Data Analytics on RAMCloud

Jonathan Ellithorpe
jdellit@stanford.edu

Abstract

MapReduce [1] has already become the canonical method for doing large scale data processing. However, for many algorithms including graph algorithms and machine learning algorithms, algorithms that process only a small fraction of the dataset, and algorithms that operate on live data, MapReduce is either computationally inefficient or incurs too much overhead to be useful. Other frameworks such as Pregel have been developed to address graph algorithms in particular, and Spark for reducing the overhead of doing small or iterative online data analytics. Both of these solutions, however, depend on loading a snapshot of the dataset into the memory of a set of servers, and therefore both suffer from this overhead.

In this project we investigate how fast table scans in a system called RAMCloud [4] may be able to allow us to have the best of both worlds: being able to do a variety of data analytic computations in-place on a live dataset (e.g. the facebook graph, amazon customer data, etc.). We believe that with such a system, instant analytics on the order of wep-page load times will be possible, thereby enabling the development of traditionally infeasible web-apps and features.

Introduction

For nearly a decade now, the MapReduce parallel computation paradigm has been the dominant method for doing large scale data processing. While many data processing algorithms can be expressed in the map and reduce task pattern, it is not always a good fit. Graph algorithms, in particular, are known to work more efficiently when expressed as “vertex programs” that are run by servers in parallel over partitions of the graph that reside statically at a single server during the course of the computation. Applications that only process small parts of a large datasets suffer from the sheer overhead of MapReduce, which typically must load the entire dataset into memory and unnecessarily increases the overall processing time. Moreover, MapReduce was not designed to process and modify “live data”, but rather the computation is performed separately on a snapshot of the dataset, incurring snapshotting and data loading overheads.

Various systems have been designed to try and address these challenges. For example Pregel [2] is a computation framework that was specifically designed for graph algorithms. By storing partitions of the graphs on servers and carefully controlling and optimizing the communication between servers, Pregel is able to easily express and efficiently compute many common graph algorithms. Spark [3] is a system for general large scale “on the fly” data analytics from simple grep and filter to more complicated machine learning algorithms. By doing lazy loading of data into volatile memory and keeping track of data lineage, Spark is able to offer fast and fault-tolerant in-memory processing across many servers in parallel. These systems, however, invariably perform these computations on a snapshot of the data and therefore incur related snapshot and data loading overheads. No system yet exists that can provide these same features on live datasets.

For this project we chose RAMCloud as a starting point for realizing a system that can both serve a live dataset and perform fast computation on that dataset in-place, or a small subset thereof, and with the goal of doing these computations within web-page load times. RAMCloud as a

storage system is already well positioned for durably storing order of terabytes of live data and serving those data at very low latencies and at large scale. We propose that by implementing fast table-scans in RAMCloud, we can do many of the same data analytic computations as Spark and pregel, but avoid the snapshot and data loading overheads by doing this processing in-place.

Counting Objects

Overview

To begin we implemented a simple object counting RPC in RAMCloud. This RPC is sent to every server holding a tablet of a particular table. That master server then scans its hash table to find, and count, every object that belongs to this table. Each master server returns a count of the objects in that tablet to the client, and the clients sums up these counts giving a total count to the application.

