Paper Citation Analysis

Large-Scale Parallel Data Processing

https://github.ccs.neu.edu/cs6240f18/Big-Hero

1. Team Members

HSIANGWEI CHAO, JIEYU SHENG

2. Input Data

2.1 Data Sourse: https://aminer.org/citation, https://dblp.uni-trier.de/

2.2 Data Description

This dataset is consisted of metadata of 3,079,007 papers, each record contains at most 7 attributes: *paper title, authors, year, publication venue, index id, id of references, abstract*. In this project, we performed link analysis on the citation network, hence only index id and references will be used. There are two types of input in our datasets. Dataset v5 and v8 have the format of text blocks. An example of a block was shown in the next bullet point. Dataset v10 are well-organized json objects, each line consists of one object. The year of papers ranges from 1936 to 2011. In order to analysis the different size of the dataset, we analyzed a total of 3 different citation datasets to get more results and compared with each other to get deeper into our questions.

Data set	#paper	#Citation Relationship	Size
DBLP-Citation-network V5	1,572,277	2,084,019	845.4 MB
DBLP-Citation-network V8	3,272,991	8,466,859	817.6 MB
DBLP-Citation-network V10	3,079,007	25,166,994	4.39 G

2.3 An Example of Dataset V5 and V8:

#index 00---- index id of this paper

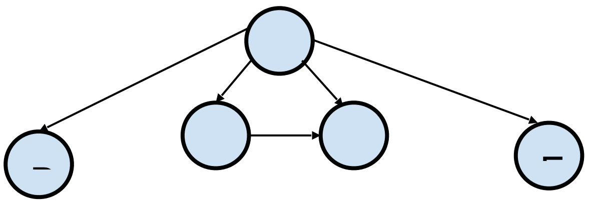
#% ---- the id of references of this paper (there are multiple lines, with each indicating a reference)

2.4 An Example of Dataset V10:

Field Name	Type	Description	Example
id	string	paper ID	013ea675-bb58-42f8-a423-f5534546b2b1
references	strings	citied ID	["4f4f200c-0764-4fef-9718-b8bccf303dba", "aa699fbf-fabe-40e4-bd68-46eaf333f7b1"]

3. Overview

We would like to estimate, for each paper, what is the estimated prior probability that a directly cited paper is also directly cited by other directly cited paper. In other words, we would like to explore for each paper A, and a citation C, what is the probability that there exists a paper B that creates the following *special triplet* relationship:



For example, from the graph above, we know paper A cites paper B, C, D, E. There only exits one special triplet ABC, then we could conclude that for paper A, the probability is 0.25. In this dataset, we aim to calculate the overall prior probability for each paper.

Our main idea to use to joins to find such special triplet, first we join A → B and B → C and get A → B → C, then join A → B → C and A → C to get this triplet ABC.. In this project, we compare two different kinds of join implementation and compare their performance. The implementation of joins to be used are 1. Reduce-Side Join 2. HBase-Assisted Join.

4. Pseudo-Code

4.1 HBase Join

```
def main(args: Array[String]): Unit = {
    // load row id and references from HBase
    val metadata = dataset.loadFrom(HBase)
    // reverse the outgoing edges
    val b_a = metadata.flatMap{case (a, bs) => bs.map(b => (b, a))}
    // A->B->C
    val abc = b_a.groupByKey.flatMap{case (b, as) =>
        // join hbase on key b
        val cs = HBase.getRow(b).getValue()
        cs.flatMap(c => a.map(a => (a, b, c))) }
    // A->B->C and A->C
    // (The special triplet we are looking for)
    val triplets = abc.map{case (a, b, c) => (a, c)}.
        distinct.reduceByKey.
        flatMap{case (a, cs) =>
            // join hbase on key a
            val referred_cs = HBase.getRow(a).getValue().toSet
            cs.filter(c => refferred.cs.contains(c))}
    println("count(a->c): " + triplets.count)
    println("count(total): " + b_a.count)
}
```

