TopK

MapReduce

Standalone word count program

- The program loops through all the documents. For each document, the words are extracted one by one using a tokenization process. For each word, its corresponding entry in a multiset called wordCount is incremented by one. At the end, a display() function prints out all the entries in wordCount.

```
define wordCount as Multiset;
for each document in documentSet
{
        T = tokenize( document )
        for each token in T
        {
                wordCount[token]++;
        }
}
display( wordCount )
```

Distributed word count program

- The central documents need to be split and different fractions of the documents need to be distributed to different machines

```
// first phase
define wordCount as Multiset;
for each document in documentSet
{
        T = tokenize( document )
        for each token in T
        {
                wordCount[token]++;
        }
}
display( wordCount )

// second phase
define totalWordCount as Multiset;
for each wordCount received from first phase
{
        multisetAdd( totalWordCount, wordCount )
}
```

- Need to replace in-memory wordCount with a disk-based hashmap
- Need to scale second phase
    - Need to partition the intermediate data (wordCount) from first phase.
    - Shuffle the partitions to the appropriate machines in second phase.

Interface structures

- In order for mapping, reducing, partitioning, and shuffling to seamlessly work together, we need to agree on a common structure for the data being processed.

Phase	Input	Output
map	<K1,V1>	List(<K2, V2>)
reduce	<K2,list(V2)>	List(<K3, V3>)

- Examples
    - Split: The input to your application must be structured as a list of (key/value) pairs, list (<k1,v1>). The input format for processing multiple files is usually list (<String filename, String file_content >).

    The input format for processing one large file, such as a log file, is list (<Integer line_number, String log_event >).
  - Map: The list of (key/value) pairs is broken up and each individual (key/value) pair, <k1, v1> is processed by calling the map function of the mapper. In practice, the key k1 is often ignored by the mapper. The mapper transforms each < k1,v1 > pair into a list of < k2, v2 > pairs. For word counting, the mapper takes < String filename, String file_content ;&gt and promptly ignores filename. It can output a list of < String word, Integer count >. The counts will be output as a list of < String word, Integer 1> with repeated entries.
  - Reduce: The output of all the mappers are aggregated into one giant list of < k2, v2 > pairs. All pairs sharing the same k2 are grouped together into a new (key/value) pair, < k2, list(v2) > The framework asks teh reducer to process each one of these aggregated (key/value) pairs individually.

MapReduce steps

1. Input: The system reads the file from GFS
2. Split: Splits up the data across different machines, such as by hash value (SHA1, MD5)
3. Map: Each map task works on a split of data. The mapper outputs intermediate data.
4. Transmission: The system-provided shuffle process reorganizes the data so that all {Key, Value} pairs associated with a given key go to the same machine, to be processed by Reduce.
5. Reduce: Intermediate data of the same key goes to the same reducer.
6. Output: Reducer output is stored.

Transmission in detail

- Partition: Partition sorted output of map phase according to hash value. Write output to local disk.
  - Why local disk, not GFS (final input/output all inside GFS):
    - GFS can be too slow.
    - Do not require replication. Just recompute if needed.

- External sorting: Sort each partition with external sorting.
- Send: Send sorted partitioned data to corresponding reduce machines.
- Merge sort: Merge sorted partitioned data from different machines by merge sort.

Word count MapReduce program

```
public class WordCount
{

    public static class Map
    {
        // Key is the file location
        public void map( String key, String value, OutputCollector
        {
            String[] tokens = value.split(" ");

            for( String word : tokens )
            {
                // the collector will batch operations writing to
                output.collect( word, 1 );
            }
        }
    }

    public static class Reduce
    {
        public void reduce( String key, Iterator<Integer> values,
        {
            int sum = 0;
            while ( values.hasNext() )
            {
                    sum += values.next();
            }
            output.collect( key, sum );
        }
    }
}
```

Offline TopK

Algorithm level

- HashMap + PriorityQueue
- Parameters
  - n: number of records
  - m: number of distinct entries
  - K: target k
- TC: O(n + mlgk) = O(n)
  - Count frequency: O(n)
  - Calculate top K: O(mlgk)
- SC: O(n + k)
  - HashMap: O(n)
  - PriorityQueue: O(k)

System level

All data is kept in memory

- Potential issues
  - Out of memory because all data is kept inside memory.
  - Data loss when the node has failure and powers off.
- Solution: Replace hashmap with database
  - Store data in database
  - Update counter in database

Too slow for large amounts of data - MapReduce

- Scenarios
  - Given a 10T word file, how to process (Need hash)
  - Each machine store word files, how to process (Need rehash)

