WTF: The Who to Follow Service at Twitter

Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, Reza Zadeh
Twitter, Inc.
@pankaj @ashishgoel @lintool @aneeshs @dongwang218 @reza_zadeh

ABSTRACT

WTF ("Who to Follow") is Twitter's user recommendation service, which is responsible for creating millions of connections daily between users based on shared interests, common connections, and other related factors. This paper provides an architectural overview and shares lessons we learned in building and running the service over the past few years. Particularly noteworthy was our design decision to process the entire Twitter graph in memory *on a single server*, which significantly reduced architectural complexity and allowed us to develop and deploy the service in only a few months. At the core of our architecture is Cassovary, an open-source in-memory graph processing engine we built from scratch for WTF. Besides powering Twitter's user recommendations, Cassovary is also used for search, discovery, promoted products, and other services as well. We describe and evaluate a few graph recommendation algorithms implemented in Cassovary, including a novel approach based on a combination of random walks and SALSA. Looking into the future, we revisit the design of our architecture and comment on its limitations, which are presently being addressed in a second-generation system under development.

Categories and Subject Descriptors: H.2.8 [Database Management]: Database applications—*Data mining*

General Terms: Algorithms, Design

Keywords: graph processing, link prediction, Hadoop

1. INTRODUCTION

The lifeblood of a vibrant and successful social media service is an active and engaged user base. Therefore, maintaining and expanding the active user population is a top priority, and for Twitter, this is no exception. At the core, Twitter is an information platform remarkably simple in concept: a user "follows" other users to subscribe to their 140-character tweets, which may be received on a variety of clients (e.g., the twitter.com website, mobile clients on iPhones and Android devices, etc.). The vibrancy of the service, whether in informing users of relevant breaking news or connecting them to communities of interest, derives from its users—all 200 million of them, collectively posting over 400 million tweets every day (as of early 2013).

One important way to sustain and grow Twitter is to help users, existing and new, discover connections. This is the goal of WTF ("Who to Follow"),[1] the Twitter user recommendation service. In the current interface, the WTF box is prominently featured in the left rail of the web client as well as in many other contexts across multiple platforms. WTF suggests Twitter accounts that a user may be interested in following, based on shared interests, common connections, and a number of other factors. Social networking sites such as Facebook and LinkedIn have comparable offerings as well. We identify two distinct but complementary facets to the problem, which we informally call "interested in" and "similar to". For example, a user interested in sports might follow @espn, but we probably wouldn't consider that user similar to @espn. On the other hand, two users might be considered similar based on their shared interest in, say, basketball, or if they follow many of the same users. Twitter also exposes profile similarity as a product feature, visible when visiting a user's profile page. Throughout this paper, our discussion of user recommendations covers both these aspects. Based on the homophily principle, similar users also make good suggestions. Besides powering user recommendations, WTF is also used for search relevance, discovery, promoted products, and other services as well.

This paper provides an overview of Twitter's WTF service, a project that began in spring 2010 and went into production the same summer.[2] Quite explicitly, our goal here is not to present novel research contributions, but to share the overall design of a system that is responsible for creating millions of connections daily and lessons that we have learned over the past few years.

We view this paper as having the following contributions:

- First, we explain and justify our decision to build the service around the assumption that the entire graph will fit in memory *on a single machine*. This might strike the reader as an odd assumption, especially in the era of "big data", but our approach has worked well and in retrospect we believe the decision was correct in the broader context of being able to launch the product quickly.
- Second, we describe the complete end-to-end architecture of the WTF service. At the core is Cassovary, our open-source in-memory graph processing engine.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.
WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.

[1] The confusion with the more conventional expansion of the acronym is intentional and the butt of many internal jokes. Also, it has not escaped our attention that the name of the service is actually ungrammatical; the pronoun should properly be in the objective case, as in "whom to follow".

[2] http://blog.twitter.com/2010/07/discovering-who-to-follow.html

- Third, we present a user recommendation algorithm for directed graphs based on SALSA [17], which has performed well in production. We have not seen the algorithm used in this manner before, to the best of our knowledge.
- Finally, recognizing the limitations of the WTF architecture, we outline a second generation user recommendation service based on machine-learning techniques we are currently building as an eventual replacement.

In our view, there remains a large gap between, on the one hand, the academic study of recommender systems, social networks, link prediction, etc., and, on the other hand, the production engineering aspects of these systems and algorithms operating "in the wild", outside controlled laboratory conditions, especially at large scale. It is our goal to bridge this gap by sharing our experiences with the community.

2. THE TWITTER GRAPH

The Twitter graph consists of vertices representing users, connected by directed edges representing the "follow" relationship, i.e., followers of a user receive that user's tweets. A key feature of Twitter is the asymmetric nature of the follow relationship—a user can follow another without reciprocation. This is unlike, for example, friendship on Facebook, which can only be formed with the consent of both vertices (users) and is usually understood as an undirected edge. Although in some cases follow relationships are formed based on social ties (kinship, friendships, etc.), in most cases a user follows another based on shared interests—for example, John follows Mary because John is interested in Hadoop and big data, on which Mary is an expert. Thus, it is more accurate to speak of the Twitter graph as an "interest graph", rather than a "social graph".

To provide the reader better context, we share some statistics about the Twitter graph. As of August 2012, the graph contains over 20 billion edges when considering only active users. As expected, we observe power law distributions of both vertex in-degrees and out-degrees. There are over 1000 users with more than one million followers, and more than 25 users with more than 10 million followers.

The Twitter graph is stored in a graph database called FlockDB,[3] which uses MySQL as the underlying storage engine. Sharding and replication are handled by a framework called Gizzard.[4] Both are custom solutions developed internally, but have been open sourced. FlockDB is the system of record, holding the "ground truth" for the state of the graph. It is optimized for low-latency, high-throughput reads and writes, as well as efficient intersection of adjacency lists (needed to deliver @-replies, or messages targeted specifically to a user and are received only by the intended recipient as well as mutual followers of the sender and recipient). The entire system sustains hundreds of thousands of reads per second and tens of thousands of writes per second.

