PREDICTIVE ANALYSIS AND CRITICAL EVENT MONITORING IN LARGE DYNAMIC NETWORKS

BY

YAN LI

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

AUGUST 2023

PREDICTIVE ANALYSIS AND CRITICAL EVENT MONITORING IN LARGE DYNAMIC NETWORKS

BY

YAN LI
B.S. SHANDONG UNIVERSITY OF TRADITIONAL CHINESE MEDICINE (2008)
M.S. UNIVERSITY OF MASSACHUSETTS LOWELL (2020)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

Signature of
Author: [illegible] Date: 05/16/2023

Signature of Dissertation Chair: [illegible]
Name Typed: Tingjian Ge

[illegible]
Name Typed: Cindy Chen

Signature of Other Dissertation Committee Members

Committee Member Signature: Xiangnan Kong
Name Typed: Xiangnan Kong

PREDICTIVE ANALYSIS AND CRITICAL EVENT MONITORING IN LARGE DYNAMIC NETWORKS

BY
YAN LI

ABSTRACT OF A DISSERTATION SUBMITTED TO THE FACULTY OF THE
DEPARTMENT OF COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

AUGUST 2023

Dissertation Supervisor: Dr. Tingjian Ge
Professor, Computer Science

Dissertation Supervisor: Dr. Cindy Chen
Associate Professor, Computer Science

ABSTRACT

Many modern data applications, such as social networks, road traffic networks, web, telecommunications, biological interactions, and sensory data over a large area, have data that are most naturally represented as large dynamic graphs. One specific example of such graphs is heterogeneous knowledge graphs that are commonly used, and that are known to be incomplete. In general, such large, possibly highly dynamic, networks have two types of significant information to be predicted and queried: (1) *attribute values* and (2) *unobserved links/relationships*. In addition, complex event monitoring and prediction are also critical problems in data streams that have drawn much attention in both research and industry. This thesis proposes a comprehensive approach to address these challenges by introducing dynamic graph analytics for predictive querying and event monitoring in complex data streams.

In order to efficiently (in real-time) answer queries on such high dynamic graphs and sufficiently capture the correlations inherent with graph topology and with time, we propose a machine learning approach using a novel probabilistic graphical model to leverage the spatio-temporal nature of the data. We perform an extensive experimental study using two real-world dynamic graph datasets to demonstrate the effectiveness and efficiency of our approach. For entity queries based on incomplete relationships in knowledge graphs, the existing work cannot obtain the results in an *efficient* manner, due to the gigantic number of candidate entities. We propose an incremental index scheme to be able to efficiently and accurately answer top-k entities and aggregate queries over a virtual knowledge graph. Experiments show that our cracking index only performs a very small fraction of the node splits compared to a full index, and is very efficient in answering queries with accuracy guarantees.

Moreover, treating event time orders as relationship types between event entities, we build a dynamic knowledge graph and use it to predict future event timing. We represent the event timing knowledge in training data using a knowledge graph where nodes are events, and edges encode timing relationships such as “happening soon after” and “happening long after”. On top of this knowledge graph, we need the notion of “active state” to characterize the state information, as well as ephemeral nodes and edges along with attention parameters for learning the embedding vectors containing latent features. With this novel embedding, we are able to achieve high precision and recall values in predicting event timing, ranging from about 0.7 to nearly 1, significantly outperforming a baseline approach.

Finally, in contrast to previous work that assumes that the user knows a complex event pattern for the system to continuously monitor, we study a novel problem which is to discover subtle and complex event patterns prior to critical events in data streams. We propose an imminence monitoring approach that uses a representation-learning based dynamic knowledge graph to predict the timing of future events. The approach captures the temporal dependencies between events, and the proposed representation learning method effectively models the heterogeneous substreams and diverse sets of attributes in the data. Our extensive experiments demonstrate the effectiveness and advantages of our method over previous work.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my supervisor Dr. Tingjian Ge, Professor of the Computer Science department at the University of Massachusetts Lowell, for providing me with invaluable guidance and support throughout my Ph.D. journey. His vast knowledge, critical insights, and unwavering encouragement have been instrumental in shaping the direction and scope of my research. I would like to express my sincere gratitude to my supervisor Dr. Cindy Chen for allowing me to do research and providing invaluable guidance throughout my study.

I would also like to thank the member of my thesis committee: Dr. Xiangnan Kong, for his valuable feedback and insightful comments on my work. Their constructive criticism and intellectual contributions have greatly enhanced the quality and rigor of my research.

I am grateful to my colleagues and friends who have provided me with a stimulating and supportive academic environment. Their ideas, feedback, and camaraderie have helped me to navigate the many challenges and obstacles that come with pursuing a Ph.D.

Finally, I would like to thank my family for their unwavering love, support, and encouragement. Their belief in me and my abilities has been a constant source of strength and inspiration throughout my Ph.D. journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

The data in urban systems such as transportation, roads and bridges, water and energy systems, and telecommunication networks can often be modeled as large dynamic graphs, i.e., graphs/networks that change over time. As data are acquired, unfortunately, they are rarely complete observations of the whole system. It is important to reliably infer the unobserved attribute values anywhere in the graphs, at certain times–either in the past or in the future.

Another example of large dynamic networks is the knowledge graphs. A knowledge graph is a knowledge base represented as a graph. It is a major abstraction for heterogeneous graph representation of data with broad applications including web data [1], user and product interactions and ratings [2], medical knowledge and facts [3], and recommender systems [4]. Due to the enormous and constantly increasing amount of information in such knowledge bases and the limited resources in acquiring it, a knowledge graph is inherently incomplete to a great extent [5]. There are two types of significant information to be predicted and queried in such large dynamic networks, (1) *attribute values* and (2) *unobserved links/relationships*. It is important, yet challenging, to efficiently answer queries on such graphs due to the highly dynamic nature of the data. Now consider the following examples that emphasize the necessity for adapting these queries to dynamic networks.

Example 1. *We are working with civil and mechanical engineering researchers in Massachusetts on traffic and driving guidance and control with smart vehicles under severe weather, such as snow storms. Traffic, road, and weather conditions over a large road network form a rich dynamic graph. What can be sensed and detected from smart vehicles and* *roadway sensors is only a small portion of the overall dynamic network environment. In order to provide travel guidance and traffic control, we need to query the traffic condition at unobserved road segments, or the slippery condition of certain roads at an unobserved time instant, among many other queries.*

Example 2. *Figure 1 illustrates a heterogeneous knowledge graph where we have different vertex types including users (Amy, Bob, and so on), restaurants, grocery stores, and styles of food (Italian, Mexican, and so on). The graph also has different relationship types as edges (illustrated with various colors) such as "rates high" (of a restaurant), "frequents" (a grocery store), and "belongs to" (a style of food). This knowledge graph is incomplete. For example, it misses the information that Amy likes Restaurant 2 (i.e., would give a high rating) with a certain probability, and she likes Restaurant 3 with a certain probability, illustrated as red dashed edges in Figure 1. We envision that one of the major uses of a*

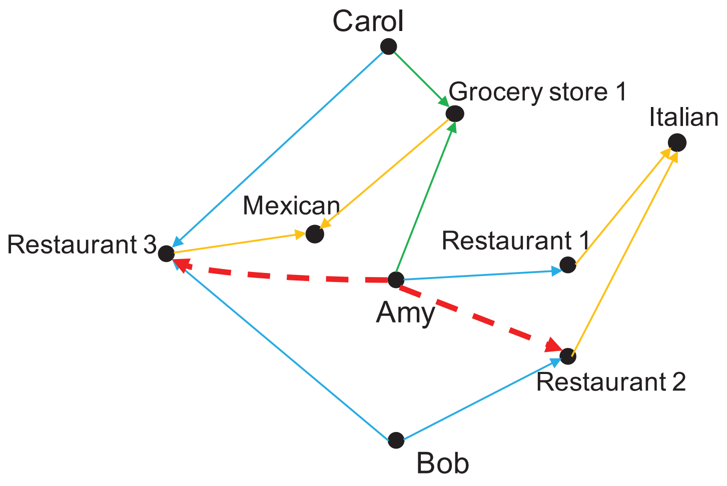


Figure 1: Illustrating a virtual knowledge graph.

virtual knowledge graph is to answer entity information queries given another entity and a relationship. For instance, in Figure 1, a useful query may be ***(Q1)*** *"What are the top-5 most likely restaurants Amy would rate high but has not been to yet?". Another query,* *which involves aggregation, is* ***(Q2)*** *"What is the average age of all the people who like Restaurant 2?".*

Event matching, as part of complex event processing (CEP), is one of the most important topics in data streams [6]. Many commercial systems (e.g., [7, 8]) have implemented event matching and CEP. However, little work has been done on *predicting the timing* of an interesting event, which we call a *target event*—whether it will happen soon or long after the current time. Let us look at an example.

Example 3. *Outpatient monitoring and management of Type 1 diabetes (T1D), also known as juvenile diabetes, is a critical issue for the treatment of T1D patients [9]. Patients with T1D are insulin deficient. Outpatient management of T1D relies principally on three interventions: diet, exercise, and exogenous insulin. Diabetes patient information is obtained from an automatic electronic recording device. The automatic device has an internal clock to timestamp events. We consider signals from a short period of time (e.g., two hours) as a tuple, and there are multiple attributes in a tuple, such as regular insulin dose, NPH insulin dose, blood glucose measurement at a pre-meal or post-meal, hypoglycemic symptoms, typical or more/less-than-usual meal ingestion, and typical or more/less-than-usual exercise activity. The continuous monitoring data from an outpatient form a multi-attribute data stream. Doctors may match interesting event patterns from the data; they may also want to predict some events of interest, such as hypoglycemic conditions in the near future, or a blood glucose measurement that will increase significantly. Doctors and/or outpatients may be notified when such predictions occur, and critical interventions can be performed to prevent undesirable events.*

While there is much previous work on time series forecasting [10], little has been done on multi-attribute data stream discrete-event timing prediction—whether a target event will happen soon, or whether an event will happen much later, using limited (and recent) training data to promptly build a model and predict. There is previous work on sequential association rules [11]. However, our experiments show that our proposed approach provides much better prediction accuracy in terms of precision and recall.

Complex event processing has been an active direction in research and in practice for data streams [12]. Nearly all work thus far assumes that the user knows and provides the complex event pattern to search for. However, we argue that oftentimes we do not know the event or state transition patterns of the data stream to search for, but merely know that some interesting or critical events happen *after the fact* (e.g., something undesirable such as security attacks). Moreover, there may not be an easy way to represent the context of critical events, not to mention their imminence monitoring.

Example 4. *Consider a hospital that has patients of certain serious diseases. The electronic treatment records are almost all digitized today. The information is very heterogeneous—there are people of different ages and backgrounds, and there are many diseases and sub-categories. Some urgent situations and events happen without well-known or uniform patterns to search for or to monitor.*

For instance, the Dutch Academic Hospital dataset [13] is a real-life event log. The log contains events related to the treatment and diagnosis steps for a heterogeneous mix of patients with cancer pertaining to the cervix, vulva, uterus, and ovary. Attributes include event name, treatment code, diagnosis, date and time, specialism code, activity code, *department/lab, and age. From time to time, there are emergent events in the treatment sequence, such as "potassium flame photometric" and "hemoglobin photoelectric". Due to the heterogeneity of patient attributes and diseases, there is no uniform or even easily known pattern to search for in order to better monitor and foresee the emergent situations.*

In general, there are two challenges: **(1)** the pattern before the critical event is often subtle, complex, and not precisely known to users; **(2)** it is difficult to use a simple language such as variants of *regular expressions* [12] to express the pattern. We propose an approach that discovers a pattern expressed as a *probabilistic state machine*, and use it to understand the current state of the stream and predict how imminent a particular critical event is, in what we call the *imminence monitoring*. One salient feature is that we handle multiple substreams (which we call a *multistream*), where each substream may correspond to a source of data generation (e.g., a person or object). The pattern that we learn must incorporate the pattern variety of substreams. This is analogous to a personalized recommender system where, for example, people with similar interests are recommended similar movies.

In Summary, various real-world applications can be treated as large-scale, dynamic graphs. Efficiently and effectively querying and predicting events in such large, dynamic graphs is very challenging. We propose a machine learning approach using a probabilistic graphical model to leverage the spatiotemporal nature of the data, and also an incremental index scheme to efficiently answer top-k entity and aggregate queries over a virtual knowledge graph. We also address the problem of event timing and complex event monitoring, proposing a representation learning-based dynamic knowledge graph to predict the timing of future events. In the end, we presents extensive experimental studies using real-world datasets to demonstrate the effectiveness and efficiency of the proposed approaches, which significantly outperform previous work.

