RecService: Distributed Real-Time Graph Processing at Twitter

Ajeet Grewal,[1] Jerry Jiang,[1] Gary Lam,[1] Tristan Jung,[1] Lohith Vuddemarri,[1]
Quannan Li,[1] Aaditya Landge,[1] Jimmy Lin[2]

[1] *Twitter, Inc.* [2] *University of Waterloo*

Abstract

We present RecService, a distributed real-time graph processing engine that drives billions of recommendations on Twitter. Real-time recommendations are framed in terms of a user's social context and real-time events incident on that social context, generated from ad hoc point queries and long-lived standing queries. Results form the basis of downstream processes that power a variety of recommendation products. A noteworthy aspect of the system's design is a partitioning scheme whereby manipulations of graph adjacency lists are local to a cluster node. This eliminates cross-node network traffic in query execution, enabling horizontal scalability and avoiding "hot spots" caused by vertices with large degrees.

1 Introduction

Twitter users wish to stay informed about the world, and the goal of the company's recommendation products is to notify them about what's happening in real time. These recommendations identify relevant accounts, Tweets, and other content in a personalized manner, delivered via a variety of mechanisms, including emails and push notifications to users' mobile devices. In this paper, we present RecService, a distributed real-time graph processing engine that powers billions of recommendations in production today.

RecService frames recommendations in terms of two central constructs: a user's social context and real-time events incident on the user's social context. A simplified description of a user's social context is the set of accounts that the user follows (more details later). Real-time events, which we also refer to as engagements, include Tweets, likes, replies, and other Twitter "verbs" that characterize different types of user actions. RecService is designed to power two types of queries: (1) ad hoc "point" queries that reference a particular social context and events incident on it, and (2) long-lived standing queries that trigger when some property of real-time events and a user's social context is satisfied.

RecService is a distributed real-time graph processing engine at Twitter that is provided to teams within the company as a service: on top of RecService, internal customers build applications whose output powers downstream products. This paper describes the architecture of RecService and a number of applications that have been built on top of it. We view our work as having two main contributions: First, framing the real-time recommendation problem as intersections between social context and real-time events is novel, and we provide examples of how such a formulation supports different ways to generate recommendations. Second, we describe a distributed, horizontally-scalable graph-processing engine designed to handle point and standing queries organized around lookup, traversal, and intersection of adjacency lists. A noteworthy aspect of our design is that all graph manipulations are performed on a particular cluster node; that is, we avoid all cross-node network traffic and the shuffling of graph edges. This partitioning scheme is specifically designed to avoid "hot spots" that arise from activity on graph vertices with large degrees.

2 Motivation and Problem Formulation

Twitter has substantial experience building and deploying graph-based recommendation engines. We can identify five distinct systems, listed in roughly chronological order in Table 1, summarizing their most salient features. Perhaps the single biggest lesson we've learned over the years is that recommendations delivered in real time, based on the dynamic state of the graph (i.e., real-time signals), are highly engaging (denoted by the "RT" column)—compared to Cassovary [3], our first system that computed recommendations in batch on static graph snapshots. Our first foray into such real-time systems, MagicRecs [4], can be thought of as a system hard-coded to execute a single standing query and hence highly in-

System	RT?	Algorithm	Impl.	Scaling
Cassovary [3]	no	random walks	custom	up
RealGraph [1]	no	random walks	Hadoop	out
MagicRecs [4]	yes	hard-coded motif	custom	out
GraphJet [11]	yes	random walks	custom	up
RecService	yes	intersections	custom	out

Table 1: Comparisons between several generations of graph-based recommendation systems at Twitter.

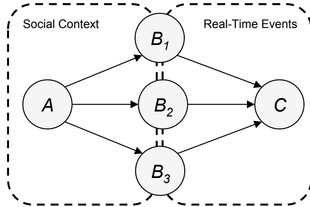

Figure 1: Schematic illustration of social context ($A \rightarrow B$ edges) and real-time events ($B \rightarrow C$ edges).

Trends for you · Change

#Oscars LIVE
Viewing party for the #Oscars

#HereWeAre
@enunge, @mrdonut and 5 more are Tweeting about this

Figure 2: Example use of social proof.

flexible. It paved the way for two systems that are still in production today: GraphJet [11] and RecService, the focus of this paper. The former focuses on random-walk style algorithms, whereas RecService recommendations are based on traversals and intersections of adjacency lists, a generalization of MagicRecs. Other salient characteristics of interest: We have tried building systems on top of Hadoop [1], and general data processing platforms remain involved in pre-processing today, but custom infrastructure provides a better solution for our needs. Finally, Twitter has experimented with both scale-up as well as scale-out architectures; consistent with McSherry et al. [9], another lesson we learned is that scale-up solutions should not be readily dismissed.