FlockDB and Gizzard are, unfortunately, not appropriate for the types of access patterns often seen in large-scale graph analytics and algorithms for computing recommendations on graphs. Instead of simple get/put queries, many graph algorithms involve large sequential scans over many vertices followed by self-joins, for example, to materialize egocentric follower neighborhoods. For the most part, these operations are not time sensitive and can be considered batch jobs, unlike graph manipulations tied directly to user actions (adding a follower), which have tight latency bounds. It was clear from the beginning that WTF needed a processing platform that was distinct from FlockDB/Gizzard.

This architecture exactly parallels the OLTP (online transaction processing) vs. OLAP (online analytical processing) distinction that is well-known in databases and data warehousing. Database researchers have long realized that mixing analytical workloads that depend on sequential scans with short, primarily seek-based workloads that provide a user-facing service is not a recipe for high performance systems that deliver consistent user experiences. Database architectures have evolved to separate OLTP workloads and OLAP workloads on separate systems, connected by an ETL process (extract–transform–load) that transfers data periodically from the OLTP to OLAP components. This analogy also holds for the Twitter architecture: FlockDB and Gizzard serve the "OLTP" role, handling low-latency user manipulations of the graph. Our first design decision was to select the most appropriate processing platform for WTF: we discuss this in the next section.

3. DESIGN CONSIDERATIONS

The WTF project began in spring 2010 with a small team of three engineers. The lack of a user recommendation service was perceived both externally and internally as a critical gap in Twitter's product offerings, so quickly launching a high-quality product was a top priority. In this respect the WTF team succeeded: the service was built in a few months and the product launched during the summer of 2010. This section recaps some of our internal discussions at the time, leading to what many might consider an odd design choice that has proved in retrospect to be the right decision: to assume that the graph fits into memory *on a single server*.

3.1 To Hadoop or not to Hadoop?

Given its popularity and widespread adoption, the Hadoop open-source implementation of MapReduce [6] seemed like an obvious choice for building WTF. At the time, the Twitter analytics team had already built a production data analytics platform around Hadoop (see [21, 15] for more details).

Although MapReduce excels at processing large amounts of data, iterative graph algorithms are a poor fit for the programming model. To illustrate, consider PageRank [28], illustrative of a large class of random-walk algorithms that serve as the basis for generating graph recommendations. Let's assume a standard definition of a directed graph $G = (V, E)$ consisting of vertices V and directed edges E, with $S(v_i) = \{v_j | (v_i, v_j) \in E\}$ and $P(v_i) = \{v_j | (v_j, v_i) \in E\}$ consisting of the set of all successors and predecessors of vertex v_i (outgoing and incoming edges, respectively). PageRank is defined as the stationary distribution over vertices by a random walk over the graph. For non-trivial graphs, PageRank is generally computed iteratively over multiple timesteps t using the power method:

$$\text{PR}(v_i; t) = \begin{cases} 1/|V| & \text{if } t = 0 \\ \frac{1-d}{|V|} + d \sum_{v_j \in P(v_i)} \frac{\text{PR}(v_j; t-1)}{|S(v_j)|} & \text{if } t > 0 \end{cases} \quad (1)$$

where d is the damping factor, which allows for random jumps to any other node in the graph. The algorithm it-

[3] http://engineering.twitter.com/2010/05/introducing-flockdb.html

[4] http://engineering.twitter.com/2010/04/introducing-gizzard-framework-for.html

erates until either a user defined maximum number of iterations is reached, or the values sufficiently converge.

The standard MapReduce implementation of PageRank is well known and is described in many places (see, for example, [20]). The graph is serialized as adjacency lists for each vertex, along with the current PageRank value. Mappers process all vertices in parallel: for each vertex on the adjacency list, the mapper emits an intermediate key-value pair with the destination vertex as the key and the partial PageRank contribution as the value (i.e., each vertex distributes its present PageRank value evenly to its successors). The shuffle stage performs a large "group by", gathering all key-value pairs with the same destination vertex, and each reducer sums up the partial PageRank contributions.

Each iteration of PageRank corresponds to a MapReduce job.[5] Typically, running PageRank to convergence requires dozens of iterations. This is usually handled by a control program that sets up the MapReduce job, waits for it to complete, and then checks for convergence by reading in the updated PageRank vector and comparing it with the previous. This cycle repeats until convergence. Note that the basic structure of this algorithm can be applied to a large class of "message-passing" graph algorithms [22, 25] (e.g., breadth-first search follows exactly the same form).

There are many shortcoming to this algorithm:

- MapReduce jobs have relatively high startup costs (in Hadoop, on a large, busy cluster, can be tens of seconds). This places a lower bound on iteration time.
- Scale-free graphs, whose edge distributions follow power laws, create stragglers in the reduce phase. The highly uneven distribution of incoming edges to vertices creates significantly more work for some reduce tasks than others (take, for example, the reducer assigned to sum up the incoming PageRank contributions to `google.com` in the webgraph). Note that since these stragglers are caused by data skew, speculative execution [6] cannot solve the problem. Combiners and other local aggregation techniques alleviate but do not fully solve this problem.
- At each iteration, the algorithm must shuffle the graph structure (i.e., adjacency lists) from the mappers to the reducers. Since in most cases the graph structure is static, this represents wasted effort (sorting, network traffic, etc.).
- The PageRank vector is serialized to HDFS, along with the graph structure, at each iteration. This provides excellent fault tolerance, but at the cost of performance.

To cope with these shortcomings, Lin and Schatz [22] proposed a few optimization "tricks" for graph algorithms in MapReduce. Most of these issues can be addressed in a more principled way by extending the MapReduce programming model, for example, HaLoop [3], Twister [7], and PrIter [31]. Beyond MapReduce, Google's Pregel [25] system implements the Bulk Synchronous Parallel model [29]: computations reside at graph vertices and are able to dispatch "messages" to other vertices. Processing proceeds in supersteps with synchronization barriers between each. Finally, GraphLab [23] and its distributed variant [24] implement an alternative graph processing framework where computations can be performed either through an update function which defines local computations or through a sync mechanism which defines global aggregation in the context of different consistency models. While these systems are interesting, we adopted a different approach (detailed in the next section).

