Meteor: Fast Analytics for a Highly Distributed Environment

Umar Javed, Thierry Moreau

December 10, 2013

Abstract

We present Meteor, a system optimized for MapReduce tasks on geographically dispersed data. We envision a highly distributed environment such as a network of cell towers or data centers on a WAN. Traditional solutions for this problem involve all or most of the input and intermediate data backhauled to a central location, making analytics slow and expensive. This is particularly the case for MapReduce-like systems that rely on an all-to-all communication model for input and aggregation. Meteor avoids this by performing as much of the processing and aggregation at the level of an individual datacenter as possible, thus minimizing communication on bandwidth-limited links. Since the aggregation and the resulting reduction on data movement comes at the expense of accuracy of the final result for most analysis tasks, we evaluate the tradeoffs between bandwidth, run-time, and quality of results in Meteor. We implement Meteor as an extension to Apache Spark, an open source in-memory cluster computing stack [14].

1 Introduction

Big data is getting even bigger due to explosive growth in computing devices, infrastructure and the amount of time users spend online. There's huge value to analyzing this data in a timely manner. For example, fast analytics over machine logs in a datacenter, or phone records collected at cell towers, will help the operator find and debug the root cause of performance problems quickly.

Much of this data is widely distributed due to the global presence of major cellular and cloud service providers. For example, the data generated and stored by Googles internal infrastructure, as well as user data collected through its search and advertising services are spread over hundreds of data centers spread throughout the world. Similarly a cellular provider's switching and serving infrastructure in a single metropolitan area alone consists of hundreds of base stations, each collecting gigabytes of data. The principal bottleneck in transferring and analyzing data over inter-cluster links in such systems is the available bandwidth. Not only is the raw capacity on these links limited, most

of this capacity is used for background, bulk transfers for replication [9]. [9] further reports that the cost of deploying this wide-area infrastructure is much more than the cost of networking equipment within a single data center. Therefore backhauling data from every datacenter to a central location for on-the-fly analytics is significantly slow and expensive.

We present Meteor, a system designed to fulfill the need for fast analytics across multiple data centers. We use the term 'data center' loosely here that reflects a highly distributed environment with significant bandwidth limitations between either individual nodes or clusters of nodes. Each node or groups of node may have its own local storage or share a networked storage with the rest of the system. We build Meteor on top of the Spark cluster computing framework [14]. Spark offers a general execution model for in-memory computing which lets query data faster than disk-based engines like Hadoop.

We use a two-pronged approach to achieve our goal of providing fast analytics across multiple data centers. First, we modify the Spark scheduler to add data locality and bandwidth aware work-stealing. Traditional MapReduce-like frameworks assume a tightly coupled cluster with a shared filesystem equally accessible from all nodes. Therefore they are not suited to a heterogeneous environment where the input data is distributed among multiple clusters, possibly on separate filesystems, and limited inter-cluster bandwidth makes timely replication nearly impossible. Meteor addresses these concerns by exposing a richer API such that the application is able to specify co-location of data with compute nodes. The scheduler then enforces these constraints by scheduling map tasks for the co-located data only on the specified hosts, instead in a round-robin fashion. In addition to data locality, Meteor implements an adaptive bandwidth-aware work stealing policy as part of the Spark scheduler. Although strict data locality already significantly improves the responsiveness of the system, data skew likely at different locations will result in the cluster with the smallest data split remaining idle for a major part of the execution. To take advantage of idle processor resources, the scheduler lets an idle host request and operate on a remote data partition with a user-defined probability p. Meteor makes this work-stealing bandwidth-aware by continuously measuring the available bandwidth on all links, and sending the remote block only for the highest bandwidth link.

Second, we explore the potential of approximate iterative queries at reducing communication bandwidth on expensive nodes the application level. When processing large amounts of physically distributed data, users may prefer a quick and dirty result over a correct answer that could take much longer to compute [1, 6]. To this end, we extend Spark's MapReduce framework to provide approximate iterative analytics on wide-area networks that minimizes communication on expensive links at the expense of result correctness. Iterative map-reduce jobs may require all-to-all communication between each iteration, leading to expensive bandwidth requirements on links that route messages between two geographically distant nodes. We wish to explore how running interative jobs by partitioning the data and computing it in isolation, while performing periodic updates can affect result correctness.