4.2 Reduce-Side Join

```
  def v2[U: ClassTag](spark: SparkContext, metaData: RDD[dataset.Paper[U]]): (Long, Long)
= {
    // reduce side map (a,[b,c]) -> [(a,b),(a,c)]
    val outgoing = metaData.mapPartitions(iter => {
      iter.flatMap(x => {
        x.references.map(y => (x.id, y))
      })
    }).distinct().persist()
    // (a, b) -> (b, a) then join (b, c)
    val first = my_join[U,U,U](outgoing.map { case (a, b) => (b, a) }, outgoing).map {
case (b, (a, c)) => (a, (b, c)) }

    //val second = first.join(outgoing).filter { case (a, ((b, c0), c1)) => c0 == c1 }
    val second = my_join(first, outgoing).filter { case (a, ((b, c0), c1)) => c0 == c1 }
      .map { case (a, ((b, c2), c3)) => (a, c2) }.distinct()
    (second.count(), outgoing.count())
  }

def my_join[K: ClassTag, U: ClassTag, W: ClassTag](myself: RDD[(K, U)], other: RDD[(K,
W)]): RDD[(K, (U, W))] = {
    val left = myself.groupByKey().map[(K, (Iterable[U], Iterable[W]))] { case (k, iter)
=> (k, (iter, Seq())) }
    val right = other.groupByKey().map[(K, (Iterable[U], Iterable[W]))] { case (k, iter)
=> (k, (Seq(), iter)) }
    val join = left.union(right).reduceByKey { case ((iterU1, iterW1), (iterU2, iterW2))
=> (iterU1 ++ iterU2, iterW1 ++ iterW2) }

    join.flatMap { case (key, (iterU, iterW)) => {
      var x: List[(K, (U, W))] = Nil
      for (leftU <- iterU) {
        for (rightW <- iterW) {
          x = (key, (leftU, rightW)) :: x}}
      x
    }}}
```

5. Algorithm and Program Analysis

5.1 Program Analysis

Regardless of what the join method we use, we have to perform 2 joins to acquire the special triplet pattern we are looking for. Reduce side join will fetch input from HDFS once only, but it will take one time for HBase join whenever it joins an key in the basic setup. Reduce-side join will have two times data duplication, while HBase join only has one since it simply lookups the its datastore when performing join.

When running the programs, we assume the data is already stored on its storage system. For reduce side join, we assume the data is stored as text file on s3, and for HBase join, we assume the data is stored on hbase on s3. The key difference is that, when reading text file from distributed storage, we still have to parse each record to obtain the information we are interested in; while in HBase, since it a columnar data store, we can directly fetch the columns we are interested in.

We use reduce side join as baseline and see what is the performance and behaviour of using HBase to join. In the initial setup, HBase join is extremely slow. The reason is that it communicates with HBase server for each key it is looking up. Although looking up a single key using HBase is fast, making this communication for every key leads to high latency. Hence, we propose two methods to reduce the latency, and speed up the join operation.

5.2 Optimization

5.2.1 Improvement 1 -- Use multiple gets

```
val abc = b_a.groupByKey.mapPartitions{iterator =>
    // Establish one connection per partition
    getConnectionToHBase()
    iterator.grouped(cacheSize).flatMap{oneBatch =>
        // Send one request for multiple rows per batch
        val gets = oneBatch.map{case (b, _) => new Get(b)}
        val result = HBase.getRows(gets).map(_.getValues)
        // zip the result and original batch together
        (oneBatch zip result).flatMap{case ((b, as), cs) =>
            cs.flatMap(c => as.map(a, b, c))
}}}
```

The first method is to issue a get request to HBase server with multiple row keys. This hugely lowers the average latency used to communicate with hbase server. This method, however, does not utilize the design of HBase which the row keys are sorted not only within but across each storage.

5.2.2 Improvement 2 -- Use scan

```
// Use RangePartitioner + Sort within Partitions
// So that closer rows will be grouped together sorted incrementally
val partitioner = new RangePartitioner(partition_size, b_a)
val b_a_sorted = b_a.repartitionAndSortWithinPartitions(partitioner)
val abc = b_a_sorted.mapPartitions{iterator =>
    // Establish one connection per partition
    getConnectionToHBase()
    // Used to store the cached rows fetched so far
    var results = Nil
    result.flatMap{ case (b, a) =>
        // Discard useless results if no b corresponds to its id
        while(results.nonEmpty && b > results.head.id)
            results = results.tail
        // Fetch result of row if it is not fetched yet
        if (results.isEmpty)
            results = new Scan(startRow = b, limit = cacheSize)
        // If row id does not exist on hbase
        if (results.isEmpty || results.head.id != b)
            None
        // row id exists, perform join
        else {
            val cs = results.head.references
            cs.map(c => (a, c))
}}}
```

In the second method, we propose a solution that utilizes the design decisions of row key distribution in hbase. Before we performed any join operation by using range partition + sort within partition on the row key. This helped us collect the row keys within each partition that aligns with the distribution on HBase, and allows us to use scan operation on HBase. Scan is a much more efficient operation to fetch records from HBase comparing to multiple gets.