1.1 Related Work

Dealing with missing or incomplete data is an important problem in database and data mining research, as well as in many specific domain fields of study. For example, Osborne [14] and Tan et al. [15] give good surveys on this topic. In specific domain studies, Honaker and King [16] study this problem in political science. They build a multiple imputation model for time-series cross-section data structures common in the political science. Horton and Kleinman [17] study this problem specifically focusing on a large health services research dataset. Ghahramani and Jordan [18] study how to perform supervised machine learning in the presence of missing data using an EM approach, especially for high-dimensional datasets.

Predictive database and query processing are also studied in the literature, e.g., [19], [20]. In particular, Akdere et al. [19] argue that the next generation DBMS should incorporate a predictive model management component to effectively support both inward-facing applications, such as self management, and user-facing applications such as data-driven predictive analytics. The Fa system [20] addresses forecasting queries in data stream and time series systems.

All the previous work discussed above, however, cannot be applied to our problem since we specifically target dynamic networks, where both correlations implied by graph topology and time-wise state evolvement patterns over the whole large graph must be incorporated. There are a few exceptions which we have discussed and compared against in experimental section. Kim and Leskovec [21] study the "network completion" problem, which bears some similarity with our work. However, their work is about inferring missing nodes and edges in large graphs, while we do not intend to change the graph structure, but only focus on the value graph that we define.

Probabilistic graphical model (PGM) has seen its many uses in the data management literature. For example, Deshpande et al. [22] give a survey of the usage of PGM in managing and querying probabilistic databases, where PGM is to model the correlations among the uncertain data entities. One particular project is the BayesStore at UC Berkeley [23], where Wang et al. use Bayesian networks extensively in their system. It has also been proposed by Deshpande et al. to use PGM to guide *data acquisition* [24] in systems such as sensor networks.

Hidden Markov models (HMM) have been used in various contexts in data management and mining, such as in mining co-evolving time sequences [25], where Matsubara et al. solve the problem that, given a number of co-evolving time series, how to find the patterns and segments over all of them. The Baum-Welch algorithm is used in learning parameters. A hierarchical hidden Markov model is proposed by Fine et al. [26]. However, it is a totally different model than ours and targets multi-scale structure in *sequence* data; it does not capture correlations based on graph topology and time as in our model. Akdere et al. [27] use dynamic Bayesian networks (DBN) to support continuous prediction queries over streaming data. Note that, in our work, we do not use HMM or DBN off the shelf. Instead, we devise a novel PGM with layers of latent variables, which scales up to large dynamic graphs, and which captures both the evolvement over time and the correlations associated with data graph structures.

A special form of dynamic graphs—graph streams—a topic which used to be mostly explored in the computer science theory literature (see the survey by McGregor [28]), have been recently also empirically studied in data analytics and mining. For example, Aggarwal et al. [29] mine frequent and dense patterns in graph streams. Song et al. [30] study event pattern matching in graph streams, where event patterns are defined both with graph pattern structure and with timing constraints. Tang et al. [31] tackle the problem of graph stream sketching and summarization to enhance the efficiency of query processing over graph streams. In this dissertation, we study a completely different problem than the ones above.

Knowledge graphs and graph embedding. One of the earliest uses knowledge graphs is by Google [1]. Google builds a knowledge base as a graph with its services to enhance its search engine's results with information gathered from a variety of sources. Knowledge graphs have become a promising strategy to bridge various information sources and organizations such as health care, law enforcement, public education and the military. For example, the Open linked data project [32] represents Web entities and relationships with RDF graphs. Google knowledge vault [33] presents a knowledge graph that contains 1.6B triples over 43M unique entities from 1.1K types.

Graph embedding represents a graph in a low dimensional space that preserves as much graph property information as possible. What differs one graph embedding algorithm from another mainly lies in how it defines the graph property to be preserved—which often depends on the problem to be solved. In a way, graph embedding is an elegant method to accomplish automatic latent feature extraction (each value in a vector is a feature) and dimensionality reduction (reducing high-dimensional graph topology and attributes to relatively low-dimensional vectors). Early-days graph embedding is based on matrix factorization. It represents graph property (e.g., node pairwise similarity) in the form of a matrix and factorizes this matrix to obtain node embedding. This line of work includes [34]. More recently, the researchers have proposed deep learning based graph embedding methods such as using random walks [35] and autoencoders [36].

Most knowledge graph embedding is based on edge reconstruction based optimization, in particular minimizing margin-based ranking loss. Such embedding methods include TransE [15], TransA [37], KGE-LDA [38], SE [39], TransD [40], MTransE [41], puTransE [42], among others. We have discussed TransE earlier, but our methods can be adapted for most of the other knowledge graph embedding methods, as they all minimize some loss function on h, r, and t for all the edge triplets in the training graph.

Cracking B+ tree and spatial indexing. A cracking B+ tree index in a relational database [43, 44] aims to reduce index maintenance costs. It amortizes the index maintenance costs over query processing and dynamically modify a half-way built B+ tree index during query processing. There are fundamental differences between a cracking B+ tree index and our proposed indexing. First of all, we work with a spatial index R-tree, with completely different techniques. Second, our work has a different goal—we are not trying to reduce index maintenance costs during updates; our R-tree index cracking is to avoid splitting R-tree nodes as much as possible based on the query search space.

R-tree is typically the preferred method for indexing spatial data. Objects are grouped using the minimum bounding rectangle (MBR). Objects are added to an MBR within the index that will lead to the smallest increase in its size. We use a bulk-loading algorithm of R-tree [45], which is commonly used for efficiently loading data into the index. Note that our method can be easily adapted for other variants of R-tree index (e.g., R+ tree or R* tree) as well.

Dimensionality reduction & nearest neighbors. The main linear technique for dimensionality reduction is Principal Component Analysis (PCA) [46]. Other dimensionality reduction techniques include Non-negative Matrix Factorization (NMF) and Linear Discriminant Analysis (LDA) [47]. High dimensionality is a great challenge for k-nearest neighbors (k-NN) queries. Compared to previous dimensionality reduction methods, some more efficient ones include random projection [48] and locality sensitive hashing (LSH) [49]. We use a particular type of random projection, JL transform [50], for transforming to space $\mathbb{S}_2$, but modify it significantly to get a low dimension, and provide theoretical guarantees. Different LSH schemes are based on different similarity metrics. The one that is closest to our work is H2-ALSH [51]; we have discussed it in detail and compare against it in the experiments.

Time series forecasting. Our work bears some similarity with time series forecasting, which has been a major focus for research in other fields. In particular, valuable tools for forecasting and time series processing appear in statistics and signal processing. [52] is a comprehensive review of this research over the past years. [53] and [54] present additional work in this area. We, however, are not dealing with single numerical attribute time series forecasting. We focus on discrete events over multiple attributes in data streams, and their timing relationship modeling and predictions.

Functional dependencies and soft functional dependencies. Correlations and association between different attributes have been traditionally studied in functional dependency theory [55]. More recently, soft functional dependencies [56] have been proposed, which aim to relax the strict dependency requirements. One can distinguish between "data driven" and "query driven" approaches to detecting and exploiting dependencies among database columns [57]. However, this line of work does not deal with data and information correlation over time, or the timing relationship prediction, which is the goal of our work.

Sequential association rules. Sequential association rule mining has been studied in data mining [11]. The basic idea is an extension from the traditional association rule mining to find patterns of correlation over time. Sequential association rules are typically also frequency based with support and confidence. We have done extensive comparisons with sequential association rule approach in the experiments.

Data streams and event processing. Event matching and complex event processing have been well studied in both research and the industry, and is used by practitioners (e.g., [8, 58, 59]). They typically extend the regular expression syntax to define a complex event that is a sequence of simple events. However, this line of work does not deal with predicting future events or predicting the timing relationships among events.

2 PROBLEM FORMULATION

In this section, we will introduce important definitions and preliminaries for the problem we focus on.

2.1 Attribute Values Prediction

A dynamic graph $\mathbb{G}$ consists of an infinite sequence of graph snapshots $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_t, ...$, where $\mathcal{G}_t$ is the graph snapshot at time t. All the snapshots are based on the same underlying graph $G = (V, E)$. That is, all vertices and edges in $\mathcal{G}_t (t \geq 1)$ come from G. In the event that the graph topology also changes over time, we may consider G as the *union* of the vertices (resp. edges) of all graph snapshots—thus, this formulation is general enough. Note that we use the terms *dynamic graph* and *dynamic network* interchangeably.

We focus on an attribute a of either V or E (i.e., either a vertex attribute or an edge attribute). which we call a *target attribute*. What may differ between two adjacent snapshots $\mathcal{G}_{t-1}$ and $\mathcal{G}_t$ is the attribute value a at each vertex or edge. The correlation of a in the dynamic graph is both across the structure of the graph, and over time. Topologically, edges or paths in $\mathbb{G}$ are indications of correlation, while temporally, attribute a evolves over time with some pattern.

If the target attribute a is of the edges, conceptually we create a value graph $G_v = (V_v, E_v)$ in which each edge in G corresponds to a vertex in G_v, and two vertices in G_v are connected by an undirected edge if their corresponding edges in G intersect. If the target attribute a is of the vertices in G, the value graph G_v is the same as G.

Given a set of m observations of the target attribute a within a time interval $T = [t_b, t_e]$ at (v_i, t_i), where $1 \leq i \leq m$, v_i is a vertex in the value graph, and $t_i \in T$, our goal is to return the value (distribution) of the target attribute a at an unobserved vertex/time pair (v_q, t_q), where v_q is a vertex in the value graph and t_q may be a time step within T, after T, or before T.

2.2 Unobserved Links/relationships Prediction

Heterogeneous knowledge graphs are commonly used, and they are known to be incomplete. For entity queries based on incomplete relationships in knowledge graphs, the existing work cannot obtain the results in an efficient manner, due to the gigantic number of candidate entities.

A *knowledge graph* $G = (V, E)$ is a directed graph whose vertices in V are *entities* and edges in E are *subject-property-object* triple facts. Each edge is of the form (*head entity*, *relationship*, *tail entity*), denoted as (h, r, t), and indicates a relationship r from entity h to entity t.

Definition 1. ***(Virtual Knowledge Graph)*** *A virtual knowledge graph $\mathcal{G} = (V, \mathcal{E})$ induced by a prediction algorithm $\mathcal{A}$ over a knowledge graph $G = (V, E)$ is a (probabilistic) graph that has the same set of vertices (V) as G, but has edges $\mathcal{E} = E \cup E'$, where each $e \in E'$ is of the form (h, r, t, p), i.e., a triple (h, r, t) extended with a probability p determined by algorithm $\mathcal{A}$. Accordingly, edges in E have a probability 1.*

We focus on two types of queries over a virtual knowledge graph $\mathcal{G}$, namely top-k queries and aggregate queries. A *top-k* query is that, given a head entity h (resp. t) and a relationship r, we return the top k entities t (resp. h) with the highest probabilities in E'. In the same context, an *aggregate* query returns the *expected* aggregate value (COUNT, SUM, AVG, MAX, or MIN as in SQL) of the attributes of entities t (resp. h) in E'. Q1 is an example of a top-k query, while Q2 is an aggregate query.

2.3 Event Timing Prediction and Critical Event Monitoring

2.3.1 Event Timing Prediction

We are given a data stream $\mathcal{S}$ that consists of a sequence of records $(r_1, t_1), (r_2, t_2), ...$ where $t_1 < t_2 < \cdots$. An *event* at time t_i is a Boolean predicate over the record r_i, involving one or more attributes of r_i.

Let the time of record r^* be t^*. We say that event e will occur *soon* after r^* (or t^*) if it is true in record (r_i, t_i) and $0 < t_i - t^* < \delta_1$, for a (small) constant value δ_1. On the other hand, if e is false in each record (r_i, t_i) for $t_i \leq t^* + \delta_2$ (where $\delta_2 > \delta_1$ is a constant), we say that e happens *long* after r^* (or t^*). Note that the semantics on the event timing gaps δ_1 and δ_2 can either be count-based (i.e., number of records) or time-based. Without loss of generality, we assume the time-based semantics (the count-based one is essentially integer timestamps).

Given a set of interesting events $e_1, e_2, ..., e_k$, and the current record (r^*, t^*), and a window of most recent data before r^*, the problem is to predict whether each event e_j $(1 \leq j \leq k)$ will happen soon after r^*, or whether e_j will happen long after r^*.

