client).

Merges are then assigned to the same compute nodes, and each compute node will perform n-way merging. **To ensure the system is able to merge large files**, we open each file as a special stream that is peekable. This ensures we do not have to read in all files in memory, reducing the possibility of running out of memory. By utilizing lazy streams into a priority queue we can ensure that we are able to merge up to `k` number of files.

Intermediate files in the `intermediate_dir` that have been successfully merged and sorted are deleted to save space and the final output is sent to the `output_dir`.

While we did not have to increase the amount of Java Heap Space allocated by default, this might be necessary if the chunksize exceeds the amount of memory available, as Sort tasks on each chunk are done in memory, while merges are not.

The Server

The Server acts as dispatcher to all of the compute nodes. When the client gives the server a job, it breaks it into 4 steps.

```
1. Processes the file into chunks and turning those chunks
into tasks that keep track of the offsets.
(The actual file is not modifed or broken up)
2. Distribute those sort tasks among all compute servers an
d then watch the queue for
any tasks that need to be redistributed because they went d
own. And when enough annoucements
have been called it moves on to the next step.
3. Create merge tasks and distribute the tasks.
4. Distribute those merge tasks among all compute servers a
nd then watch the queue for
any tasks that need to be redistributed because they went d
own. And when enough annoucements
have been called it finishes and returns the final file nam
e ot the server.
```

The key data structures in the server are the `TaskQueue` and the `InProgress` Map. The `TaskQueue` uses Java's `ConcurrentLinkedList` so that it supports concurrent operation by the `heartbeat` thread and the `server` thread, and the Thrift RPC thread pool. The `TaskQueue` is a queue of Task objects and that need to be sent to compute servers to process. The InProgress map is a mapping from machines to a list of tasks they are working on.

Since the heartbeat is not polling every milisecond there is a chance that if a node goes down the server could assign a task to it before the heartbeat recovers it from the system. To account for this the server actively wraps any calls to a compute node with a try catch. If any errors occur then it uses the same recover function that HeartBeat uses to recover the tasks and the node from the system.

HeartBeat

The heartbeat thread checks all running compute nodes to see if they are down. The heartbeat is a server thread and pings each compute node. When pinging the compute server if there is a error or exception that occurs then the node is down.

After it detects this, the heartbeat calls a recovery function on the node. What this does is take the node out of the system and reassigns the node's running tasks into the task queue. The heartbeat thread performs RPC calls to other alive compute nodes and reassign Sort/Merge tasks.

Compute Nodes

The compute node is responsible for processing the sort and merge tasks. It has two components; A thread responsible for maintaining the queue of tasks and a thread pool that puts tasks from the server into the server. The server makes RPC calls to the compute node to add Sort and Merge tasks into the request queue.

While maintaining the queue the `QueueWatcher` thread sees if there are any tasks to be executed and when there is, it starts a new SortMerge thread to handle it. The SortMerge thread will sort a file or merge k files depending the taks descriptions. When it is done it will announce back to the server so that the server can keep track of the progress.

The `Sort` and `Merge` RPC calls made by the server to the compute node do not actually perform the sorting and merging, rather they atomically add a sort/merge task to the task queue and complete the RPC call, ensuring that the Server does not block when assigning multiple tasks to compute nodes that can possibly fail.

Fault Injection

Faults are introduced into the compute server when it pops a task from the queue. Before actually creating a thread to run the task, it generates a random number to compare against the given chance to fail. If the number is less than the given chance to fail then it will call System.exit(). In Java when a thread calls this the entire process exits.

So the QueueWatcher is able to end execution of the Compute Node and all SortMerge threads.

The chance of a fault is accessed before a beginning to execute the task previously assigned to the `computeNode`. **The probability of a fault can be increased or decreased** by editing the `build.xml`, at the following parameter.

```
<property name="node.chanceToFail" value="0.01"/>
```

A `computeNode` that has been simulated to fail based on the random number drawn will perform a `System.exit(0)` without any clean up or notification, creating a `TException`. The `HeartBeat` thread will ensure tasks assigned to a particular node that has already failed to another ComputeNode.

If all nodes die before an entire merge sort operation is completed, the server aborts the job and mentions that nodes have died. The assignment description mentions that the main server should not fail and hence we did not introduce fault probability to the main server.

Client

- The Client is a terminal to the Server.

- The Client will establish a connection to the main server.

- If the Client is successful, an simple interactive terminal asking for user input is then launched.

- If the Client is unsuccessful, the Client will go to sleep for one second before retrying indefinitely.