2 Design

To achieve fast analytics of massive datasets spread over the wide area, we have implemented several key components as part of Meteor, as described below.

2.1 Location-Aware Map

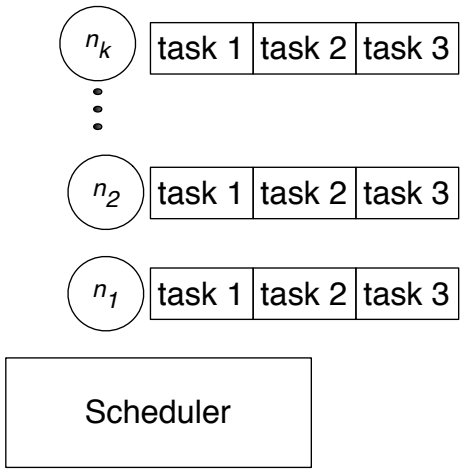

Figure 1: Scheduler keeps separate task queues for each cluster/host in location-aware map.

The system should reduce reliance on network bandwidth since it's a scarce resource. To prevent backhauling all the data from every location to a centralized location at the beginning of the job, input data should remain in place. To prevent map inputs shuttled over the wire, we implement Location-Aware Map (LAM). LAM scheduling achieves data locality by essentially forbidding a node executing a map task over a data partition residing in a remote cluster. Conceptually LAM scheduling is shown in Figure 1. Instead of a FIFO queue of tasks, the scheduler has a separate task queue for each cluster n. Therefore a node in cluster n executes a map task that belonged to its own queue. We provide an API call in Spark to let the programmer attach each data source, i.e. RDD, to a set of nodes. The scheduler puts each map task for partitions for this data source in the appropriate queue. It is interesting to note that Location-Aware Map is equivalent to Delay-Scheduling [13] with infinite delay. We show in §**??** that LAM provides significant latency and bandwidth improvements over vanilla Spark/MapReduce.

2.2 Bandwidth-Aware Work Stealing

Datasets in the wild are split equally; in fact, there's is a lot of skew in the size of geographically dispersed datasets. With strict data locality, this will

result in some set of clusters rendered idle after finishing their local computation, and hence opportunities for parallelism will go amiss. Careful scheduling of tasks on remote nodes in this case could lead to even better runtime results. To this end, we implement an adaptive work-stealing policy based on the measured bandwidth between clusters as part of the Spark scheduler. We call this Bandwidth-Aware Work Stealing (BAWS). BAWS scheduling is shown in Figure 7. BAWS is implemented on top of LAM. During the LAM stage, all nodes operate in normal. When a node n's task queue gets empty, instead of remaining idle, it tries to steal a task off a remote node's queue with probability p. p is configurable by the user. This work-stealing is bandwidth-aware since the candidate node to steal from is the one that the stealer has the highest measured bandwidth at that time. Therefore BAWS is adaptive to network conditions. We show in §**??** that BAWS improves LAM scheduling in terms of job latency.

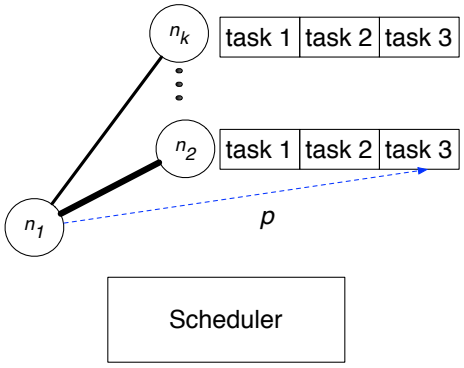

Figure 2: As part of Bandwidth-Aware Work Stealing, Meteor constantly measures the available bandwidth between each pair of clusters. Work Stealing only kicks in when a node's task queue is empty. It then steals a remote node's task with the highest bandwidth with probability p.

2.3 Communication-avoiding iterative MapReduce

There has been a great deal of research on improving iterative MapReduce jobs [4, 8, 14]. by maintaining intermediate data in memory, and consistently partitioning data across iterations. We explore in details how these optimizations help reduce communication in the case of PageRank in section 3.

Further to these techniques, combiners help reduce inter-node communication [6]. Combiners work in the same way as reduce functions, but are used in intermediate stages of a MapReduce job in order to summarize input values it was passed for a given key. Combiners essentially perform aggreagation on the map side in order to reduce the amount of network traffic required between map and reduce phases.

