The Implementation of Hadoop-based Crawler System and Graphlite-based PageRank-Calculation In Search Engine
郭清沛, 201428015029038, ISCAS
徐超, 2014E8015061086, ISCAS, 宋扬, 2014E8015061082, ISCAS

Abstract
Nowadays, the size of the Internet is experiencing rapid growth. As of December 2014, the number of global Internet websites has more than 1 billion and all kinds of information resources are integrated together on the Internet, however, the search engine is to be a necessary tool for all users to retrieve useful information from vast amounts of web data.

Generally speaking, a complete search engine includes the crawler system, index building systems, sorting systems and retrieval system. At present there are many open source implementation of search engine, such as lucene, solr, katta, elasticsearch, solandra and so on. The crawler system and sorting system is indispensable for any kind of search engine and in order to guarantee its efficiency, the former needs to update crawled vast amounts of data and the latter requires real-time to build index on newly crawled web pages and calculate its corresponding PageRank value. It is unlikely to accomplish such huge computation tasks depending on a single hardware implementation of the crawler system and sorting system, from which aspect, the distributed cluster technology is brought to the front. In this paper, we use the hadoop Map - Reduce computing framework to implement a distributed crawler system, and use the GraphLite, a distributed synchronous graph-computing framework, to achieve the real-time computation in getting the PageRank value of the new crawled web page.

Key Words: Hadoop; Crawler System; Graphite; PageRank; Search Engine

1. Introduction
1.1 Framework of Hadoop Map-Reduce
The Map - Reduce is a programming model based on Hadoop. In the Map - Reduce distributed computing framework, the programmer only writes a serial program and ensure the correctness of the serial program and then the system will complete the execution in a parallel and distributed way, which is transparent for programmers. The hadoop-based distributed computing framework is as shown in the figure 1.1.1 below:

```
| Application | ······ | Application |
|-------------|-------|-------------|
| Map/Reduce API |
|               | HDFS  |
| Node         | ···   | Node        | ···   | Node |
```

Figure 1.1.1 Hadoop Map-Reduce Framework
The lowest layer is a cluster composed of many physical nodes, each Node in the cluster is divided in logic, and the implementation of each node is just a running process so that multiple nodes can be distributed in one or more physical hosts. HDFS
and MAP - REDUCE tasks run on the cluster. HDFS defines a NameNode, usually with a Secondary NameNode for redundancy backup, which are commonly responsible for storing metadata and data backup, other DataNodes are responsible for the specific file operations such as reading and writing. The Map - Reduce tasks need to run on HDFS for sharing data between different physical host nodes and storing intermediate results. When a user submits a Map-Reduce task, the Map - Reduce framework can decompose a task into subtasks and assign them running on corresponding nodes in cluster. In such a way distributed computation is achieved by programmers without caring about any specific distributed implementation details.

### 1.2 Framework of GraphLite

GraphLite uses a called BSP (Bulk Synchronous Processing) programming model. As shown in figure 1.2:

![BSP programming model](image)

In GraphLite framework, the computation on a graph will be divide into multiple supersteps. Between two supersteps are distributed computation without any reliance, in such a way, the goal of "general serial, parallel partial" is achieved. All dependencies of nodes operation in GraphLite are classified as "data dependence" and "temporal dependence", the former can be solved through the message-sending mechanism during the initial stage of a superstep, "temporal dependence" can be resolved by a serial of sequenced SuperSteps. At the end of each SuperStep, GraphLite will collect the messages sent by all the nodes, and sending them to the corresponding destination nodes before the next superstep begins, then starts the next round of distributed computation.

### 2. Our Implementation

#### 2.1 Hadoop-based Crawler System

##### 2.1.1 How we use Map-Reduce In Our Crawler-System

We use the Map-Reduce framework to implement the distributed crawler system as shown in figure 2.1.1:
CrawlerTask is a text file that stores the seed urls to be crawled. InputFormat is the pretreatment process before performing the of the Map operation, in which the data file will be cut into small shards, each we call it a InputSplit, defaulted with the size of 64M. Each of the InputSplits will be analyzed to a pair of <key,value>. The key of every <key,value> pair outputted by InputFormat is the starting offset of each line, while the value is the URL to be crawled.

In Map process, The Mapper class input is formatted as a set of <offset,url> pairs that analyzed by InputFormat. In our system what implemented by Map function is swapping the value of key and value in input key-value pairs. We set the url of each input-pair as the output key, whose value dedicate the extracted crawling URL. While the output value is offset. The result will be written into the intermediate files, which exists in the HDFS.

