GraphJet: Real-Time Content Recommendations at Twitter

Aneesh Sharma[1], Jerry Jiang[1], Praveen Bommannavar[1], Brian Larson[1], and Jimmy Lin[2]

[1] Twitter, Inc., San Francisco, California, USA
[2] University of Waterloo, Waterloo, Ontario, Canada

@aneeshs @jerryjiang @praveenbom @larsonite @lintool

ABSTRACT

This paper presents GraphJet, a new graph-based system for generating content recommendations at Twitter. As motivation, we trace the evolution of our formulation and approach to the graph recommendation problem, embodied in successive generations of systems. Two trends can be identified: supplementing batch with real-time processing and a broadening of the scope of recommendations from users to content. Both of these trends come together in GraphJet, an in-memory graph processing engine that maintains a real-time bipartite interaction graph between users and tweets. The storage engine implements a simple API, but one that is sufficiently expressive to support a range of recommendation algorithms based on random walks that we have refined over the years. Similar to Cassovary, a previous graph recommendation engine developed at Twitter, GraphJet assumes that the entire graph can be held in memory on a single server. The system organizes the interaction graph into temporally-partitioned index segments that hold adjacency lists. GraphJet is able to support rapid ingestion of edges while concurrently serving lookup queries through a combination of compact edge encoding and a dynamic memory allocation scheme that exploits power-law characteristics of the graph. Each GraphJet server ingests up to one million graph edges per second, and in steady state, computes up to 500 recommendations per second, which translates into several million edge read operations per second.

1. INTRODUCTION

Graph-based recommendations form an integral component of Twitter in helping to promote an active and engaged user base by suggesting new connections between users and between users and content. Recommendations can be generated based on shared interests, correlated activities, topological configurations, and a multitude of other signals. This paper traces the evolution of several systems for tackling this challenge at Twitter, culminating in a look at GraphJet, a recently-deployed system for generating content recommendations from the interaction graph in real time.

In terms of how the graph recommendation problem has been formulated and tackled within Twitter, we can identify two overall trends:

1. We observe the evolution from batch processing to real-time processing. Initially, our algorithms were formulated to operate on periodic graph snapshots, generating recommendations in batch. Newer algorithms, in contrast, are designed to operate on the real-time graph.
2. We observe a broadening of the scope of graph recommendations. The first-generation service, Wtf ("Who to Follow") [17], had a name that reflected its scope: to recommend accounts that Twitter users should follow. In later systems, we realized that Wtf algorithms could be generalized to graphs derived from user behavior and other contextual signals, allowing us to recommend not only users, but content (i.e., tweets) as well.

In terms of recommendation algorithms, we have found that random walks, particularly over bipartite graphs, work well for generating high-engagement recommendations. Although conceptually simple, random-walk algorithms define a large design space that supports customization for a wide range of application scenarios, for recommendations in different contexts (web, mobile, email digests, etc.) as well as entirely unrelated applications (e.g., social search). The output of our random-walk algorithms can serve as input to machine-learned models that further increase the quality of recommendations, but in many cases, the output is sufficiently relevant for direct user consumption.

In terms of production infrastructure for generating graph recommendations, the deployed systems at Twitter have always gone "against the grain" of conventional wisdom. When many in the community were focused on building distributed graph stores, we built a solution (circa 2010) based on retaining the entire graph in memory on a single machine (i.e., no partitioning) [17]. This unorthodox design decision enabled Twitter to rapidly develop and deploy a missing feature in the service (see Section 2.1). Later, when there was much activity in the space of graph processing frameworks rushing to replace MapReduce, we abandoned the in-memory system and reimplemented our algorithms in Hadoop MapReduce (circa 2012). Once again, this might seem like another strange design decision (see Section 2.2). Most recently, we have supplemented Hadoop-based recommendations with custom infrastructure, first with a system called MagicRecs [18] (see Section 2.3) and culminating in GraphJet, the focus of this paper.

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

Contributions. We view this paper as having two main contributions. First, we trace the evolution of graph-based recommendation services at Twitter across successive generations of systems. Although others have written about production graph systems (e.g., [30, 5, 40]), we attempt to go beyond just describing a single system to comparing several generations of systems with similar goals. Why did we abandon a particular system to build a completely new one from scratch (and multiple times)? The answer is related to how we have reformulated and reconceived of the graph recommendation challenge over the years, which we share in this paper. Although there are path dependencies and other factors idiosyncratic to Twitter and our operational context, to a large extent the evolution of our thinking reflects (and in some cases, leads) general industry trends. As such, our experiences are perhaps of interest to a broad audience.

The second contribution of this paper is a detailed look at GraphJet, a recently-deployed system for real-time content recommendations. The storage engine maintains a real-time bipartite interaction graph between users and tweets. Similar to Cassovary [17], a previous graph recommendation engine developed at Twitter, GraphJet assumes that the entire graph can be held in memory on a single server. The system organizes the live interaction graph into temporally-partitioned index segments that hold adjacency lists. GraphJet is able to support rapid ingestion of graph edges while serving concurrent lookup queries through a combination of compact edge encoding and a dynamic memory allocation scheme that exploits power-law characteristics of the graph. The storage engine implements a simple API, but one that is sufficiently expressive to support a range of recommendation algorithms based on random walks that we have refined over the years. We describe a few of the recommendation algorithms that are built on top of the GraphJet storage engine, along with the production deployment environment and some performance figures. Each individual GraphJet server is able to ingest up to one million graph edges per second, and in steady state, each server computes up to 500 recommendation per second, which translates into several million edge read operations per second.

2. SYSTEM EVOLUTION

2.1 WTF and Cassovary

Graph recommendations began at Twitter in spring 2010 with the Wtf ("Who to Follow") project [17], which focused on providing user recommendations. Prior to that, no such feature existed in Twitter, which was a significant product gap. Thus, quickly launching a high-quality service was a top priority. Wtf was a success: the service was built in a few months and the product launched in summer 2010. It has contributed substantially to enriching connections between Twitter users [14] and it has inspired subsequent generations of systems within the organization.

One of the key enablers of this rapid deployment was what many might consider an unconventional design choice: to assume that the entire graph fits into memory *on a single server*. While the prevailing wisdom was (and is) to design distributed, horizontally-partitioned, scale-out infrastructure, we took exactly the opposite approach of scaling up on individual large-memory (but still commodity) servers. The critical question was: Is the graph growing slower than Moore's Law is providing commodity servers with more memory? If so, we can simply buy new machines periodically (or even just upgrade memory) and stay ahead of the growth curve. Projections we had run at that time (based on historical data) answered this question in the affirmative out to a reasonable time horizon, giving us the empirical support needed to proceed with this design choice.

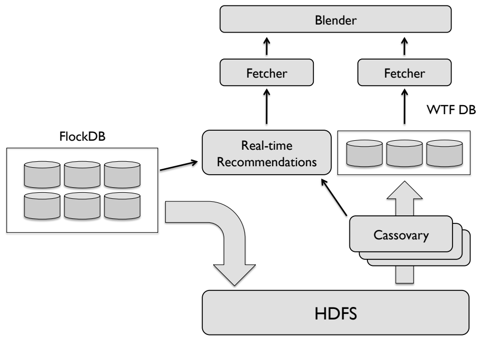

Figure 1: Overall architecture of the "Who to Follow" user recommendation service.

