Mining Dynamic Graph Streams for Predictive Queries Under Resource Constraints

Xuanming Liu and Tingjian Ge(✉)

University of Massachusetts, Lowell, USA
{xliu,ge}@cs.uml.edu

Abstract. Knowledge graph streams are a data model underlying many online dynamic data applications today. Answering predictive relationship queries over such a stream is very challenging as the heterogeneous graph streams imply complex topological and temporal correlations of knowledge facts, as well as fast dynamic incoming rates and statistical pattern changes over time. We present our approach with two major components: a Count-Fading sketch and an online incremental embedding algorithm. We answer predictive relationship queries using the embedding results. Extensive experiments over real world datasets show that our approach significantly outperforms two baseline approaches, producing accurate query results efficiently with a small memory footprint.

AQ1

1 Introduction

The *knowledge graph* model is widely used to represent online data [5,13,15]. It consists of (h, r, t) triples, where h is the head entity, t is the tail entity, and r is their relationship. Each triple is called a *knowledge fact*, and there are typically multiple types of vertices and relationships. With an ever-increasing amount of ubiquitous data captured online, this model also provides rich structural semantics to data streams. For example, in communication networks, road traffic graphs, and user-product-purchase real-time graphs used by companies such as Amazon [4], dynamic knowledge facts stream in. Thus, the model comprises a dynamic portion which is a *graph stream* [18]—a sequence of knowledge-fact edges with timestamps, as well as an optional static graph portion.

Let us look at some examples. **Traffic and commute** are an integral part of people's life. Dense and dynamic traffic data has been collected and made available online as a service, such as the New York taxi cab data [2]. It contains taxi trip information, including pick-up and drop-off locations, trip start time and duration, number of passengers, among others.

This knowledge graph stream is illustrated in Fig. 1(a). We partition the whole geographic area into a dense grid, where each vertex corresponds to a 0.5 mile by 0.5 mile square area. Two neighboring vertices are connected by a *static*

This work is supported in part by NSF grant IIS-1633271.

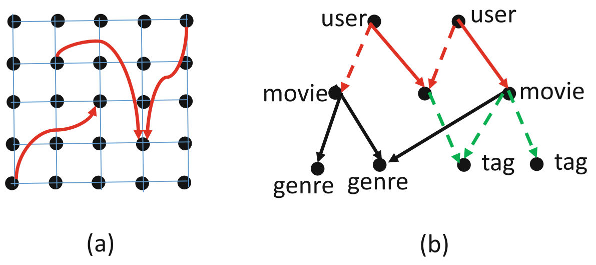

Fig. 1. (a) Taxi trip information network, (b) dynamic user-movie information graph. (Color figure online)

thin edge in the figure, indicating the "proximity" relationship (an undirected edge is treated as two directed edges in both directions). The bold red edges denote trip information, i.e., from a grid point to another during a time interval. These are dynamic edges and can be of two relationships: "fast", or "slow" (compared to past statistics).

As an example in a different domain, companies, e.g., Amazon, are collecting information about **users, product items, purchase/rating history** as online data for business and service. Another example is dynamic movie rating and tagging as in MovieLens [1], illustrated in Fig. 1(b). There are four types of vertices—users, movies, genres, and tags, and four relationship types: movie $\rightarrow$ genre (black solid edges), movie $\rightarrow$ tag (green dashed edges), user "likes" a movie (red solid edges), and user "dislikes" a movie (red dashed edges), with the former two being static and the latter two being dynamic with timestamps.

Predictive Relationship Queries. Given a knowledge graph stream, we focus on *predictive relationship queries*. As in previous work such as [13], we follow the *local closed world* assumption: for a target knowledge (h, r, t) not in the graph, if there exists (h, r, t') where $t' \neq t$, then we consider (h, r, t) as false; otherwise it is unknown—indeed, knowledge facts are too abundant and a system may not have the time/resource to acquire all. The relationship queries are based on the predictive results of unknown edges; the details are in Sects. 2 and 4. To see a simple example, for relationship temporal joins, in Fig. 1(a), one may be interested in querying the correlation of two trips over time—when each trip's fastness/slowness property is treated as a sequence. The result of such queries would be useful for traffic planning and management.