Given the thought that the overhead of communication between the nodes in Hadoop Cluster usually costs much in efficiency. We combine each Mapper output in Combiner stage of temporary files in the repeat key on the local merging, so we can reduce the amount of traffic between nodes and reduce the pressure of subsequent reducers.

What implemented in the Partitioner process is partitioning the intermediate results. According to the value of the result-key, the results could be divided into R intermediate results after the combiner process, each will be sequently processed by a Reducer. The partition algorithm we use is aimed at calculating the hash value of each URL corresponding to the host so that the URLs belonging to the same host will be partitioned into the same bin, which will be then processed by the same Reducer. So that the same host URL will be crawled exactly on the same machine. In the Reduce phase, the URLs will be used to multithreaded downloading and the crawled web pages will be written in the HDFS.

### 2.1.2 Our Running Results

The crawled results based on Map-Reduce framework is stored in the HDFS, files of which are distributed saved at different host nodes. In order to display the results conveniently, we use a database visualization tool connected HDFS to display the crawled data. Figure 2.2 shows the results crawled by our distributed Crawler-System, and Figure 2.3 roughly display the specific content of a crawled web-page.
Distributed Crawler-System is running on the Hadoop cluster, each node in the cluster are definitely a centralized crawler, controlled by a master node to work together, so the efficiency of Distributed Crawler-System is much higher than the centralized Crawler-System. In our System, We adopt three nodes, each node of the reducer crawling with 16 threads, 48 threads totally. Experimental results show that with the system running for 30 minutes, the size of web page stored on HDFS data is 872M (HDFS data redundancy backup number is set to 1). The single machine 16 threads run 30 minutes crawl web data of about 300M, which shows that the distributed crawler performance is much better than single node.

2.2 Graphlite-based PageRank-Calculation

2.2.1 our method to calculate PageRank

In the search engine, each web page newly crawled by Crawler-System needs to be real-time calculated its weight among all web Pages in Ranking-System, namely PageRank, according to the number of up and down links. PageRank reflects the importance of web pages, which is critical for improving the user search-experience. Here we use GraphLite, a distributed System for large-scale graph processing to calculate PageRank.
In our Crawler-System, after de-emphasis of each page, we will assign a unique Id for each of them. Each page is deemed as a vertex in graph computation System. The formula to calculate each vertex is as follows:

\[ R_v = 1 - d + d \sum_{u \in B(v)} \frac{R_u}{L_v} \]

Among them:
- \( R_v \): PageRank * N of vertex N
- \( L_v \): in-degree of vertex V
- \( B(v) \): out-degree of vertex u
- \( d \): web links Probability
- \( N \): the number of all pages

In the program, we initialize the weight of all the pages as 1.0, and perform iterate computation until the result comes to convergence. We enabled four worker nodes in our program, each vertex is assigned to four worker nodes based on the result of its vertex Id mod 4, the organization format of the input file is shown as Figure 2.2.1.1. Take the input data of Worker3 for example, the input data format is shown in Figure 2.2.1.2.

Figure 2.2.1.1 shows the organization format of the input file, Figure 2.2.1.2 shows the input data format for each worker.

In Figure 2.2.1.2, the number 1010 in the first line represents that this worker's entry page nodes is 1010, the 21037 in the second line represents that output edge associated with entry page node amount to 21,037. Each row of data from the beginning of the third line represents an edge, for example, "2 20" represents the page, the starting id of which is 2, and the end Id is 20.

The main function of each worker is as follows:
class VERTEX_CLASS_NAME(): public Vertex <double, double, double>
{
    public:
    void compute(MessageIterator* msgs) {
        double val;
        /*
         * initialize all web pages weight as 1.0
         */
        if (getSuperstep() == 0) {
            val = 1.0;
        }
        else {
            if (getSuperstep() >= 2)
            {
                double global_val = *(double*)getAggrGlobal(0);
                if (global_val < EPS)
                {
                    /*
                     * if all accumulated PageRank biases are in EPS, then stop computing.
                     */
                    voteToHalt();
                    return;
                }
            }
            double sum = 0;
            for (; !msgs->done(); msgs->next()) {
                /*
                 * sum up all neighbors' contributes
                 */
                sum += msgs->getValue();
            }
            /*
             * dumping factor are set as 0.85
             */
            val = 0.15 + 0.85 * sum;

            double acc = fabs(getValue() - val);
            accumulateAggr(0, &acc);
        }
        *mutableValue() = val;
        const int64_t n = getOutEdgeIterator().size();
        sendMessageToAllNeighbors(val / n);
    }
};
2.2.2 Our Running Results