Finally, the Wtf project required an online serving component for which Hadoop does not provide a solution. Although user recommendations, for the most part, can be computed as offline batch jobs, we still needed a robust, low-latency mechanism to serve results to users.

3.2 How Much Memory?

An interesting design decision we made early in the Wtf project was to assume in-memory processing on a *single* server. At first, this may seem like an odd choice, running counter to the prevailing wisdom of "scaling out" on cheap, commodity clusters instead of "scaling up" with more cores and more memory. This decision was driven by two rationales: first, because the alternative (a partitioned, distributed graph processing engine) is significantly more complex and difficult to build, and, second, because we could! We elaborate on these two arguments below.

Requiring the Twitter graph to reside completely in memory is in line with the design of other high-performance web services that have high-throughput, low-latency requirements. For example, it is well-known that Google's web indexes are served from memory; database-backed services such as Twitter and Facebook require prodigious amounts of cache servers to operate smoothly, routinely achieving cache hit rates well above 99% and thus only occasionally require disk access to perform common operations. However, the additional limitation that the graph fits in memory on a single machine might seem excessively restrictive.

To justify this design decision, consider the alternative: a fully-distributed graph processing engine needs to partition the graph across several machines and would require us to tackle the graph partitioning problem. Despite much work and progress (e.g., [12, 11, 26, 4], just to name a few) and the existence of parallel graph partitioning tools such as ParMETIS [12], it remains a very difficult problem [18], especially in the case of large, dynamically changing graphs. A naïve approach such as hash partitioning of vertices is known to be suboptimal in generating excessive amounts of network traffic for common operations and manipulations of the graph (for example, in the context of MapReduce, see experimental results presented in [22]).

Distributed GraphLab [24] implements a two-stage graph partitioning algorithm that attempts to minimize the number of edges that cross partitions (machines). This is accomplished by first over-partitioning the graph into k clusters (where k is much larger than the number of machines m), and then mapping those k small clusters onto m partitions. For the Twitter graph, or any graph that exhibits power law distribution of vertex degrees, this partitioning algorithm (or any similar approach) would yield unsatisfactory partitions. For example, consider Barack Obama, who has over 28 million followers—any placement of that vertex would necessarily result in many edges crossing partitions (machines). This means that graph operations would involve network communication, with latencies measured in the hundreds of microseconds at best.

One proposed solution to reduce network traffic for graph processing is to provide an n-hop guarantee (e.g., [11]). In this approach, the graph is selectively replicated such that

[5] This glosses over the treatment of the random jump factor, which is not important for the purposes here, but see [20].

neighbors within n hops of every vertex are guaranteed to reside on the same machine. For example, a 2-hop neighbor guarantee ensures that all graph random walks within two steps from any source vertex would not require network communication. While an attractive property, there are two major challenges: First, since some vertices and edges need to be replicated, graph algorithms that mutate the graph require some mechanism to enforce consistency. Second, the presence of vertices with large out-degrees makes the n-hop guarantee very costly to maintain in terms of storage, since large portions of the subgraph connected to those vertices would need to be replicated in every partition.

Beyond the challenges of graph partitioning, building a production distributed graph processing engine would have required tackling a whole host of engineering issues related to distributed systems in general. For example, to achieve fault tolerance, replication (of partitions) is a common strategy. But this requires mechanisms for load balancing, coordinating the layout (i.e., which machines serve which partitions), as well as dynamic failover and recovery. Although complex, these are solvable problems (e.g., see our discussion in [16]), but various solutions introduce a layer of (perhaps unneeded) complexity. Architecturally, fitting the entire graph on a single machine means that replication is sufficient for fault tolerance and scale-out in a production environment—we simply need a fleet of servers, each with an identical copy of the complete graph. Load balancing and failover in this context would be much simpler to handle.

As plans were being made for building Wtf, speed of execution was an important concern. The lack of a follower recommendation service created an excessive burden on users to curate an interesting timeline by themselves. Comparable functionalities existed in sites such as LinkedIn and Facebook, and the lack of such a service in Twitter was viewed both internally and externally as an important missing product feature. While building a distributed graph processing engine would have been technically interesting, it would have no doubt substantially delayed the product launch. Our goal was to deliver a high-quality product that would scale with a growing user base, and to accomplish this as quickly as possible. Thus, the entire-graph-on-one-machine design choice boiled down to two questions: could we actually fit the graph in memory on a single server, and could we realistically expect to continue doing so moving forward?

Our design targeted higher-end commodity servers. In terms of hardware configurations, there is a region where memory cost rises roughly linearly with memory capacity. Beyond that, costs start rising super-linearly as one approaches the realm of "exotic" hardware, characterized by low volume chips and custom memory interconnects. We aimed for servers at the higher end of that linear cost region. Additional constraints were imposed by the operational staff, based on the form factor of the servers, which determined the number of DIMM slots available, and the staff's preference for homogeneous system configurations to simplify management.

During the development of the system, we targeted servers with dual quad core processors with 72 GB of RAM and later moved to servers with 144 GB of RAM. We ran some back-of-the-envelope calculations to determine the size of the graph such a server could manage. The simplest representation of a graph consists of an enumeration of edges, with source vertex and destination vertex per edge. Allowing for 2 billion users, one could storing each vertex id as a 32-bit (signed) integer, in which case each edge would require eight bytes. On a machine with 72 GB memory, we could reasonably expect to handle graphs with approximately eight billion edges: 64 GB to hold the entire graph in memory, and 8 GB for the operating system, the graph processing engine, and memory for actually executing graph algorithms. Of course, storing the source and destination vertices of each edge in the manner described above is quite wasteful; the graph could be much more compactly stored as adjacency lists, so these estimates are quite conservative. In practice, we observe a memory requirement of about five bytes per edge. Note that these calculations do not take into account the possibility of graph compression [5], which would reduce storage requirements substantially, although at the cost of slower access and restrictions on the types of graph operations supported.

This analysis does not account for metadata that we might wish to store for each edge, such as an edge weight. Even with standard "tricks" such as quantization to represent floating point numbers in a small numbers of bits, edge data can rapidly blow up the memory requirements, since each datum is a multiplicative factor. We recognized this weakness in the design, and that it could limit the sophistication of the algorithms we could use, but ultimately decided the risk was acceptable, considering that the graph-on-a-single-server design significantly reduced the amount of engineering effort necessary to build the product.