System performance (e.g., latency and throughput) and output quality (e.g., relevance) are both important in recommendation products. Production algorithms at Twitter comprise two main phases: candidate generation and refinement. The first identifies a pool of candidates that are then further refined: additional processing involves application of business rules (e.g. don't notify users too frequently) and machine-learned models. Final outputs frequently involve ensembles and draw from different variants as part of ongoing experimentation. The primary role of RecService (and indeed all the systems listed in Table 1) is to provide candidate recommendations to feed downstream pipelines, so we focus only on system performance in this paper.

A schematic illustration of our real-time context approach to recommendation with its two central constructs is shown in Figure 1. Generically, we denote the user for which we're making a recommendation A, and the user's social context as A to B_i edges, where the B_i's can be thought of as the "influencers" of A. In the simplest case the B_i's are accounts that user A follows, but in production this is slightly more complex as we incorporate engagement metrics to refine a user's social context (e.g., if A rarely engages with Tweets from a B_i, it may be pruned from A's social context). Real-time events (engagements) are shown as the B_i to C edges: these edges are typed, capturing activities such as Tweets, likes, replies, engagements with hashtags and entities, etc. RecService supports an arbitrary number of real-time graphs, whatever can be extracted from logs.

It is worth emphasizing that Figure 1 is drawn schematically to illustrate how we frame the recommendation problem. In reality, every user is an A (we aim to make recommendations to everyone); every user is potentially a B (i.e., influences someone) and also a C (involved in some type of activity), although C vertices can represent other objects such as Tweets, hashtags, etc.

Given this model, we provide two examples of point queries for generating recommendations:

- Social Proof. Given a specific user A_q and a hashtag C: who are the people that A_q follows that engaged with C? We call this "social proof" because these queries help contextualize the relevance of certain algorithmic output. For example, Figure 2 shows how such a query can "explain" personalized trends—in this case, because people that you follow are also Tweeting about a particular hashtag.
- Content recommendation. A generalized version of the social proof query described above can be used to provide recommendations for different types of content. For example, what hashtags are a user's influencers including in their posts? This could provide a recommendation to explore new content.

Similarly, we provide two examples of recommendation algorithms based on standing queries:

- Notification of live video broadcasts. We wish to notify users if someone in their social context (i.e., an influencer) starts a live video stream, in which case the notified users might want to tune in.
- Replies to engagement. We can notify a user about replies to Tweets that they have liked from people they follow. This lets users learn about conversations that they might be interested in.

3 System Design

Before detailing the architecture of RecService, we describe a number of assumptions and design choices that come together to yield a scalable, efficient solution to real-time context recommendations.

At a high-level, point queries boil down to sequences of adjacency list lookups, traversals, and intersections. Our previous system MagicRecs [4] demonstrated that standing queries translate into graph motif detection, which can be similarly decomposed into manipulations of graph adjacency lists. However, MagicRecs was limited to a specific physical query plan, and with RecService we aimed to design a more general solution.

To make this problem more tractable, we treat the social context graph and the real-time event graphs very differently. A user's social context ($A \rightarrow B_i$ edges) is static but updated periodically via offline computations. This design choice is based on our observation that social context evolves slowly enough as to not require real-time processing. Viewing the B_i's as influencers of A, influence grows (or wanes) on a timescale that is slower than the periodic updates provided to the system. Unlike the social context, the real-time engagements ($B_i \rightarrow C$ edges) are updated at a high velocity; our design requirement targets $O(10^4)$ events (graph edges) per second.

Our design partitions the social context across multiple nodes in a distributed architecture but *not* the real-time event graphs; that is, they are replicated across all nodes. There are several reasons for this design: in general, the social context graph is larger than the real-time event graphs. Whereas everyone has influencers, engagements tend to be more sparse. Furthermore, social context grows cumulatively (i.e., influence accumulates over time), whereas real-time events are transient, since their value decays (which is the entire point of real-time recommendations). We can tune memory consumption and algorithmic quality in a way that allows full replication of the real-time graphs (more discussion later).

