Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs

Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy,
Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin

Twitter, Inc.
San Francisco, California

@pankaj @venusatuluri @ajeet @sgurumur @vzhabiuk @truthseeker1985 @lintool

ABSTRACT

We describe a production Twitter system for generating relevant, personalized, and timely recommendations based on observing the temporally-correlated actions of each user's followings. The system currently serves millions of recommendations daily to tens of millions of mobile users. The approach can be viewed as a specific instance of the novel problem of online motif detection in large dynamic graphs. Our current solution partitions the graph across a number of machines, and with the construction of appropriate data structures, motif detection can be translated into the lookup and intersection of adjacency lists in each partition. We conclude by discussing a generalization of the problem that perhaps represents a new class of data management systems.

1. INTRODUCTION

One major strength of Twitter is its ability to inform users about what's happening in the world *in real-time*, be it news of natural disasters, breaking political stories, or the latest sports scores. To keep users informed, we have developed services that proactively deliver content via push notifications to their mobile devices.[1]

Naturally, such notifications must be relevant, personalized, and timely, or else they run the risk of becoming an annoyance. We have discovered, perhaps unsurprisingly, that local network signals are important for discovering content. Twitter users curate accounts they follow, which represent their interests as well as social and professional connections. Thus, the temporally-correlated activities of a user's followings provide a rich source of recommendations. For example, suppose we wish to make "who to follow" [5] recommendations to user A: we examine the list of accounts that A follows (call them $B_1 \ldots B_n$), and if more than k of them follow an account C within a time period τ, then we recommend C to A (where k and τ are tunable parameters). The idea applies to recommending content as well, based on user actions such as retweets, favorites, etc. Empirically, we have found this approach to yield high engagement—intuitively, the temporally-correlated activities capture "what's hot" (hence, timely) among the accounts that a user follows, which is by definition the group that the user is interested in (hence, relevant and personalized).

This recommendation strategy represents a specific instance of the more general problem of network motif detection in real-time on large dynamic graphs. Network motifs are recurrent patterns in graphs [6], usually defined in terms of vertices and edges that are arranged in a specific configuration. Motifs have significance across a wide range of domains, including biochemistry, ecology, computer science, and sociology. Nearly all approaches to motif detection are based on a static graph snapshot and viewed as batch computations. Our novel "twist" is to identify motifs as they are being formed in real time and trigger appropriate actions.

How does this relate to previous work? Of course, stream-oriented databases have a long history [4], but they are designed for relational and not graph processing. Much of the work in large-scale graph analytics focuses on batch computations, although there is work in a streaming setting, e.g., on random walks [1], triangle counting [2], and clustering coefficients [3]. Our work is very much in this spirit.

This paper makes two contributions. First, we present the design of a large-scale production system at Twitter for serving real-time recommendations to tens of millions of mobile users. Second, we identify the algorithm implemented in this service as one instance of a more general problem of real-time motif detection on large dynamic graphs and discuss implications for future data management systems.

2. SYSTEM DESIGN

As a running example, we refer to the graph fragment in Figure 1. For simplicity, let us assume that a directed edge indicates a follow, and so in this case we are recommending user accounts. In terms of notation, A's refer to users for whom we wish to make recommendations; B's are the users whom the A's follow, and C's are the users whom the B's follow. We want to recommend the appropriate C's to the A's. Note that this notation is highly schematic, as the C's are also A's for a different set of B's (i.e., we want to make recommendations for everyone). For simplicity, let us assume $k = 2$ in the above definition, which means that when the edge $B_2 \rightarrow C_2$ is created in Figure 1, we want to push C_2 to A_2 as a recommendation. In other words, we are interested in the "diamond" motif involving four graph vertices (although in production, $k = 3$).

What is the scale of this problem? The Twitter follow graph (as of 2012) contains $O(10^8)$ vertices and $O(10^{10})$

[1] `https://blog.twitter.com/2013/stay-in-the-know`

Proceedings of the VLDB Endowment, Vol. 7, No. 13

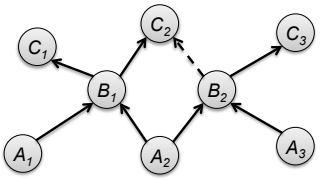

Figure 1: Sample graph fragment.

edges [7]. The system must be able to handle a highly dynamic graph—our design targets $O(10^4)$ edge insertions per second—and be able to deliver recommendations within seconds. Naturally, we desire a horizontally-scalable solution.

At the outset, we ruled out two obvious but naïve solutions. One could poll each user's network periodically to see if the motif has been formed since the last query; however, the latency would be unacceptably large. Another approach would be to keep track of each A's two-hop neighborhood; a rough calculation shows that this is impractical, even using approximate data structures such as Bloom filters.