Beyond consistent partitioning, and combining, we wanted to see if *more* could be done to minimize network traffic, potentially at the expense of result

correctness. The idea of providing a quick and dirty result over a slow but correct answer has been explored in a few works [1, 6], but not necessarily in the context of globally distributed data centers.

Our approach works as follows: assuming we are computing an iterative MapReduce job on a partitionned data set which is consistently partitionned across iterations, we avoid sending inter-node updates based on bandwidth availability. The dataflow diagram is shown in Figure **??**. The datasets at each stage of the algorithm are represented by a dotted line, while the partitions are shown as shaded rectangles. Every time a reduction executes, the nodes have to communicate with one-another in order to update their respective key values. This communication can be expensive, and depending on the partitioning of the input dataset, and the dependencies between those datasets, could require high-bandwidth. Thus, by skipping cross-partition updates, we wish to evaluate the overall impact on result quality as well as the reduction in network bandwidth requirements.

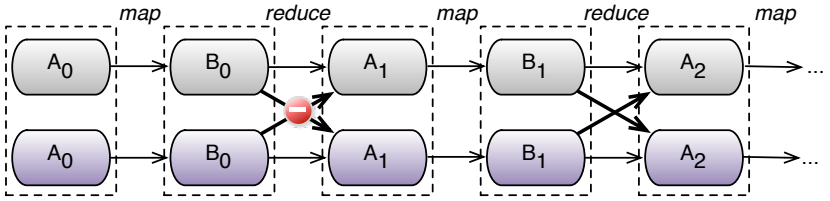

Figure 3: For iterative MapReduce jobs, Meteor can periodically avoid all-to-all communication. This technique can be used to respond to bandwidth fluctuations, at the cost of result convergence.

3 Putting it together: PageRank

The PageRank algorithm is the cornerstone of Google's search technology[3]. It is used to rank the many trillions of URLs by importance. PageRank iteratively updates a rank for each webpage by adding up contributions from each page that links to it.

PageRank is an example of a difficult to parallelize in-memory computation. Part of the complexity arises from the complex pattern of data sharing which is often encountered in many graph algorithms [3]. The computation of the PageRank score of a web page is derived from the score of the "neighboring" web pages. The most bandwidth-hungry stage during a MapReduce implementation of PageRank is the iterative reduce step that aggregates the parent URL for a URL key to compute its rank.

3.1 PageRank with MapReduce on Meteor

The iterative PageRank algorithm described using the MapReduce programming model has been summarized in Listing [**?**]. The input to each of the MapReduce job is a `(url, (outlink_list, rank))` RDD. Each pair is passed in as input to the mapper which produces a set of all outlinked pages as key with contribution weights as values computed from taking the rank of the origin page, divided by the total number of outlinks from the origin page. The reducer then updates each page rank by computing a weigted sum of the contributions each page received from the mapper stage. The PageRank algorithm is implemented in Spark using Resilient Distributed Datasets (RDDs) [14] as follows:

```
// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
    // Build an RDD of (targetURL, float) pairs
    // with the contributions sent by each page
    val contribs = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }
    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) => x+y)
        .mapValues(sum => a/N + (1-a)*sum)
}
```

We show the RDD lineage graph for the PageRank job stages along with the stage dependencies in Figure 4. Each dotted box is an RDD and partitions are shown as shaded rectangles. We would like to compute PageRanks on a datasets that are distributed across two nodes. While the goal of Meteor is to minimize communication between geographically distant data-centers, we suggest for the sake of simplicity to use the term 'node' loosely when covering the PageRank example.

The first stage of the PageRank algorithm is to load the input files, and parse them to generate a set of links from which the ranks of each page can get computed. Ideally, when loading the files, we would like the scheduler to choose an assignment that minimizes inter-node transfer while keeping the partitions balanced among the nodes.

We address this first issue by allowing the application to specify locality awareness and bandwidth aware work-stealing, as discussed in section **??**. This ensures that loading the file data into RDDs generates as low as possible inter-node communication. Once the input files have been loaded into "locality-aware" partitions, the links datasets can be generated locally without requiring any inter-node communication.