In 2010, our calculations showed that storing the entire Twitter graph in memory on a single machine was feasible. The other important question was, of course, whether it would continue being feasible given anticipated growth. This question was fairly easy to answer, since we had access to historic data on which we could build models and extrapolate. Different growth models and assumptions about the increased memory capacity of future servers suggested anywhere between 24 to 36 months of lead time before any serious concerns about insufficient memory (which roughly corresponds to the natural lifespan of a large-scale production system anyway). In other words, we had a fair bit of headroom before the growth of the Twitter graph outpaces the increase in memory on a commodity server due to Moore's Law. This gave us the confidence to proceed with our plans, quickly launch a much-needed product, and iterate on an improved solution.

4. OVERALL ARCHITECTURE

The overall Wtf architecture is shown in Figure 1. This section describes our Cassovary graph processing engine and how it fits into production workflows.

4.1 Cassovary

The heart of Twitter's Wtf service is the Cassovary in-memory graph processing engine,[6] written in Scala, a language that targets the Java Virtual Machine (JVM). The system was open sourced in March 2012.[7] As previously discussed, Cassovary assumes that there is sufficient memory to hold the entire graph. The graph is immutable once loaded into memory, and therefore the system does not pro-

[6] http://engineering.twitter.com/2012/03/cassovary-big-graph-processing-library.html

[7] https://github.com/twitter/cassovary

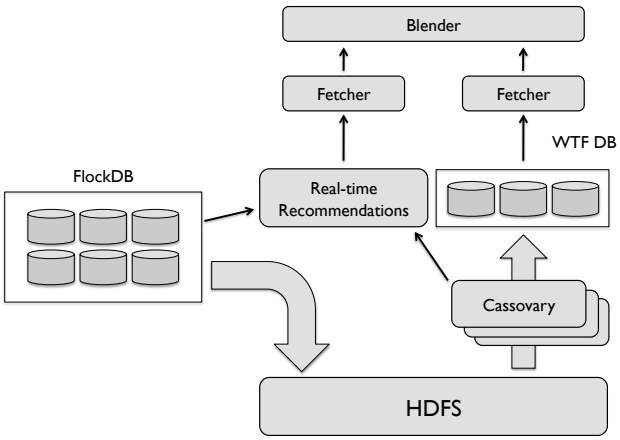

Figure 1: Overall architecture of Twitter's "Who to Follow" user recommendation service.

vide any persistence mechanisms. Fault tolerance is provided by replication, i.e., running many instances of Cassovary, each holding a complete copy of the graph in memory. Immutability is a useful property that allows us to statically optimize the layout of in-memory data structures at startup. Cassovary provides access to the graph via vertex-based queries such as retrieving the set of outgoing edges for a vertex and sampling a random outgoing edge. We have found that vertex-based queries are sufficient for a wide range of graph algorithms, including breadth-first search, random graph traversals, and components needed in a wide variety of link predictions algorithms. Efficient access is provided by storing outgoing edges for each vertex in an adjacency list, implemented as primitive arrays of integers (more details below). Cassovary is multi-threaded, where each query is handled by a separate thread.

Cassovary stores the graph as optimized adjacency lists. A naïve implementation would store the adjacency list of each vertex in a separate object, but for the JVM this is problematic since it yields a proliferation of many objects, most of which are small and all of which consume memory overhead (e.g., for object headers). We wished to minimize overhead so that as much memory can be devoted to storing the actual graph data as possible. In our implementation, Cassovary stores the adjacency lists of all vertices in large shared arrays and maintains indexes into these shared arrays. The vertex data structure holds a pointer to the portion of the shared array that comprises its adjacency list in the form of a start index and length. These shared arrays are simply primitive 32-bit integer arrays, stored in uncompressed form for fast, random lookup. Because of this data organization we are able to quickly fetch all outgoing edges of a vertex and also sample random outgoing edges efficiently. We have experimented with compressing edge data, but ultimately decided that the tradeoff was not worthwhile: not only was it unnecessary from a capacity standpoint, but the need for decompression increased query latencies substantially and complicated the sampling of random outgoing edges.

A random walk, which lies at the core of a wide variety of graph recommendation algorithms, is implemented in Cassovary using the Monte-Carlo method, where the walk is carried out from a vertex by repeatedly choosing a random outgoing edge and updating a visit counter. It is well-known that this Monte-Carlo approach will approximate the stationary distribution of random walk algorithms such as personalized PageRank [8, 2]. The convergence rate is slower than a standard matrix-based implementation, but the memory overhead per individual walk is very low since we only need to maintain the visit counters for vertices that are actually visited. This low runtime memory overhead also allows Cassovary to take advantage of multi-threading to perform many random walks in parallel.

Due to the entire-graph-on-one-machine design, algorithms run quickly on Cassovary, which allowed us to rapidly experiment with different approaches to user recommendation. This is facilitated by a dynamic class-loading mechanism that allow us to run new code without having to restart Cassovary (and reload the entire graph).

4.2 Production Flow

Daily snapshots of the Twitter graph from FlockDB are imported into our Hadoop data warehouse as part of the analytics import pipeline [15]. Once the import completes, the entire graph is loaded into memory onto the fleet of Cassovary servers, each of which holds a complete copy of the graph in memory. Because of the entire-graph-on-a-machine assumption (Section 3.2), we have no need to partition the graph. Furthermore, providing fault tolerance and scaling out for increased query throughput can both be accomplished in an embarrassingly parallel manner.

Cassovary servers are constantly generating recommendations for users, consuming from a distributed queue containing all Twitter users sorted by a "last refresh" timestamp. Typically, it takes less than 500 ms per thread to generate approximately 100 recommendations for a user; Section 5 discusses a few recommendation algorithms in more detail. Output from the Cassovary servers are inserted into a sharded MySQL database, called, not surprisingly, the Wtf DB. There is no specific time target for the Cassovary servers to cycle through the entire user base, only to process the queue as quickly as possible. Note that our design allows throughput to scale linearly by simply adding more Cassovary servers and thus user refresh latency is a resource allocation (not engineering) issue. Once recommendations have been computed for a user, the user is enqueued once again with an updated timestamp. Active users who consume (or are estimated to soon exhaust) all their recommendations are requeued with much higher priority; typically, these users receive new suggestions within an hour.