Resource Constraints. In Fig. 1(a), the number of vertices can be thousands or millions in a large area, and the total number of edges can be a quadratic function of that. Likewise in Fig. 1(b), there is a large number of high-rate concurrent purchases and rating edges. There are always resource constraints for real-time stream processing. In order to answer queries online in real time, we may only access a limited amount of memory, and cannot afford to access disks. This is more so with the trend of *edge computing* [14], where more processing is pushed to smaller or less powerful devices at the edge of a cloud system. Furthermore, even in memory-abundant server environment, *sketches* are needed as a small, lightweight component for approximate analysis within a broader stream processing pipeline [9]. Sharing similar motivations as ours, sketches are used for predictive linear classifiers over data streams [21].

We first devise a novel sketch called a Count-Fading (CF) sketch (Sect. 3); then we extend previous work on knowledge graph embedding [22] to the context of graph streams, using CF. This in turn helps us answer relationship queries (Sect. 4). Finally, we perform comprehensive experiments that demonstrate the effectiveness and efficiency of our solution (Sect. 5).

Related Work. *Sketches*, as data summaries, have been studied for data streams. It starts from Bloom filters [6]. Later ones include AMS sketch [3] and Count-Min (CM) sketch [12]. There has also been attempt to add time decay into a sketch. Cafaro et al. [10] combine a "forward decay" model with CM sketch and a Spacing-Saving algorithm to solve the problem of mining *frequent items*. Our CF sketch is significantly different from previous work, and targets a completely different problem—serving dynamic graph stream embedding. We study the choice of sketch size based on dynamic incoming rate and allowed false-positive rate. Moreover, CF sketches dynamically grow/shrink in response to the fluctuation of stream rates, which has not been addressed in previous work.

Knowledge embedding refers to a technique that models multi-relational data by constructing latent representations for entities and relationships. Researchers developed translation-based embedding model to jointly model entities and relationships within a single latent space, such as translational embedding (transE) [7], translation on hyperplanes (transH) [23], and relation-specific entity embedding (transR) [16]. However, graph stream embedding is an open problem [11]. Our work is an endeavor towards this direction. Moreover, we target a more diverse set of predictive relationship queries than link prediction. To our knowledge, these have not been studied before.

2 Problem Formulation

A *knowledge graph stream* is represented as $\mathcal{G} = (V, E_s \cup E_d, R)$, where V is the set of vertices, E_s is a set of static edges, E_d is a set of dynamic edges, and R is a set of relationships. Every edge $e \in E_d$ has a timestamp $ts(e)$ corresponding to the edge's arrival time. Each edge $e \in E_s \cup E_d$ is in the form of (h, r, t), where $h \in V$ and $t \in V$ are called the *head* and *tail*, respectively, and $r \in R$ is a *relationship*. Given $\mathcal{G}$, we answer predictive relationship queries below.

A basic one is *local relationship queries*. At time t, given vertices $u, v \in V$, and a relationship $r \in R$, a local relationship query asks for the probability that a knowledge fact (u, r, v) holds at time t. Another close one is called a *relationship ranking query*. Given multiple edges (u_1, r_1, v_1), ..., (u_k, r_k, v_k) (e.g., a special case is $r_1 = r_2 = ... = r_k$), the query asks to rank these k predictive edges in the order of their probabilities. We call the above two types *basic predictive relationship queries*. In Sect. 4, we also discuss *relationship temporal joins*, *user defined relationships*, and *global relationship queries* as *extended types*.

3 A Count-Fading Sketch

Basic Scheme. We devise a novel time-adaptive dynamic-size sketch based on the Count-Min (CM) sketch [12], and call it a *Count-Fading* (CF) sketch. Some background on CM is in the Supplementary Material [17]. CF significantly extends the CM in Fig. 2(a). Our goal is to track ever-increasing continuous counts of items (knowledge fact edges), with higher weights for recent ones. Secondly, we need our sketch to last, remaining low errors, and adapt to data stream bursts. As illustrated in Fig. 2(b), CF has dynamic *versions* that grow/shrink over time. Let us first focus on a single version, version 1 in Fig. 2(b) (to the left of a vertical divider corresponding to time up to some value T_1). Each row i corresponds to a hash function $h_i (1 \leq i \leq d)$, and there are w_1 columns for version 1.