2.3.2 Critical Event Monitoring

A data stream $\mathbb{S} = \mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_s$, which we call a *multistream*, consists of s substreams $\mathcal{S}_i$ $(1 \leq i \leq s)$. Each substream $\mathcal{S}_i$ is a (possibly infinite) sequence of tuples with schema $(A_1, ..., A_c)$, corresponding to the subsequence of data from a *data-generation source*, such as each individual person or object. We are given a period of history $\mathbb{H}$ of $\mathbb{S}$. Some tuples in $\mathbb{H}$ are labeled as *critical tuples* containing a *critical event* $e_{\mathcal{C}}$. Note that $e_{\mathcal{C}}$ may in fact be the *union* of several events (i.e., sub-events) that are all critical—we use a single $e_{\mathcal{C}}$ for notational convenience.

Our goal is to learn a pattern $\mathcal{P}$ that characterizes the sequence context with respect to the critical event $e_{\mathcal{C}}$, such that for an unseen portion of $\mathbb{S}$, using $\mathcal{P}$ and a continuous function $f_{\mathcal{P}}$, we can continuously monitor and predict how imminent $e_{\mathcal{C}}$ is. Specifically, $f_{\mathcal{P}}$ is a *cost function* that takes the context of the current tuple t_{now} as input and returns a real value (indicating the estimated "cost" to reach $e_{\mathcal{C}}$), where a smaller value indicates that $e_{\mathcal{C}}$ is more imminent. The *context* of t_{now} may be defined in different ways. In this work, we use a short sequence of tuples (e.g., of length 10) prior to t_{now} as its context.

3 METHODOLOGY

After formulating our problems in section 2, we will introduce the details about the solutions and algorithms for them.

3.1 Attribute Values Prediction

For answering the queries of unobserved attribute values in dynamic networks, our basic idea is to devise a novel probabilistic graphical model (PGM) that scales and that leverages the topology of the original data graph. In other words, the topology of the graph in the PGM is derived from the topology of the original data graph. Our model consists of multiple levels of latent variables, each capturing the co-evolvement of a small number of nodes over time (see Figure 2 for an illustration).

As the first step of model building, we propose an efficient vertex grouping algorithm based on value correlations along graph edges. The state transitions of each group of vertices are characterized by a latent variable node at its upper level. We prove that our algorithm ensures the size balance of groups with each group of size in $[\alpha, 2\alpha)$, where α is a parameter, and that the amortized cost [60] is optimal—linear to the graph size, and independent of α.

We then study how to learn the parameters of our model. We devise a novel algorithm that nontrivially extends Baum-Welch algorithm [61] at *each node* of our model, and uses

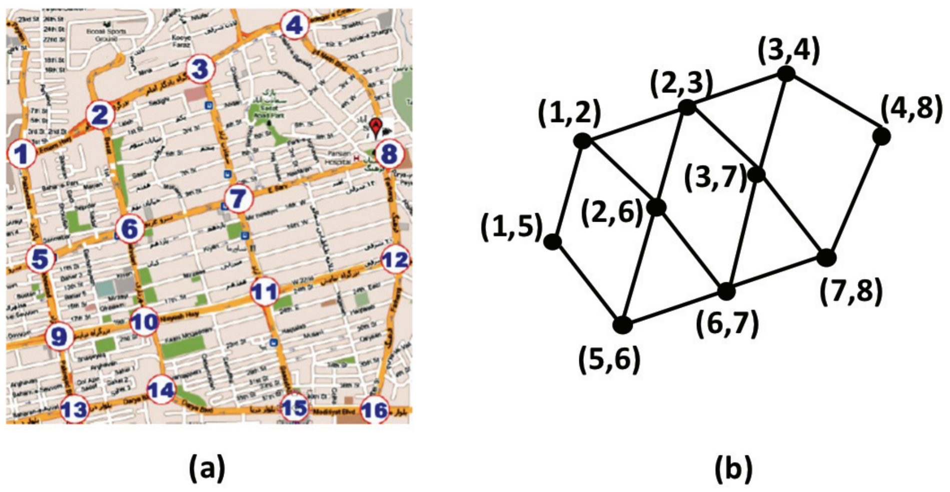


Figure 2: Value graph G_v. (a) is the original road network graph G (b) is part of the value graph G_v

variable elimination [62] to propagate the information from training data. Moreover, we use the minimum description length (MDL) principle [63] to determine the optimal number of states for the latent variable at each node.

After learning the parameters, we present an algorithm to answer queries on unobserved variables at a certain time step. The algorithm first carves out, from the full model, a *subtree* that encompasses the evidence and query nodes. It then "unrolls" the subtree into a dynamic Bayesian network and performs *belief propagation* [64] over it. Moreover, we propose a few refinements to further improve the performance.

Due to the dynamic nature of the network, the model parameters that we have learned may change over time. A frequent full learning/rebuilding of the model is expensive, while an infrequent one may risk working with an outdated and inaccurate model. Ideally we should be able to automatically and incrementally rebuild the model as needed. We propose a lightweight approach that exactly does so. Whenever there are some observed evidence values, we perform cross-validations using our query processing algorithm, and only partially re-learn the model parameters related to the observed variables. Thus, the more accessed some vertices are, the more up-to-date the model parameters surrounding them are.

3.1.1 The Model

In our solution, we devise a novel probabilistic graphical model. We intend to model the time-variant nature of the whole value graph. To do this in a scalable manner, we use one latent variable to characterize the correlation and evolvement of a *small number* of values in the graph. Thus, we have many such latent variables–these are the first tier latent variables. For each first-tier latent variable, we use a Markov chain to model its state changes over time, where a *state* is treated as a value of the latent variable. Each first-tier latent variable "emits" a number of values in the graph (through conditional distributions). This is illustrated in Figure 3, where each first-tier latent variable emits five base-level graph values (in a red solid oval).

The intuition is that each first-tier latent variable is not overwhelmed with too much information from the base-level graph values, and that we can have a model of a reasonable size and complexity. Analogously, to model the correlation and co-evolvement of the network values at a greater scope, we have the second-tier latent variables, each of which follows a Markov chain and emits a small number of correlated/co-evolving first-tier latent variables. Figure 3 only depicts one second-tier latent variable (the root node) that emits five first-tier latent variables in the dashed oval. For clarity, Figure 3 only shows a small snippet of the graph; we may have more tiers/levels.

The intuitions of our whole model are as follows. There is a reason for a real network

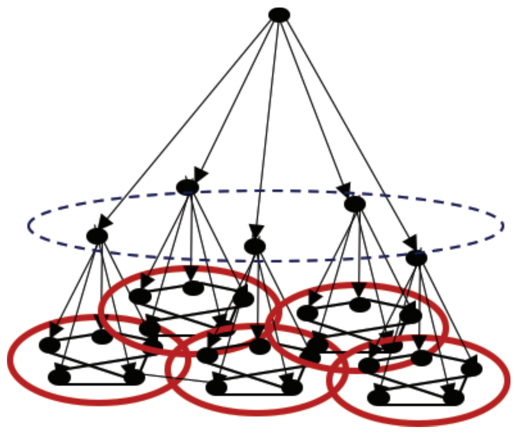


Figure 3: Probabilistic graphical model.

to have its graph structure—its edges typically contain all important relationships and correlations; otherwise missing edges should be added. Based on graph topology, our model *directly* encodes stronger correlations of random variables that are closer in the graph, and *indirectly* (at a higher level of the model) encodes weaker correlations of random variables that are distant in the graph. The hierarchical structure of our model also helps inference performance and scalability—which can be run on a local subtree of the whole model.

Note that if we "unroll" each internal node of the model across adjacent time steps (as each internal node follows a Markov chain), our model is essentially a dynamic Bayesian network. This is exactly our treatment in Section 3.1.3 for query processing, illustrated in Figure 4.

3.1.2 Data-Driven Model Learning

Recall that at the base level of our model, we need to divide the value vertices of the graph into *balanced* groups, such that vertices inside a group tend to be more correlated, and the groups should have similar sizes. Note that these are "soft" requirements. For example, because we model each group with a latent variable, it is desirable to have the groups with close sizes, but they do not have to be the exact same size. Intuitively, other conditions being equal, the more variables/vertices in a group, the more complex it is for the latent variable to characterize their state co-evolvement over time through a hidden Markov model. This motivates us to have approximately balanced group sizes.

Thus, for our specific purpose, we devise a very efficient algorithm (with a linear time amortized cost), as shown in OneLevelVertexGrouping. The key idea of the algorithm is to pick the next heaviest weight edges (i.e., most correlated vertices) to assign into a group, and to simultaneously balance the group sizes and ensure that each group size be in $[\alpha, 2\alpha)$, where α is a parameter of a small value (e.g., $\alpha = 5$).

Algorithm 1: OneLevelVertexGrouping(G)

```
Input: $G$: graph, where we add a weight to each edge $(u, v)$ as the Pearson correlation
       coefficient [15] of the values at $u$ and $v$
Output: groups of the vertices in $G$
1  initially let each vertex be its own group
2  create a priority queue $Q$ for all edges $E$ (weight as priority)
3  while pop $e$ from $Q$ with highest weight do
4      if $e$ connects two groups $c_1$ and $c_2$ where $|c_1| \leq |c_2|$ then
5          if $|c_1| < \alpha$ then
6              merge $c_1$ and $c_2$ into one group $c_{1,2}$
7              if $|c_{1,2}| \geq 2\alpha$ then
8                  let $e = (v_1, v_2)$ where $v_1 \in c_1$ and $v_2 \in c_2$
9                  $n \leftarrow \lfloor \frac{|c_{1,2}|}{2} \rfloor - |c_1|$ //number of vertices to move
10                 BFS from $v_2$ in $c_2$ to get $n$ vertices $V_{2\rightarrow 1}$
11                 move $V_{2\rightarrow 1}$ from $c_2$ to $c_1$, resulting in $c_1'$ and $c_2'$
12                 replace $c_{1,2}$ by $c_1'$ and $c_2'$ (split)
```

Line 2 of the algorithm creates a priority queue of edges based on their weights. We use the Pearson correlation coefficient [15] as the weight. Thus, edges connecting most correlated value vertices will be popped out first in line 3. As long as one of the groups connected by the current edge e is smaller than α, we merge the two groups in line 6, making e an intra-group edge. If it turns out that the other group is too large and the total size exceeds the "invariant" range $[\alpha, 2\alpha)$, lines 8-12 split the merged group into two, effectively balancing the sizes of the original two groups. Line 9 computes the number of vertices to be moved from the bigger group to the smaller one, while line 10 does a breadth-first search (BFS) in the bigger group to get the set of vertices $V_{2\to 1}$ to move.

Note that although this algorithm is initially similar to hierarchical clustering [15], the bulk of the algorithm (lines 4-12) is completely different, for a different goal—hierarchical clustering is only concerned with homogeneous clusters regardless of their sizes, while our algorithm aims at two simultaneous targets: (1) high correlations within each group (which could be negative correlation, not necessarily homogeneous), and (2) approximate balance of group sizes.

In addition, the related problem of graph partitioning is an NP-hard problem; see [65] for a survey of approximate solutions. Most of them are more expensive than linear cost. In particular, a state-of-the-art algorithm [66] achieves a (k, v) balanced partition, i.e., it partitions G into k components of at most size $v \cdot (n/k)$, where $v > 1$ is a constant. It has an expensive computation cost of $O(n^2)$. More importantly, it would not meet our need since it does not bound from below the number of nodes in a partition (it could be 1). We have two different aims, as discussed above, in addition to being efficient. We prove the following properties.

Theorem 1. *After running* OneLevelVertexGrouping*, every group has at least α but less than 2α vertices, unless there is an isolated small group with less than α vertices and no edges coming out of it.*

Proof. We say that an edge is an *inter-group* edge if its two endpoints are in two groups, and an *intra-group* one if its two endpoints are in one group. Initially, all edges are inter-group ones as each vertex is a group. Then in descending weight order, all these edges are traversed exactly once.

First, no group will have size 2α or more in the end. We prove this by induction over the while loop in line 3. Initially it must be true as each group is of size 1. Now suppose it is true after the previous while loop (line 3); we show that it is true after the current loop. As each group is less than 2α, the merged group size $|c_{1,2}|$ in line 7 is no more than $4\alpha - 2$. Since the two groups from the split in lines 11-12 have sizes $\lfloor \frac{|c_{1,2}|}{2} \rfloor$ and $\lceil \frac{|c_{1,2}|}{2} \rceil$, respectively, they are both less than 2α again.

We next show that at the end of the algorithm, no inter-group edge is connected to a group that is smaller than size α. Initially all inter-group edges are put in the priority queue Q (line 2), and each of them is visited in descending weight order (line 3). When a merge happens in line 6, the inter-group edge e in the current iteration (line 3) will become an intra-group one. On the other hand, when a merge does not happen, it could be one of the two reasons: (1) edge e has already become intra-group when a previous iteration merges the two groups it connects; (2) the two groups it connects are already of size at least α. In either case, it still holds that no inter-group edge is connected to a group that is smaller than size α.