Due to the size of the Twitter graph and the load incurred on the frontend databases by the snapshot and import process into HDFS, it is not practical to update the in-memory Cassovary graphs much more frequently than once a day. This, however, is problematic for new users, since recommendations will not be immediately available upon joining Twitter. In general, making high-quality recommendations for new users is very challenging, since their egocentric networks are small and not well connected to the rest of the graph—this data sparsity issue is generally known as the "cold start" problem in the recommender systems community. Converting new users into active users is an immensely important problem for social media services, since user retention is strongly affected by the user's ability to find a community with which to engage. Furthermore, any system intervention is only effective within a relatively short time

window. Quite simply, if users are unable to find value in a service, they are unlikely to return.

We address this challenge in two ways: First, new users are given high priority in the Cassovary queue, so that recommendations based on whatever information is available since the last snapshot will be generated with only a brief delay. In the common case, the delay is smaller than the amount of time it takes a new user to go through the steps in the new user welcome flow, thus allowing the user to see some follow recommendations as soon as account setup is complete. Second, we have implemented a completely independent set of algorithms for real-time recommendations, specifically targeting new users. The real-time recommendation module takes advantage of both Cassovary as well as FlockDB to ensure that the most up-to-date graph information is available (in this case, FlockDB load is minimal).

At the front end of the Wtf architecture is a service endpoint known as the "Blender", which is responsible for integrating results from the Wtf DB and real-time recommendation code paths (via "Fetchers" that, as the name suggests, fetch the recommendations). The Blender service endpoint is called by Twitter clients (e.g., the twitter.com site, iPhone and Android mobile apps, etc.), which handle the final rendering of recommendations. The Blender is also responsible for handling A/B testing and recording feedback data that allow us to analyze the quality of deployed algorithms.

5. RECOMMENDATION ALGORITHMS

In this section, we describe a few of the user recommendation algorithms implemented in Cassovary and discuss our general approach to evaluation.

5.1 Circle of Trust

A useful "primitive" that underlies many of our user recommendation algorithms is what we call a user's "circle of trust", which is the result of an egocentric random walk (similar to personalized PageRank [8, 2]). Cassovary computes the circle of trust in an online fashion given a set of parameters: the number of random walk steps to take, the reset probability, pruning settings to discard low probability vertices, parameters to control sampling of outgoing edges at vertices with large out-degrees, etc.

A user's circle of trust provides the basis for our SALSA-based recommendation algorithm (described below). Cassovary also provides the circle of trust "as a service" for any user (or a set of users) given a set of parameters. This provides a flexible network-based personalization mechanism used in a variety of Twitter products. For example, in search and discovery, content from users that are in one's circle of trust are upweighted in the relevance algorithms.

Note that Cassovary dynamically computes a user's circle of trust by performing the random walk from scratch *every time*. This provides two main advantages: we're guaranteed to have fresh results (to within the most recent daily snapshot of the graph) and we can dynamically adjust the random walk parameters and the target of personalization based on the particular application. For example, it is just as easy to compute the circle of trust for a set of users as it is for a single user. One application of this is in generating real-time recommendations: for example, if a user u has just followed v_1 and v_2, one possible online algorithm for generating recommendation candidates would be to simply pick among the circle of trust computed for the set $\{u, v_1, v_2\}$.

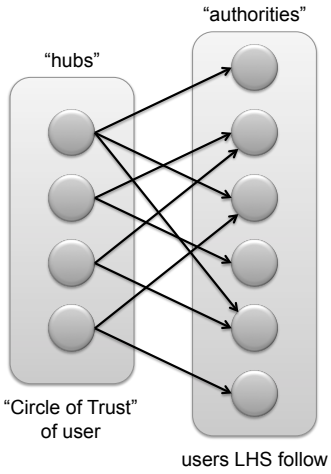

Figure 2: Illustration of our SALSA-based algorithm for generating user recommendations.

5.2 SALSA for User Recommendations

One unique aspect of Twitter is the asymmetric nature of the follow relationship—a user can receive messages from another without reciprocation or even the expectation thereof. This differs substantially from other social networks such as Facebook or LinkedIn, where social ties can only be established with the consent of both participating members. For the purposes of user recommendation, these graph edges are usually treated as undirected; this is also the assumption in most academic studies. As a result, much of the literature on user recommendation assumes undirected edges, though a straightforward observation is that the directed edge case is very similar to the user–item recommendations problem where the 'item' is also a user.

After much experimentation, we have developed a user recommendation algorithm that has proven to be effective in production (see evaluation in the next section), based on SALSA (Stochastic Approach for Link-Structure Analysis) [17], an algorithm originally developed for web search ranking. SALSA is in the same family of random-walk algorithms as PageRank [28] and HITS [13]. The algorithm constructs a bipartite graph from a collection of websites of interest ("hubs" on one side and "authorities" on the other). Unlike in a "conventional" random walk, each step in the SALSA algorithm always traverses two links—one forward and one backward (or vice-versa). That is, starting on the left-hand side of the bipartite graph, the random walk will always end up back on the left-hand side; similarly, when starting on the right-hand side, the random walk will always end up back on the right-hand side.

We have adapted SALSA for user recommendations in the manner shown in Figure 2. The "hubs" (left) side is populated with the user's circle of trust, computed based on the egocentric random walk described in the previous section. In our implementation, we use approximately 500 top-ranked nodes from the user's circle of trust. The "authorities" (right) side is populated with users that the "hubs" follow. After this bipartite graph is constructed, we run multiple iterations of the SALSA algorithm, which assigns scores to both sides. The vertices on the right-hand side are then ranked by score and treated as standard ("interested

in") user recommendations. The vertices on the left-hand side are also ranked, and this ranking is interpreted as user similarity. Based on the homophily principle, we can also present these as suggested recommendations, although they are qualitatively different. We also use these as one source of candidates for Twitter's "similar to you" feature.