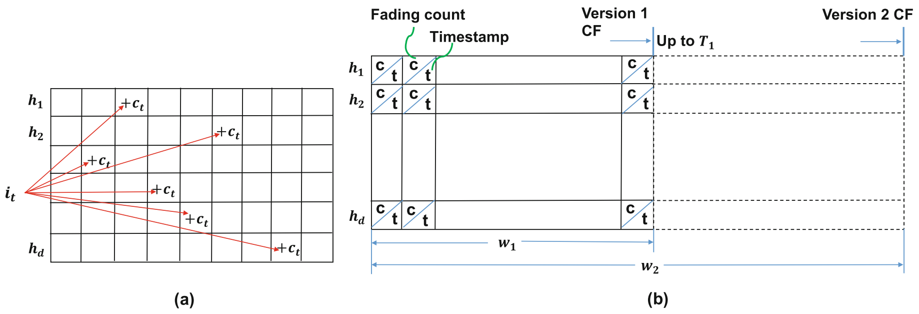

Fig. 2. (a) A Count-Min sketch, (b) A CF sketch.

Each cell has two fields, a count $c \in \mathbb{R}$ and a timestamp t. By design, logically, at each time step, we deduct 1 from the count c in each cell of the CF with probability p_-. Intuitively, old counts decay over time, while new counts keep being added—so our embedding learning will be adaptive. Additionally, as shown below, it has an important purpose—the count errors become bounded and fade out with time, in contrast to the unbounded/accumulated errors without the fading. However, it is too costly to update all the counts at every time step. We defer the deductions and batch them until a particular cell is needed. This is achieved by adding a "time" field to each cell (t in Fig. 2(b)), indicating the last time step when the count c is up to date.

To **increment the count** of an item x, we first locate the cell in row i and column $h_i(x)$ mod w_1, for each $1 \leq i \leq d$. For each of these d cells with content (c, t), we update its c to $\max(\varepsilon, c - p_- \cdot (t_{now} - t)) + 1$ and its t to t_{now}, where t_{now}

is the current time, and ε is a tiny constant (e.g., 0.01). Similarly, to**look up the count** of an item x, for each of those d cells, we get a count $\max(\varepsilon, c - p_- \cdot (t_{now} - t))$, and update its t to t_{now}. The minimum of these d values is returned. We save the update cost by batch updates using expectations, sufficient for our incremental embedding. The reason for the constant ε is to record the fact that the edge did once exist. We hash a triple (u, r, v) for an item. We may choose to clear ε cells to 0 after a certain amount of time. We now quantify the errors from CF. The proofs of all theorems are in the technical report [17].

Theorem 1. *Let the load factor of a CF (i.e., the fraction of cells that are nonzero) be ρ, and the graph stream edge incoming rate be λ per time step. Then the probability that the count of an item x has any error is ρ^d. Moreover, the probability that the error is greater than α is no more than $(\frac{\lambda w p_- + \lambda^2}{2w^2 \alpha p_-})^d$.*

Dynamic Growth/Shrinkage. We propose the dynamic growth/shrinkage of CF based on the incoming rate. The basic idea is to make CF "elastic" to the incoming edge rate λ and the required load factor ρ. Intuitively, when the stream rate is high, we increase the width of CF, as illustrated in version 2 in Fig. 2(b), and decrease the width when the rate is too low.

Theorem 2. *Setting the width of CF sketch to $w = \frac{\lambda}{\rho p_-}$ gives an expected load factor ρ of the sketch, where the graph stream average input rate is λ edges per time step, and p_- is the fading parameter.*

We start with width w_1 as shown in Fig. 2(b). When Theorem 2 suggests a $w > w_1$, we increase it to $w_2 = 2w_1$. Later on, when we access an edge e in version 1, we get the count c_e of e from version 1 as before, and deduct c_e from each of the d cells in version 1 for e. Then we add e with count c_e into version 2. Each version i is associated with a cut-off timestamp T_i, i.e., version i contains edges inserted up to time T_i. A version i is removed when $c_{\max} - p_- \cdot (t_{now} - T_i) \leq 0$, i.e., $t_{now} \geq T_i + \frac{c_{\max}}{p_-}$, where t_{now} is the current time, and $c_{\max}$ is the maximum count of any cell in version i. Intuitively, it is the time when the maximum cell count fades to 0. The version sequence can either grow or shrink (i.e., halve in width) from version i to version $i+1$, depending on the change direction of λ. At any time we keep at most k versions (e.g., $k = 5$). When we look up the count of an edge, we examine the versions in reverse order starting from the latest, and return the count as soon as it is found to be non-zero. We increase p_- according to Theorem 2 if we reach the memory constraint.