As an aside, although distributed graph stores and graph processing engines are interesting, we wonder if the focus and effort that the academic community places on this class of solutions overstates its importance and relevance for solving real-world problems. Consider a graph with ten billion edges: even a naïve representation as an edge list would occupy a mere 80 GB, which is well in the range of memory available on commodity servers today. Along these lines, see additional commentary by Lin [25].

2.1.1 Overall Architecture

The overall Wtf architecture is shown in Figure 1. At the core is the Cassovary in-memory graph processing engine [37], a custom system we wrote from scratch and later open sourced.[1] Cassovary operates on snapshots of the follower graph loaded from HDFS; the graph snapshots in HDFS are in turn ingested from the frontend graph store (FlockDB) daily. The storage layer of Cassovary provides access to the graph via vertex-based queries and a recommendation engine computes the actual who-to-follow suggestions. These are materialized and stored in a sharded MySQL database, called, not surprisingly, the Wtf DB.

In steady state, Cassovary servers constantly generate recommendations for users, consuming from a distributed queue containing users sorted by a "last refresh" timestamp. The actual API endpoints fetch recommendations from the Wtf DB in response to requests from the web frontends or users' mobile apps. Since the graph is stored entirely in memory on each server, the production architecture is straightforward: we simply replicate Cassovary server instances as needed to achieve a particular throughput. Each instance reads from the work queue and writes its output to the Wtf DB, so there is no need for any explicit coordination mechanism.

Due to the load incurred on the frontend graph store by the snapshot and import process into HDFS, it is not practical to update the in-memory Cassovary graphs more frequently than once a day. This, however, is problematic for new users, since recommendations will not be immediately available upon joining Twitter. Making high-quality recommendations to new users is challenging since they have few

[1] `https://github.com/twitter/cassovary`

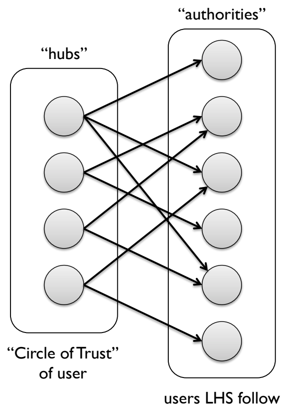

Figure 2: Illustration of our SALSA-based algorithm for generating user recommendations.

connections to begin with—this issue is generally known as the "cold start" problem in the recommender systems community. This challenge is addressed by a completely different code path (and set of algorithms) for real-time recommendations, specifically targeting new users (see Figure 1).

2.1.2 Recommendation Algorithms

We describe two Cassovary algorithms that have proven to be effective in production and are surprisingly robust. Similar algorithms are also implemented in GraphJet. Note that in reality, production deployment involves constant A/B tests that combine a wide range of techniques, so these algorithms are better thought of as primitive "building blocks" from which end-to-end algorithms might be composed.

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 (i.e., personalized PageRank [13, 4]). Cassovary computes the circle of trust on demand given a set of parameters: reset probability, edge pruning settings, etc.

SALSA for User Recommendations. We have developed a user recommendation algorithm based on SALSA (Stochastic Approach for Link-Structure Analysis) [23], a random-walk algorithm in the same family as HITS [19] originally developed for web search ranking. The algorithm constructs a bipartite graph from a collection of websites of interest ("hubs" on one side and "authorities" on the other). Each step in SALSA traverses two edges—one forward and one backward (or vice-versa).

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 (from above). 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 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 recommendations (i.e., "similar to you") in different contexts.

We believe that this algorithm is effective because it captures the recursive nature of the user recommendation problem. 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. SALSA operationalizes 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 vertices in both directions. See Gupta et al. [17] for some metrics comparing the engagement of SALSA, personalized PageRank, and a few standard recommendation algorithms.

2.2 Hadoop and the RealGraph

After launching Wtf and gaining production experience with graph recommendations, we began building its successor, which was deployed circa 2012 [15]. The impetus for building a Cassovary replacement was the desire to exploit a broader range of signals to generate recommendations, in particular, signals contained in behavior logs. For such data, it seemed clear that the assumption of being able to hold everything in memory on a single machine would no longer be realistic. At that time, we were very much of the following mindset: How do we scale the algorithms that had proven effective in Cassovary to a larger, richer graph comprised of not only user follows, but behavioral signals as well? As we retrospectively tell the story now, the gap might seem obvious, but back then we had yet to realize the importance of generating graph recommendations in real time, and thus were still focused on batch solutions.

For this second generation graph recommendation system, we made the interesting design choice of building on top of the Hadoop analytics platform within the organization, which at that time primarily used Pig [33] (and thus, MapReduce) for data processing. This design decision warrants some discussion: although by that time the shortcomings of MapReduce for graph analytics were well known (e.g., [17, 28], among many other places), no production-ready processing platform specifically designed for graph analytics existed (although there were a few research systems); see Section 7 for additional discussion.

We began by performing a detailed analysis of where the bottlenecks were in our existing algorithms: as it turned out, most of our algorithms could be expressed without much iteration. Poor performance in handling iterative algorithms is one of the biggest shortcomings of MapReduce—but we were able to engineer around this. In the most naïve implementation of the Cassovary algorithms, we would materialize the neighborhoods of the vertices for which we are generating the recommendation and run the random walks, which could be encapsulated within a user-defined function (UDF). The bottleneck was in the materialization of the neighborhoods, which required shuffling an enormous amount of data (adjacency lists) across the cluster nodes—for this, alternative processing frameworks (even a graph-centric framework like Pregel [30]) would not be of much help.

What we needed was an *algorithmic* advance to deal with the data shuffling issue, not just a more efficient processing framework. The solution came from a variety of sampling techniques [15] that significantly decreased the inherent complexity of our recommendation algorithms (although the tradeoff was the introduction of approximations). Once we had addressed the data shuffling issue, it really didn't matter what processing framework we used, since the algorithms were mostly encapsulated in UDFs. Using Pig, running on Hadoop, made sense because it exploited existing

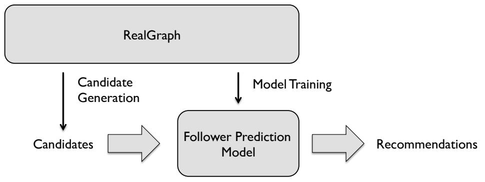

Figure 3: The RealGraph and the Hadoop-based recommendation pipeline.

production infrastructure for managing the feature pipeline, job scheduling, error reporting, etc.

There were additional non-technical considerations pushing us toward a Hadoop solution. Although Cassovary ran on commodity servers, they nevertheless had substantially more memory than most other servers in the datacenter (at least initially, although later on, other parts of the organization began experimenting with scale-up solutions). From an operations perspective, it is easier to manager large fleets of homogeneous servers with identical configurations. At the time, Twitter was also transitioning to a new framework for allocating computing resources, and it made sense to exploit general-purpose Hadoop infrastructure.

The overall setup of the Hadoop-based recommendation pipeline is shown in Figure 3. What we call the RealGraph is a composite representation of the follower graph and interaction graphs that are induced from behavior logs. This representation is built from multiple feature pipelines that extract, clean, transform, and join features from raw log data; see Lin and Ryaboy [27] for more details. The RealGraph is stored on HDFS, and hence we have the freedom to incorporate as many signals as we have available.