A noteworthy situation arises when a split is done after a merge (lines 11-12). This is the only situation where some intra-group edges may revert back to be inter-group. However, such an inter-group edge must be connecting c'_1 and c'_2, both of which are of size at least α, because their sizes are $\lfloor \frac{|c_{1,2}|}{2} \rfloor$ and $\lceil \frac{|c_{1,2}|}{2} \rceil$ and $|c_{1,2}| \geq 2\alpha$ (line 7). To summarize the proof above, by the time we iterate through all the edges in Q, there is no inter-group edge connected to a group smaller than size α. Thus, Theorem 1 is true. $\square$

Most real world large graphs are connected (such as social networks or road networks). Hence it is not a problem to have groups all of size $[\alpha, 2\alpha)$. Even if the graph were not connected, the group size invariant $[\alpha, 2\alpha)$ in Theorem 1 would only require each *component* have size at least α. If indeed there are smaller than α components/groups, that is the nature of the graph and we can just leave them as they are. We next prove the complexity of the algorithm.

Theorem 2. *The amortized cost of* OneLevelVertexGrouping *is* $O(|E|+|V|)$. *In particular, it is not a function of* α.

Proof. A naïve analysis will give the cost $O(\alpha \cdot |E| + |V|)$, as we may have to traverse and move $O(\alpha)$ vertices (lines 10-11) in an iteration of an edge. We use an amortized analysis to give a tighter bound. First, observe that each of the new groups c'_1 and c'_2 from the split (lines 11-12) always has a size in $[\alpha, \frac{3}{2}\alpha)$. We already show the lower bound of α as part of the proof of Theorem 1. The upper bound $\frac{3}{2}\alpha$ is because c'_1 and c'_2 have sizes $\lfloor \frac{|c_{1,2}|}{2} \rfloor$ and $\lceil \frac{|c_{1,2}|}{2} \rceil$, and because $|c_{1,2}| < 3\alpha$ as $|c_1| < \alpha$ (line 5) and $|c_2| < 2\alpha$ (proof of Theorem 1).

We say that a vertex is *on the small side* if it is in a group of size less than α, and *on the big side* if it is in a group of size at least α. Initially all $|V|$ vertices are on the small side. Observe that once a vertex goes from the small side to the big side, it never goes back, because even a split (line 12) will have groups of size at least α. Thus, this is a one-way move. Initially, at least 2α small-side points are "consumed" (i.e., moved to the big side) before triggering a split, and a split costs $O(\alpha)$ (for BFS and moving vertices). After a split, since each of the two groups has size less than $\frac{3}{2}\alpha$ (proven above), it will consume at least $\frac{\alpha}{2}$ small-side points before triggering another split. Overall, $O(\alpha)$ small-side points are consumed to trigger a split that also costs $O(\alpha)$. The initial number of small-side points is $|V|$; thus the total split cost in the whole algorithm is $O(|V|)$. The amortized cost of the algorithm is $O(|E| + |V|)$. $\square$

We repeat the same algorithm for upper levels of the model, where each node is the parent of a group of nodes in the lower level, and each edge (v_1, v_2) in the lower level has a corresponding edge between v_1 and v_2's parents in the upper level (if they are different).

Given training data for a period of time, we now discuss how to determine our model's parameters. The parameters at each internal node of the model consist of three parts: (1) the number of states, (2) the transition probabilities between states, and (3) the emission probabilities to lower level child nodes.

Recall that the bottom level of our model is the original value graph nodes, which we call the leaf nodes. The parameter learning proceeds in a bottom-up order, starting from each node N right above the bottom level. To learn the parameters of N, our basic idea is to extend the Baum-Welch algorithm [61] to multiple emission variables, and hence multiple emission probability distributions, one for each child node of N. In order to do this, we propagate up a distribution function $f_{t,N}(j)$, i.e., the probability of state/value j at node N at time step t. In a way, this f function bottom-up propagation can be regarded as a bottom-up order *variable elimination* [62] for the Bayesian network at time step t. The algorithm is shown in AllParameterLearning.

The loop in line 2 is to iterate over all possible numbers of states of node N from 2 to the maximum number of states σ allowed for a variable (e.g., $\sigma = 32$). Later on, in lines 21-26, we will use the minimum description length (MDL) principle [63] to determine the optimal number of states. Lines 3-4 initialize the state transition probabilities at node N, $A_N[i][j]$ (from state i to state j), and the emission probabilities to each child C_i, $p_{C_i}[j][k]$ (from state j of N to state/value k of C_i) to uniform distributions.

Then the "repeat" loop in lines 6-20 performs the E and M steps (as in the *expectation maximization* method of the extended Baum-Welch algorithm described earlier), until the distributions A_N and p_{C_i}'s converge. Lines 8-14 are the E-step. In line 9, the f function value, as discussed above, is propagated from all child nodes to N, taking into account the f values at the children and the emission probabilities. The α, β, and γ functions in lines 10-14 are similar to Baum-Welch, except that we use f function values as the emission probabilities. Likewise, lines 16-19 are the M-step where we re-estimate A_N and p_{C_i}'s.

Finally, in lines 21-26, we calculate and compare the description length dl, and pick the number of states that results in the minimum dl, following the minimum description length (MDL) principle [63] (which basically says that the best model should be the most concise one).

To summarize, the AllParameterLearning algorithm is correct since (1) it runs a variant/generalization of the Baum-Welch algorithm at each internal node, and (2) instead of using emission probabilities as in Baum-Welch, we use f functions as in bottom up variable elimination (VE) to express the likelihood of producing all the leaf nodes of the subtree rooted at N. The f functions take the observed values at the leaf nodes.

We analyze its complexity as follows.

Algorithm 2: AllParameterLearning(G)

```
Input: $G$: graph with observed values in times 1 to $T$
Output: all parameters of the model
1  for each internal node $N$ in a bottom-up order do
2      for $|S| \leftarrow 2, ..., \sigma$ do
3          initialize $A_N[i][j]$ to $\frac{1}{|S|}(1 \le i, j \le |S|)$
4          for each child node $C_i$ do
5              initialize $p_{C_i}[j][k]$ to $\frac{1}{|S_i|}(1 \le j \le |S|, 1 \le k \le |S_i|)$   //prob at state $j$
               of $N$ emitting state $k$ of $C_i$
6          repeat
7              //E-step
8              for $t \leftarrow 1...T$ and $j \leftarrow 1...|S|$ do
9                  $f_{t,N}[j] \leftarrow \prod_{C_i} \{\sum_{C_i=k} f_{t,C_i}[k] \cdot p_{C_i}[j][k]\}$
10                 $\alpha[t][j] \leftarrow \sum_{i=1..|S|} \alpha[t-1][i] \cdot A_N[i][j] \cdot f_{t,N}[j]$
11             for $t \leftarrow T...1$ and $i \leftarrow 1...|S|$ do
12                 $\beta[t][i] \leftarrow \sum_{j=1..|S|} \beta[t+1][j] \cdot A_N[i][j] \cdot f_{t,N}[i]$
13             for $t \leftarrow 1...T, i \leftarrow 1...|S|, j \leftarrow 1...|S|$ do
14                 $\gamma_t[i][j] \leftarrow \alpha[t-1][i] \cdot A_N[i][j] \cdot f_{t,N}[j] \cdot \beta[t][j]$
15             //M-step
16             for $i \leftarrow 1...|S|, j \leftarrow 1...|S|$ do
17                 $A_N[i][j] \leftarrow \frac{\sum_{t=1}^{T} \gamma_t[i][j]}{\sum_{k=1}^{|S|} \sum_{t=1}^{T} \gamma_t[i][k]}$
18             for each child node $C_l, j \leftarrow 1...|S|, k \leftarrow 1...|S_l|$ do
19                 $p_{C_l}[j][k] \leftarrow \frac{\sum_{i=1}^{|S|} \sum_{t=1}^{T} f_{t,C_l}[k] \cdot \gamma_t[i][j]}{\sum_{i=1}^{|S|} \sum_{t=1}^{T} \gamma_t[i][j]}$
20         until converged
21         $dl \leftarrow 0$; $z \leftarrow \sum_{i,j} A_N[i][j] + \sum_{C_l,j,k} p_{C_l}[j][k]$
22         for $i \leftarrow 1...|S|, j \leftarrow 1...|S|$ do
23             $p \leftarrow \frac{A_N[i][j]}{z}$; $dl \leftarrow dl + p \log \frac{1}{p}$
24         for each child node $C_l$, $j \leftarrow 1...|S|, k \leftarrow 1...|S_l|$ do
25             $p \leftarrow \frac{p_{C_l}[j][k]}{z}$; $dl \leftarrow dl + p \log \frac{1}{p}$
26         choose the min $dl$ among different $|S|$ for the $S$ of node $N$
```

Theorem 3. *Suppose that it takes θ iterations for the loop in lines 6-20 to converge, and that we try a constant number of states at each node. Then the complexity of* AllParameterLearning *is $O(\theta T|V|)$.*

Proof. The reason of this complexity result is due to a couple of key points: (1) The algorithm goes through every internal node once to learn its parameters. (2) Even though there are loops in line 4 and line 18 to iterate through all child nodes of N, each node has only one parent. Thus, the code in the loop is only done once for every node during the whole run of the algorithm, i.e., the amortized (aggregate) cost of this part is linear to $|V|$. $\square$

3.1.3 Inference to Answer Queries

We now study the query answering problem, provided that we have learned the model from the previous section. We are given a number of observed values of variables within the time interval $[t_b, t_e]$, and we have a number of queries $q_1, ..., q_c$. Each query q_i $(1 \leq i \leq c)$ inquires about the (unobserved) value of a variable at time t (where t is either inside or outside $[t_b, t_e]$). For each query q_i, the user may specify the minimum set of observations E_i that must be used to answer the query.

The basic idea of the query processing algorithm is to first "unroll" the model across multiple time steps in the observed evidence interval T and the queried time steps. This is illustrated in Figure 4, where the model is unrolled across 4 time steps. The roots of the four trees are just one node (the *lowest common ancestor* of all evidence nodes and query nodes) unrolled at four adjacent time steps. The green oval leaf nodes are the observed variables, while the 4 leaf nodes marked with "?" (one at each time step) are the query nodes. In this example, all observations are between time step 1 and 3. Query nodes could be before observation start time too (e.g., time step 0).

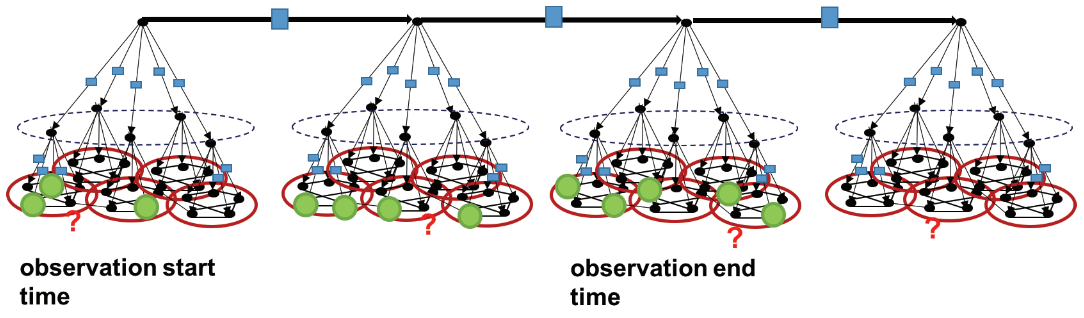


Figure 4: Part of graphical model

The unrolled model is essentially a dynamic Bayesian network, and fortunately it does not have any cycles, which means we can perform *exact inference* by belief propagation [64]. To do this, we first add auxiliary "factor" nodes, one in the middle of each edge, as shown in Figure 4. Informally, a factor node for a Bayesian network just contains the conditional probability function between the nodes it connects. The product of all factor node functions is the joint distribution of the whole model.

We present the basic query processing algorithm in BeliefPropagationOverBridges. The multiple versions of the lowest common ancestor (LCA) of the evidence and query nodes are connected by "bridges", which are from the Markov model of the LCA, shown as the root nodes in Figure 4. We will later present some optimizations on top of this basic algorithm.