The iterative stages of PageRank require inter-node communication at each MapReduce interations. In Spark, communication can be minimized by controlling how the RDDs get partitionned. By using consistent partitioning across each iteration of the algorithm, specifically on the `links` and `ranks`, the `join` operation will require no inter-node communication. On top of consistent parti-

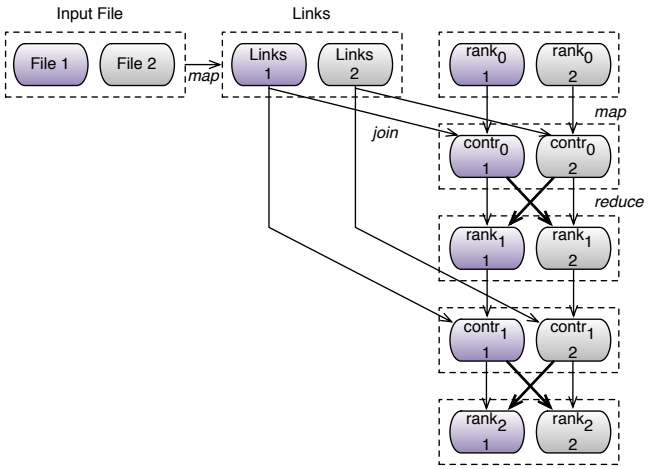

Figure 4: Lineage and dependency graph for datasets in PageRank. Each dotted box is an RDD and partitions are shown as shaded rectangles.

tioning, Spark permits the use of combiners [6] in order to minimize inter-node traffic by aggregating values by keys on the map side.

Beyond consistent partitionining and combining, Meteor avoids inter-node traffic during the reduce stage by dropping updates performed at each MapReduce iteration. This technique effectively isolates the iterative PageRank computation on each partition until there is bandwidth available to update PageRanks across nodes.

4 Evaluation

In this section, we evaluate the following questions regarding Meteor's design.

- To what extent data locality achieved through LAM decrease job time and bandwidth consumed?
- What's the improvement in latency when BAWS is used on top of LAM?

For our evaluation, we use the Emulab testbed [11]. Emulab gives us the ability to configure arbitrary network topologies and link bandwidths. In this initial evaluation, each node in our Emulab topology represents a datacenter and a link connecting two nodes represents a link in the wide-area. We plan to extend the setup to contain multiple nodes in a cluster as future work. We have configured our environment in Emulab by installing Spark and its dependencies. Note that Spark requires the data to be stored on a shared filesystem, such as HDFS. To give our system this illusion while ensuring that data is accessed over the traffic-shaped links, we set up NFS servers at each node and disabled client-

side caching. Each emulab node has 2GB RAM, and two Intel Xeon 3.00GHz processors.

For data, we use publicly available wikipedia data. The total size of our dataset is 4GB. We chose this size for fast prototyping, considering emulab's status as a highly shared and thus resource-constrained testbed. Each node has a split of the total data on its local storage. Our example query, unless otherwise stated, is 'word count'. The size of each data block is 64MB. Unless otherwise stated, each data point is the mean of three runs.

4.1 Job Latency and Bandwidth Usage for LAM

For this experiment, we use a two-cluster 'dumbbell' topology with each 'cluster' containing two nodes, and connected to the other cluster with a WAN link. Data is split equally between the two clusters. Figure 5 shows how the job execution time changes as the link bandwidth is varied, with MapReduce/Spark compared with Location-Aware Map (LAM). It clearly shows that the job runtime is very sensitive to the amount of available bandwidth between workers. The Spark runtime is improved by almost 100 secs when the available bandwidth is increased from 5 to 10 Mbps, with it further decreasing as the bandwidth is increased. In contrast, LAM achieves significantly better runtime even when the bandwidth is really small. This is expected since data locality is strictly enforced and hence no map input block travels over the WAN link. The slight decrease in runtime as the link bandwidth goes up is due to the fact that reduce input still uses the link due to the all-to-all nature of the reduce phase.

Figure 6 splits the runtimes shown in Figure 6 into the individual map and reduce stages. It's evident that most of the improvement comes from enforcing strict data locality through LAM and hence make the map runtime independent of the link bandwidth.

4.2 Job Latency for BAWS

Two measure the effect of Bandwidth-Aware Work Stealing, we use a topology containing three nodes with every pair of nodes connected together. Again each node represents a data center and each link represents a WAN link. The overall dataset is of size 5.5GB, with splits of 3GB, 2GB and 0.5GB sitting at individual nodes, to mimic the data skew common in the wild. This skew will result in the node with the smallest split, i.e. At a given time, the bandwidth for all three links is the same. Figure 7 shows the percentage job runtime decrease for BAWS scheduling compared with LAMP scheduling as well as vanilla Spark, as a function of bandwidth.