The RealGraph is used in two main ways: First, reimplementations of Cassovary algorithms in Pig generate user recommendations in batch. Second, the RealGraph and log data are used to train a follower prediction model, operationalized as a classifier; see Lin and Kolcz [26] for more details on how classifier training is integrated into Pig. The output of the first stage provides candidates that are fed into the second stage classifier to generate the final recommendations; see Goel et al. [15] for more details.

2.3 MagicRecs

At this point in the evolution of graph recommendation systems at Twitter (circa 2012), nearly all recommendations were generated in batch at roughly daily intervals. This, of course, was dissonant with the core "Twitter experience", which emphasized the real-time aspects of the global conversation, be it about celebrity gossip, world affairs, or the activities of loved ones. Day-old batch recommendations did not seem to exploit the advantages of Twitter.

The metrics supported this intuition—recommendations perform well for new users and users whose recommendations had been recently recomputed. In the first case, WTF had a dedicated code path for new users, which by necessity uses real-time signals (see Figure 1). In the second case, the recommendations were generated using "fresh" signals based on users' recent activities. We observed that recommendations based on more recent signals tended to be more engaging. Taking this to the logical conclusion, what if we generated recommendations in real time?

These ideas were prototyped in a Twitter account (initially protected, i.e., not publicly visible) called @magicrecs.

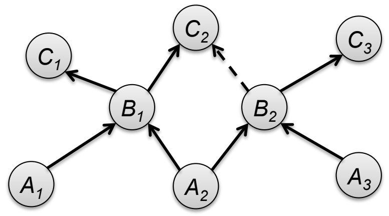

Figure 4: Sample fragment of the follow graph illustrating MagicRecs: when the edge $B_2 \rightarrow C_2$ is created, we want to push C_2 to A_2 as a recommendation.

The account sent recommendations computed in real time to followers via direct messages. As a proof of concept, the initial prototype implemented a brute-force approach that wasn't particularly scalable, but the point was to demonstrate the viability of real-time recommendations. Initial experiments were successful, which led to a formal plan to build a real-time recommendations product. This included addressing the scalability challenges, which could be characterized as a streaming graph processing problem.[2]

Here's how @magicrecs worked: Suppose we wish to make user recommendations to a particular user A (see Figure 4). 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). 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 4, we want to push C_2 to A_2 as a recommendation. In practice, the recommendations are delivered via push notifications to users' mobile devices [38]. Note that the above 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). Furthermore, although we have explained this idea in terms of the follow graph, it applies to recommending content as well, based on user actions such as retweets, likes, etc.

Why does this technique work? Push notifications must be relevant, personalized, and timely, or else they run the risk of becoming an annoyance. Our experience with previous systems highlighted the importance of local network signals for content discovery. Twitter users curate accounts they follow, which represent their interests as well as social and professional connections. Temporally-correlated activities of a user's followings 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). Indeed, we empirically observe that real-time recommendations generated in this manner achieve high engagement.

Interestingly, this particular formulation of the graph recommendation problem is completely different from the traditional setup. In the standard formulation, we are given a user u and the system's task is to compute a list of recommendations. In our case, we observe a stream of graph edges, where we ask the question: given the addition of a particular edge e from u to v, what recommendations can we make to whom? The recommendations usually do not directly involve either u or v, but rather their respective local neighborhoods.

[2]For a more personal account of how MagicRecs came about, see a blog post by Gupta [16].

We explored a wide range of potential solutions to this real-time recommendation problem on streaming graphs. It was obvious that our Hadoop-based infrastructure could not be reused; Cassovary assumed a static graph and thus was not applicable either. Initial explorations quickly confirmed what we already suspected: that a general-purpose, real-time processing platform such as Storm [36] was not adequate for our needs. Beyond not achieving the latency guarantees that we were after, the tuple-oriented nature of the framework made it difficult to express graph-centric operations. Ultimately, it became clear that we needed to build custom infrastructure to solve this problem.

Our solution was described in Gupta et al. [18]; here we briefly summarize the technical insights. We realized that the task of identifying B to C edges within some temporal interval (Figure 4) can be reformulated as an intersection of adjacency lists. That is, if we stored the outgoing edges of the A vertices and the incoming edges of the C vertices, we can intersect the adjacency lists of A and C to arrive at the B's and check the cardinality. Thus, for each incoming edge we perform these intersection queries and are able to identify the graph configuration in real time. The final system partitions the graph in such a way that all these adjacency list intersections can be performed locally on each node, thereby eliminating cross-node traffic for computing the recommendations. Overall, the system achieves a median latency of $\sim$7s, 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. GRAPHJET OVERVIEW

3.1 Goals and Approach

Our success with MagicRecs demonstrated the power of real-time recommendations, which take advantage of Twitter's strength as the medium in which users discuss "what's happening right now". The MagicRecs implementation, however, was designed for a specialized task, and we felt it was necessary to build a more general-purpose graph storage engine that is able to support richer recommendation algorithms. Given our previous experiences, we had a pretty good idea of the range of operations that needed to be supported. Our philosophy was to design the API to support the minimal set of operations necessary to express our target recommendation algorithms. As a consequence of this philosophy, many might be surprised by some of the missing features in our system (which we discuss in detail). These decisions, however, gave us the ability to optimize in ways not otherwise possible. The final product is GraphJet.

One early design decision that we revived from Cassovary is to assume that the entire graph will fit in memory on a single machine. This assumption is a particularly good fit for the interaction graph as it evolves rapidly, and user attention in Twitter content wanes with time. Thus, the focus on real-time recommendations imposes natural restrictions on the size of the graph and its potential growth—real-time algorithms necessarily depend on recent signals, which means that we only need to maintain the graph across a moving window, and that interactions beyond a certain point can be discarded. Therefore, the graph for our purposes does not grow without bound.

3.2 Data Model and API

Formally, GraphJet manages a dynamic, sparse, undirected bipartite graph $G = (U, T, E)$, where the left-hand side U represents users, the right-hand side T represents tweets, and the edges E represent interactions over a temporal window. For simplicity, we assume that vertices (on the U and T sides) are identified by 64-bit integers—this choice ensures that we are not in danger of exhausting the id space anytime soon. We assume that edges have a particular type $r \in R$, but that the cardinality of $|R|$ is relatively small and fixed—these types correspond to Twitter "verbs" such as likes, retweets, etc. and so this assumption is realistic. Note, quite explicitly, that edges are not timestamped (more discussion below). Most of the memory requirements for storing the graph comes from the continuous insertion of edges that correspond to user interactions. GraphJet maintains vertex metadata, but the memory requirements for these are not particularly onerous.

Although conceptually, the bipartite graph is undirected, in practice the edges are stored internally in adjacency lists that implicitly convey directionality. We can build a left-to-right index, which for $u \in U$ provides the edges that are incident to u, with their destination vertices and edge types, i.e., (t, r) tuples where $t \in T$ and $r \in R$. Correspondingly, we can build a right-to-left index, which for $t \in T$ provides the edges that are incident to t, with their destination vertices and edge types, i.e., (u, r) tuples where $u \in U$ and $r \in R$. Depending on the desired data access pattern, we can build a left-to-right index, a right-to-left index, or both—with different memory requirements.

At the storage layer, GraphJet implements a simple API comprised of five main methods:

`insertEdge(u, t, r)`: inserts an edge from user `u` to tweet `t` of type `r` into the bipartite graph.