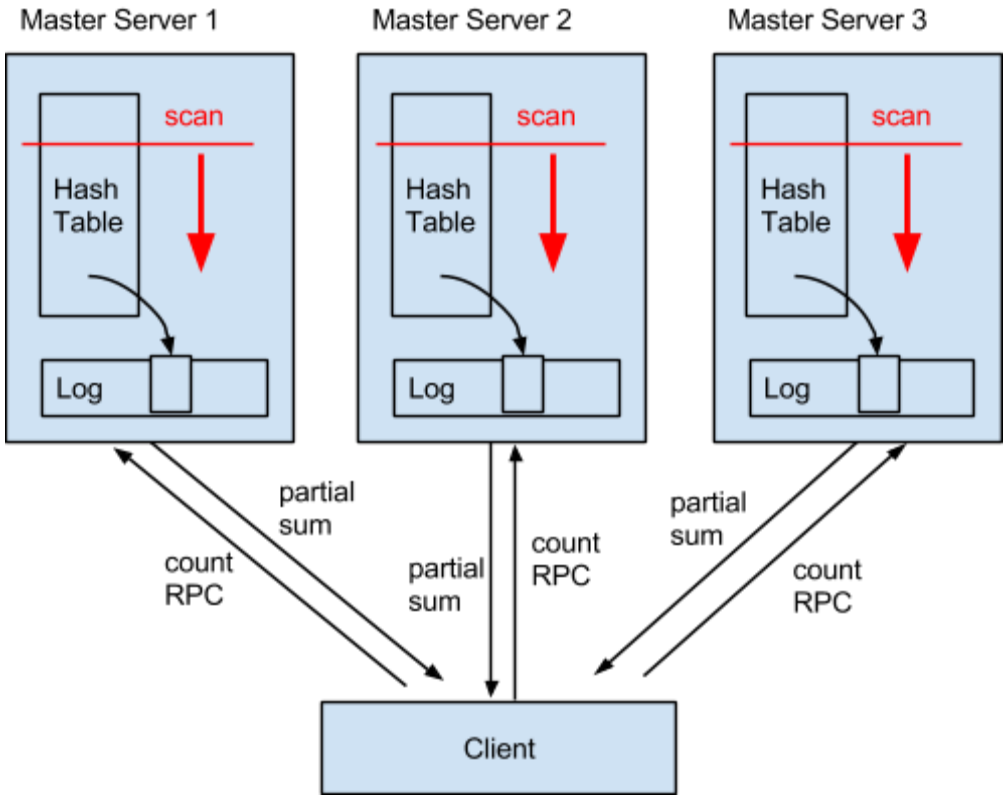

Figure 1. Count RPC. The table to be scanned is partitioned three ways onto servers 1, 2, and 3. Each master server scans its hash table from top to bottom, looking at every object, and counting all objects belonging to the given table with ID TableID.

Performance

We tested the performance of this RPC by loading 4GB of random data (4 million objects with 8B keys and 1KB value blobs) into a table and executing the count RPC while varying the number of servers across which we spread the table. Figure 2 shows our results.

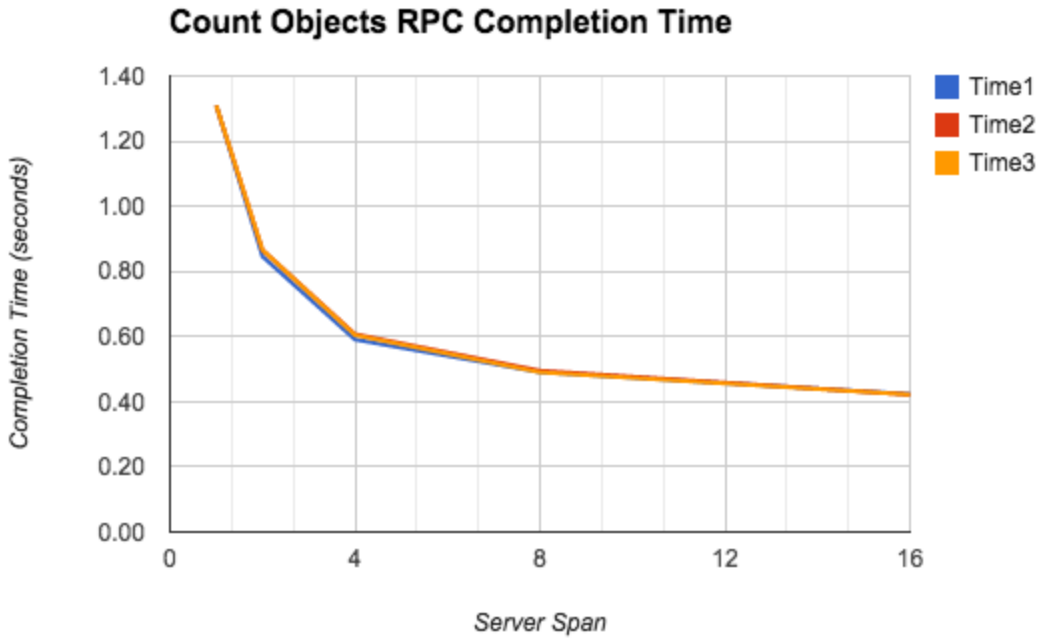

Figure 2. 4GB of random data loaded into a single table. Plots count objects RPC completion time versus the number of servers used to store the table. Each experiment was run 3 times.

We observed diminishing returns for parallelization as we increase the server span, seeming to plateau around 400ms. For instance, going from 8 server to 16 servers we see only a 16% speedup. We found that the reason for this is due to the fixed overhead of scanning the hash table. Even with a hash table with only 1 resident object, because the master server does not know a priori the hash bucket in which it lives, it must scan all hash buckets. We measured this overhead in Figure 3 below.

What is also interesting about Figure 3 is that we see a linear increase as we increase the amount of data. This implies a fixed cost for reading objects out of the in-memory log. We calculated this to be approximately 235ns per object for our 1KB objects, which on the machines we used is roughly the time it takes for 2 cache misses. Scanning an empty hash table in our configuration took a total of 350ms for 32 million buckets, averaging 11ns per hash table bucket.

Limitations

In implementing the basic count objects RPC we therefore noted some interesting limitations for scanning the RAMCloud hash table. One, reading every hash bucket is unavoidable, leading at a lower bound latency of 350ms in our measurements. Two, we found that hash bucket entries actually do not contain tableID information (they are essentially simply a pointer into the in-memory log). This means that scanning any table in the system requires reading out all live data in the log.

To address the first problem, there are a few potential partial remedies. One is keeping the hash table small. Already with 32 million buckets, if each bucket contained one 1KB object, then that is

already 32GB of data indexed by the hash table (our servers had 24GB of main memory). Decreasing the hash table size means potentially increasing read latencies, as the bucket lists become longer and hash collisions become more likely. Another is maintaining per-table bit vectors indicating which buckets contain live table data. This would allow a more focused search of the hash table.