Line 1 of the algorithm finds the minimum subtree, rooted at the LCA of the relevant nodes. Line 2 is the "unrolling" of the model across the relevant time steps. Lines 3-6 add the factor nodes that are either the transition probability function of the Markov chain at the LCA, or the emission functions from a parent to a child. Line 7 picks a single "root" R at the "middle" of the times of the observed nodes, as the root of the whole unrolled

```
Algorithm 3: BeliefPropagationOverBridges
    Input: $M$: model, $q$: a query, $E$: observed evidence nodes for $q$
    Output: estimated probability distribution of $q$
 1  identify the minimum subtree $T_{eq}$ that contains $E$ and $q$
 2  build a dynamic graph by connecting the roots of $T_{eq}$ at different times
 3  for each bridge edge do
 4      add a factor node that has the transition function of the chain
 5  for each edge in $T_{eq}$ in all time steps do
 6      add a factor node that has the emission function
 7  $R$ ←root of $T_{eq}$ at time $\lceil \frac{t_b+t_e}{2} \rceil$
 8  do belief propagation from each leaf of the trees to $R$, taking $E$
 9  do belief propagation from $R$ to all leaves
10  return the product of the messages to $q$
```

dynamic Bayesian network. This root R can in fact be any node and belief propagation will give the same result for the query node [64]. We pick this particular root R for the ease of presentation of the algorithm in lines 8-9. Basically, belief propagation comprises two rounds: from all leaves to root (line 8), and then from root to all leaves (line 9). We illustrate how this works via an example below.

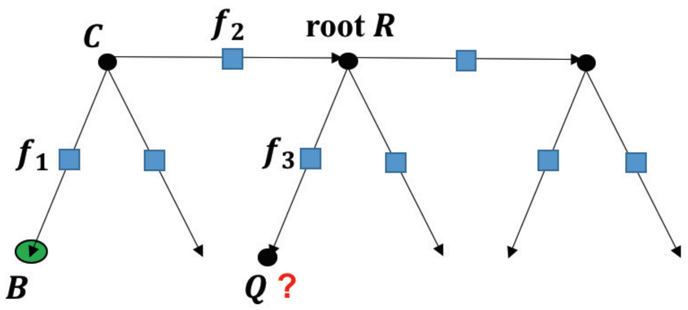


Figure 5: BeliefPropagationOverBridges algorithm.

Example 5. *We illustrate the algorithm through a simple example in Figure 5, where we show three time steps and the chosen root R of this unrolled model is in the middle. For the ease of illustration, each tree only has 2 levels, and we focus on the observed evidence node B (marked with a green oval) and the query node Q (marked with a "?"). Suppose there* *are only two states (values) of each node (0 and 1) and the transition matrix of the Markov chain at* R *is* $A = \left[\begin{smallmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{smallmatrix}\right]$ *, and that the emission distribution from* C *to* B *(or* R *to* Q*) is* $P = \left[\begin{smallmatrix} 0.2 & 0.8 \\ 0.8 & 0.2 \end{smallmatrix}\right]$*. That is, factor node* f_2 *contains* A*, while* f_1 *and* f_3 *contain* P*. Suppose the observed value at* B *is* 0*. Our goal is to answer* Q*.*

First consider round 1 of belief propagation from leaves to R *(line 8). For leaf* B*, we have* $\sum_B \mu_{B \to f_1}(B) Pr[B|C] = \mu_{f_1 \to C}(C)$*, where* $\mu_{B \to f_1}(B)$ *stands for the message from* B *to* f_1*, and it is a function of* B*, and* $Pr[B|C]$ *is just the function* f_1 *has (i.e.,* P*). Thus* $\mu_{f_1 \to C}(0) = P(0|0) = 0.2$ *and* $\mu_{f_1 \to C}(1) = P(0|1) = 0.8$*. This is because we are given the observed* B *value* 0*; hence* $\mu_{B \to f_1}(0) = 1$ *and* $\mu_{B \to f_1}(1) = 0$*. We get* $P(0|0)$ *and* $P(0|1)$ *from looking up the* P *matrix above.*

Once we get $\mu_{f_1 \to C}(C)$*, round 1 message passing to root continues. Next,* C *will collect the messages it receives from all its neighbors except* f_2 *(which is on the path to* R*), multiply them together, and pass it to* f_2*. Since the subtree under* C*'s right branch does not contain observed nodes, the message to* C *from its right child is simply 1. Thus,* $\mu_{C \to f_2}(C) = \mu_{f_1 \to C}(C)$*, which is obtained earlier.*

Then similar to the message from f_1 *to* C*, we can compute the message from* f_2 *to* R*,* $\mu_{f_2 \to R}(R)$*, as follows:* $\mu_{f_2 \to R}(0) = 0.2 \cdot A(0|0) + 0.8 \cdot A(0|1) = 0.32$ *and* $\mu_{f_2 \to R}(1) = 0.2 \cdot A(1|0) + 0.8 \cdot A(1|1) = 0.68$*. Here, the conditional probabilities such as* $A(0|0)$ *are looked up from the transition matrix* A *above.*

Now we proceed to round 2 of belief propagation (line 9). R *will send a message to* f_3*, i.e.,* $\mu_{R \to f_3}(R)$*, as a product of all messages* R *receives from all other neighbors. This is just the* $\mu_{f_2 \to R}(R)$ *we have above (since, again, other neighbors give a message 1). Then finally, we compute our goal* $\mu_{f_3 \to Q}(Q)$ *as:* $\mu_{f_3 \to Q}(0) = 0.32 \cdot P(0|0) + 0.68 \cdot P(0|1) = 0.608$, *and* $\mu_{f_3 \to Q}(1) = 0.32 \cdot P(1|0) + 0.68 \cdot P(1|1) = 0.392$*, which is the estimated distribution of* Q.

The correctness of BeliefPropagationOverBridges is due to the fact that the unrolled graph is still a tree without loops, and hence we can perform exact inference. It is easy to see that the complexity of the algorithm is linear to the number of vertices in the value graph and to the length of the time interval T, as message passing is linear to the tree size.

We propose a few refinements and optimizations to the query processing algorithm. First, we observe that, rather than spending $O(n \cdot |T|)$ time to process one query at a time, we batch a set Q of many queries that can be answered together using the same set of observed evidence E. The rationale of doing this is analogous to the multi-query optimization, i.e., to share the computation efforts among concurrent queries.

Our second refinement is for the first round of belief propagation in line 8 of BeliefPropagationOverBridges. We only initiate the messages from the observed evidence nodes E, but ignore all other leaves. We will show that the end result is the same.

The third refinement is for the second round of belief propagation in line 9 of BeliefPropagationOverBridges. We only propagate messages along the paths to the set of query nodes Q. To figure out the paths first, we can start from the query nodes and trace back towards R. The message passing paths will be the reverse of these paths. Again, we show that the final query result is equivalent with this optimization.

Theorem 4. *Assuming that the Markov chain at root* R *in* BeliefPropagationOverBridges *has a stationary distribution, our query processing algorithm will return exactly the same results with or without the three refinements.*

Proof. It is clear that the first refinement does not change query results. Let us consider the second refinement. In the first round of message passing, a leaf without an observed value passes a message of constant value “1” to the factor node above by the definition of belief propagation. Recall that, as illustrated in Example 5, $\sum_B \mu_{B\to f_1}(B)Pr[B|C] = \mu_{f_1\to C}(C)$. When $\mu_{B\to f_1}(B) = 1$, we have $\mu_{f_1\to C}(C) = \sum_B Pr[B|C] = 1$. Thus, this constant message 1 keeps getting propagated throughout the whole subtree that does not have any observed evidence. If the message 1 gets to the root of the tree at a particular time step, e.g., node C in Figure 5, the message will need to be passed towards the designated root R. This message passing could be in two directions: in increasing time order (the same as the Markov chain) or in decreasing time order (reverse of the Markov chain) if it is from a node to the right of R in Figure 5. It is well known that, if a Markov chain has a stationary distribution, the reverse chain is also a Markov chain. Hence, in either direction, the summation equation as shown above still holds, and message “1” will be propagated all the way.

Next we consider the third refinement (for round 2). We can prove this inductively. Consider a query node (leaf) q. There is only one path from the root to q. To get the message to q, we only need to consider its parent node p (as there is only one edge to q). For p to pass a message to q in round 2, p needs to multiply messages from all its other children (from round 1) and from its parent. This argument goes on inductively, until we reach the root node of q's subtree, e.g., node C in Figure 5. C's messages from its neighbors are all either from round 1 or from the path from R. This holds all the way to the designated root R. $\square$

Note that the refinements proposed here make it convenient to handle other types of queries such as aggregates and joins, which we plan to study in detail in future work.

In massive dynamic networks, the model parameters may change after some time. We propose an approach that addresses this issue. The basic idea is that during query processing, we do leave-one-out cross-validation [15] for a small constant number of observed leaves. If the accuracy is below a threshold, we decide to partially rebuild the model around the observed evidence E. To partially (incrementally) rebuild the model, we invoke the AllParameterLearning method, but only learn the nodes that are ancestors of the observed evidence nodes E. Therefore, this is much more efficient than a complete rebuild/re-learn of the model. Effectively, the part of the model that is accessed more frequently will be maintained more up-to-date. The algorithm is presented in CrossValidationPartialUpdate, where BPOB is short for BeliefPropagationOverBridges, the algorithm that answers queries.

Algorithm 4: CrossValidationPartialUpdate

```
1  repeat while running BPOB
2      choose one evidence leaf $e_q$ uniformly at random
3      let the path from $R$ to $e_q$ be $P_{eq}$
4      round 1 of BPOB (line 8) is exactly the same as before
5      in round 2 of BPOB (line 9), create a second version of messages for nodes in $P_{eq}$ by
         treating $e_q$ as a query
6      if $e_q$ result differs from the observed within a bound then
7          return
8  until $k$ times
9  $A \leftarrow$set of ancestors of the leaves in $E$
10 run AllParameterLearning but only for nodes in $A$
```

The loop in lines 1-8 does leave-one-out cross validation test on k (a small constant number) values randomly chosen from the observed evidence values. If all k tests have errors greater than some threshold, we partially re-build the model over the observed values E.

One may wonder about the correctness of cross-validating the query result on e_q, as line 4 of the algorithm does not exclude e_q from passing its value/evidence messages to R in round 1. For clarity, we formally summarize the correctness of the algorithm and its complexity in the following theorem.

Theorem 5. *The cross-validation in lines 4-5 is sound: specifically, the e_q result we get is based on all observed evidence except e_q itself. Furthermore, the cross-validation does not increase the asymptotic complexity of BPOB. Let the probability that a query result has error exceeding a bound be ϵ (line 6). Then the probability that a partial update is run is ϵ^k, and the expected cost is $\epsilon^k \theta e |T| \log |V|$, where θ is the number of iterations for the loop in BPOB to converge, e is the number of evidence nodes, and $|T|$ is the time interval length of evidence.*

Proof. We first show the soundness of cross-validation. The key point is that, in line 4, even though we do not keep a second version of messages without e_q participating in round 1 of message passing, any messages that the observed e_q value contributes to will not affect the message eventually passed to e_q in round 2. To see why this is true, consider the messages that goes through path P_{eq} (from R to e_q) in round 2. Regardless of whether such a message is "horizontal" (e.g., from R to f_2 to C in Figure 5) or "vertical" (from a root down to e_q), the messages from the neighbors that form the product do not contain any factors from the message of e_q in round 1.

Since the loop in lines 1-8 is repeated at most a constant number of times, and the extra message passing in line 5 is subsumed by the asymptotic cost of BPOB query processing itself, the complexity is the same. Lines 6-7 and the loop in lines 1-8 indicate that only if all

k tests fail, does it do the partial model update; hence the probability is ϵ^k. It is also clear that the expected cost is $\epsilon^k \theta e |T| \log |V|$, similar to the analysis in Theorem 3. $\square$

3.2 Unobserved Links/relationships Prediction

After introducing the solutions to attribute values prediction on large dynamic graphs, let us move to the problem of efficiently answering queries according to 2.2.

3.2.1 Transform of Embedding Vectors

We first apply an existing knowledge graph embedding scheme, such as TransE [15] or TransA [37], to get the embedding vectors of each node and relationship type of the graph. This embedding is the algorithm $\mathcal{A}$ that induces the virtual knowledge graph. We aim to index these embedding vectors. However, typically they are of tens or hundreds of dimensions, which is too inefficient for commonly used spatial indices. Hence, in this section, we apply the Johnson-Lindenstrauss (JL) transform [50] to convert the embedding vectors into a low dimension (such as 3) before indexing them. A major technical challenge is how to provide accuracy guarantees, since classical JL transform and its proof of distance preservation only apply to the situations of mapping to at least hundreds of dimensions. In Sections 3.2.2 and 3.2.3, we will further discuss indexing and query processing.