`getLeftVertexEdges(u)`: returns an iterator over the edges incident to `u`, a user on the left-hand side. More precisely, the iterator provides (t, r) tuples where $t \in T, r \in R$. The method makes no guarantee with respect to the order in which the edges are returned, but in the current implementation the iteration order is the order in which the edges are inserted into the graph.

`getLeftVertexRandomEdges(u, k)`: returns `k` edges uniformly sampled with replacement from the edges incident to vertex `u`, a user on the left-hand side. Results are returned as an iterator over (t, r) tuples where $t \in T, r \in R$. Note that since we're sampling with replacement, edges might be returned more than once, particularly in cases where the degree of the vertex is smaller than `k`.

`getRightVertexEdges(t)`: returns an iterator over the edges incident to `t`, a tweet on the right-hand side, with the same contract as the left variant.

`getRightVertexRandomEdges(t, k)`: returns `k` edges uniformly sampled with replacement from the edges incident to vertex `t`, a tweet on the right-hand side, with the same contract as the left variant.

The storage engine guarantees that calls to the get methods will return an iterator that corresponds to a consistent state of the graph at call time, and that all sampling probabilities are accurate. State necessary to ensure these guarantees is captured in the iterator directly. This means, for example, that newly arrived edges will not be visible while the iterator is being traversed. The storage engine, however, makes no guarantees with respect to the state of the graph *across* multiple get calls—multiple gets on the same vertex may indeed return different numbers of edges.

There are a few aspects of our data model and API worth discussing. First, our API does not support edge deletions. This was an explicit design decision and is justified by our data model. Edges in our bipartite graph capture interactions between users and tweets—a user retweeting a post, for example—and such interactions represent point events that have no duration. Furthermore, it is not meaningful to talk about deleting such edges, since such interactions cannot usually be undone.[3] From the implementation perspective, not needing to support deletes greatly simplifies many aspects of the implementation.

If the API does not support deletes, what prevents the graph from growing without bound? Our bipartite graph captures user–tweet interactions over a temporal window of the previous n hours: stale interactions beyond the window are not considered as those edges are periodically pruned in a coarse-grained manner (described in Section 3.3).

Another design decision worth discussing is the fact that edge timestamps are not stored. This represents a straightforward tradeoff between space (memory consumption) and the marginal quality gain of recommendation algorithms that take advantage of edge timestamps. Offline experiments show that varying the size of the temporal window of the graph *does* have an impact on recommendation quality, but beyond knowing that an interaction happened within the last n hours, we've found it hard to design algorithms that provide substantial gains by exploiting edge timestamps. One factor is that most interactions occur close to the tweet creation time (which is already encoded implicitly in the tweet id since they are assigned in a monotonically increasing manner); if we, for example, weighted edges based on time in our random walk algorithms (see Section 5), the end results would not be substantially different. Given the substantial increase in storage requirements necessary to hold the timestamps, we made a tradeoff against this in the current system, but it is a potential future direction.

Although the data model of GraphJet is an undirected bipartite graph, the system can be adapted to store standard directed or undirected graphs in a straightforward manner. If we set the vertices on the left-hand and right-hand sides to be the same, and build a left-to-right index, we can model a standard directed graph, where the `getLeftVertexEdges` method (and the sampling variant) retrieve outgoing edges with respect to a vertex. If we build a right-to-left index, we can retrieve incoming edges to a vertex using the `getRightVertexEdges` method (and the sampling variant).

In our experience, the directionality of the edges is to a large extent an application-specific semantic property. If we wish to access a vertex in terms of incoming edges, it is entirely equivalent to reversing the directionality of the edges and accessing outgoing edges. As a concrete example, it might make sense to store users who have retweeted a tweet (edges from tweets to users) or tweets that have been retweeted by a particular user (edges from user to tweets), depending on the desired access patterns.

[3]Although it is possible to undo some actions such as "likes", accidental clicks are almost always undone immediately, which means that they can be handled as part of the ETL process prior to data ingestion.

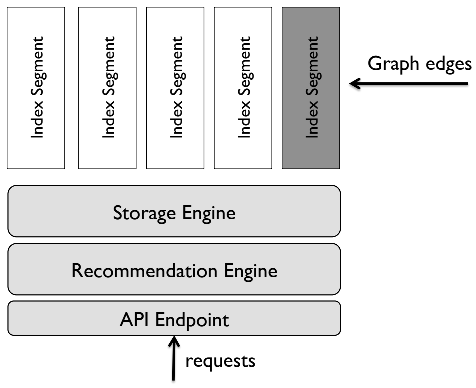

Figure 5: Overall architecture of GraphJet.

To summarize, GraphJet maintains a bipartite graph that keeps track of user–tweet interactions over the most recent n hours. By supporting the simplest possible API that is sufficient to capture our problem domain, the system is able to achieve high performance in both ingestion of real-time interactions and generation of recommendations.

3.3 Overall Architecture

The overall architecture of GraphJet is shown in Figure 5. The entire system is implemented in Java. The storage engine maintains a list of temporally-ordered index segments comprised of adjacency lists that store a partition of the bipartite graph. In particular, the left-to-right index maps from a vertex $u \in U$ to an adjacency list, which is an arbitrary number of $(t \in T, r \in R)$ tuples each representing an edge incident to the vertex. Since the right-to-left index is exactly symmetric, we can discuss the index structures in a generic manner. The storage engine provides raw access to the graph in support of the recommendation engine. The entire system presents an API endpoint for external clients to request recommendations.

GraphJet temporally partitions the adjacency lists by periodically creating a new index from scratch, as shown in Figure 5. In our current implementation, new partitions are triggered once the number of edges reaches a threshold, so that each partition is roughly the same size, but alternative criteria could include temporal windows or a combination of size and temporal window. Centrally, GraphJet maintains the degree distribution for each vertex in each temporal segment—as we describe later, this is used for efficient sampling (see Section 4.2). In the current implementation, each GraphJet instance maintains a handful of segments.

In this design, only the latest index segment is actively being written into (see Figure 5); the remaining are read-only. To eliminate the need for complex synchronization mechanisms, all edge insertions are handled by a single writer thread, reading from a Kafka queue (although read queries are served by multiple threads). We have found that a single writer thread supports more than sufficient write throughput (see Section 6). Once an index segment stops accepting new edges, we can optimize its contents to support more efficient reads (see Section 4.1.3). Periodically, an entire index segment is discarded if it is older than n hours, so that memory consumption does not increase without bound. Thus, the pruning of the graph happens in a rather coarse-grained manner. Experiments show that this does not have a noticeable impact on recommendation quality.

Temporal partitioning of the bipartite graph has the additional advantage that the number of unique vertices contained in each partition can be bounded, and thus we can map global 64-bit ids into segment-internal ids that require less space (discussed in detail below). This simple optimization yields substantial space savings.

4. GRAPH OPERATIONS

4.1 Edge Insertions

We begin by describing how edge insertions are handled. Since we adopt a single-writer, multi-reader design, there is no need to worry about write–write conflicts. The input to GraphJet is a stream of edge insertions in the form of (u, t, r) triples, where the user u and the tweet t are 64-bit integers (longs); r is the edge type, whose cardinality we assume is small. Our biggest challenge is that the system must concurrently support rapid ingestion of edges as well as high-volume reads, and thus we need to strike a balance between write-optimized and read-optimized data structures.