Why has this algorithm proven to be effective in production? It is difficult to say with absolute certainty, but note that our application of SALSA mimics the recursive nature of the problem itself. A user u is likely to follow those who are followed by users that are similar to u. These users are in turn similar to u if they follow the same (or similar) users. The SALSA iterations seem to operationalize this idea—providing similar users to u on the left-hand side and similar followings of those on the right-hand side. The random walk ensures equitable distribution of scores out of the nodes in both directions. Furthermore, the construction of the bipartite graph ensures that similar users are selected from the circle of trust of the user, which is itself the product of a reasonably good user recommendation algorithm (personalized PageRank).

5.3 Evaluation

A robust framework for continuous evaluation is critical to the long term success of any recommender system, or web service for that matter. We need to continuously monitor core metrics of deployed algorithms to ensure that they are performing as expected, but maintaining smooth operations is not sufficient. We must also constantly tweak old algorithms and experiment with new ones to improve output quality. This process of empirical, data-driven refinement is well-accepted in the community, and benefits from accumulated experience of previous successful (and not-so-successful) algorithms.

Broadly speaking, we conduct two types of evaluations:

1. Offline experiments on retrospective data. A typical evaluation would use a graph snapshot from, say, a month ago, on which an algorithm is run. Relevant metrics such as precision and recall (more on metrics below) would be computed against the current graph, using the edges that have been added since the graph snapshot as the ground truth. This is the experimental approach adopted by most academic researchers.
2. Online A/B testing on live traffic, which has emerged as the gold standard evaluation for web-based products. In the typical setup, a small fraction of live users are subjected to alternative treatments (e.g., different algorithms). Prospectively (i.e., after the evaluation has begun), relevant metrics are gathered to assess the efficacy of the treatments, as compared to a control condition. While simple in concept, proper A/B testing is as much an art as it is a science; see, e.g., [14] for more discussion.

Early on after the initial launch of the Wtf service, we relied almost exclusively on A/B testing, but have since added offline retrospective experiments to our evaluation repertoire. The initial focus on online A/B testing made sense because we wanted experiments to match production conditions as much as possible, and placing an experimental algorithm in a live setting allowed us to evaluate the entire context of deployment, not only the algorithm itself. The context is critical because user recommendations appear in a wide variety of settings across many different platforms (e.g., on the web, on mobile devices, etc.), each of which may evoke different user behaviors. For example, different types of users such as new vs. old might react differently to a new experimental algorithm. As another example, a user's decision to accept a recommendation might be influenced by how the results are rendered—specifically, cues that we provide to "explain" why the suggestion was made. The only way to evaluate the end-to-end pipeline, from algorithm to user interface, is by online testing.

Online A/B testing, however, is time-consuming because we must wait for sufficient traffic to accumulate in order to draw reliable conclusions, which limits the speed at which we can iterate. Over time, as we have developed better intuitions for the deployment contexts of the algorithms, we have augmented A/B testing with offline retrospective experiments, which are much faster to conduct. Thus, they are invaluable for rapid formative evaluations, before deciding whether it is worthwhile to deploy a full online A/B test.

In our evaluation framework, each experimental condition is identified by a unique algorithm id, which specifies not only the algorithm-under-test, but also the set of associated parameters. Each recommendation carries with it the unique algorithm id through the compute, storage, and serving stages of the pipeline, all the way to the user's client (web browser, mobile phone, etc.). When rendering recommendations, the client logs impression details, which include the recommended user, rank in the list, display context, and the algorithm id. These log data are gathered and automatically post-processed to derive relevant metrics, which are broken down across various display contexts and also aggregated in various ways to provide high-level overviews. The algorithm id enables fine-grained measurements of the algorithm under various test buckets with different user and context scenarios. We maintain an internal dashboard where we can monitor and explore the performance of various algorithms and algorithm ensembles.

Most of our experiments fall into three categories: First, we can vary the recommendation algorithm (including fusion algorithms that combine results from different algorithms). Second, we can vary algorithm parameters. Finally, we can selectively target the result of a particular algorithm, for example, showing recommendations only to a particular user population or in a specific context.

A key question in any evaluation is the metric or the set of metrics to be tracked. The simplest metric we use to compare different algorithms is "follow-through rate" (FTR), which is calculated by dividing the number of generated follows by the number of impressions of the recommendations attributed to a particular condition. Although useful and easy to understand, we stress that FTR is not sufficient, and far from being the final word on evaluating recommendation engines. While it is a reasonable measure of precision, it fails to adequately capture recall. Furthermore, users' following behavior is subjected to natural lifecycle effects—newer users, who are still exploring Twitter, are more receptive to following additional accounts. On the other hand, since attention is a limited quantity, an experienced user who already receives information from many accounts will be hesitant to follow additional users. Unfortunately, FTR does not adequately capture these effects.

However, the biggest issue with FTR is that it does not measure the *quality* of the recommendations, since all follow edges are not equal. For Twitter, the primary goal of

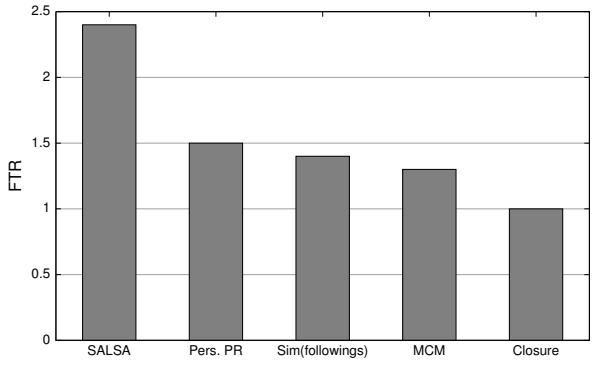

Figure 3: Comparing the relative effectiveness of various follower recommendation algorithms.

user recommendations is to help the user create a quality timeline and increase overall engagement. We approximate this notion using a metric called "engagement per impression" (EPI). After a user has accepted a recommendation, we measure and quantify the amount of engagement by the user on that recommendation in a specified time interval called the observation interval. From this we derive a metric that attempts to capture the impact of a recommendation. The downside of this metric is that it is available only after the observation interval, which slows the speed at which we can assess deployed algorithms.