4 Graph Stream Embedding and Query Answering

To answer predictive relationship queries, we devise a general method based on graph embedding. However, it is an open problem to do embedding for dynamic graphs or graph streams [11]. There are a number of embedding algorithms for static knowledge graphs, including TransE, TransH, and TransR, among others

[22], and the list is still growing. However, this is an orthogonal issue, as our techniques are independent and can be readily plugged into any of them. For clarity, we present our techniques with TransE [7]. Let us start with the static knowledge graph embedding. The goal is to give an embedding vector (e.g., a vector of 100 real values) to each *node* in the knowledge graph, as well as to each relationship. The idea is to convert a knowledge fact edge (u, r, v) in the training set into a soft constraint $\boldsymbol{u} + \boldsymbol{r} = \boldsymbol{v}$. Thus, the following conditional probability is a key component of our objective function:

$$P(v|u, \boldsymbol{r}) = \frac{\exp\left((\boldsymbol{u} + \boldsymbol{r}) \cdot \boldsymbol{v}\right)}{\sum_{\boldsymbol{v'} \in \boldsymbol{V}} \exp\left((\boldsymbol{u} + \boldsymbol{r}) \cdot \boldsymbol{v'}\right)} \tag{1}$$

Intuitively, if nodes u and v satisfy the relationship r, then the vector $\boldsymbol{u} + \boldsymbol{r}$ should be close to vector $\boldsymbol{v}$; hence the numerator in Eq. 1 should be large, and the conditional probability is high. We optimize the log likelihood of the observed

Algorithm 1: OnlineKGStreamEmbedding ($\mathcal{G}$)

```
   Input: $\mathcal{G}$: the graph stream
   Output: evolving embedding-vectors of nodes and relationships
1  for each incoming edge $(u, r, v) \in \mathcal{G}$ do
2      increment the count of $(u, r, v)$ in CF
3      if $(u, r, v) \notin E_m$ then
4          if $c_{(u,r,v)} > \min_{e \in E_m} c_e$ then
5              add $(u, r, v)$ into $E_m$, and remove one with $\min_{e \in E_m} c_e$
6      while time remains do
7          $\mathbb{N}^+ \leftarrow \emptyset$, $\mathbb{N}^- \leftarrow \emptyset$
8          sample an edge $(x, r, y)$ from $E_m$ weighted by counts
9          $\mathbb{N}^+ \leftarrow \mathbb{N}^+ \cup \{(x, r, y)\}$
10         with probability $p_S$
11             sample edge $e_S$ from $E_S(x) \cup E_S(y)$
12             $\mathbb{N}^+ \leftarrow \mathbb{N}^+ \cup e_S$
13         repeat
14             sample $y' \in V$
15             if $c_{(x,r,y')} > 0$ in CF then
16                 $\mathbb{N}^+ \leftarrow \mathbb{N}^+ \cup \{(x, r, y')\}$
17             else
18                 $\mathbb{N}^- \leftarrow \mathbb{N}^- \cup \{(x, r, y')\}$
19         until $|\mathbb{N}^-| = k^-$
20         update node and relationship embeddings w.r.t. the gradients of
            $\sum_{(x,r,y) \in \mathbb{N}^+} \log \sigma((\boldsymbol{x} + \boldsymbol{r}) \cdot \boldsymbol{y}) + \sum_{(x,r,y) \in \mathbb{N}^-} \log \sigma(-(\boldsymbol{x} + \boldsymbol{r}) \cdot \boldsymbol{y})$
21 embedding vectors of all vertices and relationships are returned upon queries
```

edges using Eq. 1:

$$\log \mathcal{L}(\mathcal{G}) = \sum\nolimits_{(u,r,v)\in\mathcal{G}} \log P(v|u, \boldsymbol{r}) \tag{2}$$

Our goal is to perform online, adaptive, and incremental graph embedding as edges stream in, as an *anytime algorithm* [25]. Any memory constraint must also be satisfied. Thus, we maintain the following data structures: (1) the **embedding vectors** of each node and each relationship type, (2) the **top-m edges** E_m with respect to the accumulated counts, and (3) the **CF sketch** of all edges. We use a CF sketch to hash stream edges, and maintain an accumulated/decayed count for each edge. In addition, using a priority queue, we maintain a buffer that consists of the top-m edges with the highest dynamic counts. The algorithm is in OnlineKGStreamEmbedding.

