CS 6381 Final Report - Apache Cassandra Team:

1. Overview
2. Experiment Criteria and Design

We created a set of applications to test the read-write efficiency of the Apache Cassandra database based on our overall understanding of the experiment designers’ guidelines and the time available. Each set of applications connects to a created keyspace within the Cassandra database before interactions with the database are made. On Ubuntu 18.04, we installed Cassandra and ran it using the “sudo service cassandra start” command, checked its active status using the “sudo service cassandra status” command, and restarting by running “sudo service cassandra stop” and the start command as mentioned here. With this, we were able to run our sets of applications.

One set of applications revolves around calling read and write queries to a table in the Cassandra database we called table1 using the Python file, cassandraRW.py. Each write query inserts a record to the database that contains a string parameter, message, which consists of randomly generated characters to a specified length that is supplied to the function call. Each read query selects 1 record from the database. Time differences are saved to a CSV with the title including the indication of read or write, number of iterations, host name and process ID, which are obtained using terminal commands, current timestamp, and UUID to distinguish them apart. We ran tests using shell files to call these queries a number of times based on the following conditions: Each reader and writer works independently (neither have to wait for the other to finish to start running). A write query collection contains 5 write queries that vary in terms of total character lengths as follows: 128, 254, 512, 1024, and 10000.

Our expectations for running the tests are as follows: For the 1writerTest, we run 1 write query collection once. For the 1readerTest, we run 1 read query once. For the 10writer1readerTest, we run 10 write query collections then 1 read query, 10 times. For the 1writer10readerTest, we run 1 write query collection then 10 read queries, 10 times. For the 25writer25readerTest, we run 25 write query collections then 25 read queries, 5 times.

We tried to implement a set of applications to test the CAP analysis efficiency of Apache Cassandra, but we fell short likely due to unstable connection to Cassandra. The applications revolve around capAnalysis.py and testing the efficiency of readers based on sample text files of varying character lengths. There are 5 sample files, each of which is named “sample” concatenated with a number from 1 to 5, followed by the “.txt” file extension. Among the queries, there are various parameters as follows: payload refers to the text from the sample file and recordIndex refers to the numerical value in the sample file name. There are 3 different queries, insert, update, and read, each of which is accessed by a URL request and involves looping through a folder directory to find a sample file with matching numerical value as that specified by the provided recordIndex. insertQuery() accepts a payload and a recordIndex, appends the payload to the current timestamp as a new message, and inserts the message into the database with the id equal to the recordIndex. updateQuery() acts like insertQuery() except the difference is it updates the message of the record with the id that matches the given recordIndex. readQuery() accepts a recordIndex, notes the timestamp before selecting for a record from the table where the id matches the recordIndex, gets the timestamp after, splits the message to get the timestamp from the record, and returns this information along with the string, “reader”, concatenated to the recordIndex. Time differences are saved to a CSV with the title including the number of iterations, host name and process ID, which are obtained using terminal commands, current timestamp, and UUID to distinguish them apart. The tests we run revolve around readIteration.py. We run capAnalysis.py as a server that can accept incoming URL requests. Next, we run “http://localhost:5000/u/1” to keep the write query running on the server. Then we run readIteration.py with an argument of 10 to get 10 iterations of send URL requests to the server to run read queries. We do this 10 times.

We used calculateStatistics.py to read the data from the time differences CSV files and save a CSV filled with statistics and calculateStatisticsAll.py for similar purposes with the additional feature of generating graphs based on the calculations. Here, we set a for-loop to check for CSV files in a folder. When there’s a match, we call readCSV(), which takes in the file and opens it, iterates through it, and appends the write and read times to lists as well as write and read success rates, which are based on how many time rows there are divided by the total iterations for a file. For latency and throughput, using this collected information, we get the minimum and maximum values for times by using the min() and max() functions respectively, average by summing the tuple-converted data list of float-converted times and dividing that by the length of the data list, and standard deviation by getting the variance for the result of dividing the sum of the data list by the length of the data list and setting it to the power of 0.5. These findings are added to a calculations CSV file with the current timestamp and UUID in the title to distinguish the files apart. calculateStatisticsAll.py featured generating graphs by passing in the file name that is parsed from the path name. Using matplotlib and pandas, graph settings are configured with the X-axis showing the various test conditions and the Y-axis showing the values for the parameter of interest as graphs are generated.

We used hardwareMeasure.py to continuously see the latest hardware statistics, including CPU, RAM, and storage. In a “while True” loop, we printed the functions that return a dictionary including the max average values for each of these parameters. In each function, we appended the parameter fetching using psutil to its own list and from that list, we find the maximum value using the max() function and average value by summing up the tuple-converted data list and dividing that by the length of the list.

1. Experiment Results

The following includes our observations of trends among the graphs, which are stored in the “graphs\_ApacheCassandra1” folder. In terms of average, max, min, and standard deviation for read latency and throughput and write throughput, all results appear high around the same range in the graphs. Upon closer observation, the lowest yields come from the 1st, 2nd, and 10th 10writer1reader tests, the moderate yields come from the 3rd to 8th 10writer1reader tests, and higher yields come from the rest of the conditions. For write latency, the exception is the min, where the 1st 10writer1reader test result yields the highest result, followed by the 2nd, 3rd, 4th, and 10th, followed by the 5th, 6th, and 7th, and the rest of the conditions yield lower results around the same range. From these observations, it seems that Apache Cassandra, especially when there are more readers, performs well in throughput and has increased latency. An exception occurs at least in our case as generally when there are more readers, the write latency is lower. We think one possibility for increased latency could be due to our connection issues to Cassandra.

For the base file, it appears the hard disk memory stays roughly stable while the readIteration.py test is run. RAM decreases and CPU increases from the 1st to the 2nd load. The computer we used had other activities going on outside of the VM, so perhaps this contributed to the RAM and CPU fluctuations. The computer has a lot of hard disk memory, so changes to the memory should appear minimal as long as large-sized items are not added or removed.

1. Conclusions
2. References
   1. https://github.com/jamesvu1/cs6381finalprojectapachecassandra