- The terminal contains a few simple commands to interact with the File Server.
    - `sort <filename> <chunk size> <number of merges>` - performs a map-reduce merge sort on file `filename` with `chunk_size` byte chunk sizes and that are merged with `number of merges` number of files per merge.
    - `ls` - lists all files in the `data` directory
    - `exit` - closes the connection to the Server and quits the interactive terminal
- **Example command**: `sort 200000 200 12` tells the entire cluster to perform merge-sort on a file named `200000` with chunk sizes of `200` bytes, which are finally merged with `12` files per merge.

SortTask / MergeTask

We used object oriented programming principles in designing this framework by having an abstract class called `Task` and 2 classes `SortTask` and `MergeTask`. `SortTask` objects specify which file read as well as the start and end offset, while `MergeTask` holds a list of paths to intermediate files to merge. Both tasks have a unique filename assigned by the server. Through this method we can ensure that tasks that failed can be easily reexecuted and that files being written to are unique (avoiding any possibility of a race condition in a distributed system).

SortMerge

We start off a thread for every sort/merge task on the compute node, ensuring that we are able to perform concurrent sorts and merges on each compute node. We have a `QueueWatcher` thread that maintains a lock-free access on a concurrent linked queue within a `computeNode`. The QueueWatcher starts up a sort/merge task in an individual thread, and then performs an RPC call to the server to `announce` that the task has been completed. This ensures that we are able to perform multiple tasks concurrently. Since each task does not depend on the previous task like in most map reduce cases, each task is individually carried out.

Sorting is first done by advancing (`skip`) a BufferedReader to a specific postion of the data file (which is also being read concurrently by other `ComputeNode` processes, and reading in a specific number of bytes (based on the calibration and chunk size). After spliting the `character` buffer that is transformed into a `String` and parsing the entire `String` array into `Integers`, we sort the numbers (using Java's internal `Colections.sort` to ensure performance efficiency).

Merging is then done after all sorting is complete. After being assigned a `MergeTask`, merging is executed in another thread. Previously our design was to perform multiple merges in which only 2 intermediate files are merged each time as we did not completely understand the assignment goal. After rereading the assignment description, we realized we had to perform merging on any number of files, this is known as `k-way` merging. This was achieved by using a special form of a scanner that is able to read the next number in a scanner (stream), in a lazyway, and exploiting a Priority Queue that was able to automatically order each stream from smallest to largest.

Merging is then done by writing out the smallest number in the heap of a Priority Queue and repeated until a stream no longer has anymore numbers before it is discarded. The resulting file is then announced to the server.

How to Run the Project

The entire Map Reduce project was built with the help of Ant. A prerequisite to run this project requires Ant. Additionally, the generated thrift files were also kept in a version control, but the `Thrift` compiler is also needed if Thrift files need to be generated (else only the Java Thrift library is needed).

Ant was used to handle automatic building and generation of class files (into a separate `bin` directory) as well as creating short targets for rapid developement.

The compute nodes connect to the server to enroll into the MapReduce server located at `localhost` on port `9090`, **to override this default, on Step 6 do**:

```
ant -Dserver.address='x32-XX' -Dserver.port='XXXX' start-no
de
```

Please see Ant Targets and Overriding Properties for more information.

To start the Coordinator, Servers, and Client on localhost

(This starts all processes on one terminal using forked processes and will cause all processes to print to the same `System.out`, which would cause very messy output as all threads attempt to print to `stdout` in an unsynchronized way.)

```
ant start-all
```

To start the File Server across multiple machines

```
1. ssh username@x32-05.cselabs.umn.edu
2. cd into project directory
3. ant start-server
4. ssh username@x32-XX.cselabs.umn.edu (open another termin
al)
5. cd into project directory
```

```
6. ant start-node
7. Repeat steps 4 - 6 for the other Nodes
8. ssh username@x32-XX.cselabs.umn.edu (or open another ter
minal)
9. cd into project directory
10. ant start-client
```

Example of running the entire project across different computers.

On the root directory where `build.xml` is located.

```
1. (Machine A) `ant start-server`
2. (Machine B) `ant -Dserver.address=IP_MachineA start-node
`
3. Repeat step 2 for other machines.
4. (Machine Z) `ant -Dserver.address=IP_MachineA start-clie
nt`
5. `ls`
6. Performing a sort on any of the files, like `sort 200000
 200 4`
```

You can edit the fail probability in `build.xml` under `<property name="node.chanceToFail" value="0.01"/>` where `0.01` equals to 1.0 percent. (`0.0061` equals to 0.061 percent).

Ant Targets

The fastest way to test out the entire DHT is to create 1 SuperNode and multiple Nodes on the same machine (but with a different port). Multiple targets have been provided for automating the process of starting everything.

Note: Run commands from the project directory, where `build.xml` is localed.