Lines 3–5 of the algorithm maintain the top-m edge buffer E_m. The loop in lines 6–20 performs *stochastic gradient descent* (SGD) [8] to optimize Eq. 2, using both positive samples (stream edges) and negative samples [19]. The loop continues as time allows; when a new edge arrives, the algorithm will pause the loop and handle the new edge first, before returning to SGD loops. The sets $\mathbb{N}^+$ and $\mathbb{N}^-$ in line 7 are for positive and negative edges, respectively. In lines 10–12, with some probability we sample a *static* edge that intersects with either node x or node y. $E_S(x)$ denotes the set of static edges with one endpoint being x, which are neighbors of the current dynamic edge used in optimization. Lines 17–18 follow the aforementioned *local closed world* assumption [13]—the unobserved (x, r, y') is considered negative since we have (x, r, y) in the stream. In line 19, k^- is the number of negative edges required in each round. Line 20 does the SGD update based on Eq. 2, where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. The following lemma follows from the edge buffer maintenance of the algorithm.

Lemma 1. *The edges in the top-m store are always the edges with top-m highest expected weight conditioned on all the readings of edges from the CF.*

In summary, our algorithm prioritizes the top-m edges based on the decayed counts for adaptive and incremental embedding. Importantly, CF keeps the counts of *all edges*, not just those in top-m, which is essential as we need to know if an edge is negative for the negative sampling.

Answering Predictive Queries. Based on the embedding, we now discuss several types of analytical predictive queries. The first two types are closely related, namely **local relationship queries** and **relationship ranking queries**. Their formal semantics are already presented in Sect. 2. An example of relationship ranking query in Fig. 1(a) is to rank several trips from one source node (e.g., home) to several destination nodes based on the probabilities of being slow. Or for the same pair nodes, rank the probabilities that the trip will be slow versus fast. To answer such queries, we use the embedding vectors and Eq. 1, where the denominator is estimated based on negative sampling discussed earlier (i.e., estimating the average of $\exp((\boldsymbol{u}+\boldsymbol{r})\cdot\boldsymbol{v}')$, and hence the sum). For ranking we may only need to compare the numerators when the denominators are the same.

We also find it useful to answer **relationship temporal join queries**. Such a query asks for the correlation of these two predictive edges (relationships) over time. Specifically, every Δt time, r is either true or false for (u_1, v_1), giving us a binary sequence s_1. Similarly, we get another binary sequence s_2 for (u_2, v_2) at the same time. Then the query asks to measure the correlation/similarity between s_1 and s_2. For example, in Fig. 1(a), we may want to find out the temporal correlation between the traffic of a pair of locations (l_1, l_2) and (l_3, l_4), helpful for traffic analysis, planning, and management. To answer such a query, at every Δt time, we estimate the probabilities that (u_1, r, v_1) and (u_2, r, v_2) hold using the embedding vectors. Then we can use Pearson correlation coefficient [20] to measure their similarity. Alternatively, we compare the binary sequences s_1 and s_2 using the Sokal-Michener similarity [24], which is defined as $\frac{S_{11}+S_{00}}{N}$, where N is the length of the two sequences, and S_{11} (resp. S_{00}) is the number of time instants when both values are true (resp. false).

A novel type of analytical predictive query that we study is based on what we call a **user defined relationship (UDR)**. A user may first define a new relationship r, based on existing ones. Then the system learns the embedding vectors of r along with other relationships and nodes to answer queries. For instance, in Fig. 1(b), one may define a relationship $(user, tag)$ which indicates that the *user* likes movies bearing the *tag*. Essentially this corresponds to a two-edge path in the original graph. UDR gives users flexibility and convenience in querying novel relationships.

Finally, we also extends our study to what we call **global relationship queries**. The idea is that we treat each relationship as a relational *table* with three columns: *from* (vertex), *to* (vertex), and *time*. Each observed edge corresponds to one tuple in one of the tables. Thus, each table has an *observed part* and an *extended part* predicted to be likely (e.g., from embedding vectors). Such queries would be useful for a global view of relationships. In the example in Fig. 1(b), a global query may ask the fraction of the user population who will "like" a particular movie, which may include both the observed and predicted tuples. The result can be estimated by sampling all user nodes and performing a local relationship query with the movie node. One may also ask a join query between two such global tables. We have more examples in the experiment section.

5 Experiments

5.1 Datasets and Setup

We use two real datasets, **New York taxi data** and **movie data** as described in Sect. 1 (Fig. 1). Some statistics are shown in Table 1. We implement all the algorithms in Java (with maximum heap size 256 MB), as well as two baseline algorithms described below. For graph embedding, as in previous work [7], we use a default dimensionality of 50 and a learning rate of 0.01. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4 and a 2.5 GHz Intel Core i7 processor.

Table 1. The statistics of the two datasets.