During the cluster initializing state, the master node distribute all of the data to the 4 workers, as is shown in Figure 2.2.2.1.

```
hadoop@ubuntu:~/GraphHite$ start-graplite example/PageRankVertex.so ~/GraphHite/ 
/home/hadoop/GraphHite/engine/graphHite 0 /home/hadoop/GraphHite/engine/start-worker 
/home/hadoop/GraphHite/example/PageRankVertex.so /home/hadoop/GraphHite/Input 
/tinygraph_4w /home/hadoop/GraphHite/Output/out 
master run 
parseCmdArg 
loadUserFile 
startWorkers 
worker 1 cmd: ssh localhost '/home/hadoop/GraphHite/engine/start-worker /home/hadoop/ 
/graphHite/engine/graphHite 1 /home/hadoop/GraphHite/example/PageRankVertex.so 
/home/hadoop/GraphHite/Input/tinygraph_4w /home/hadoop/GraphHite/Output/out 1' 
worker 2 cmd: ssh localhost '/home/hadoop/GraphHite/engine/start-worker /home/hadoop/ 
/graphHite/engine/graphHite 2 /home/hadoop/GraphHite/example/PageRankVertex.so 
/home/hadoop/GraphHite/Input/tinygraph_4w /home/hadoop/GraphHite/Output/out 2' 
worker 3 cmd: ssh localhost '/home/hadoop/GraphHite/engine/start-worker /home/hadoop/ 
/graphHite/engine/graphHite 3 /home/hadoop/GraphHite/example/PageRankVertex.so 
/home/hadoop/GraphHite/Input/tinygraph_4w /home/hadoop/GraphHite/Output/out 3' 
worker 4 cmd: ssh localhost '/home/hadoop/GraphHite/engine/start-worker /home/hadoop/ 
/graphHite/engine/graphHite 4 /home/hadoop/GraphHite/example/PageRankVertex.so 
/home/hadoop/GraphHite/Input/tinygraph_4w /home/hadoop/GraphHite/Output/out 4' 
init 
manageSuperstep 
Receiver: accept all client success 
received WM_BEGIN 
MW_BEGIN: 1 
step into sendall 
```

Figure 2.2.2.1 initialization of the cluster

Figure 2.2.2.2 shows that the program converges when proceeding to the 19th step. Computing 4039 nodes, 88,234 edges of a directed graph on Ubuntu 64bit 3.2 GHz dual-core 4 thread machine costs only 4.56s. It is almost single-node operation consuming 1/3! Though it seems still a little time consuming for real-time computation, but it’s obvious that we can use more machines in our cluster to achieve faster computation. It is shown that the use of distributed computing that can improve the efficiency of PageRank efficiency greatly, and the use of a distributed computing architecture experiment can greatly reduce the requirements for in-memory of a single node during the computation process.

```
superstep: 19 
received WM_CURSSFINISH 
MW_END: 4 
step into sendall 
sent MW_END to worker[1] 
sent MW_END to worker[2] 
sent MW_END to worker[3] 
sent MW_END to worker[4] 
sent MW_END 
received WM_END 
terminate 
Receiver: closeAllSocket 
Sender: closeAllSocket elapsed: 2.574161 
```

Figure 2.2.2.2 Running Results
3. Conclusion

In this paper, we design and implement two distributed systems to solve the real-time problem of big data processing in search engine. The Hadoop-based Crawler System and Graphlite-based PageRank-Calculation System running on a cluster are both proven highly effective than a single machine in big data processing and can be used in real Industrial production environment as a solution.

References

【1】薛振华，杨艳娟. 分布式搜索引擎的研究[J]. 中国搜索研究中心,Vol.18,1999
【2】Luiz André Barroso , Jeffrey Dean , Urs Hölzle, Web Search for a Planet: The Google Cluster Architecture, IEEE Micro, v.23 n.2, p.22-28, March 2003 [doi>10.1109/MM.2003.1196112]
【3】Sanjay Ghemawat , Howard Gobioff , Shun-Tak Leung, The Google file system, Proceedings of the nineteenth ACM symposium on Operating systems principles, October 19-22, 2003, Bolton Landing, NY, USA [doi>10.1145/945445.945450]
【4】Jeffrey Dean , Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, p.10-10, December 06-08, 2004, San Francisco, CA
【5】Mike Burrows, The Chubby lock service for loosely-coupled distributed systems, Proceedings of the 7th symposium on Operating systems design and implementation, November 06-08, 2006, Seattle, Washington
【6】Sergey Brin , Lawrence Page, The anatomy of a large-scale hypertextual Web search engine, Proceedings of the seventh international conference on World Wide Web 7, p.107-117, April 1998, Brisbane, Australia
【7】Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed Web crawler. In Proc. AusWeb02. The Eighth Australian World Wide Web Conference, 2002. To appear in Software: Practice & Experience.