- `ant start-all`
  - Create a MapReduce server (port 9090) and 10 Nodes and a Client on the same machine.
  - All processes run in their own Java VM and as a forked process.
  - Client waits for 4 seconds before connecting the MapReduce server
  - Note: see Ant Properties on how to start targets with different values.
- `ant start-server`
  - Starts the coordinator on the current machine on the port specified by `server.port`
- `ant start-node`
  - Starts a compute node that will connect to the server

located at `server.address` and `server.port`.

- `ant start client`
    - Starts the client that connects to the server specified under `server.address` and `server.port`
- `ant thrift`
    - Fires up the thrift compiler to generates Java versions of the `.thrift` files located in `src` and saves them to `src/gen-java`.
    - This target should be called everytime the new *thrift* interfaces have been defined pr to update to a newer version of *thrift*.
    - Targets must be modified if additional *thrift* files are created, as I could not find a way for *Ant* to pass files individually to the *Thrift* compiler without introducing Ant plugins or Maven.
- `ant build`
    - Builds but doesn’t run the project. Note all targets that run depend on this target, so there is no need to run `ant build` before running the target.
- `ant clean`
    - Deletes all compiled bytecode in `bin`
- `ant diagnostics`

- Prints out the Java Class Path defined in *master-classpath*

Ant properties (and overriding them)

Many of the above targets depend on some known information about the SuperNode and the ports to connect to. Sometimes these constants should be overridden (for example when starting multiple nodes/clients on the same machine). **To Override these constants**, use the format:

```
-D(propertyName)=(value)
```

To start the Server with a minimum of `10` nodes:

```
ant -Dserver.port=5050 start-server
```

With a different port number `5555`:

```
ant -Dserver.port=8 -Dserver.port=5555 start-server
```

Alternatively open up `build.xml` and edit the values:

```
 <!--Default Coordinator and Server properties used unless
provided-->

 <property name="server.address" value="localhost" />
 <property name="server.port" value="9090"/>

```

```
<property name="node.port" value="5000" />

<property name="src.dir" value="src" />
<property name="build.dir" value = "bin"/>
<property name="thrift.install" value="/usr/local/Thrift/"
/>

<!-- CHANGE THESE VALUES -->
<property name="node.chanceToFail" value="0.01"/>
```

Additional point to note

At times the target `ant start-all` may be problematic as there is a race condition in the `javac` compiler and JVM starting multiple instances of ComputeNodes at the same time. Ant attempts to rebuild classes it finds that needs rebuilding, while some threads begin to execute a process before it has completed. While artificial delays have been introduced between starting each compute nodes, at times the JVM is unable to start a class (with exceptions like `ClassFormatError`). The only known way to resolve this is to run `ant start-all` again. This does not occur if instances are started manually or across different machines.

Performance Results

Stress Testing

We performed stress testing across 15 compute nodes, each on a differenct virtual machine. The tests were done using lots of tasks (small chunksize compared to filesize) and few tasks (large chunksize).

The tests were performed on the largest sample file. Because if the entire cluster is able to handle a sort merge on the largest sample file, we are able to deduce that the system is robust enough to handle smaller files. Tests with a probable failure was also carried out in Performanace testing and Fault testing.

The raw data collected from stress testing can be found in `results/StressTestingData.txt`.

Parameter for stress test 1 (Large chunksize)

`sort 20000000 500000 4`

Where we are sorting the file 20000000 (of approx 98 mb) with a chunksize of 500000 (500kb) with 4 intermediate files per merge.

This yielded a total of 196 sorts, and then 67 merges in total. The total runtime for this was 67303ms (~67 seconds).

Parameter for stress test 2 (Large number of chunks, small

chunks size)

`sort 20000000 5000 4`

Where we are sotring the same file of 98 mb with a chunksize of 5000 bytes (5kb) and a total of 4 intermediate files per merge.

This yielded a total of 19548 sorts and a total of 1633 merges. The total runtime was 262814 ms (262 seconds, or ~4.5 minutes).

Parameter for stress test 3 (Large number of merges on large chuhnks)

`sort 20000000 500000 59`

Where we are sorting the same file of 98 mb with a chunksize of 500000 bytes (500kb) with 59 intermediate files per merge.

This yielded a total of 196 sorts and 5793 merges. The total runtime was 301657 ms (301 seconds, 5 minutes).

Analysis and conclusion

In conclusion, with these stress tests, our Map Reduce program does not fail on large chunksizes as well as large file sizes, or with a large number of sort tasks. We found that the larger the number of sort tasks to perform, the longer it takes as more network RPC calls.

We have proven that our system can handle large number of requests and large number of files, hence as it works at scale, it will handle smaller files and chunks just fine. We avoided increasing the Java Heap space with the given test files, this may be required if the chunksize exceeds the default amount of memory available to the JVM.

Performance Testing

Due to time constraints we perform tests on the 20mb (half one), we can assert that if it works on larger files it can work on smaller files We chose the percentages based on the a ratio we wanted of nodes failing. This was performed over 10 computeNode instances on different machines.