A knowledge graph embedding scheme results in vectors of dimensionality d in the vicinity of hundreds, in an embedding space $\mathbb{S}_1$. There is one vector for each vertex (entity) and for each relationship type. We then perform JL transform on these vectors, except that the classical JL transform and its analysis require the resulting dimensionality to be typically rather high, at least in the hundreds—while we need it to be a small number α (e.g., 3). Let this α-dimensional space be $\mathbb{S}_2$. Specifically, the mapping of this transform is:

$$\mathbf{x} \mapsto \frac{1}{\sqrt{\alpha}}\mathbf{A}\mathbf{x}$$

where the $\alpha \times d$ matrix **A** has each of its entries chosen i.i.d. from a standard Gaussian distribution $N(0, 1)$. Intuitively, each of the α dimensions in $\mathbb{S}_2$ is a random linear combination of the original d dimensions in $\mathbb{S}_1$, with a scale factor $\frac{1}{\sqrt{\alpha}}$, so that the L_2-norm of **x** is preserved.

Note that our proof of the following result, Theorem 6, is inspired by, but differs significantly from that in [50]. In particular, the analysis and proof in [50] only apply to the case when the ε below is between 0 and 1 for the upper bound; moreover, we obtain a tighter bound for a small dimensionality α and a relaxed ε range.

Theorem 6. *For two points **u** and **v** in the embedding space $\mathbb{S}_1$ that are of Euclidean distance l_1, their Euclidean distance l_2 in space $\mathbb{S}_2$ after the transform has the following probabilistic upper bound:*

$$Pr[l_2 \geq \sqrt{1+\varepsilon} \cdot l_1] \leq \Delta_u(\varepsilon) \doteq \left(\frac{\sqrt{1+\varepsilon}}{e^{\varepsilon/2}}\right)^{\alpha} \tag{1}$$

for any $\varepsilon > 0$, where α is the dimensionality of $\mathbb{S}_2$. Similarly, the probabilistic lower bound is

$$Pr[l_2 \leq \sqrt{1-\varepsilon} \cdot l_1] \leq \Delta_l(\varepsilon) \doteq \left(\sqrt{1-\varepsilon} \cdot e^{\varepsilon/2}\right)^{\alpha} \tag{2}$$

for any $0 < \varepsilon < 1$.

Proof. First, each of the α entries in the vector **Ax** is a random variable with the same distribution

$$W_i = \sum_{j=1}^{d} x_j A_{ij} (1 \le i \le \alpha) \tag{3}$$

where $A_{ij} \sim N(0,1)$. For **x** $=$ **v** $-$ **u**, due to the property of normal distributions, W_i also follows a normal distribution $N(0, l_1^2)$, where l_1 is as stated in the theorem. This is because each $x_j A_{ij} (1 \le j \le d)$ follows $N(0, x_j^2)$, and hence their sum is also a normal distribution with mean 0 and variance $\sum_{j=1}^{d} x_j^2 = l_1^2$.

Next, according to the definition of the transform, the distance square l_2^2 correspond to a random variable

$$Y = \frac{1}{\alpha} \cdot \sum_{i=1}^{\alpha} W_i^2 \tag{4}$$

Note that $E[Y] = E[W_i^2] = Var[W_i] + E[W_i]^2 = l_1^2$, which means that the transform is unbiased in preserving the Euclidean distance.

Towards a probabilistic upper bound, resorting to moment generating function, we have, for $t > 0$,

$$\begin{aligned} Pr[Y \ge (1+\varepsilon) l_1^2] &\le Pr[e^{t\alpha Y} \ge e^{t\alpha(1+\varepsilon)l_1^2}] \\ &\le \frac{\mathbf{E}[e^{t\alpha Y}]}{e^{t\alpha(1+\varepsilon)l_1^2}} = \prod_{i=1}^{\alpha} \frac{\mathbf{E}[e^{tW_i^2}]}{e^{t(1+\varepsilon)l_1^2}} \end{aligned} \tag{5}$$

where the second inequality is due to the Markov inequality [67], and the last equality is based on Equation (4). Recall that $W_i \sim N(0, l_1^2)$ from Equation (3), which gives

$$\begin{aligned} \mathbf{E}[e^{tW_i^2}] &= \frac{1}{\sqrt{2\pi l_1^2}} \int e^{tw^2} \cdot e^{-\frac{w^2}{2l_1^2}} dw = \frac{1}{\sqrt{2\pi l_1^2}} \int e^{-\frac{w^2}{2l_1^2}(1-2l_1^2 t)} dw \\ &= \frac{1}{\sqrt{2\pi l_1^2}} \int e^{-\frac{z^2}{2l_1^2}} \frac{dz}{\sqrt{1-2l_1^2 t}} = \frac{1}{\sqrt{1-2l_1^2 t}} \end{aligned} \tag{6}$$

where the first equality is based on the definition of Gaussian distribution, the third equality is based on defining a new variable $z \doteq w \cdot \sqrt{1-2l_1^2 t}$, and the last equality is due to the fact that the integral of a Gaussian variable z is 1. Note that this requires an additional constraint $1 - 2l_1^2 t > 0$; we now require $0 < t < \frac{1}{2l_1^2}$. Plugging Equation (6) into Equation (5), we have

$$Pr[Y \geq (1+\varepsilon)l_1^2] \leq \left(\frac{1}{\sqrt{1-2l_1^2 t} \cdot e^{t(1+\varepsilon)l_1^2}} \right)^{\alpha} \tag{7}$$

Thus, we are free to choose t in its range $(0, \frac{1}{2l_1^2})$ in order to get the tightest bound from Equation (7). Towards this goal, we define y to be the denominator in Equation (7), and by setting $\frac{dy}{dt} = 0$, we get the optimal $t^* = \frac{\varepsilon}{2l_1^2(1+\varepsilon)}$ (satisfying the range constraint of t for any $\varepsilon > 0$), which gives the probability bound $\Delta_u(\varepsilon) \doteq \left(\frac{\sqrt{1+\varepsilon}}{e^{\varepsilon/2}}\right)^{\alpha}$ as required in Equation (1). Note that the event $Y \geq (1+\varepsilon)l_1^2$ in Equation (7) is equivalent to $l_2 \geq \sqrt{1+\varepsilon} \cdot l_1$ in Equation (1).

The derivation of the probabilistic lower bound is similar. By requiring $t < 0$, we have $Pr[Y \leq (1-\varepsilon)l_1^2] \leq Pr[e^{t\alpha Y} \geq e^{t\alpha(1-\varepsilon)l_1^2}]$, and the rest of reasoning (which we omit) is analogous to the above for the upper bound. Finally, we get the optimal setting $t^* = \frac{-\varepsilon}{2l_1^2(1-\varepsilon)}$ (satisfying the constraint $t < 0$ for any $0 < \varepsilon < 1$), giving $\Delta_l(\varepsilon) \doteq (\sqrt{1-\varepsilon} \cdot e^{\varepsilon/2})^{\alpha}$ as in Equation (2). $\square$

To see an example of the upper bound, we set $\varepsilon = 3$, and suppose the JL transform has dimensionality $\alpha = 3$, then with confidence 91.2%, $l_2 < 2l_1$. For an example of lower bound, by setting $\varepsilon = \frac{15}{16}$ (again $\alpha = 3$), we have that, with confidence at least 94%, $l_2 > \frac{l_1}{4}$.

3.2.2 Cracking and Uneven Indices for Virtual Knowledge Graphs

Once we transform all entity points into the low-dimensional space $\mathbb{S}_2$, we perform indexing in order to answer queries on the virtual knowledge graph. The basic idea is that we bulk-load/build an R-tree index in a top-down manner, and continue the "branching" on demand—only as needed for queries. As a result, regions in the embedding space $\mathbb{S}_2$ that are more relevant to queries (e.g., $\mathbf{h} + \mathbf{r}$) are indexed in finer granularities, while the irrelevant regions stay at high levels of the tree. Thus, unlike the traditional R-trees that are balanced, the index that we build is unbalanced (uneven).

We are indexing a set of rectangular objects D (for our problem, they are actually just a set of points—a special case of rectangles—in space $\mathbb{S}_2$). The basic idea of BulkLoad-Chunk is to first sort D into a few *sort orders* $D^{(1)}, D^{(2)}, ..., D^{(S)}$, for example, based on the 2α coordinates of the α-dimensional rectangles (e.g., $S = 2\alpha$). Note that each $D^{(i)}$ is a sorted list of rectangles.

Then BulkLoadChunk performs a greedy top-down construction of the R-tree. Due to dense packing, it is known in advance how many data objects every node *covers*. Each time, we perform a binary split of an existing *minimum bounding region* (MBR) at a node based on one of the *sort orders* and a *cost model*. A cost model penalizes a potential split candidate that would cause a significant overlap between the two MBRs after the split. The cost model (which we omit in the pseudocode) is invoked in the $cost$ function in line 4 of the BestBinarySplit function. This binary split is along one of the $M - 1$ boundaries if we are partitioning a node into M child nodes (thus it requires $M - 1$ binary splits). This choice is greedy.

Algorithm 5: BulkLoadChunk $(\mathbf{D}, h)$

```
Input: D: rectangles in a few sort orders $D^{(1)}, ..., D^{(S)}$ ($D^{(i)}$ is a list of rectangles
          in a particular sort order)
h: the height of the R-tree to build
Output: the root of the R-tree built
1 if $h = 0$ then
2     return BuildLeafNode($D^{(1)}$)
3 $m \leftarrow \lceil \frac{D^{(1)}}{M} \rceil$ //$M$ is capacity of a nonleaf node; $m$ is # rectangles
    per child node's subtree
4 $\{\mathbf{D}_1, ..., \mathbf{D}_k\} \leftarrow$Partition $(\mathbf{D}, m)$ //$k = M$ unless in the end
5 for $i \leftarrow 1$ to $k$ do
6     $n_i \leftarrow$ BulkLoadChunk($\mathbf{D}_i, h - 1$) //Recursively bulk load lower
        levels of the R-tree.
7 return BuildNonLeafNode($n_1, ..., n_k$)

  Function Partition $(\mathbf{D}, m)$ //partition data into $k$ parts of size $m$
1 if $|D^{(1)}| \leq m$ then
2     return D //one partition
3 L, H $\leftarrow$ BestBinarySplit $(\mathbf{D}, m)$
4 return concatenation of Partition($\boldsymbol{L}, m$) and Partition($\boldsymbol{H}, m$)

  Function BestBinarySplit $(\mathbf{D}, m)$ //find best binary split of D
1 for $s \leftarrow 1$ to $S$ do
2     $F, B \leftarrow$ ComputeBoundingBoxes($D^{(s)}, m$)
3     for $i \leftarrow 1$ to $M - 1$ do
4         $i^*, s^* \leftarrow i$ and $s$ with the best $cost(F_i, B_i)$
5 $key \leftarrow$ SortKey($D^{s^*}_{i^* \cdot m}, s^*$) //sort key of split position
6 for $s \leftarrow 1$ to $S$ do
      //split each sorted list based on key of $s^*$
7     $L^{(s)}, H^{(s)} \leftarrow$ SplitOnKey($D^{(s)}, s^*, key$)
8 return $\boldsymbol{L}, \boldsymbol{H}$
```

Some functions are omitted for succinctness. For instance, ComputeBoundingBoxes takes as input a sorted list of rectangles and the size m of each part to be partitioned, and returns two lists of bounding rectangles F (front) and B (back), where F_i and B_i are the two resulting MBRs if the binary split is at the ith (equally spaced) position ($1 \leq i \leq M - 1$). Lines 3-4 of BestBinarySplit get the optimal split position with the least cost—position i^* of sort order s^*, and line 5 retrieves the split key from that sort order list. Based on this binary split, we maintain (i.e., split) all the S sorted lists in lines 6-7, and return them.

Instead of doing offline BulkLoadChunk, we build a cracking, partial, and uneven index online upon the arrival of a sequence of queries. In the complete R-tree bulk loading algorithm BulkLoadChunk, all nodes are fully partitioned top-down until the leaves at the bottom level, resulting in a balanced R-tree. In our online incremental build, however, we only grow the partitions of nodes that contain the data points in query region $\mathcal{Q}$. Thus, we end up seeing an R-tree that is unbalanced with some partitions that are not split yet, and possibly also with some leaves (having the smallest partitions).

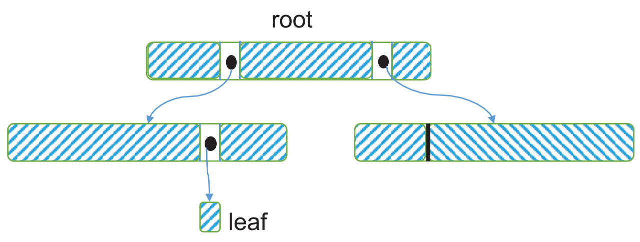


Figure 6: The notion of a contour

Definition 2. ***(Contour)***. *The* contour $\mathcal{C}$ *of a cracking R-tree is the set of current partitions (inside nodes) that do not have a corresponding child node, together with any terminal leaf nodes. We say that each such partition or leaf node is an* element e *of the contour.*

Figure 6 illustrates a contour of a cracking R-tree, where the contour is shaded and has eight elements, one of which is a leaf node (the number of data points that it covers is small enough).