The red curve represents BAWS's improvements over LAM. As can be seen, the overall runtime actually *increases* a little bit for very low bandwidth values. This is because work-stealing results

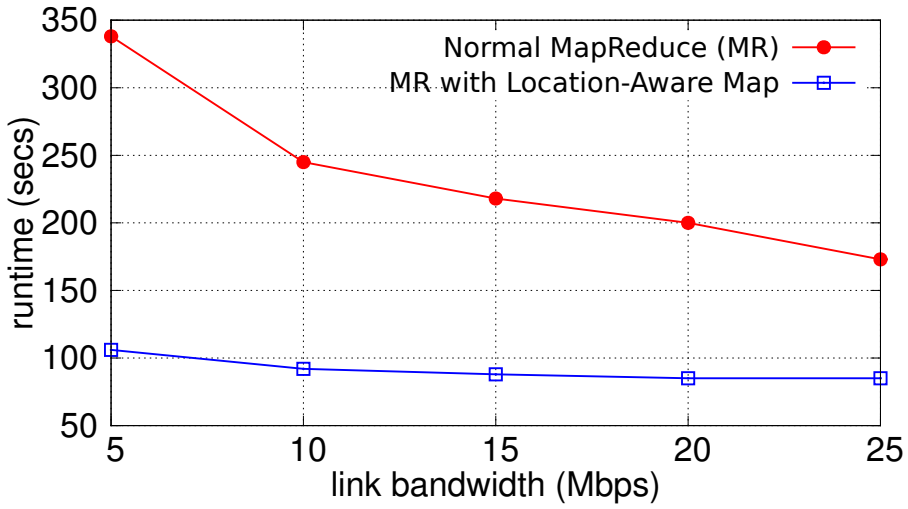

Figure 5: Spark job execution time increases drastically as the wide area link bandwidth decreases. Latency for the Location-Aware Map is significantly better.

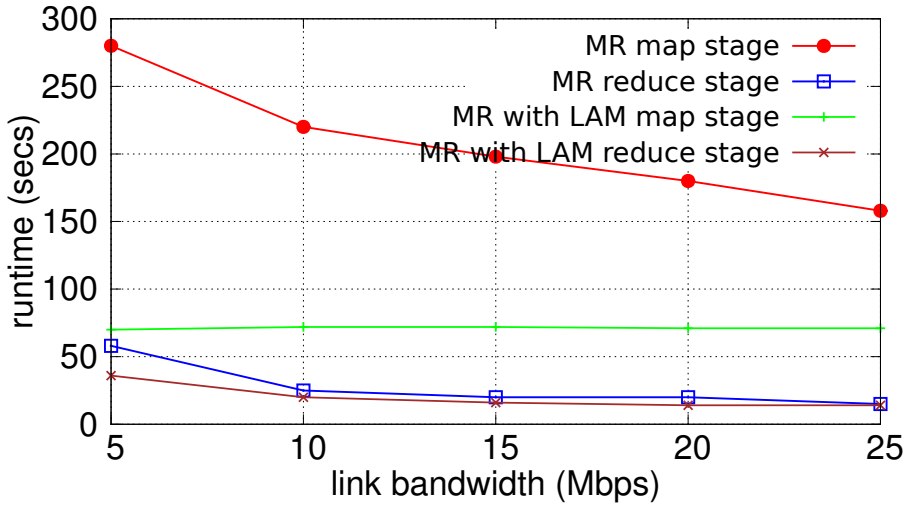

Figure 6: Enforcing data locality for the map stage makes map runtime constant no matter the available bandwidth.

