COMPSCI 590S Systems for Data Science **Fall 2017**

Lecture 16: Morris Counter and Spark

Lecturer: Emery Berger *Scribe(s): Harshith Padigela, Nitin Kishore*

16.1 Morris Counter

Certain data structures allow you to trade a bit of accuracy for immense decrease in memory usage. These are very efficient for analytic applications, when for example, you want to calculate how many unique users have visited the URL. In the previous class we looked at Bloom filters - offers a set membership query only, where the value of lookup is one of two values: definite "No", meaning item doesn't exist in bloom filter and "Maybe", meaning that there is a probability that the item exists and HyperLogLog which is another data structure, that enables you to ask questions about cardinality. For example, how many unique users have visited the URL. Calculating the exact cardinality of a multiset requires an amount of memory proportional to the cardinality, which is impractical for very large data sets(The traditional way to calculate this type of aggregation will require use of hashtable, which could explode your data storage need). HyperLogLog has much smaller memory footprint, but the trade-off is again accuracy.

Now we look at Morris counter, an approximate counting algorithm by Robert Morris(early unix researcher and NSA cryptographer, father of the RTM of "internet worm" fame). Consider the problem of counting events. The counters are monotonically increasing, but we do not care about their exact precision. We relax the exactness for fastness and what we need is an approximate count.

Counter API:
The counter(ctr) has two apis: ctr.increment() and ctr.getValue();
For a traditional exact counter the apis look something like below.

```
increment(){
  count++;
}

getValue(){
  return count;
}
```

16.1.1 Problems in Traditional Approach

Imagine a distributed server setting where we maintain the count for a key and the servers can frequently increment the count. We can either use a centralized count or use distributed counts and reconcile them frequently. Although the second way looks like a good idea, with large geographically distributed servers it gets complicated with problems like how do we maintain these counts, how often do we reconcile for consistency, handling cases where each server sees different counts, etc.

Eventual consistency, also called optimistic replication, is a model used in distributed computing that is a weak guarantee that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. We can treat this as an eventual consistency problem but eventual consistency is a very generic solution. Since we have a very specific use case we can do something more optimized.

16.1.2 Algorithm

The idea is to count logarithms instead of discrete events. We start the counter at 2. On next increment, count will update to 4 with prob. of (1/2). So the expected number of successes in next 2 increment calls is 1. Similarly from 4, we will need 4 increment calls so that the expected update to 8 happens.

```
increment(){
  p = 1/count;
  with prob p:
     count = 2*count;
}
```

We can see the counter only takes powers of 2 and at any point gives us an approximation within a factor of 2. As the count gets bigger, we need more calls for the update to happen. So when we are using a centralized counter, we only have 1 expected update when 'count' (say 10 Million) number of increment calls happen to a counter value currently at 'count'. In essence what would have been a write every single time is now mostly a read. Since writes are costly this is a significant benefit when we are dealing with large counts and provides other benefits like better caching, less network messages in distributed setting along with reducing contention in centralized counters.

16.2 Inverted Index using Bigtable

We discussed in previous class about how google uses spidering for indexing web pages and generating inverted indices. Now we look at why BigTable is good for this task.

We have set of documents and their contents say - D1(A B C D), D2(E B D), D3(A,D). The inverted index for these documents will look as below.

Word	Documents
A	D1, D3
B	D1, D2
C	D1
D	D1, D2, D3
E	D2

We can see that keeping the inverted index sorted by word/key will help in quicker lookup.

The map-reduce jobs for creating inverted index are below.
Map Task: Input - [(url1, "the first content"), ("the second content")........]
Output - [("the",url1), ("first",url1), ("content", url1), ("the",url2), ("second",url2), ("content", url2)........]

Reduce Task: The reduce task merges all the values of a key and the outputs are sorted by key.
["content": [url1,url2...], "first": [url1...], "second": [url1...], "the": [url1, url2...] ....]

The output of map-reduce is the inverted index and we store it in the SSTables (sorted string tables) in BigTable. The structure is shown below.

Word	URLS	Timestamp
content	url1, url2 ...	1234
first	url1 ...	1234

Whenever we need documents for a word, we need to do a binary search and find the urls. The content of the urls can be similarly stored in another SSTable with url as the key, content as value followed by timestamp. The timestamp is the timestamp of the operation and is used for garbage collection and getting the latest content.