4.1.1 Id Mapping

Since each index segment holds a small set of vertices relative to the 64-bit id space, we can reduce memory requirements by building a (bi-directional) mapping between the external (i.e., canonical) vertex id and an internal vertex id that is unique only within the particular index segment. The is accomplished by hashing the external vertex id and treating the hash value as the internal vertex id. We use a standard double hashing technique, an open-addressing scheme whereby the initial probe site in the hash table (i.e., array) is determined by one hash function, and the skip interval between linear probes in case of collisions is determined by a second hash function. We store the external vertex id in the hash table, so that recovering those ids can be accomplished by a simple array lookup. For convenience, we size the hash table to be a power of two so that mod can be accomplished via efficient bit operations.

Because we are using the hash value as our internal vertex id, we cannot rehash the entries (e.g., to grow the table). Thus, we must take care in the initial sizing of our hash table. Fortunately, we can use historical data as a guide. Suppose we wish to store n vertices at a load factor f: we size our initial hash table to be 2^b where $b = \lceil \lg(n/f) \rceil$. We fill the hash table until we insert $f \cdot 2^b$ vertices, after which we allocate another hash table of size 2^{b-1} to hold the next $f \cdot 2^{b-1}$ vertices. For these new vertices, the internal vertex id is the hash value in the new table plus 2^b, so that we can unambiguously determine the external vertex id when performing the mapping in reverse. The process can be repeated as often as necessary, but we can tune the index segment size and the initial table size so that additional allocations are rare events.

We further optimize by bit-packing both the edge type and internal vertex id in a single 32-bit integer. This optimization is safe because we can tune the segment size so that we never overflow the allocated space. For example, we might reserve three bits to support eight edge types, which gives us room for 2^{29} (approximately 537 million) unique vertex ids. This straightforward optimization means that, within each index segment, an adjacency list is simply an array of 32-bit integers. Inserting new edges into the graph simply becomes the process of looking up the adjacency list (array) associated with the vertex and writing an integer into the next available location. Since we only have a single insertion thread, there are no consistency issues that we need to worry about. However, the challenge we must address is memory allocation for the adjacency lists.

4.1.2 Memory Allocation

Let us first sketch out the design space for allocating memory for adjacency lists: At the highest level, the adjacency list for a vertex is either maintained in a physically contiguous region in memory, or it is not. Maintaining physical contiguity optimizes for reads by eliminating pointer chasing and maximizing processor pre-fetching opportunities. However, it is impossible to maintain contiguity without relocating existing adjacency lists as the graph evolves, since there is no way to predict the degree distribution of a specific vertex in advance. Unfortunately, copying around (potentially large) memory regions is a slow operation. In giving up memory contiguity, a system makes the opposite tradeoff: edge insertions can be significantly faster, but reads might involve pointer chasing due to fragmentation. This is indeed the decision we made for GraphJet, to optimize data structures for edge insertion. We are able to tolerate discontiguous adjacency lists because sampling edges is a common operation, and the nature of sampling means that we do not have regular data access patterns to begin with (and hence pointer chasing is less of a concern).

Accepting that adjacency lists will be broken into discontiguous pieces (which we call *slices*) does not actually answer the question of how to allocate memory for storing edges. In particular, how much memory should we allocate for a given vertex? Selecting a value that is too large leads to inefficient memory utilization, because most of the allocated space will remain empty (most tweets have little interaction). On the other hand, selecting a value that is too small leads to inefficiencies. Imagine a tweet by a celebrity that receives a large number of interactions very quickly (in the left-to-right index): the number of edges might grow very quickly, and if we are overly conservative in allocating space, we might have to repeatedly allocate new arrays, and memory allocation is relatively costly. Furthermore, highly fragmented adjacency list slices are wasteful in terms of memory usage due to the need for addressing pointers that link the slices together. In the limit, allocating space for one edge at a time is the same as a linked list, which is obviously inefficient.

Our solution is to allocate memory for adjacency lists in slices that double in size each time we run out of space, i.e., the slice sizes grow as powers of two. The first time we encounter a vertex, space for two edges (i.e., an array for two integers) is allocated. The inserted edge fills the first available position in that array, leaving room for another edge. By the time we encounter a third edge, we allocate a new slice for storing $2^2 = 4$ edges; after we fill that, we allocate a new slice for $2^3 = 8$ edges; and so on. Such a memory allocation strategy is sensible because many aspects of Twitter graphs follow power-law distributions [31]. Our allocation strategy implicitly assumes some type of preferential attachment effect, since it is an easy way to explain the existence of power-law distributions: the more that we observe an edge incident to a vertex, the more likely that more edges will follow. Hence, it makes sense to exponentially increase the amount of allocated space each time. Expanding by two is simply the most convenient.

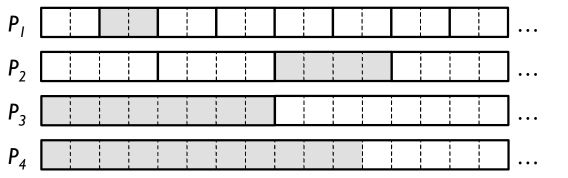

Figure 6: Illustration of the edge pools. Shaded boxes represent the adjacency list slices of a vertex with degree 25, specified by $P_1(1), P_2(2), P_3(0), P_4(0)$.

We organize the adjacency list slices by grouping all slices of the same size together in what we call edge pools, which are implemented as large arrays. The first edge pool P_1 is of length $2^1 \cdot n$, where n is the number of expected vertices in a particular segment (the same value for tuning the hash table size above, estimated from historical data). We denote the k-th "slot" in the first edge pool as $P_1(k)$—the actual offset in the underlying array can be easily computed from the slice size and k. The second edge pool is denoted P_2 and holds $n/2$ adjacency list slices, each of size 2^2. Generalizing, the r-th edge pool is denoted P_r and holds $n/2^{r-1}$ slices of size 2^r. Thus, the adjacency list of a vertex v with degree d (in the active segment) can be specified as follows:

$$v \rightarrow d : P_1(k_1), P_2(k_2), P_3(k_3), P_4(k_4), P_5(k_5)...$$

That is, the first 2^1 edges are in slot k_1 in pool P_1, the next 2^2 edges are in slot k_2 in pool P_2, etc. Furthermore, since the slice sizes are fixed, we know from the vertex degree where to insert the next edge and how much space is left in the current slice.

To provide a concrete example, suppose vertex v_1 with degree 25 has the following adjacency list:

$$v_1 \rightarrow 25 : P_1(1), P_2(2), P_3(0), P_4(0)$$

The situation is illustrated in Figure 6, where the light gray areas correspond to filled adjacency list slices of v_1 (slots are indexed from zero). Since the degree of v_1 is 25, we know that there is space for 5 more edges in the 2^4 slice in P_4 (since $2^1 + 2^2 + 2^3 + 2^4 = 30$).

A few more details suffice to round out the description of how GraphJet manages memory allocation for adjacency lists. Edge pools containing slices for a particular size are allocated lazily—that is, the pools themselves are not created until they are necessary. In practice, the maximum number of edge pools is bounded because we control the size of each index segment, so there is no danger of overflow. Finally, each edge pool itself has an initial size of $2n$ (as described above); if this space is exhausted, we grow the pool size by 10% each time, as many times as needed. Note that as with the hash table size, we can tune the index segment size so that these are rare events.