The combination of the above design choices leads to the most interesting and novel aspect of our architecture: a partitioning scheme in which all postings intersection operations are node local. That is, all query operations related to a user A are completely handled by the cluster node responsible for user A. Specifically, we avoid all cross-node network traffic, for example, shuffling of graph adjacency lists (particularly problematic for vertices with large degrees). This ensures horizontal scalability, as we can simply increase the number of partitions to distribute query load without needing to worry about network crosstalk. For point queries, we rely on a service proxy to route the request to the node responsible for user A. For standing queries, each node computes recommendations for a partition of the user space in parallel.

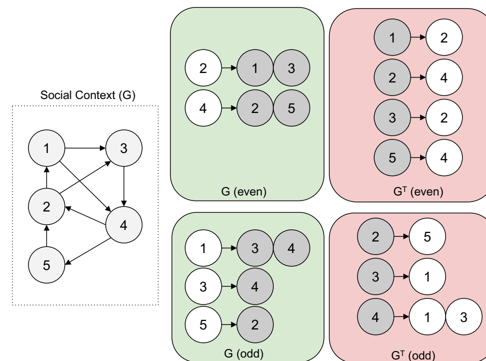

Figure 3: Partitioning of the social context.

In more detail, we co-partition the social context graph and its transpose by A, as shown in Figure 3 (in this case, we partition even and odd A's). The co-partitioning of the graph and its transpose allows easy traversal, both forward and backward, of the social context. That is, we can query who influences A_q as well as who B_q influences. Note that since the social context is computed offline, graph transposes are easy to compute using any batch processing framework.

4 System Architecture

The overall architecture of RecService comprises a service proxy and a number of query processing nodes. Each partition of the social context graph is assigned to a node (see Figure 3). The real-time events are consumed from Kafka by all nodes. Each query processing node is replicated to provide robustness. The overall architecture takes advantage of Zookeeper [5] for service discovery, partition and replica assignment, and graceful failover. We adopt standard best practices, some of which are described in Leibert et al. [7].

A service proxy provides the external API endpoint for point queries. The proxy is responsible for dispatching requests to the node that is responsible for the partition containing the user A_q. The service proxy also transparently handles load balancing across replicas, service discovery as nodes fail and are reinitialized, and a number of common issues associated with distributed systems. The standing queries are "registered" at all the processing nodes, and the triggering of those queries yields candidate recommendations that are written to Kafka, driving downstream processes in a loosely-coupled manner. This represents a completely different code and dataflow path from the point queries.

The architecture of each query processing node is shown in Figure 4. In the middle we show the core graph data structures: the social context on the bottom and the real-time graphs on the top.

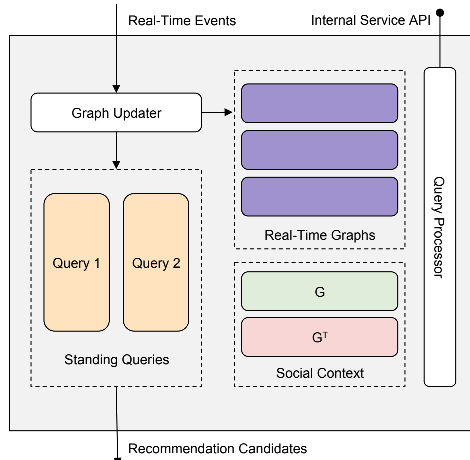

Figure 4: Detail of a query processing node.

4.1 Graph Data Structures

The social context graph is static and computed offline, via a series of MapReduce jobs from the source-of-truth follow graph. These jobs deposit the adjacency lists along with its transpose on HDFS. Each RecService node periodically checks for a new version and loads the new graph if necessary. The MapReduce jobs constructing the social context pre-partition the graph on HDFS, simplifying the bulk load process into RecService. Storage of the graph data as adjacency lists supports querying by A_q to retrieve all outgoing edges (corresponding to the influencers of A_q) as well as by B_q to retrieve all incoming edges (corresponding to the users influenced by B_q).

The social context graph partitions are stored using an implementation inspired by cdb, short for "constant database", which provides an on-disk associative array that can be memory mapped, allowing us to swap versions of the social context atomically. The query performance of our implementation is on par with in-memory hashmaps, but data loading is two orders of magnitude faster since we do not need to insert key–values pairs one by one. The MapReduce jobs creating the graph partitions are orchestrated to write the format needed by RecService directly.