16.3 Map Reduce Problems

Albeit some pros like simplicity and abstracted fault tolerance that make Map Reduce look good, they don't necessarily outweigh the cons, like performance issues. There are several intermediate disk reads and writes. One might suggest SSD, but they are still way slower than RAM. Lets see what problems are involved if we use RAM.

· Not having enough RAM (smaller compared to disk)

· It is not fault tolerant. Volatility would result in data loss during a power outage or other failures

· Financially impractical (nearly 40 times as expensive)

· RAM takes up more space than disk and each CPU has limit of around 32-64 GB of RAM

FlumeJava improves map-reduce by lazily building a dataflow graph and optimizing it to reduce number of steps (fusion) resulting in less disk IO. But still there is considerably disk IO and we want to achieve the RAM level performance. That is the subject of RDDs (Resilient Distributed Datasets), which form the basis of Spark.

16.4 Spark

Spark is built on insights from BigTable, FlumeJava and primarily Microsoft's DRYADLINQ. DRYAD converts a program into a dataflow graph which gets executed in a distributed fashion. LINQ - Language integrated query, is the idea of doing SQL operations in a regular language (VB & C#), by exposing types and preventing SQL injection.

RDDs and Lineage:
Spark is the main abstraction for resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. The important idea of RDD is lineage. When we are doing map reduce operations, the data gets read from disk, is transformed and is written to disk and this process is repeated. In spark the chunks of data/RDDs we operate on are immutable (like in SSTable) and these RDDs are stored in memory and applying an operation on one RDD gives a new immutable RDD.

Consider the following map-reduce operations:

1. Create a new RDD from a text file on disk - first RDD

2. Apply map operation(one to one) on first RDD to get second RDD
3. Apply shuffle operation on second RDD to get third RDD
4. Apply count(reduce) operation on third RDD to get fourth RDD

When you apply a function, Spark does not execute it immediately, instead it creates a lineage. A lineage keeps track of what all transformations have to be applied on that RDD, including from where it has to read the data. A lineage graph of the above operations looks like:

$$\text{Disk} \xrightarrow[\text{read}]{} \text{Ram (1st RDD)} \xrightarrow[\text{map}]{} \text{RAM (2nd RDD)} \xrightarrow[\text{shuffle}]{} \text{RAM (3rd RDD)} \xrightarrow[\text{reduce}]{} \text{RAM (4th RDD)}$$

We can observe that if the operations in the graph are deterministic and side-effect free, we only need to store operations to disk instead of the data. If any of the RDD partitions are lost, we can reconstruct them by redoing the chain of operations. The notion of lineage allows us to spend all the time in RAM and get performance and fault tolerance.

RDDs are best suited for batch applications that apply the same operation to all elements of a dataset, so we can achieve fault tolerance without logging lot of data. Otherwise it is better to use systems which provide traditional update logging and data checkpointing.

Dependencies:
The operations in spark are not point-to-point operations but instead aggregate operations which operate on a bunch of data (RDD partitions). In spark the operations are categorized based on dependency as narrow and wide. Narrow is a one to one dependency where each child RDD partition depends on one parent partition. ex: map operation
In wide dependency multiple child partitions can depend one parent partition. ex: join or groupby

API:
Spark provides API for defining the RDDs, for invoking actions and applying transformations on them. Spark automatically distributes the RDDs across the network through partitions and tracks the RDDs' lineage. Spark allows for persistent RDDs, which persist in RAM across operations. Otherwise the RDDs are aged out in an LRU fashion and are flushed out to disk.

An interesting optimization Spark employs is serialization before writing data to disk. Serialization compresses data and is hence faster than deserialized data.

Spark Streaming:
Similar to Map-Reduce, Spark is about batch processing and is not aimed at interactive performance. Doing things for batch allows Spark to build the lazy lineage graph which can be optimized and create the RDDs only when needed.

Spark Streaming (will not be discussed in this class), on the other hand can be used when you have large amount of data coming in as a stream. Examples include sensor data coming from various weather stations every second, parsing tweets to predict stock price, etc. In these cases the rate of data coming in exceeds the bandwidth to write and we cant afford to store it or go back and look at it. The data needs to processed as it comes in, in one pass.