4.1.3 *Read-Only Optimizations*

In our design, the GraphJet storage engine manages multiple temporally-partitioned index segments, only one of which (the most recent one) ingests new edges. This means that the rest of the index segments are immutable, and hence we can reorganize the physical memory layout of the various data structures to support more efficient read operations. Once an index segment stops accepting new edges, a background thread begins optimizing the storage layout of that segment, creating a shadow copy. When the process is completed, the storage engine atomically swaps in the new segment and the original version is dropped.

The current optimizations in the read-only segments are relatively straightforward: since the graph partition is now immutable, we no longer need the edge pool structure to store the adjacency lists. Because we know the final degree of each vertex, we can lay out each adjacency list end to end in a large array without any gaps. To access the edges, we simply keep a pointer to the beginning of each adjacency list. This layout guarantees that iteration over the edges of a particular vertex will touch contiguous regions in memory, thereby eliminating the overhead of pointer chasing. When sampling edges, we also increase the probability that multiple samples reside on the same cache line.

4.2 Edge Lookups and Sampling

We turn our attention next to how read operations are handled in GraphJet. Since there is only a single writer ingesting new graph edges, judicious use of memory barriers is sufficient to address memory visibility issues across multiple threads. Memory barriers are sufficiently lightweight that the performance penalties are acceptable.

The `getLeftVertexEdges` method returns an iterator that traverses all edges incident to a vertex on the left-hand side of the bipartite graph (there is also the symmetric call for fetching vertices from the right-hand side, implemented in the same way). The returned iterator is a composition of iterators over edges in all index segments, from earliest to latest. Since in each segment the edges are stored in insertion order, the overall iteration order is also insertion order. Within each segment, a vertex is associated with its degree and a list of pointers into edge pools that comprise the actual adjacency list (i.e., the slices). The actual index positions into the integer arrays backing each edge pool can be easily constructed from the degree information and the list of slice pointers. To guarantee consistency, the iterator is initialized with the degree of the vertex in the most recent (active) index segment, which prevents the client from reading edges inserted after the API call.

The `getLeftVertexRandomEdges` method returns an iterator over k edges uniformly sampled with replacement from the edges incident to a vertex on the left hand side (and the corresponding call for the right-hand side is exactly symmetric). Sampling randomly from within a particular index segment is easy: we know the degree d and so we can sample a random integer uniformly from the range $[0, d-1]$, which can then be easily mapped into an array location in the underlying edge pools. What about sampling across index segments? Recall that the storage engine keeps track of vertex degrees across all segments. We can normalize these degrees into a probability distribution from which we sample, where the probability of selecting an index segment is proportional to the number of edges in that segment. If we sample index segments in this (biased) manner and then sample uniformly within each segment, it is equivalent to drawing a random sample from all the edges.

Fortunately, sampling from a discrete probability distribution efficiently is a problem that has been well studied. We use the well-known alias method, which takes $O(n)$ preprocessing time (where n is the number of index segments), after which values can be drawn in $O(1)$ time [39]. The alias method requires the construction of two tables, a probability table and an alias table, which is performed when the

API is called to capture the degree distribution across the segments at that time. The two generated tables are encapsulated in the iterator, along with other state information to ensure that edges added after the API call are not visible. To sample each edge, the iterator first uses the alias method to determine the index segment to sample from, an then draws a uniformly random sample from edges in that index segment.

5. REAL-TIME RECOMMENDATIONS

GraphJet supports two types of queries: content recommendation queries—for a user, compute a list of tweets that the user might be interested in, and similarity queries—for a given tweet, compute a list of related tweets. Here, we describe three different algorithms, which should be thought of as "building blocks" from which the actual deployed algorithms are built. For certain use cases, GraphJet output is best viewed as a set of candidates that are further reranked and filtered by machine-learned models.

5.1 Full SALSA

A large portion of the traffic to GraphJet comes from clients who request content recommendations for a particular user. An example is our home timeline recommendations service, where we wish to compute a list of tweets that a user might be interested in at a given moment. To serve this query, we run a personalized SALSA algorithm, similar to the WTF algorithm described in Section 2.1.2, on the interaction graph in GraphJet. The main difference, of course, is that the algorithm exploits real-time signals.

The simplest form of this query is to begin with the vertex in the bipartite interaction graph that corresponds to the query user and run SALSA [23]. This involves performing the following random walk: starting from a vertex on the left-hand side u, select one of its incident edges uniformly to visit t, a vertex on the right-hand side. Then, from t, select one of its incident edges uniformly to walk back to the left-hand side. This is repeated an odd number of steps. To introduce personalization, we include a *reset* step on the left-to-right edge traversals: with a fixed probability α, we restart from the query vertex to ensure that the random walk doesn't "stray" too far from the query vertex, which is the same intuition behind personalized PageRank [13, 4]. At the end of this random walk, we can compute a distribution of visits for the right-hand vertices.

On top of this starting point, we introduce an additional refinement. In some cases, the query user may not exist in the interaction graph—if the user hasn't been active recently. To address this, we start the random walk from a *seed set* instead of a single user vertex. The seed set is configurable, but a common choice is the circle of trust of the user, as discussed in Section 2.1.2. Intuitively, this makes sense, since users are interested in current consumption patterns in their egocentric networks. When starting from a seed set, for the random walk restarts, we select a random vertex from the seed set with equal probability.

The output of this full SALSA algorithm is a ranked list of vertices on the right-hand side of the bipartite interaction graph. Note that these tweets may not have any direct interactions with the seed set at all. This is a strength of the algorithm for users with a sparse seed set, but it can lead to noisy results for users with a dense seed set. Furthermore, having direct public interactions (such as retweets and likes) enables the product to serve a *social proof*—explaining why a particular recommendation was generated. This typically results in better user understanding of the recommendations, leading to higher engagement.

5.2 Subgraph SALSA

Suppose we wish to only serve recommendations for which we can generate some "social proof". This can be accomplished in our random walk algorithm by restricting the output set to only neighbors (on the right-hand side) of the seed set. The fanout of the seed set is usually not very large, and moreover we can downsample edges from seed set vertices to bound the vertex degrees (to avoid creating overly-unbalanced graphs). This produces a subgraph of the complete bipartite interaction graph that we can then run our SALSA algorithm on.

This particular algorithm takes a seed set (for example, a user's circle of trust), constructs (i.e., materializes) a small subgraph in memory, and runs the following PageRank-like algorithm: We begin with a uniform distribution of weights in the seed set (summing to one). In each left-to-right iteration, we take the weight $w(u)$ of each vertex u and divide it equally among u's neighbors, i.e., each neighbor t of u receives weight $w(u)/d(u)$ where $d(u)$ is the degree of u. Each right-hand vertex sums the weights received from the left-hand side. The same weight distribution process from the right-hand side vertices to their left-hand neighbors constitutes a symmetric right-to-left iteration. We iterate until convergence.

The output of this algorithm is a list of ranked right-hand vertices (tweets), as in the full SALSA case. We make a few observations: From the performance standpoint, this algorithm is much faster than the full SALSA version since it only needs to access the complete interaction graph once to materialize the subgraph. The subgraph usually fits into cache, and so the weight distribution steps benefit from good reference locality. Additionally, this algorithm only requires a left-to-right segment index, so the memory consumption in GraphJet is roughly half of the fully-indexed case. From the algorithmic perspective, note that this approach ignores second-order interaction information: specifically, it ignores all paths from the seed set to the fanout vertices through other user vertices that are outside the seed set. This in essence is the tradeoff between the full and subgraph versions of our SALSA algorithm—both have their uses, but in different circumstances.