TopK

```
class Pair
{
        String key;
```

```
        int value;

        Pair(String key, int value) {
                this.key = key;
                this.value = value;
        }
}

public class TopKFrequentWords
{

        public static class Map
        {
                public void map( String kkey, Document value, Outp
                {
                        int id = value.id;
                        StringBuffer temp = new StringBuffer( "" )
                        String content = value.content;
                        String[] words = content.split( " " );
                        for ( String word : words )
                        {
                                if ( word.length() > 0 )
                                {
                                        output.collect( word, 1 );
                                }
                        }
                }
        }

        public static class Reduce
        {
                private PriorityQueue<Pair> maxQueue = null;
                private int k;

                public void setup( int k )
                {
                        // initialize your data structure here
                        this.k = k;
                        maxQueue = new PriorityQueue<>( k, ( o1, o
                }

                public void reduce( String key, Iterator<Integer>
                {
                        // Write your code here
                        int sum = 0;
```

```
                        while ( values.hasNext() )
                        {
                                sum += values.next();
                        }

                        Pair pair = new Pair( key, sum );
                        if ( maxQueue.size() < k )
                        {
                                maxQueue.add( pair );
                        }
                        else
                        {
                                if ( maxQueue.peek().value < pair.
                                {
                                        maxQueue.poll();
                                        maxQueue.add( pair );
                                }
                        }
                }

                public void cleanup( OutputCollector<String, Integ
                {

                        List<Pair> pairs = new ArrayList<>();
                        while ( !maxQueue.isEmpty() )
                        {
                                Pair qHead = maxQueue.poll();
                                output.collect( qHead.key, qHead.v
                        }
                }
        }
}
```

Online TopK

Algorithm level

TreeMap

- TC: O(nlgm)
- SC: O(m)

HashMap + TreeMap

- TC: O(nlgk)
  - Update hashMap O(n)
  - Update treeMap O(nlgk)
- SC: O(n + k)
  - HashMap: O(n)
  - TreeMap: O(k)

Approximate algorithms with LFU cache

- Data structure: DLL + HashMap
- Algorithm complexity:
  - TC: O(n + k)
  - SC: O(n)

System level

All data is kept in memory

- Problems and solutions are same with offline

Too slow for large amounts of data because of locking

- Distribute the input stream among multiple machines 1, ..., N
- Get a list of TopK from machines 1, ..., N
- Merge results from the returned topK list to get final TopK.

Thundering herd problem

- Problem: What if one key is too hot, writing frequency is very heavy on one node?
- Solution: Add a cache layer to have a tradeoff between accuracy and latency. More speicifically, count how many times an item appears in a distributed way.
  - For each slave, maintain a local counter inside memory. Every 5 seconds, these slaves report to the master node. Namely, each slave

will aggregate the statistics of 5 seconds and report to master. Then the master will update the database. Although the cache layer adds a five seconds latency, it does not have any central point of failure anymore.
    - What if the master node fails?
        - Use another machine to monitor the master, if the master dies, issue a command to restart the machine.

Low frequency words take up so much space

- Solution: Approximate topK. Sacrifice accuracy for space
    - Flexible space
    - O(logk) time complexity
- Disadvantage:
    - All low frequency will be hashed to same value, which will result in incorrect result (low possibility)
    - Some low frequency words will come later, which will have a great count, then replace other high frequency words (bloom filter)
        - HashMap will have 3 different hash functions
        - Choose the lowest count from hashmap

How to calculate topk recent X minutes

Storage

- Write intensive like 20K QPS. NoSQL database suited for this purpose.
- Do not need data persistence. Use in-memory data store.
    - Redis
    - Memcached
- Redis supports more complex data structures
    - Use a key to sorted time mapping.
        - The keys are timestamps
        - The values are sorted set. The sorted set member is the Key and score is the count.

Multi-level bucket

- One bucket maps to one key inside Redis.
- How to calculate the records in last 5 minutes, 1 hour and 24 hours
  - 6 1-minute bucket
  - 13 5-min bucket
  - 25 1-hour bucket
- Retention:
  - Every one minute, a background job will put the oldest 1-min bucket into 5-min bucket and reset the clear up the bucket.
  - Every five minutes, a background job will put the oldest 5-min bucket into 1-hour bucket and reset the clear up the bucket.
  - Every one hour, a background job will put the oldest 1-hour bucket into 1 hour bucket and reset the clear up the bucket.
- How to get the latest 5 minutes: Merge the five key spaces

Final data structure

- Multi-level bucket structure + TreeMap