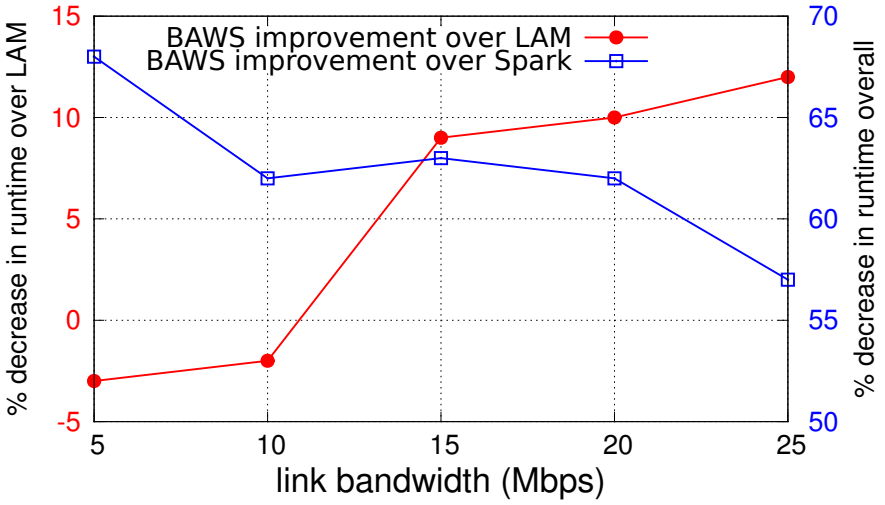

Figure 7: BAWS improves the job runtime even further than LAM unless the bandwidth is really low. The BAWS probability is this case 0.5

5 Related Work

Data-centric programming models for parallel in-memory applications. Spark, and Spark-streaming [5] are recently developed systems for big-data processing, with Spark-streaming an extension to Spark for streaming applications. The primary innovation of these systems is to keep intermediate data for MapReduce jobs in memory, providing quicker response times. Although these systems provide real-time data analytics with fault tolerance, they assume stable, high bandwidth shared storage, such as HDFS. Meteor, on the other hand, provides fast analytics over multiple datacenters connected by a WAN. We will primarily use Spark as the base for Meteor.

Piccolo [12] exposes mutable state to allow the user to perform in-memory computation via a key-value table interface. Piccolo lets applications specify their locality policies to exploit the locality of shared state. When loading data from distributed files, Piccolo chooses an assignment that minimizes inter-rack transfer. Additionally, Piccolo implements a greedy heuristic-based work-stealing approach to load-balancing. Piccolo's reliance on on fine-grained updates to a mutable state makes it difficult to implement in an efficient and fault-tolerant on commodity clusters.

Approximate Queries. BlinkDB [1] is a massively parallel approximate query engine for running interactive SQL queries on large volumes of data. It allows users to trade-off query accuracy for response time by running queries on data

samples. The results are presented to the results to users with meaningful error bars. BlinkDB relies on an optimization framework that maintains multi-dimensional stratified samples from original data over time in order to perform dynamic selection. BlinkDB therefore only applies to centralized databases where dynamic stratification is made possible. It also is restricted to aggregate queries, instead of a broader set of MapReduce jobs.

HOP [6] is an online version of the MapReduce architecture that supports online aggregation, which allows users to see early returns from a job as it is being computed. Online aggregation is a technique that provides initial estimates of results several orders of magnitude faster than the final results, by pre-emptively applying the reduce function to the data that a reduce task has received so far at a given point in time. Meteor borrows the idea of invoking the reduce function on partial map tasks outputs, but applies it in order to reduce the amount of data exchange between workers not in order to perform interactive analysis.

Theres has been a great deal of theory research on streaming algorithms for distributed queries. These algorithms try to limit message-passing by introducing tolerance for inaccurate final results. For example, distributed top-k monitoring [2] and approximate quantiles [7]. We intend to use many ideas contained in this paper as we figure out the operators that Meteor should support.

Resource-constrained computing. Theres has also been a great deal of research in sensor networks on resource-constrained computing, such as [10]. Although we intend to study and use some of these techniques, our focus is primarily on efficient use of network bandwidth rather than the traditional sensor network focuses such as limited power and/or processing.

References

[1] S. Agarwal, B. Mozafari, A. Pand, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, 2003.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. 1998.

[4] Y. Bu, B. Howe, M. Balazinska, and M.D. Ernst. The haloop approach to large-scale iterative data analysis. *The VLDB Journal*, 2012.

[5] Apache Software Compay. Spark - http://spark.incubator.apache.org/.

[6] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[7] C. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *SIGMOD*, 2005.

[8] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *HPDC*, 2010.

[9] A. Greenberg, J. Hamilton, David A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. 2009.

[10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[11] University of Utah. Emulab - http://emulab.net/.

[12] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[13] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I Stoica. Resilient distributed dataests: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.