EE542 - Reading Assignment – 07

Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark

Presenter: Boyang Xiao

USC id: 3326-7302-74

Email: boyangxi@usc.edu

Index

Introductions and background

- Hadoop introductions and suspicions
  - Hadoop is a very popular general purpose framework for many different classes of data-intensive application, which provides an open source implementation of MapReduce
  - Hadoop is not good for iterative operations because of the cost paid for the data reloading from disk at each iteration
- Spark introductions and suspicions
  - Spark, which is designed to have a global cache mechanism, can achieve better performance in response time since the in-memory access over the distributed machines of cluster will proceed during the entire iterative process
  - Although Spark can achieve tremendous speedup, it is suspected that it requires high memory cost for the introduction of RDDs

System overview

- System overview for Hadoop
  - Hadoop is composed of two layers
    - HDFS : a data storage layer called Hadoop distributed file system
    - MapReduce: a data processing layer called Hadoop MapReduce Framework
  - HDFS is a block-structured file system managed by a single master node. A processing job in Hadoop is broken down to as many Map tasks as input data blocks and one or more Reduce tasks. An iterative algorithm can be expressed as multiple Hadoop MapReduce jobs.
- System overview for Spark
  - Spark is a novel cluster computing framework that is designed to overcome Hadoop's shortages in iterative operations
  - Spark uses Hadoop supported storage systems (e.g. HDFS) as its input source and output destination
  - Spark introduces a data structure called resilient distributed datasets (RDDs) to cache data

Experiment environment

- Experiment settings
  - Cluster size: 8 computers
  - OS: Ubuntu 12.04.2 (GNU/Linux 3.5.0-28-generic x86 64)
  - Network switch: H3C S5100 w/ port rate 100Mbps
  - Cluster manager: Apache Mesos 0.9.0
  - Hadoop ver. 0.20.205.0
  - Spark ver. 0.6.1
- Experimental data
  - five real graph datasets
  - five generated synthetic graph datasets to do comparative experiments

Experiment implementations

- PageRank
  - The basic idea behind PageRank is that a node which is linked to by a large number of high quality nodes tends to be of high quality.
  - PageRank of node n:

$$P(n) = \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

  - Two problems to address when benchmark with PageRank:
    - **Spider trap**: which are group of nodes that have no links out of the group. This can be solved by adding a random jump factor α to the formula above
    - **Dangling nodes:** which are nodes in the graph that have no outlinks. This can be solved by redistributing PageRank mass "lost" at dangling nodes across all nodes in the graph evenly.

Experiment implementations

- PageRank on Hadoop
  - The basic process of each PageRank iteration can be divided into two Hadoop MapReduce jobs.
  - In the first job, each node m first divides its current PageRank value P(m) evenly by C(m), the number of nodes m links to and passes each share to them separately.
  - Before the second job, the total PageRank loss l at dangling nodes needs to be calculated.
  - In the second job, each node n updates its current PageRank value P(n) to P(n) according to formula below:

$$P(n)' = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\left(\frac{l}{|G|} + P(n)\right)$$

Experiment implementations

- PageRank on Spark
    - All the iterative operations can be implemented in a Spark driver. Each iteration includes four steps.
    - The first step distributes the PageRank value of each node to its neighbors using the join and flatMap transformation.
    - The second step does PageRank aggregation for each node using reduceByKey transformation.
    - In the third step, PageRank loss at dangling nodes is computed.
    - The fourth step deals with PageRank loss distribution and random jump factor

Experiment results

- PageRank Scalability
  - When datasets are smaller than kronecker22, the PageRank implementations exhibit near-linear scalability on both Hadoop and Spark whether the graphs are real or synthetic
  - When datasets are larger than kronecker22, slope of Spark line is greater than that of Hadoop line which means that the performance advantage of Spark becomes smaller as the graph size increases
- Running Time Comparison
  - When dataset is too small, Spark can outperform Hadoop by 25x-40x. In this case, the computing resources of all the slaves are underutilized.
  - When dataset is relatively small, Spark can outperform Hadoop by 10x-15x. In this case, the computing resources of all the slaves are moderately utilized.

Experiment results

- Running Time Comparison (cont'd)
  - When dataset is relatively large, the advantage of Spark becomes small and Spark can only outperform Hadoop by 3x-5x. In this case, for Hadoop the memory of each slave in the cluster is still moderately utilized while for Spark the memory of each slave in the cluster is full which result in Spark's reduced performance advantage.
  - When dataset is too large, e.g., kronecker23, Hadoop beats Spark. In this case, for Hadoop, the memory of each slave in the cluster is fully utilized while for Spark the memory of each slave in the cluster is not enough for PageRank running
  - When dataset is extremely large, e.g., Twitter, PageRank on Spark crashes halfway with JVM heap exceptions while PageRank on Hadoop can still run

Experiment results

- Memory Usage Comparison
  - For Hadoop platform, when dataset is fixed, memory usage exhibits periodically increase and decrease with the increase in the number of iterations. There is also a gradual increment between two consecutive iterations
  - For Hadoop platform, when iteration number is fixed, the shapes of memory usage plots are similar with the increase in the size of dataset
  - For Spark platform, when dataset is fixed, memory usage only exhibits periodically gradual increase with the increase in the number of iterations
  - For Spark platform, when iteration number is fixed, the shapes of memory usage plots are similar with the increase in the size of dataset.

Q&A

- **Questions:** What are the two typical problems when using PageRank on Hadoop and how to solve them?
- **Answer:**

  The two typical problems are:

  - **Spider trap**: which are group of nodes that have no links out of the group. This will cause the PageRank calculation to place all the PageRank within the spider trap.
  - **Dangling nodes:** which are nodes in the graph that have no outlinks. The total PageRank mass will not be conserved due to dangling nodes.

  To solve them respectively:

  - **Spider trap** can be solved by adding a random jump factor α to the formula above
  - **Dangling nodes** can be solved by redistributing PageRank mass “lost” at dangling nodes across all nodes in the graph evenly.

Thanks for watching!

Presenter: Boyang Xiao

USC id: 3326-7302-74

Email: boyangxi@usc.edu