Two methods each have its advantage and disadvantage. While scanning is a more efficient to fetch data, it puts more workload on mapreduce/spark program. Range partition + sorting is an expensive method because the whole dataset is being shuffled around each machine. We can say that these two methods are a tradeoff between putting more workload on HBase server or mapreduce/spark program.

6. Experiments and Results

6.1 Report the running time for each of two programs, on both cluster

6 machines	**ReduceSideJoin**	**HBaseJoin (scan)**	**HBaseJoin (gets)**
DBLP-Citation-network V5	1 min 10 sec	1 min 27 sec	3 min 55 sec
DBLP-Citation-network V8	2 min 04 sec	5 min 02 sec	10 min 00 sec
DBLP-Citation-network V10	5 min 22 sec	29 min 59 sec	14 min 07 sec
11 machines			
DBLP-Citation-network V5	1 min 08 sec	1 min 21 sec	1 min 44 sec
DBLP-Citation-network V8	1 min 32 sec	4 min 00 sec	3 min 51 sec
DBLP-Citation-network V10	3 min 24 sec	9 min 43 sec	10 min 24 sec

6.2 Report the running time for local machine

	ReduceSideJoin	**HBaseJoin (scan)**	**HBaseJoin (gets)**
DBLP-Citation-network V5	1 min 05 sec	2 min 26 sec	2 min 34 sec
DBLP-Citation-network V8	1 min 30 sec	8 min 43 sec	8 min 40 sec
DBLP-Citation-network V10	21 min 14 sec	-	-

6.3 Report the number of triplets

Dataset	Result	Count	Proportion
DBLP-Citation-network V5	references that can construct triplet	932,315	40.0%
	total references	2,327,450	
DBLP-Citation-network V8	references that can construct triplet	3,405,499	39.4%
	total references	8,650,089	
DBLP-Citation-network V10	references that can construct triplet	12,591,863	50.0%
	total references	25,166,994	

6.4 program parameter settings

	ReduceSideJoin	**HBase Join (scan)**	**HBase Join (gets)**
hbase cache size	-	1000	1000
partitions (6 nodes)	default	500	200
partitions (11 nodes)	default	300	200

6.5 Speedup

	ReduceSideJoin	**HBaseJoin (scan)**	**HBaseJoin (gets)**
DBLP-Citation-network V5	1	1.07	2.25
DBLP-Citation-network V8	1.35	1.26	2.6
DBLP-Citation-network V10	1.57	3.08	1.36

6.6 Scalability

6 machines	**ReduceSideJoin**	**HBaseJoin (scan)**	**HBaseJoin (gets)**

V5 → V8 **#nodes*2 | #edges*4**	1.77	3.47	2.55
V8 --→ V10 **#nodes*1 | #edges*3**	2.60	5.96	1.41

7.Task

- ❏ Set Up HBase on AWS
- ❏ ReduceSide Join Implementation

8. Conclusion

Through this project, we discovered some strategies for using HBase to perform join operation. Initially, we thought using HBase as indexing would be more efficient than ReduceSide Join, because it is extremely fast for looking up for a single row key. However, after some trials, we found out that HBase is not suitable for using as a general-purpose join, but only works on few cases. The cases where HBase is suitable for this task are when HBase is used for the reason it is built for: get and scan. If we have a small table, and would like to perform a single-side join on a large table storing on HBase, it can utilize its fast get request. Besides, if we have a dataset with sorted row keys, it works well with using scan to fetch records in its original order.

We learn that when designing a distributed program in reality, it is crucial to give proper configuration on the hardware. We fail multiple times at using HBase join on the large dataset partly because we had wrong expectation on the amount of hardware resources. We also learn that despite spark and hadoop mapreduce have similar use cases, they might perform differently due to their design decisions. For example, spark is not good at doing huge amount of shuffling together with producing a huge amount of extra records during mapping phase, while mapreduce might be able to handle this problem since its output is directly written to local storage/HDFS.

On the other hand, it is interesting to discover this special graph pattern in citation graph. Around 40~50 percent of references are also referred by a paper that is directly referred. This result is out of our expectations, and gives us another aspect to understand citation relationship in academic papers.

In the future, we believe the pattern that is discovered in this project can be used in paper recommendation system. This analysis of citation relationship can give academic researchers a better recommendation of interesting/related papers.