Lemma 1. *Consider a contour of the R-tree at any time instant. Each element of the contour contains a mutually exclusive set of data points, and together they contain all the data points.*

Proof. This is true by an inductive argument. Initially a contour only has a single root node (one partition) that contains all the data points. Now suppose the lemma is true at some time instant. Any change next is either to perform a binary split of an existing partition or to create a new node. Creating a new node merely passes down all the data points from an existing partition to the new node, while after a binary split it is still true that the elements of the contour contain all data points and they are mutually exclusive. $\square$

Definition 3. ***(Leaf Distance).*** *At time t during the lifetime of a cracking R-tree index, if two data points d_1 and d_2 either are already in the same leaf node, or have a non-zero probability to be in the same leaf node in the future (as more queries arrive), then we say that their* leaf distance *at time t, denoted as $l_t(d_1, d_2)$, is 0. Otherwise, their leaf distance is 1.*

Thus, leaf distance is a time-variant binary random variable that depends on the sequence of incoming queries. We next have the following observation.

Lemma 2. *Consider two data points d_1 and d_2 in a partition. After a binary split at time τ, if d_1 and d_2 are still in the same partition, then in every sort order s, the positions of d_1 and d_2 can only be closer or stay the same due to the split. If d_1 and d_2 are separated into two partitions due to the split, then $l_t(d_1, d_2) = 1$ for any $t > \tau$.*

Proof. Upon a binary split of a partition, the algorithm only removes data points from every sort order s in a partition in order to maintain the sort orders at a child partition (i.e., the data points in a sub-partition is a strict subset of those in a partition). Therefore, the positions of d_1 and d_2 in s can only be closer or stay the same. On the other hand, once d_1 and d_2 are separated, they will never be united into the same node. Hence their leaf distance is 1 for any $t > \tau$. $\square$

From Lemma 1, we know that all the required data points in the query region $\mathcal{Q}$ must be in the current contour $\mathcal{C}$ of the index. Thus, our incremental index building algorithm will locate each element $e \in \mathcal{C}$ that overlaps with $\mathcal{Q}$ and determine if we need to further split e for the query. At this point, we need to revise the cost model to optimize the access cost for the current query region $\mathcal{Q}$, in addition to the previous cost function that penalizes the MBR overlap due to the split. Intuitively, after the split, the data points in $\mathcal{Q}$ should be close to each other to fit in a minimum number of pages (i.e., their leaf distance should be small). Based on the *principle of locality* in database queries [68], this optimization has a lasting benefit.

The key idea is that we extend the cost model into a two-component cost $(c_{\mathcal{Q}}, c_{\mathcal{O}})$, where $c_{\mathcal{Q}}$ is the cost estimate for accessing query region $\mathcal{Q}$, while $c_{\mathcal{O}}$ is the cost incurred by overlaps between partitions. At a contour $\mathcal{C}$ of the index at any time instant, we define $c_{\mathcal{Q}}$ to be the minimum number of leaf pages to accommodate all the data points in $\mathcal{Q}$, given the current configuration of elements in $\mathcal{C}$ and any possible future node splits. Lemma 2 implies that after every split, the leaf distance between any $d_1, d_2 \in \mathcal{Q}$ still in the same partition is expected to either get smaller or stay the same, while two separated points will be in different leaf nodes. This leads to the following result.

Lemma 3. *At a contour $\mathcal{C}$ of an index, a lower bound of the number of leaf nodes that we need to access and process is $\sum_{e \in \mathcal{C}} \lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil$, where $\mathcal{Q} \cap e$ is the set of data points in the element e (of $\mathcal{C}$) that are also in the query region $\mathcal{Q}$, and N is maximum number of data point entries that can fit in a leaf node.*

Proof. Lemma 1 and Lemma 2 show that: (1) $\forall e_1, e_2 \in \mathcal{C}, e_1 \neq e_2$, all the data points in e_1 must end up in different leaf nodes than those in e_2. (2) Moreover, the distance in position between any two target data points in $\mathcal{Q} \cap e$ monotonically decreases or stays the same in every sort order throughout the runs of our algorithm until they get to the leaves (if they are not split apart). Thus, $\lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil$ is a lower bound of the number of leaf nodes that need to be accessed and processed for each element $e \in \mathcal{C}$, and the lemma holds. $\square$

Note that for each leaf node accessed for $\mathcal{Q}$, we will need to convert each data point in it to the original embedding space $\mathbb{S}_1$ and calculate the distance to the query center point (e.g., **h+r**, to be detailed in Section 3.2.3). Hence, the number of leaf nodes accessed and processed is a reasonable measure of the first cost component $c_{\mathcal{Q}}$. The second component of cost is $c_{\mathcal{O}}$, the cost for overlaps between partitions. We increment $c_{\mathcal{O}}$ by $\beta^h \cdot \frac{\|O\|}{min(\|L\|, \|H\|)} (\beta \geq 1)$ at each binary split during the runs of the algorithm, where O is the overlap region between two resulting partitions L and H of the binary split, $\| \cdot \|$ denotes the volume of a region, h is the height of the R-tree where the split happens, and $\beta \geq 1$ is a constant indicating that an overlap higher in the R-tree has more impact and is more costly (as an R-tree search is top-down).

A remaining issue is that $c_{\mathcal{Q}}$ and $c_{\mathcal{O}}$ are two types of cost measured differently—

making the whole node-splitting cost a composite one. However, it is important to be able to compare two node-splitting costs, as required by our index building algorithms. We observe that, for our problem, the query region $\mathcal{Q}$ is derived from a ball around the center point (e.g., **h+r**), and is hence continuous in space and should not be too large (otherwise the links are too weak). Thus, it is reasonable to attempt to achieve optimal $c_{\mathcal{Q}}$ as a higher priority. As a query-workload optimized approach, we treat $c_{\mathcal{Q}}$ as the *major order* and $c_{\mathcal{O}}$ as the *secondary order* when comparing two composite costs.

Having developed the cost model, we are now ready to present our online cracking index algorithms. The algorithms incrementally build an index and use it to search at the same time. The idea is that for the initial queries more building of the index is done, while it is mainly used for search (and little is changed to the index) for subsequent queries. Overall, the cracking index only performs a very small fraction of the binary splits performed by the full BulkLoadChunk, as verified in our experiments in Section 4.

We describe our main cracking index algorithm, IncrementalIndexBuild, based on the key functions given under BulkLoadChunk. IncrementalIndexBuild takes as input a query region $\mathcal{Q}$ and the current index $\mathcal{I}$ (with contour $\mathcal{C}$). Initially $\mathcal{I}$ (and $\mathcal{C}$) is just a root node containing all the data points.

1. Instead of a top-down complete bulk-load, upon a query region $\mathcal{Q}$, we perform an incremental partial top-down build of index $\mathcal{I}$ to the elements in the current contour that overlap $\mathcal{Q}$. We store a set of data points contained in each element e of the current contour $\mathcal{C}$ in addition to its MBR. A non-leaf element e contains the S sort orders of the data points.

2. The incremental algorithm probes $\mathcal{I}$ until it reaches an element e in $\mathcal{C}$ that has data points contained in $\mathcal{Q}$, and calls Partition over e.

3. The Partition function simply returns its input **D** if it satisfies the *stopping condition*, which is $\mathcal{Q} \cap e = \emptyset$ *or* $\lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil = \lceil \frac{|e|}{N} \rceil$, where e is the current element of $\mathcal{C}$ that has the **D** partition.

4. If a Partition call returns from its line 2 (i.e., already smallest partition at its level), we then call BulkLoadChunk over it (the same as line 6 of BulkLoadChunk).

5. The *cost* function in line 4 of BestBinarySplit is revised as stated in Section 3.2.2.

In (3) above, the stopping condition of binary partition over an element e in the current contour is either $\mathcal{Q} \cap e = \emptyset$ indicating that e is irrelevant to $\mathcal{Q}$, or $\lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil = \lceil \frac{|e|}{N} \rceil$ indicating that almost all the data points in e are in $\mathcal{Q}$. A special case of the latter stop condition is when e is a single leaf node that has data points in $\mathcal{Q}$, although in general it may stop at an element larger than a leaf without splitting it. In (4) above, the top-down recursive algorithm proceeds to the next lower level of the tree.

As part of this top-down probing process, the qualified data points are found and returned. Note that we can start processing the first query when the index only has a root node. As more queries come, the index grows incrementally and the node splits are optimized for the query usage. In our case, it is optimal for the queried embedding vectors (e.g., $\mathbf{h} + \mathbf{r}$) in $\mathbb{S}_2$. As shown in our experiments in Section 4, only a very small fraction of the splits are performed compared to BulkLoadChunk, since the space of queried embedding vectors is much smaller than that of all data points. Thus, the amortized cost of incremental index building is much smaller than bulk loading.

Our main indexing algorithm makes a greedy choice to select a locally optimal cost for each split, as the original bulk loading algorithm does. However, we observe that, since we are now only incrementally build the index for each query, we may afford to explore more than one single split choice. We will iterate over a small number (e.g., $k = 2$ to 4) of split choices, with the goal of getting a good global index tree. Furthermore, we will use A^* style aggressive pruning to cut down the search space for a query.

The key idea is to use a priority queue $\mathbb{Q}$ (a heap) to keep track of "active" contours as *change candidates*. We do not adopt a change candidate until it completely finishes its splits for the current query and is determined to be the best plan based on A^* pruning. The sort order of the priority queue is the two-component cost of a contour. The minimum cost contour (i.e., change candidate) is popped out from $\mathbb{Q}$, and is expanded (with the top-k choices for the next split). The algorithm is shown in Top-kSplitsIndexBuild.

In lines 1-3, we create the priority queue $\mathbb{Q}$ for the very first query. Initially, the index only has the root node, and this contour is added into $\mathbb{Q}$ with the two-component cost (the overlap cost is 0). Line 5 pops out the head of the queue which has the least cost. The stopping condition of processing an element e in line 7 is the same as that in IncrementalIndexBuild.

Lines 6 and 8 are based on a certain traversal order such as depth-first-search. If all elements of the top change candidate (contour) satisfy the stopping condition, and hence the next element is null (lines 9-12), this candidate must be the best among all. Otherwise, in lines 13-19, we will continue to process (split) this change candidate and expand it with the top-k splits. We add these k new candidates into $\mathbb{Q}$ with updated costs as their weights.

Algorithm 6: Top-kSplitsIndexBuild ($\mathcal{I}$, $\mathcal{Q}$)

```
Input: $\mathcal{I}$: current index; $\mathcal{Q}$: query region
Output: revised index
1  if $\mathbb{Q}$ does not exist yet then
2      create a priority queue $\mathbb{Q}$
3      add into $\mathbb{Q}$ the initial contour $\mathcal{C}$ with only the root node of $\mathcal{I}$ and cost
         $(\sum_{e \in \mathcal{C}} \lceil \frac{\mathcal{Q} \cap e}{N} \rceil, 0)$ as weight
4  while $\mathbb{Q}$ is not empty do
5      $\mathcal{C} \leftarrow$ pop head of $\mathbb{Q}$
6      $e \leftarrow$ first element of $\mathcal{C}$ whose MBR overlaps $\mathbb{Q}$
7      while $\mathcal{Q} \cap e = \emptyset$ or $\lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil = \lceil \frac{|e|}{N} \rceil$ do
           //stopping condition
8          $e \leftarrow$ next element of $\mathcal{C}$ whose MBR overlaps $\mathbb{Q}$
9          if $e = null$ then
10             break
11     if $e = null$ then
12         return the index with $\mathcal{C}$ //all $e \in \mathcal{C}$ are exhausted
13     process $e$ as IncrementalIndexBuild does, except:
14     in BestBinarySplit, we get top-$k$ best splits
15     for each of the $k$ splits do
16         $\mathcal{C}' \leftarrow \mathcal{C}+$ the split
17         $c_{\mathcal{Q}} \leftarrow \mathcal{C}.c_{\mathcal{Q}} - \lceil \frac{|\mathcal{Q} \cap e|}{N} \rceil + \lceil \frac{|\mathcal{Q} \cap e'|}{N} \rceil + \lceil \frac{|\mathcal{Q} \cap e''|}{N} \rceil$ //$e$ is split into $e'$ and
             $e''$
18         $c_{\mathcal{O}} \leftarrow \mathcal{C}.c_{\mathcal{O}} + \beta^h \cdot \frac{\|O\|}{min(\|L\|, \|H\|)}$
19         add $\mathcal{C}'$ into $\mathbb{Q}$ with cost ($c_{\mathcal{Q}}$, $c_{\mathcal{O}}$) as weight
```

3.2.3 Algorithms to Answer Queries

Let us now proceed to discussing the algorithms to answer queries on a virtual knowledge graph, given the indexing algorithms in Section 3.2.2.