Fault % represents: Perfect run, at least 1-2 servers faulting , and half the servers faulting.

`sort 2000000 100000 2 0%` `sort 2000000 100000 2 1.3%` `sort 2000000 100000 2 3.3%`

`sort <filename> <chunkszie> <no merges> <fault percentage>` `sort 2000000 100000 50 0%` `sort 2000000 100000 50 2%` `sort 2000000 100000 50 5%`

Fault Testing

```
Max acceptable bound for fault tolerance.
NumTasks * % Chance to Fail = Number of Servers that will f
ault.
```

In our testing we will be testing the above hypothesis with these scenarios. F = Chance to fault. N = number of servers. T = number of tasks.

1. T * F = 2N
2. T * F = N
3. T * F = .5 N
4. T * F = 1/15 * N

With this we will figure out a acceptable fault rate.

Results

Please refer to “results/FaultTestingData.txt” for raw data.

Looking at the data someone who is concerned about the longevity of the computing cluster should figure how the average amount of tasks that are ran in a day,minute,second…etc. After that is determined a person is able to use that number of tasks and figure out what would be a acceptable fault percentage in their system.

In our tests we’ve found that the acceptable fault percentage to have is typically safer when T * F < N. Where T is the number of tasks and F is the fault percentage and N is number of servers. This is because if there are statistically more faults than there are nodes there wont be any nodes left to do any sort or merge tasks.

This leaves operators a couple options when trying to lessen risk. Either decrease the amount of tasks done, or increase the amount of servers. Increasing and decreasing the fault rate would be harder in a real life situation.

Unit Testing

Here are a few tests we perfomed during the development of this project, we have considered and accounted for these possibilities.

Scenario : Data file not found

Description: The client submits jobs to the server indicating the filename of a data file, which may not exist in `data/`

Solution: The client and server both actively check if the file exists before performing the entire merge sort operation notifying all files.

Scenario : Race conditions overriding filenames

Description: Assigning intermediate output filenames to the compute nodes for each task might cause files to be overridden if multiple threads attempt to increment a unique counter.

Solution: Ensure only one thread increments the unique counter for filenames, ensuing each ComputeNode instance is able to write out to a unique file.

Scenario: Number of short tasks exceeding the unique data type.

Description: We initially used `int` primitive types as the atomic counter for unique intermediate filenames. This number could be very large and overflow (i.e only 65535 unique counters can be created in a `char` type)

Solution: To avoid any possibility of overflowing and overwriting intermediate files we decided to use a long counter, effectively increasing the maximum number from 2147483647 to 9223372036854775807, eliminating any chance of overflow.

Scenario : ComputeNode instance is killed/dies when enrolling with the server.

Description: A computenode dies or loses connection to the network when enrolling into the cluster.

Solution: HeartBeat thread will remove unresponsive nodes. This case is accounted for.

Scenario : ComputeNode instance is killed/dies while server is assigning tasks.

Description: A compute node dies or loses connection to the server when the server is assigning `Tasks` (Merge tasks or Sort Tasks) to the nodes.

Solution: The HeartBeat threads checks to see if the node is still reachable and functional at regulat intervals, and performs recovery by reassigning tasks to other healthy nodes, tasks are all kept track off and node jobs are accounted for in the server.

Scenario: ComputeNode instance crashes when merging intermediate (sorted files) Scenario: ComputeNode instance crashes when sorting intermediate files. Scenario: Forced exit of a compute node when merging and sorting files.

Description: ComputeNodes can fail when performing jobs that require intensive computation.

Solution: Handled by the heartbeat thread as well.

Scenario: ComputeNodes runs out of memory when merging/sorting

Description: Due to hard limits set by Machines in CSELabs, performing MergeSort creates a large number of intermediate files and an output file, having a possibility of hitting the 1GB quota.

Solution: Intermediate files are deleted upon successful completion of jobs (after the ComputeNode has successfully announced this to the

Server). Deleting additional files on the account also reduces this possibility.

Scenario : All Compute Nodes die before the completion of an object

Description : When increasing the fault probability, there is a high chance of all compute nodes on the cluster crashing.

Solution : The server notifies the job has failed and mentions that no more healthy compute nodes are in the cluster, and then exits.

Scenario : Client begins another Merge Sort after a job has been completed.

Description : The Server previously kept leftover state after a job has been completed, causing issues when another job is started.

Solution : The server has a `cleanup()` function that ensures upon successful completion of each job it is kept stateless, as with compute nodes.

Scenario : User inputs bad parameters/ bad input for commands

Description : The entire system would throw an exception if a user does not entire perfect parameters.

Solution : The client sanitizes and performs checks before submitting the job, ensuring any user-input problems are resolved client side.