The real-time events are read from Kafka and also organized as adjacency lists, maintained using techniques similar to GraphJet [11], Twitter's other real-time graph processing engine. This is handled by the graph updater in the upper left corner of Figure 4. Note that these edges are typed, corresponding to different engagements (e.g., likes, replies, etc.). Each real-time graph is stored separately, and the only mode of accessing each of these graphs is querying by B_q to retrieve all outgoing typed edges to C's (i.e., the adjacency lists). Quite explicitly, we do not store the transpose of the real-time graphs, for efficiency considerations.

Since the stream of engagement events is endless, we must bound memory usage. This is managed in a coarse-grained manner by dividing each graph into temporal segments with a maximum number of edges. Once a segment is full, a new segment is created, and the old segment is dropped (also similar to GraphJet [11]). Conceptually, we are maintaining a moving window of recent events. As discussed in the context of GraphJet, we have noticed that the value of engagements in providing high-quality recommendations falls off as the event ages. Thus, storage management of the real-time graphs is a balancing act between resource consumption (primarily memory) and algorithmic quality.

4.2 Query Processing

On the right in Figure 4 is the query processor for point queries. Each node exposes an internal service API, which receives dispatches from the service proxy.

Point queries translate into adjacency list lookups, traversals, and intersections on graph structures described above. For example, in the "social proof" query (see Section 2), the $B \rightarrow C$ edges are users and the hashtags they have engaged with. Given a user A_q and a hashtag C_h, the service proxy first dispatches the query to the node containing the user A_q. The system looks up all the influencers of A_q and checks if any of them have engaged with the hashtag C_h by consulting the real-time graph. As another example, suppose we wish to recommend hashtags to a user A_r. The system looks up the top influencers for A_r and examines the hashtags (i.e., the C's) that they have engaged with. This is an intersection operation on the adjacency lists of the B_i's, which yields a list of N potential hashtag recommendations.

The final component of each query node is the executor for the standing queries (the left side of Figure 4). After every engagement event is updated in the real-time graphs, it is dispatched to all registered standing queries. As an example, we describe the execution of the "live video broadcast" query (see Section 2), where we want to notify users of live video streams by their influencers: For this particular application, a $B \rightarrow C$ edge represents a user B who just initiated a live video broadcast C. Each node receives this engagement event, looks up all the A_i users that are influenced by the broadcaster B, and writes these results to a Kafka topic.

This particular application also illustrates how our partitioning scheme distributes query load across all processing nodes, particularly for "supernodes" with large vertex degrees. For example, if a highly-popular broadcaster B (i.e., with millions of followers) begins streaming video, we may need to dispatch notifications to many users. Since we partition by A, these users are distributed across all the query processing nodes; the recommenda-

tions are computed in parallel without a single "hot spot" and written to a Kafka topic.

Note that for both point and standing queries, RecService provides the initial candidate recommendations. There is substantial downstream processing beyond the scope of this paper—refinement by machine-learned models and the application of business logic—before recommendations are finally delivered to users.

5 Deployment and Evaluation

Development of RecService began in late 2015 and the system first went into production in early 2016. Currently, we logically partition the social context graph into 256 partitions, distributed among 64 physical machines. For robustness we employ 2× replication, so in total we have 128 query processing nodes. Each of these machines are commodity servers with two Intel Xeon processors (E5-2620 v2) and 32 GB of memory.

As detailed in Section 4, each query processing node holds a partition of the social context graph and the real-time graphs. Memory pressure for the social context graph can be managed by increasing the number of partitions. Memory consumption of the real-time graphs is managed separately for each type of engagement. For example, we store about a day's worth of hashtag engagements but only about eight hours of engagements associated with topic entities extracted from Tweets. These temporal thresholds are heuristically tuned to meet product requirements. Storing more events doesn't necessarily yield better results, since events age at different rates in terms of contributions to algorithmic quality.

The real-time graphs are built by reading engagement events from Kafka. On average, about 12k events per second are ingested with a peak of about 17k events per second. The hashtag engagement graph receives around 2.8k edges per second and the topic entities graph receives around 1.9k edges per second. The median latency between an event being written to Kafka (by an upstream process) and the corresponding graph edge being accurately reflected in the real-time graphs is 150ms, with a 99th percentile latency of 430ms.