Dataset	#vertices	#edges	Time span	Data size
NY taxi	5,654	169,100,000	1 year	11 GB
Movie	317,887	26,914,247	22.5 years	1.08 GB

5.2 Experimental Results

We first use the taxi data, preprocessed as described in Sect. 1 (for Fig. 1(a)), to evaluate local queries. For predictive local queries, in every 1000 incoming edges, we predict 20 edges uniformly at random—whether an edge is a fast trip (the algorithms have a warm start after running the first 10,000 edges). We remove all the occurrences of these 20 test edges from the dataset when answering queries.

Baselines. We compare against two baseline algorithms. Baseline 1 performs embedding without using edge time information, as in previous work—note that, although there is no absolute winner of link prediction for all applications, network embedding is currently considered as the state-of-the-art method for link prediction [11]. For each type of edge, it stores the occurrence count as the weight, which is used for weighted sampling of edges for iterative training. Baseline 2 maintains a sliding window of the most recent edges (using the same amount of memory as our approach), and iteratively performs embedding over those edges.

We vary the edge inter-arrival time and show the result accuracy in Fig. 3. As our query processing is an anytime algorithm, the inter-arrival time specifies a time budget, exploring the tradeoff between efficiency and accuracy. Our method has an accuracy between 0.85 and 0.9, and the accuracy improves when the edge inter-arrival time increases because there is time for more iterations over the SGD optimization. The improvement eventually levels off as the iterations near convergence. Our method has a clear advantage over the two baselines. Baseline 1 is inaccurate because it does not consider the trip property's dynamic changes over time. Baseline 2, although on a sliding window, does not use CF sketch, and

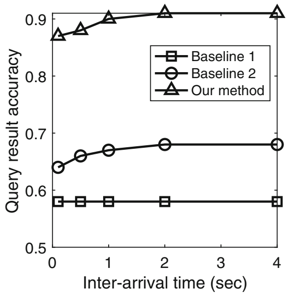

Fig. 3. Local relationship (taxi)

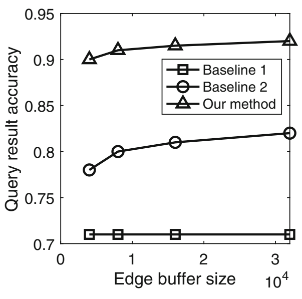

Fig. 4. Local relationship (movie)

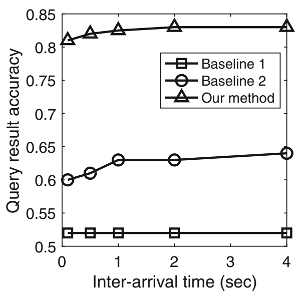

Fig. 5. Relationship rank. (taxi)

hence it only has very limited edge temporal count information. For instance, for negative sampling, it lacks information on which edges are negative.

Likewise, we work with the movie data, and define two static relationships—movie $\rightarrow$ its genres and movie $\rightarrow$ its tags with a relevance score at least 0.9. Most of the edges are dynamic with a timestamp—a user's rating to a movie. We define two relationships: a user "likes" a movie if the rating is at least 4 (in the range [0, 5.0]); a user "dislikes" a movie if the rating is ≤ 2. We predict the "like" relationship. The result is in Fig. 4, varying the number of edges in the edge buffer. Again our method has a clear advantage over the two baselines for the same reason. All the three algorithms have slightly better accuracy than the taxi data. This is because the taxi data is even more dynamic with faster changes, and is hence harder to predict.

We now examine relationship ranking queries, first with taxi data. As a trip edge $e = (u, v)$ comes in, we take its from vertex u. In subsequent edges, we take the first two edges that also start from u, i.e., edges $e' = (u, v')$ and $e'' = (u, v'')$. Our relationship ranking query is to rank e, e', and e'' in the order of their "fastness" (relative to the average statistics). We answer the predictive query by removing the three edges from data. Since the query result is a permutation of three edges, we define a metric for result accuracy: $1 - \frac{inv}{P}$, where $P = \binom{3}{2} = 3$ is the total number of pairs out of the three edges, and inv is the number of pair-wise inversions between query result and ground truth. For instance, if the ground truth ranking is e, e', e'', while the query result ranking is e', e, e'', then the accuracy is $1 - \frac{1}{3} = \frac{2}{3}$, as there is only one pair (e, e') whose order is inverted in the query result. In Fig. 5, we show the average ranking query result accuracy over 200 such queries. The accuracy from our method is significantly higher than the two baselines. One thing to note is that all three methods' accuracy for relationship ranking queries is slightly lower than that of the local queries. This is because relationship ranking result involves multiple pairs of edges and is more difficult to be all correct. Similarly, we show the ranking query result accuracy for movie data in Fig. 6, where we vary the edge buffer size.