To give the reader a sense of the effectiveness of specific user recommendation algorithms, we present some evaluation results in Figure 3, which shows normalized FTR over a small fraction of traffic in a controlled experiment lasting several months, for a few of the algorithms we have tried. The algorithms in this experiment are:

- SALSA, the algorithm described in Section 5.2.
- Personalized PageRank [8, 2].
- Sim(Followings): In order to generate recommendation candidates for a user u, we consider F, the set of users that u follows. For each $f \in F$, the set of users that are similar to f are considered and these sets are then combined into a multi-set with members ranked by their cardinality (or a score) in the multi-set.
- Most common neighbors: This algorithm seeks to maximize the number of followings of the user u that in turn follow the recommendation candidate. In other words, it picks those users c_i as candidates from the second degree neighborhood of the user u (i.e., followings of followings of u) to maximize the number of paths of length two that exist between u and c_i.
- Closure: If u follows v, v follows w and w follows u, w is a recommendation candidate produced by this algorithm. Note that all follows produced by this algorithm produce bidirectional (i.e., reciprocated) edges.

Since we need to serve many different types of users across different contexts and on different clients, the actual algorithms deployed in production tend to be ensembles that integrate results from different approaches. For example, the "standard" production algorithm blends results from approximately 20 different algorithms.

6. ONGOING WORK

The Wtf architecture built around the graph-on-a-single-server assumption was never meant to be a long term solution. As mentioned before, it was part of a deliberate plan to rapidly deploy a high-quality but stopgap measure, and then iterate to build a better replacement, incorporating all the lessons we learned with the Cassovary architecture.

At the outset, we were well aware of two significant weaknesses in the original design. The first is the scalability bottleneck, where the growth of the graph would exhaust the memory capacity of individual servers. The second, and more important, is the limitation on the richness and variety of features that could be exploited by graph algorithms due to memory limitations. For example, we wish to attach metadata to vertices (e.g., user profile information such as bios and account names) and edges (e.g., edge weights, timestamp of edge formation, etc.), and make these features available to the recommendation algorithms. In addition, we have thus far exclusively focused on the follower graph, but interaction graphs (e.g., graphs defined in terms of retweets, favorites, replies, and other user interactions) provide a wealth of valuable signals as well. A user recommendation algorithm should have access to all these features, but this is very difficult to accomplish given the memory constraints. For example, edge metadata become multiplicative factors on memory consumption: in the naïve encoding of edges as (source vertex, destination vertex) pairs (two 32-bit integers = 8 bytes), augmenting each edge with a single 8-bit quantized weight would increase memory usage by 12.5%. Available memory severely limits the diversity and sophistication of algorithms that could be brought to bear on the problem.

6.1 Hadoop Reconsidered

We have been developing a completely new architecture for Wtf that addresses the above two limitations. This section discusses our general approach, and we hope to share more details in a future paper. Instead of relying on Cassovary for in-memory graph processing, the new architecture is completely based on Hadoop—specifically, all algorithms are implemented in Pig [27, 9], a high-level dataflow language for manipulating large, semi-structured datasets that compiles into physical plans that are executed on Hadoop. Pig (via a language called Pig Latin) provides concise primitives for expressing common operations such as projection, selection, group, join, etc. This conciseness comes at low cost: Pig scripts approach the performance of programs directly written in Hadoop Java. Yet, the full expressiveness of Java is retained through a library of custom UDFs that expose core Twitter libraries (e.g., for extracting and manipulating parts of tweets). The other noteworthy aspect of our new approach is its reliance on machine-learned models to exploit a wealth of relevance signals (more below).

The first decision, to adopt the Hadoop stack, is worth discussing. At the outset of the Wtf project in early 2010, we had specifically considered, and then ruled out, an architecture based on Hadoop. What's changed?

The numerous issues discussed in Section 3.1 that make iterative algorithms inefficient in MapReduce still remain. However, the primary difference is that we have accumulated substantial intuition about the problem domain and experience with various types of algorithms. Thus, we can begin by reimplementing effective Cassovary algorithms in

Pig, and then focus on optimizing the efficiency of the component MapReduce jobs. Since we have already explored substantial portions of the solution space with Cassovary, we are less engaged in exploratory algorithm development, which makes long-running algorithms tolerable.

Indeed, the initial Pig implementation of the user recommendation pipeline was quite slow, generated terabytes of intermediate data in a mostly brute-force fashion, and ranked among the most resource-intensive jobs on Twitter's central Hadoop cluster. However, over time, we have substantially increased the efficiency of the pipeline through careful software engineering (e.g., adopting Pig best practices for join optimizations where possible) and clever optimizations. One example is a sampling-based solution to finding all pairs of similarities between D vectors, each of dimension N, where the algorithmic complexity is independent of N [30].

There have been substantial developments in large-scale graph processing frameworks since the WTF project began. Notably, Giraph began as an open-source implementation of Pregel [25]; GraphLab [23] and distributed GraphLab [24] were presented in 2010 and 2012, respectively. Why did we favor Hadoop over these frameworks specifically designed for graph processing? For one, we did not feel that these systems were sufficiently mature to run in production. GraphLab has the additional disadvantage in that it is implemented in C++, whereas Twitter is built primarily around the Java Virtual Machine (JVM); of course, JNI bindings are possible, but that adds a layer of unnecessary complexity.

Even if there existed a production-quality graph processing framework, it would not be clear if adopting a separate specialized framework would be better than using Hadoop. Building a production service is far more than efficiently executing graph algorithms. For example: the production pipeline must run regularly, thus we need a scheduler; the algorithms necessarily depend on upstream data (e.g., imports from FlockDB), thus we need a mechanism to manage data dependencies; processes need to be monitored and failures handled, or otherwise an on-call engineer needs to be notified, thus we need a monitoring framework. All of these components already exist on Hadoop, since productionizing Pig scripts is a commonplace task with well-developed processes (see [21, 15] for more details). In fact, moving WTF to Pig simplified its architecture, as we simply plugged into existing analytics hooks for job scheduling, dependency management, and monitoring. In contrast, introducing another computational framework (for graph processing) would require building parallel infrastructure from scratch—it is not entirely clear that the gains from adopting an efficient graph processing framework outweigh the additional engineering effort required to fully productionize the processing pipeline. This is similar to the "good enough" argument that Lin [19] made in a recent paper.