Ultimately, we implemented a solution that requires only standard data structures. To explain, we first consider the case where the entire graph fits on a single machine. Let us treat the edges from A's to B's as static (we'll see why later) and store the inverse as an adjacency list (for convenience, we'll refer to this data structure as $\mathcal{S}$). Given a particular B, we can query $\mathcal{S}$ to look up all A's that follow it.

We assume the existence of a data source (e.g., message queue) that provides a stream of graph edges as they are created in real-time. In our notation, these are treated as edges from B's to C's; we refer to this as the dynamic part of the graph. We maintain a data structure $\mathcal{D}$ that holds the edges pointing to C's. In other words, given a query vertex C, we can easily fetch all edges from the B's along with their creation timestamps—in this way we enforce the freshness of the recommendation (i.e., τ).

Note that in our design, all data structures are held in main memory. Given this setup, the recommendation algorithm proceeds as follows: when a $B \rightarrow C$ edge is created (in this case, $B_2 \rightarrow C_2$), we query $\mathcal{D}$ to find all other B's that also point to the C (in this case, B_1). At this point, we've compute the top half of the diamond motif. For all these B's (in this case, B_1 and B_2), we look up their incoming edges from the A's in $\mathcal{S}$ to compute an intersection, which is whom we're making the recommendation to—in this case, A_1 and A_2 point to B_1 and A_2 and A_3 point to B_2, so A_2 is the intersection. Note that since $\mathcal{S}$ is a static data structure, we can easily keep the A's sorted and thus intersections can be implemented efficiently using well-known algorithms.

To distribute this design over multiple machines, we partition by the A's. This means each partition (currently, 20) holds a disjoint set of source vertices for the $\mathcal{S}$ data structure; thus, the same B's may reside in multiple partitions. Such a design guarantees that all adjacency list intersections are local to each partition, which eliminates complex cross-partition operations at scale. Note that we can replicate the partitions for both fault tolerance and increased query throughput. The final design is a fairly standard partitioned, replicated architecture with coordination handled by brokers that fan-out queries and gather results.

We note one potential scalability bottleneck with the current design: each partition needs to keep the complete $\mathcal{D}$ data structure (holding the incoming B's to C's), since in principle any B can be in any partition. Thus, every partition needs to handle the entire stream of edge creation events, which creates both network pressure and memory pressure. In practice, we have not found network bandwidth to be an issue, and memory pressure can be alleviated by pruning the $\mathcal{D}$ data structure to only retain the most recent edges (since we desire timely results).

Currently, new incoming edges are inserted into the $\mathcal{D}$ data structures in each partition but these updates are not propagated to the $\mathcal{S}$ data structures (since in principle the new $B \rightarrow C$ edges can be viewed as $A \rightarrow B$ edges for another set of vertices). Technically, there is nothing to prevent us from keeping both data structures updated, but currently the $A \rightarrow B$ edges are computed offline and loaded into the system periodically: this allows us to take advantage of rich features to prune the graph. For users who follow many accounts, in practice we have found it more effective to limit the number of "influencers" (e.g., B's) each user can have. This has the additional benefit of limiting the size of the $\mathcal{S}$ data structures held in memory.

Our system has been serving recommendations in production to Twitter mobile users since September 2013. Each day, billions of raw candidates are generated, yielding millions of push notifications (after eliminating duplicates, suppressing messages during non-waking hours, controlling for fatigue, etc.) The system operates with a median latency of ∼7s and p99 latency of ∼15s, measured from the edge creation event to the delivery of the recommendation. Nearly all the latency comes from event propagation delays in various message queues; the actual graph queries take only a few milliseconds.

3. CONCLUSIONS

The system we have described for making real-time Twitter recommendations has two logical components: the first is the partitioned graph infrastructure that maintains the relevant data structures; the second is the "program" that performs the motif detection. Of course, beyond the "diamond" motif there may exist others that are useful for generating recommendations—these may be implemented as additional programs that use the graph infrastructure (which may need to be augmented to include other data structures). To go even further, we envision the development of a generalized framework where one can declaratively specify a motif, which would yield an optimized query plan against an online graph database. This would seem to represent an entirely new class of data management systems.

4. REFERENCES

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

[2] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. *SODA*, 2002.

[3] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader. Massive streaming data analytics: A case study with clustering coefficients. *MTAAP*, 2010.

[4] J. Gehrke. Special issue on data stream processing. *Bulletin of the Technical Committee on Data Engineering*, 26(1):2, 2003.

[5] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: The Who to Follow service at Twitter. *WWW*, 2013.

[6] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[7] S. Myers, A. Sharma, P. Gupta, and J. Lin. Information network or social network? the structure of the Twitter follow graph. *WWW Companion*, 2014.