Figure 3. Count RPC completion time on a single server, varying the amount of data in the hash table. Even with 1 object in the hash table there is a fixed cost of searching every hash table bucket.

To address the second problem, we can extend a bucket entry to contain a few bits from the tableID (say the first 8 bits), and use this to quickly filter out most of the entries that belong to other tables (we will collide with 1/256 of the other tables in the system). In fact, currently RAMCloud uses 47 bit pointers in the bucket entry. If we borrowed 7 bits from this to store TableID information, 40 bits can be used to address up to 1TB of data on a server.

Implementing Fault Tolerance

We initially set out to build our count RPC to be able to make incremental progress in the event of failure. That is, if we have some but not all partial counts returned in a given round of RPCs, then the next round of RPCs can attempt to fill in the gaps. This proved to be very complex in the face of arbitrary failure and reconfiguration scenarios, for the benefit of optimizing the uncommon case. Therefore, we chose to use a much simpler method, described below.

When a client issues the count RPCs to the master servers, it attaches what it believes to be true about the start and end key hashes for the tablet it thinks is residing on that server. When the

server receives the RPC, it checks that its current state matches the client's belief. If it does not match, then the master immediately returns a failure. Otherwise, it executes the RPC. But since recoveries of other tablets may occur in parallel on the master server while the RPC is executing, the generated count is still not guaranteed to be consistent with the initial table hash ranges. Therefore once the scan is complete, the master checks its local information about its tablets, and any ongoing recoveries. If the tablet information has changed since when the scan began, or there is currently a recovery in progress for a tablet with the same TableID, then the count information calculated by the scan is not guaranteed to be correct and the master reports failure to the client. Otherwise, the count is accurate, and success is reported to the client.

With these guarantees, if the client received a successful response from each server, then the sum of the counts is guaranteed to be accurate. Otherwise it does not have this guarantee and the client simply throws away all information, refreshes its tablet mapping information from the RAMCloud coordinator, and tries again. Note that since recovery times in RAMCloud are guaranteed to be less than 2s, there will be at most a 2s delay added to the completion time of the RPC.

Logistic Regression

$$p(x;b,w) = \frac{e^{\beta_0 + x\cdot\beta}}{1+e^{\beta_0 + x\cdot\beta}} = \frac{1}{1+e^{-(\beta_0 + x\cdot\beta)}}$$

For the last phase of the project we explored the performance of logistic regression [5]. Logistic regression is a machine learning algorithm used to generate a binary classifier. A common example would be spam classification. Given a dataset of known spam and known non-spam e-mails, called the training set, logistic regression produces a function whose input is a new unclassified e-mail and whose output is an estimated probability of this e-mail being spam. The underlying model for logistic regression is shown above. It is parameterized by beta and omega, and given a sample (say, a numerical vector of features for an e-mail) x, the model outputs the probability that this e-mail is spam. Logistic regression finds beta and omega to fit the training data, which algorithmically translates to finding the beta and omega that maximizes the likelihood of seeing this sample dataset.

$$\begin{aligned}\frac{\partial \ell}{\partial \beta_j} &= -\sum_{i=1}^{n} \frac{1}{1+e^{\beta_0 + x_i\cdot\beta}} e^{\beta_0 + x_i\cdot\beta} x_{ij} + \sum_{i=1}^{n} y_i x_{ij} \\ &= \sum_{i=1}^{n} \left(y_i - p(x_i;\beta_0,\beta)\right) x_{ij}\end{aligned}$$

Mathematically this means finding the beta and omega where the gradient of the likelihood function (above) is equal to zero. There is no closed form solution to this function, so we use the method of gradient descent to iteratively work our way to the peak. To evaluate the gradient (the equal above) involves computing on each point in the dataset, and so we implemented this directly into RAMCloud. Figure 4 below shows the completion time of evaluating this function on a single master server as we increase the amount of data to compute on (each point is 256 floating points numbers, 1KB in total size). In comparison to our previous simple object counting baseline RPC (235ns per object), the gradient calculation takes on average 1014ns per object. Although about 5x the processing time compared to our baseline counter, compared to the numbers reported in Spark [3], where each iteration takes 6s per server on a 1.5GB/server dataset, our implementation is able to

achieve about 1.75s per server per iteration with the same size dataset, and just over 4s on a 4GB dataset. Therefore, not only is our system able to save the cost of data loading, but our servers are able to compute over the data much faster, and this only gets more efficient as the dataset increases in size (therefore amortizing the cost of scanning every hashtable bucket).

Figure 4. Completion time of single iteration of logistic regression on a single server.

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

[2] Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010.

[3] Zaharia, Matei, et al. "Spark: cluster computing with working sets." *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 2010.

[4] Ousterhout, John, et al. "The case for RAMClouds: scalable high-performance storage entirely in DRAM." *ACM SIGOPS Operating Systems Review* 43.4 (2010): 92-105.

[5] http://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch12.pdf