At present, the system serves about 20k point queries per second, with a median and 99th percentile latency of 2ms and 13ms, respectively. There are currently only a couple of standing queries in production, although we have a handful of other prototypes not yet fully deployed. Once the triggering event is processed, the execution of standing queries is very similar to point queries in terms of performance. The latency between an engagement event arriving in Kafka and the output of standing queries being written back to Kafka is well under one second. At present, roughly 450 recommendations per second are generated by the standing queries.

6 Related Work

Beyond Twitter, there is of course a large body of work on graph processing, and a proper survey is beyond the scope of this paper. However, we can divide related work into two categories at a high level. On the one hand, there are a number of analytical graph processing frameworks (see MuCune et al. [8] for a survey). These are generally focused on efficiently implementing large traversals on static graphs and do not handle real-time graph updates for the most part. On the other hand, there are a number of graph stores—the most well known being Neo4j—which are targeted at real-time update workloads and point queries (although Pacaci et al. [10] argue that even traditional RDBMSes provide good performance for real-time graph workloads). Most of these graph stores do not handle standing queries, although Graphflow [6] is a notable exception.

In general, Twitter systems for graph-based recommendations, including RecService, were not designed as general-purpose graph engines, unlike nearly all systems referenced above. Instead, our focus might be best characterized as building general *domain-specific* graph processing engines. Each of the systems described in Section 2 reflect a particular formulation of the recommendation problem. For example, RecService encodes a real-time context model of recommendations; GraphJet assumes a bipartite user–content interaction graph. The inclusion of domain-specific constructs leads to designs that provide clever solutions to common challenges in graph processing—in this case, partitioning that supports node-local operations. These designs would be difficult to implement in the general case.

7 Future Work and Conclusions

RecService continues to be in active development inside Twitter and there are a number of ongoing efforts: We are continually adding new queries to power new downstream products. Point queries are easy to understand, but we find that standing queries are conceptually more challenging to formulate, which explains why there are relatively few that have been productionized. We are exploring ways to help algorithm designers reason about highly-dynamic graphs.

Overall, the capabilities that Twitter has developed around graph-based recommendations, as captured in several generations of systems, are reasonably mature. One direction of future work is how to integrate graph-based signals with other sources that have not received as much attention, e.g., Tweet content [2] and location. However, the ultimate goal remains the same: to surface relevant, personalized, and timely content to inform users around the world.

8 Acknowledgments

We'd like to thank the anonymous reviewers for their insightful comments and the guidance of our shepherd in helping to significantly improve this paper from submission to the final version.

References

[1] GOEL, A., SHARMA, A., WANG, D., AND YIN, Z. Discovering similar users on Twitter. In *Proceedings of the Eleventh Workshop on Mining and Learning with Graphs* (Chicago, Illinois, 2013).

[2] GREWAL, A., AND LIN, J. The evolution of content analysis for personalized recommendations at Twitter. In *Proceedings of the 41st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2018)* (Ann Arbor, Michigan, 2018).

[3] GUPTA, P., GOEL, A., LIN, J., SHARMA, A., WANG, D., AND ZADEH, R. WTF: The Who to Follow service at Twitter. In *Proceedings of the 22nd International World Wide Web Conference (WWW 2013)* (Rio de Janeiro, Brazil, 2013), pp. 505–514.

[4] GUPTA, P., SATULURI, V., GREWAL, A., GURUMURTHY, S., ZHABIUK, V., LI, Q., AND LIN, J. Real-time Twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment 7*, 13 (2014), 1379–1380.

[5] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX 2010)* (Boston, Massachusetts, 2010), pp. 145–158.

[6] KANKANAMGE, C., SAHU, S., MHEDBHI, A., CHEN, J., AND SALIHOGLU, S. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 2017)* (Chicago, Illinois, 2017), pp. 1695–1698.

[7] LEIBERT, F., MANNIX, J., LIN, J., AND HAMADANI, B. Automatic management of partitioned, replicated search services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC '11)* (Cascais, Portugal, 2011).

[8] MCCUNE, R. R., WENINGER, T., AND MADEY, G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv. 48*, 2 (Oct. 2015), 25:1–25:39.

[9] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! But at what COST? In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, 2015).

[10] PACACI, A., ZHOU, A., LIN, J., AND ÖZSU, M. T. Do we need specialized graph databases? Benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES'17)* (Chicago, Illinois, 2017).

[11] SHARMA, A., JIANG, J., BOMMANNAVAR, P., LARSON, B., AND LIN, J. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment 9*, 13 (2016), 1281–1292.