5.3 Similarity

In addition to content recommendation queries, GraphJet also supports similarity queries that return all vertices (of the same type) similar to a query vertex. For concreteness, we focus on queries for tweets, but the symmetric process applies to users as well.

Specifically, given a query vertex t, we wish to find other vertices that receive engagement from a similar set of users that t received engagements from. This can be formalized in terms of cosine similarity between two vertices t_1 and t_2:

$$\mathrm{Sim}(t_1, t_2) = \frac{N(t_1) \cap N(t_2)}{\sqrt{|N(t_1)| \cdot |N(t_2)|}}$$

where $N(t)$ denotes the left-side neighbors of vertex t. Thus, for a query vertex t, our goal is to construct a ranked list of tweets with respect the similarity metric.

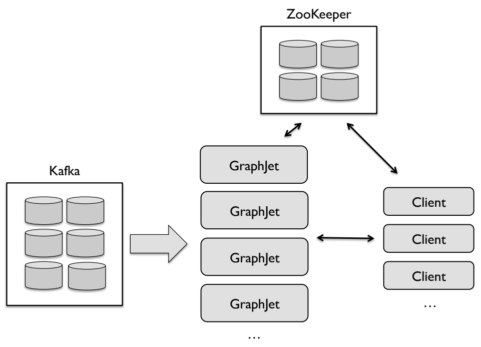

Figure 7: GraphJet deployment architecture.

Our algorithm for this is also based on random walks: given a vertex t, we first sample its neighborhood $N(t)$ to obtain a set $N_S(t)$. Then, for each vertex $u \in N_S(t)$ we sample u's neighbors to produce a set of candidates that might be similar to t. We can set weights on each of the sampling steps such that the expected number of visits for each candidate t' is proportional to $\mathrm{Sim}(t, t')$. By repeating this sampling process many times, we can compute both the candidate vertices and estimates of their similarities. The algorithm then returns these candidates sorted by similarity.

We describe this algorithm only at a high level because these sampling techniques are quite similar to ones that have been described in previous work [15]. However, we do note a few interesting aspects: The choice of cosine similarity can be easily changed to any similarity metric that can be computed via set intersections, for example, Jaccard similarity. Also, the similarity query can be augmented with a *seed set* in an analogous manner to the content recommendation queries: we observe that computing similar tweets to a set of given tweets often produces higher quality results.

6. DEPLOYMENT AND PERFORMANCE

The deployment architecture of GraphJet is shown in Figure 7. At the core, we have a fleet of servers each running a GraphJet instance. Since each individual server holds a complete copy of the interaction graph, fault tolerance is handled by replication, which also determines the overall throughput of the service. Each GraphJet instance ingests graph edges from a Kafka queue of user–tweet interaction events, which is a filtered and cleaned version of the "client events" stream from Twitter's unified logging pipeline; see Lee et al. [21] for more details. In steady state, the bipartite interaction graph stored in each GraphJet instance can hold $O(10^9)$ edges using less than 30 GB of RAM.

GraphJet provides a Thrift-based API for recommendation requests. All server instances are viewed as a ServerSet, which is an abstraction for a replicated service [22]. The ServerSet abstraction is widely used within Twitter and generically handles issues such as load-balancing (performed on the client side), retries, backoffs, etc., and is resilient to many common failure scenarios such as "thundering herds" (a scenario in which all clients are successively redirected to the same backend server, thus causing cascading failures). Service discovery is handled by ZooKeeper: GraphJet servers register with ZooKeeper at a known location when they are ready to serve traffic, and clients consult the service registry for request routing and load balancing.

GraphJet began receiving production traffic in early 2014. The first use case was to discover content beyond a new user's direct network and to inject the content into the user's home timeline. This addressed the cold start problem of integrating new users into Twitter. Today, GraphJet results are deployed much more broadly: beyond injecting interesting and relevant tweets into a user's home timeline, the system also powers "activity digest" emails that are sent periodically to users and live push notifications to mobile devices. At the beginning of 2016, the entire service receives tens of thousands of recommendation requests per second, with results reaching many millions of users each day.

In what follows, we report on the performance of a single GraphJet instance in a production environment. The following metrics were measured on our current hardware configuration, which consists of servers with two Intel Xeon 6-core processors (E5-2620 v2) running at 2.10 GHz.

With a cold start (server restart or when new servers join the service), GraphJet is able to sustain one million edge insertions per second reading from the Kafka queue to "catch up" to the current state of the graph. After this initial bootstrap phase, the ingestion rate drops down to tens of thousands of edges per second, which is the typical engagement rate for the types of interactions we are interested in.

In a stress test within a production environment, we measured client-observed latency (i.e., including network latency, RPC overhead, etc.) for a recommendation request using the subgraph SALSA algorithm. An individual GraphJet server is able to support up to 500 recommendation requests per second, with a latency profile of p50 = 19ms, p90 = 27ms, and p99 = 33ms at that load. Note that this represents the complete end-to-end latency for computing a recommendation, which includes fetching the relevant portions of the interaction graph to construct the bipartite subgraph, as well as running the recommendation algorithm itself (i.e., multiple weight-distribution passes until convergence). Each individual recommendation request translates into many API calls to the underlying storage engine, which based on our rough calculations correspond to several million edge read operations per second. This represents a lower bound on the raw performance of the underlying storage engine, since the recommendation engine needs to perform computations beyond edge lookups. It is difficult to benchmark the storage engine directly due to the challenge of generating a realistic edge query load in isolation.

In terms of availability, over a typical 30-day period, the success rate of the system in delivering recommendations, as measured from the clients, is consistently above 99.99% in terms of requests across all users. This high availability stems from our simple design. Our internal Kafka queue is fully replicated across multiple data centers, and therefore GraphJet only needs to handle read-path failures. We have implemented cross-datacenter failover in reading from Kafka; even in the case of catastrophic Kafka failure, GraphJet can continue serving recommendations (albeit with increasingly stale data in memory). Once Kafka recovers, the GraphJet instances can resume ingesting new edges and "catch up" in a completely transparent manner.

7. RELATED WORK

7.1 Production Recommendation Systems

In this paper, we have attempted to trace the broad arc of how graph recommendations systems at Twitter have evolved over the years. Although there is a tremendous amount of work on recommender systems in the academic literature, relatively few papers describe actual production systems, and those that do (e.g., [29, 10, 1]) focus mostly on algorithms, as opposed to systems infrastructure—which is the perspective we take.

A well-known case study of "real-world" recommendations comes from the Netflix Prize, which in 2006 offered an award of $1 million dollars to improve Netflix's recommendations by 10% in terms of root mean squared error (RMSE) and provided public data for training algorithms to accomplish this goal. In 2009 this milestone was achieved; the Netflix team evaluated the winning solution but found that "the additional accuracy gains that we measured [compared to relatively simple baselines] did not seem to justify the engineering effort needed to bring them [the winning methods] into a production environment" [32]. Furthermore, the Netflix team spoke of a fundamental shift from US DVDs to global streaming as reshaping the very nature of the recommendation problem—they pointed out the growing importance of real-time recommendations in this context. This experience resonates with us and the Netflix case study highlights a gap between the academic study of recommendation algorithms and the practice of building production recommender systems, and it is precisely this gap that we hope to help narrow.