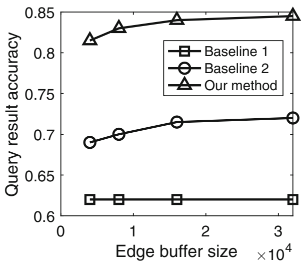

Fig. 6. Relationship rank. (movie)

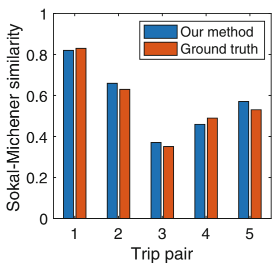

Fig. 7. Temporal join queries

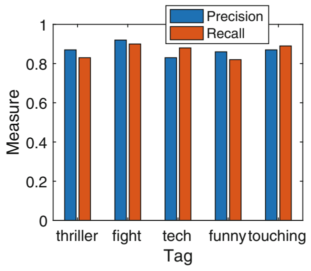

Fig. 8. UDR user–tag (movie)

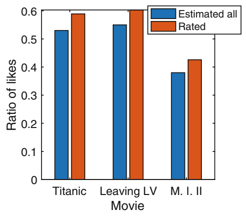

Fig. 9. Global rel. (movie)

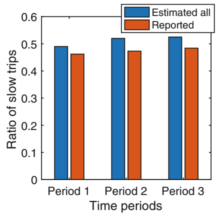

Fig. 10. Taxi data

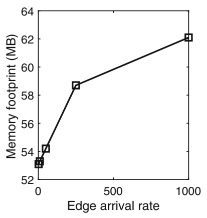

Fig. 11. Memory (movie)

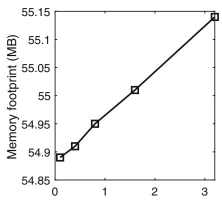

Fig. 12. Memory (movie)

We next examine temporal joins, user defined relationships, and global relationship queries. We begin with temporal joins using taxi data. We randomly pick a pair of trips (edges), e.g., (e_1, e_2). Then the query measure the correlation between e_1 and e_2 over time—between the two binary variables indicating whether they are "slow". We use the Sokal-Michener similarity [24] as described in Sec. 4. We repeat the query for five random pairs of trips, and show the result in Fig. 7 from our method, compared to the ground truth. Our result is accurate, due to the fact that our dynamic graph embedding using CF and top-m edges adaptively captures the edge transitions.

We now study UDR queries with movie data, and define a relationship *(user,tag)* to indicate that *user* likes a movie bearing *tag*. This is a two-edge path. We arbitrarily pick five tags "thriller", "fight scenes", "technology", "funny", and "touching", and evaluate the five queries for users seen in the stream. Once a UDR is defined, the rest is similar to a local query using the auxiliary edge. The accuracy results are in Fig. 8. Last we study global queries. For movie data, we query the fraction of the user population who will "like" a particular movie. The result is estimated by sampling all user nodes and performing a local relationship query with the movie node. We arbitrarily pick three movies, "Titanic", "Leaving Las Vegas", and "Mission Impossible II", and show the result in Fig. 9 (the first bar of each movie). We also compare it against the ratio calculated from the dataset but only based on those users who gave a rating to the movie (the second bar of each movie). The query results are all slightly smaller than those calculated from the users who rated the movies. A possible reason is that those who rated a movie were motivated to watch the movie in the first place, and hence had a higher chance to like the movie than the general population.

Similarly, using taxi data, we query the fraction of all possible (source, destination) pairs that are slow at some point within a time period of 10 min. We estimate the result of this query by sampling a pair of location nodes. Figure 10 shows the results for three time periods (first bar of each time period), where we also compare with the ratio of slow trips among those that are reported during that time period in the dataset. The estimated ratios are slightly higher, with the intuition that a randomly picked pair is more likely to hit a slow link.