6.2 From Real Graph to Recommendations

In the new WTF architecture, user recommendations begin with the "Real Graph", which integrates a multitude of user and behavior features into a common graph representation. Since the Real Graph is not intended to reside in memory, but rather on HDFS (Hadoop Distributed File System), there are few limitations on the richness or number of features that we can include. Vertices in the graph still represent users, but now we can attach arbitrary profile information (number of tweets, number of followers, number of followings, etc.). Edges represent relationships between users, and have been expanded to include not only following relationships, but also retweets, favorites, mentions, and other user behaviors. Edges can also be associated with arbitrary metadata such as weights. The data structures underlying the Real Graph are extensible so that additional vertex or edge features can be added at any time.

The Real Graph itself is constructed on a periodic basis through a series of Pig scripts that run on Hadoop, joining together data from disparate sources. From the Real Graph, recommendations are computed with a new generation of algorithms that reflect a distillation of experiences with Cassovary. They can be divided into two phases: candidate generation and rescoring.

- *Candidate generation* involves producing a list of promising recommendations for each user, with algorithms ranging widely in sophistication. For example, (selectively) materializing users' two-hop neighborhoods is a simple, but very inefficient approach to candidate generation. A more refined approach would be to compute the top n nodes from personalized PageRank or the SALSA-based algorithm described in Section 5.2.
- *Rescoring* involves applying a machine-learned model to the initial recommendations generated by the previous stage. We adopt a standard formulation of user recommendation as a binary classification problem [10, 1]. Since it is impractical to classify all possible non-existent edges, this stage requires a working set of edges to serve as classifier input, hence the candidate generation stage. The models are learned using the Pig-based framework described in [21]. We train logistic regression classifiers using stochastic gradient descent in a single pass.

The two-stage approach provides a nice abstraction that allows engineers to work on either component alone. Note that these two stages have complementary goals and challenges. The candidate generation stage is primarily focused on recall and diversity, whereas the rescoring stage must deliver high precision while retaining diversity. Efficiency is a key concern for candidate generation, since topological properties of the graph render naïve algorithmic implementations impractical—for example, while technically feasible, it is hugely inefficient to selectively materialize users' two-hop neighborhoods—they are surprisingly large due to the presence of supernodes, or vertices with extremely large outdegrees. Every candidate generation algorithm must contend with topological features of the Twitter graph, which require clever optimizations and sometimes approximations. In contrast, efficiency is less of a concern in rescoring since our use of linear models means that classification ultimately boils down to inner products between feature and weight vectors (which are fast operations).

In our design, the Real Graph is useful beyond generating user recommendations. It is intended to be a general resource at Twitter for any project or product that involves the graph or requires graph features, for example, search personalization, content discovery, static priors for ranking, etc. Since the complete Real Graph is very large, we materialize compact views that are customized to the specific application—these materialized graphs might, for example, be pruned to fit the memory profile of the target application.

The second generation of the WTF architecture is the focus of ongoing efforts at Twitter. Compared to the original

design, we have eliminated the scalability bottleneck and are building a flexible infrastructure to enable future exploration of graph- and content-based recommendation algorithms.

7. CONCLUSIONS

This paper shares the story of Twitter's "Who to Follow" user recommendation product. We consider the project a big success: the service was built and deployed within a short amount of time, and has been driving a substantial amount of engagement on Twitter over the past few years. In addition, the WTF architecture has allowed other Twitter products such as search and promoted products to take advantage of graph algorithms on demand. We hope this case study provides the community a fresh perspective on the "practical" aspects of building and running a large-scale recommendation service. Most academic studies focus on algorithms, but the success of a service is dependent on many additional factors such as having the right architecture and the right evaluation framework for iterative refinement. Our account fills a gap in the literature by providing a "big picture" view in the context of a large-scale production service.

8. ACKNOWLEDGMENTS

We would like to acknowledge additional members of the WTF team who are moving the project forward today: Ajeet Grewal, Siva Gurumurthy, Jerry Jiang, Venu Satuluri, Zhijun Yin. Also thanks to Ning Liang, John Sirois, Tao Tao, and Mengqiu Wang for their contributions.

9. REFERENCES

[1] L. Backstrom and J. Leskovec. Supervised random walks: Predicting and recommending links in social networks. *WSDM*, 2011.

[2] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. *VLDB*, 2010.

[3] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient iterative data processing on large clusters. *VLDB*, 2010.

[4] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. *SoCC*, 2012.

[5] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. *KDD*, 2009.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. *MAPREDUCE Workshop*, 2010.

[8] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized PageRank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[9] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of MapReduce: The Pig experience. *VLDB*, 2009.

[10] M. Hasan and M. Zaki. A survey of link prediction in social networks. In C. Aggarwal, editor, *Social Network Data Analytics*. Springer, 2011.

[11] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *VLDB*, 2011.

[12] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[13] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[14] R. Kohavi, R. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the HiPPO. *SIGKDD*, 2007.

[15] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at Twitter. *VLDB*, 2012.

[16] F. Leibert, J. Mannix, J. Lin, and B. Hamadani. Automatic management of partitioned, replicated search services. *SoCC*, 2011.

[17] R. Lempel and S. Moran. SALSA: The Stochastic Approach for Link-Structure Analysis. *ACM Transactions on Information Systems*, 19(2):131–160, 2001.

[18] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[19] J. Lin. MapReduce is good enough? If all you have is a hammer, throw away everything that's not a nail! *arXiv:1209.2191*, 2012.

[20] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[21] J. Lin and A. Kolcz. Large-scale machine learning at Twitter. *SIGMOD*, 2012.

[22] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. *MLG Workshop*, 2010.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new parallel framework for machine learning. *UAI*, 2010.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[25] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. *SIGMOD*, 2010.

[26] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. *SIGMOD*, 2012.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. *SIGMOD*, 2008.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford University, 1999.

[29] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[30] R. Zadeh and A. Goel. Dimension independent similarity computation. *arXiv:1206.2082*, 2012.

[31] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. *SoCC*, 2011.