7.2 Graph Analytics Frameworks

The shortcomings of MapReduce for graph processing are well-known in the research community [28]. Today, there are of course many graph analytics frameworks that address the limitations of MapReduce, but at the time when we were planning a replacement for Cassovary (circa 2011), there were no viable alternatives. Pregel was known [30], but the open-source Giraph implementation did not exist yet. There were a few academic research papers on iterative formulations of MapReduce [12, 41], but nothing that was close to production ready. Therefore, at the time, implementing the RealGraph on Hadoop seemed like a reasonable decision, which we discussed in Section 2.2.

Later, when we recognized the importance of real-time processing (circa 2012), we did build initial prototypes on Storm. As we discussed in Section 2.3, it was not sufficient for our needs both in terms of performance and expressivity. This was before the development of Heron, Spark Streaming, and Flink—today, there are more choices in the space of real-time processing platforms. However, these alternative are still tuple-oriented, and so the challenge of expressing graph-centric operations remains. We believe that we made the right decision to build custom infrastructure in MagicRecs and GraphJet.

7.3 Real-Time Graph Stores

The general architecture of GraphJet borrows heavily from Earlybird, Twitter's production real-time search engine [7]. Early on, we drew the connection between managing postings lists in real-time search and managing adjacency lists in real-time graphs. The issue of memory allocation for postings lists was studied in depth separately [3], thus providing us with a good understanding of the design space. Temporal partitioning of index segments was also a design borrowed from Earlybird, which allowed us to prune the interaction graph in an efficient (but coarse-grained) manner.

Tracing the intellectual lineage of Earlybird further back, there is substantial work in the information retrieval literature on incremental indexing ([9, 35, 6, 8, 24], just to provide a few examples) that is relevant to building inverted indexes in real time. Although most of the work considered on-disk indexes, a different point in the design space, many of the tradeoffs, for example, between contiguous and discontiguous index segments (Section 4.1.2), were already well known; see Asadi and Lin [2] for a recent study. In this respect, GraphJet acknowledges and builds on related work dating back at least a few decades.

It is, of course, possible to use any key–value store to hold the adjacency lists that comprise a graph, thus serving as a real-time graph store (cf. [34]). A wide range of key–value stores are available (a survey is beyond the scope of this paper), but Redis in particular supports a command (LPUSH) that inserts specified values at the head of the list stored at a key. The implementation of the command, however, lacks the memory allocation optimizations in GraphJet. Furthermore, Redis lacks a mechanism for pruning these lists; although it would be possible to implement temporal partitioning, it would basically be replicating some of the main design features in GraphJet.

In the space of graph stores specifically designed for non-static graphs, we are aware of a number of commercial offerings (e.g., Neo4j, Titan, OrientDB) and research systems. The research systems implement a range of different strategies for managing evolving graph data structures, including linked lists of edge blocks in Stinger [11], compacting circular buffers in Trinity [34], and partitioned edge buffers in GraphChi-DB [20]. We have not undertaken a study comparing these different techniques, but such an evaluation would be interesting future work.

8. CONCLUSIONS

This paper traces the evolution of graph recommendations at Twitter over four generations of systems: Cassovary, RealGraph, MagicRecs, and GraphJet. All of these systems were essentially from-scratch implementations, not because we were enamored with system building, but because each represented a reformulation of the graph recommendation problem that highlighted fundamental shortcomings in the previous systems. This progression highlights a pragmatic, solution-oriented approach to system design and building, and we hope that our experience can help bridge the gap between the academic study of recommendation algorithms and the practice of building production systems.

9. ACKNOWLEDGMENTS

We'd like to acknowledge the many engineers who have contributed to the systems described in this paper. Specifically for GraphJet, thanks to Xiaoyang Gu and Guang Xiang for early contributions, Michael Busch for helpful feedback, and team members from a hackweek project that explored an early prototype (Ashish Goel, Gilad Mishne, and Babak Hamadani). Additional thanks to Pankaj Gupta for helpful comments on an earlier draft of this paper.

10. REFERENCES

[1] D. Agarwal, B.-C. Chen, Q. He, Z. Hua, G. Lebanon, Y. Ma, P. Shivaswamy, H.-P. Tseng, J. Yang, and L. Zhang. Personalizing LinkedIn feed. *KDD*, 2015.

[2] N. Asadi and J. Lin. An exploration of postings list contiguity in main-memory incremental indexing. *WSDM Workshop on Large-Scale and Distributed Systems for Information Retrieval*, 2014.

[3] N. Asadi, J. Lin, and M. Busch. Dynamic memory allocation policies for postings in real-time Twitter search. *KDD*, 2013.

[4] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized PageRank on MapReduce. *SIGMOD*, 2011.

[5] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. *USENIX ATC*, 2013.

[6] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. *VLDB*, 1994.

[7] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. *ICDE*, 2012.

[8] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time trade-offs in dynamic information retrieval systems. *CIKM*, 2005.

[9] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. *SIGIR*, 1990.

[10] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. *WWW*, 2007.

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

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

[13] 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.

[14] A. Goel, P. Gupta, J. Sirois, D. Wang, A. Sharma, and S. Gurumurthy. The Who-To-Follow system at Twitter: Strategy, algorithms, and revenue impact. *Interfaces*, 45(1):98–107, 2015.

[15] A. Goel, A. Sharma, D. Wang, and Z. Yin. Discovering similar users on Twitter. *MLG*, 2013.

[16] P. Gupta. You can't stop real-time. `https://medium.com/@pankaj/you-cant-stop-real-time-9e8d33644e7b`, 2013.

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

[18] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin. Real-time Twitter recommendation: Online motif detection in large dynamic graphs. *VLDB*, 2014.

[19] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *JACM*, 46(5):604–632, 1999.

[20] A. Kyrola and C. Guestrin. GraphChi-DB: Simple design for a scalable graph database system – on just a PC. *arXiv:1403.0701v1*, 2014.

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

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

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

[24] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM TODS*, 33(3):19, 2008.

[25] J. Lin. Is big data a transient problem? *IEEE Internet Computing*, 19(5):86–90, 2015.

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

[27] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: The Twitter experience. *SIGKDD Explorations*, 14(2):6–19, 2012.

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

[29] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

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

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

[32] Netflix. Netflix recommendations: Beyond the 5 stars. `http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`, 2012.

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

[34] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. *SIGMOD*, 2013.

[35] A. Tomasic, H. Garcia-Molina, and K. A. Shoens. Incremental updates of inverted lists for text document retrieval. *SIGMOD*, 1994.

[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm @Twitter. *SIGMOD*, 2014.

[37] Twitter. Cassovary: A big graph-processing library. `https://blog.twitter.com/2012/cassovary-a-big-graph-processing-library`, 2012.

[38] Twitter. Stay in the know. `https://blog.twitter.com/2013/stay-in-the-know`, 2013.

[39] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM TOMS*, 3(3):253–256, 1977.

[40] R. Wang, C. Conrad, and S. Shah. Using set cover to optimize a large-scale low latency distributed graph. *HotCloud*, 2013.

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