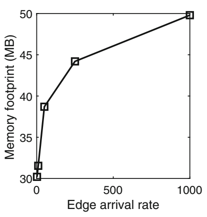

Fig. 13. Memory (taxi)

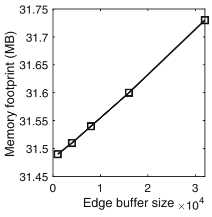

Fig. 14. Memory (taxi)

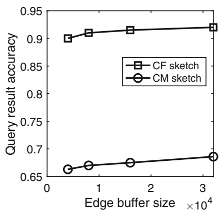

Fig. 15. CF vs CM (movie)

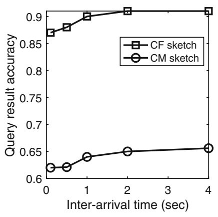

Fig. 16. CF vs CM (taxi)

We next look into the memory footprint. The results for movie data are in Fig. 11 and 12 as we vary the edge arrival rate and the edge buffer size, respectively, and in Fig. 13 and 14 for the taxi data. For both datasets, as the edge arrival rate increases, the width of CF also increases, and so does the memory footprint. In general, the movie data has a higher memory footprint than taxi data. This is because the movie data has significantly more vertices. Hence, with the movie data, more vertices tend to be loaded into memory, along with the embedding vectors of each vertex. Overall, our approach has a very small footprint. Finally, we evaluate the usage of CF in our scheme, compared to the off-the-shelf CM sketch. We examine the result accuracy of local relationship queries for the movie data in Fig. 15, and for the taxi data in Fig. 16. Using CF gives much more accurate results. This is because CM always accumulates its counts in each cell, and the false positive errors are never erased, compromising its accuracy. In addition, it does not dynamically adjust its size as CF does.

6 Conclusions

Knowledge graph streams are a common model for many applications. Predictive relationship queries are important for data analytics. We devise an approach that performs online incremental embedding using a novel sketch. Our approach is general enough to answer many types of predictive relationship queries. The experimental results show that our approach gives accurate query results efficiently and has a small memory footprint.

References

1. Movielens data (2019). https://grouplens.org/datasets/movielens/latest/
2. New york taxi data (2019). http://chriswhong.com/open-data/foil_nyc_taxi/
3. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. **58**, 137–147 (1999)
4. Amazon: Amazon neptune (2019). https://aws.amazon.com/neptune/
5. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. (2009)

6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**, 422–426 (1970)
7. Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. In: Advances in Neural Information Processing Systems, pp. 2787–2795 (2013)
8. Bottou, L.: Stochastic learning. In: Bousquet, O., von Luxburg, U., Rätsch, G. (eds.) ML -2003. LNCS (LNAI), vol. 3176, pp. 146–168. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28650-9_7
9. Boykin, O., Ritchie, S., O'Connell, I., Lin, J.: Summingbird: a framework for integrating batch and online MapReduce computations. In: VLDB (2014)
10. Cafaro, M., Pulimeno, M., Epicoco, I., Aloisio, G.: Mining frequent items in the time fading model. Inf. Sci. **370**, 221–238 (2016)
11. Cai, H., Zheng, V.W., Chang, K.C.C.: A comprehensive survey of graph embedding: problems, techniques and applications. TKDE **30**, 1616–1637 (2018)
12. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**, 58–75 (2005) AQ2
13. Dong, X., et al.: Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In: KDD (2014)
14. Garcia Lopez, P., et al.: Edge-centric computing: vision and challenges. SIGCOMM Comput. Commun. Rev. (2015)
15. Google: Google inside search (2019). https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html
16. Lin, Y., Liu, Z., Sun, M., Liu, Y., Zhu, X.: Learning entity and relation embeddings for knowledge graph completion. In: AAAI (2015)
17. Liu, X., Ge, T.: Mining dynamic graph streams for predictive queries under resource constraints (2020). http://www.cs.uml.edu/~ge/paper/gstream_predictive_queries_tech_report.pdf
18. McGregor, A.: Graph stream algorithms: a survey. ACM SIGMOD Rec. **43**(1), 9–20 (2014)
19. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. In: NIPS (2013)
20. Pearson, K.: Note on regression and inheritance in the case of two parents. Proc. R. Soc. Lond. Series **I**(58), 240–242 (1895)
21. Tai, K.S., Sharan, V., Bailis, P., Valiant, G.: Sketching linear classifiers over data streams. In: SIGMOD (2018)
22. Wang, Q., Mao, Z., Wang, B., Guo, L.: Knowledge graph embedding: a survey of approaches and applications. TKDE **29**(12), 2724–2743 (2017)
23. Wang, Z., Zhang, J., Feng, J., Chen, Z.: Knowledge graph embedding by translating on hyperplanes. In: AAAI, vol. 14, pp. 1112–1119 (2014)
24. Zhang, B., Srihari, S.N.: Properties of binary vector dissimilarity measures. In: CVPR (2003)
25. Zilberstein, S.: Using anytime algorithms in intelligent systems. AI Mag. **17**, 73 (1996)