We first consider queries that ask for top-k tail entities, given a head entity h and a relationship r (note that answering queries for top-k head entities given t and r is analogous and is omitted). The basic idea of our algorithm is to iteratively refine (reduce) the query rectangle region, until the nearest k data points to **h+r** in the original embedding space $\mathbb{S}_1$

are identified, based on the accuracy guarantees given in Section 3.2.1. The algorithm is shown in FindTop-kEntities.

```
Algorithm 7: FindTop-kEntities ($(\mathcal{I}, h, r)$)
   Input: $\mathcal{I}$: current index; $h$: head entity; $r$: relationship
   Output: top-$k$ entities most likely to have relationship $r$ with $h$
1  $\mathbf{q} \leftarrow \mathbf{h{+}r}$
2  probe $\mathcal{I}$ for smallest element of its contour $\mathcal{C}$ that contains $\mathbf{q}$, and get $k$ data points
     $N_q$
3  $r_q \leftarrow r_k^*(N_q) \cdot (1 + \varepsilon)$
4  $\mathcal{Q} \leftarrow$ region of $B(\mathbf{q}, r_q)$
5  while data points in $\mathcal{Q}$ have not been all examined do
6      $N_q \leftarrow$top-$k$ data points closest to $\mathbf{q}$ so far
7      $r_q \leftarrow r_k^*(N_q) \cdot (1 + \varepsilon)$
8      $\mathcal{Q} \leftarrow$ region of $B(\mathbf{q}, r_q)$
9  return $N_q$
```

In line 2, we probe the current index and locate a smallest element of its contour that contains **q** and randomly get k data points from it—let this set be N_q. In line 3, $r_k^*(N_q)$ denotes the kth smallest distance to **q** after mapping the data points in N_q to the original embedding space $\mathbb{S}_1$. Recall that our ultimate goal is to find closest entities in $\mathbb{S}_1$. The ε in line 3 is based on the accuracy guarantees in Theorem 6, and will be discussed in our next two theorems. The r_q in line 7 is further used in line 8 for the minimum bounding box that contains the ball with radius r_q around **q** in space $\mathbb{S}_2$. This is used as the next iteration query region $\mathcal{Q}$. As we only keep the k closest points at any time, $\mathcal{Q}$ will not grow bigger (but will most likely get smaller), until we have examined all the points in it.

Now we analyze our top-k entity query processing algorithm. Theorem 7 below provides data-dependent accuracy guarantees, while Theorem 8 addresses the performance of the algorithm.

Theorem 7. *With probability at least* $\prod_{i=1}^{k}\left[1-\frac{m_i^\alpha}{e^{\alpha(m_i^2-1)/2}}\right]$, FindTop-kEntities *does not miss any true top-*k *entities, where* $m_i = \frac{r_k^*}{r_i^*}(1+\varepsilon)$*, while the expected number of missing entities compared to the ground-truth top-*k *entities is* $\sum_{i=1}^{k}\frac{m_i^\alpha}{e^{\alpha(m_i^2-1)/2}}$.

Proof. We first note that: $\mathbf{Pr}[\text{all top-}k\text{ entities are inside}] = \prod_{i=1}^{k}\mathbf{Pr}[\text{rank } i \text{ entity is inside}] \geq \prod_{i=1}^{k}\left[1-\delta(\frac{r_k^*}{r_i^*}(1+\varepsilon))\right]$, where $\delta(\frac{r_k^*}{r_i^*}(1+\varepsilon))$ is the probability upper bound that a rank i entity (in $\mathbb{S}_1$) ends up having a distance greater than $r_k^*(1+\varepsilon)$ in $\mathbb{S}_2$, and hence is missed by our algorithm. Expressing this ratio in the form of Equation (1) in Theorem 6, we have $\frac{r_k^*}{r_i^*}(1+\varepsilon)) = \sqrt{1+\varepsilon'}$ (where ε' corresponds to ε in Theorem 6). Hence, $\varepsilon' = m_i^2 - 1$, where m_i is as defined in the theorem statement, and we can further derive the probability lower bound and the expected number of missing entities as in the theorem. □

Theorem 8. *The expected number of data points in the final query region* Q *in* FindTop-kEntities *is* $|B^*(\boldsymbol{q}, r_k^*(1+\varepsilon))|$*, i.e., the number of data points in the ball with center* $\boldsymbol{q}$ *and radius* $r_k^*(1+\varepsilon)$ *in the embedding space* $\mathbb{S}_1$ *(*B^* *denotes the region in* $\mathbb{S}_1$*). In particular, the probability that a data point with distance at least* $r_k^* \cdot \frac{1+\varepsilon}{1-\varepsilon'}$ *from* $\boldsymbol{q}$ *in* $\mathbb{S}_1$ *may get into* Q *is no more than* $(1-\varepsilon')^\alpha \cdot e^{\alpha(\varepsilon' - \frac{\varepsilon'^2}{2})}$.

Proof. First, the proof of Theorem 6 shows that the data-point transform into $\mathbb{S}_2$ is unbiased, preserving the expected Euclidean distance. Thus, the number of data points search in the algorithm in $\mathbb{S}_2$, $|B(\mathbf{q}, r_k^*(1+\varepsilon))|$, has the same expectation as the number of points in the same size/center region in the embedding space $\mathbb{S}_1$. For the second part of the theorem, we let $1-\varepsilon' = \sqrt{1-\varepsilon''}$, where ε'' corresponds to the ε in Equation (2) of Theorem 6, from which we can derive the probability upper bound stated in the theorem. □

Consider the following queries: What is the total number of restaurants that Amy may like? What is the average distance of the restaurants that Amy likes? These queries involve statistical aggregation over the virtual knowledge graph.

The relevant entities are within a ball with radius r_τ around the query center point such as **h+r**. The ball corresponds to a probability threshold of p_τ (e.g., a small value 0.05). To decide the probabilities, we let the entity closest to the query center point have probability 1 for the relationship, and other entities' probabilities are inversely proportional to their distances to the query center point. In general, for each data point in the ball, we may need to access its record with attribute information for aggregation and/or for evaluating the predicates. When there are too many such data points, we may use a sample of them to estimate the query result.

Note that in this ball, each data point only has a certain probability to be relevant (in the relationship with the head entity h), with probabilities in decreasing order from the center to the sphere of the ball. Without knowledge of the distribution of a relevant attribute in these data points, our accessed sample is the a points closest to the center (i.e., with top-a probabilities) among a total of b points in the ball.

COUNT, SUM, and AVG Queries. The estimations of COUNT, SUM, and AVG query results are similar. Let us begin with a SUM query. The expectation of a SUM query result is:

$$\mathbf{E}[s] = \frac{\sum_{i=1}^{a} v_i \cdot p_i}{\sum_{i=1}^{a} p_i / \sum_{i=1}^{b} p_i} \tag{8}$$

where a is the number of accessed data points out of a total of b data points in the ball, $p_1 \geq p_2 \geq \cdots \geq p_a \geq \cdots \geq p_b$ are the data point probabilities, and v_i ($1 \leq i \leq a$) are the retrieved attribute values to sum up. Note that we know the number of entities in each element of an index contour, and hence can estimate the $b - a$ probabilities (based on the average distance of an element to a query point). The numerator in Equation (8) is the expected sum of the attribute value in the retrieved sample. This needs to be scaled up by a factor indicated in the denominator of Equation (8), the ratio between the cardinality of the sample and the cardinality of all data points.

Note that we can maintain minimum statistics on $|v_m|$ at R-tree nodes to facilitate accuracy estimates. Alternatively, we may estimate $|v_m|$ based on the known sample values $|v_i|$ ($i \leq a$), without relying on any domain knowledge of the attribute. This is the same as how we estimate the expected MAX value discussed shortly.

MAX and MIN Queries. Such queries select the MAX or MIN value of an attribute among the b data points in the ball centered at the query point such as **h+r**. As before, we may access a sample of a closest data points to estimate the result. We only discuss MAX queries; the treatment for MIN queries is analogous. Again, we estimate the expected value, and then bound the probability that the true MAX is far away from this expectation.

First, let us estimate the expectation of MAX of the a accessed data points. Without loss of generality, we rearrange the index of the a data points so they are in non-increasing value order $\{(u_i, p_i)\}$, where $u_1 \geq \cdots \geq u_a$. Then the expectation of sample MAX is:

$$\mathbf{E}[\mathbf{M}_S] = u_1 p_1 + u_2(1 - p_1)p_2 + \cdots u_a(1 - p_1)...(1 - p_{a-1})p_a$$

Next, given an n-value sample chosen uniformly at random from a range $[0, m]$, we can estimate the maximum value m from the sample as $(1 + \frac{1}{n})m_s$ where m_s is the sample maximum value [69], which leads to the following result for expected MAX based on the

$\mathbf{E}[\mathbf{M}_S]$ above:

$$\mathbf{E}[\mathbf{M}] = \left(\mathbf{E}[\mathbf{M}_S] - \min_{1 \le i \le a} v_i\right)\left(1 + \frac{1}{\sum_{i=1}^{a} p_a}\right) + \min_{1 \le i \le a} v_i \tag{9}$$

We can then use martingale theory to bound the probability that the ground truth MAX result is far from the value in Equation (9). We omit the details–the main idea is that from data point $i-1$ to point i, the change to the expected MAX value $Y_i - Y_{i-1}$ should be bounded in a small range $[B_i, B_i + d_i]$, where $d_i = max(0, [v_i - \mathbf{E}[\mathbf{M}_S]] \cdot \left(1 + \frac{1}{\sum_{i=1}^{a} p_a}\right))$ for $i \le a$ and $d_i = \frac{\mathbf{E}[\mathbf{M}_S]}{\sum_{i=1}^{a} p_a}$ for $i > a$.

3.3 Event Timing Prediction and Critical Event Monitoring

In this section, we present our approach for event matching and complex event monitoring in data streams. First, let us discuss the problem of event timing prediction on multi-attribute data streams—whether a target event will happen soon, or whether an event will happen much later, using limited (and recent) training data to promptly build a model and predict.

3.3.1 Event Timing Prediction

We first describe the core of our approach, which is to efficiently build a event-timing knowledge graph from one pass of the training data, and to efficiently perform embedding over this graph based on the new notions of active sets, ephemeral nodes, and attentions on ephemeral edges.

The basic idea is that we represent a number of key relevant events (with respect to the given target events) as vertices V in a knowledge graph $\mathcal{G}$. The set of edges E in $\mathcal{G}$ indicates *timing* relationships. A directed edge (u, v) from event u to event v may have one of the two types of relationships: r_1 indicating that event v happens soon after event u, and r_2 indicating that event v happens long after event u.

On top of the basic graph elements described above, we define the notion of an activation graph formally below.

Definition 4. ***(Activation Graph)****. An activation graph $\mathcal{G}_S$ of a data stream $\mathcal{S}$ is a dynamic graph where each vertex is an event. $\mathcal{G}_S$ has states: at any moment, there is a subset of vertices V_a that are active, called the active set, which are the events that are true for the current tuple (r_i, t_i) in $\mathcal{S}$. The lifetime of V_a is $[t_i, t_{i+1})$, i.e., before the next tuple arrives. In addition, every V_a is associated with an ephemeral node v_a that has the same lifetime as V_a. The edges of $\mathcal{G}_S$ are of two categories, which are both ephemeral and incident to an ephemeral node at one end. The first category of edge is from each vertex in V_a to v_a, while the second category is from v_a to an event vertex u with a relationship type either r_1 or r_2, where r_1 indicates that event u happens soon after V_a, and r_2 indicates that event u happens much later than V_a.*

At time t when a tuple (r, t) of the stream arrives, it *activates* a set of event nodes V_t in $\mathcal{G}$. This is the major difference between our graph model and previous graphs or dynamic graphs. Our goal is to predict whether each of the target event nodes will be active soon or much later.

Example 6. *Figure 7 shows an example, where each solid vertex is an event node while the two hollow vertices are ephemeral nodes. The four event vertices inside the green solid* *circle are the active set V_a at some time instant t; V_a is associated with an ephemeral node v_a. Similarly, the three vertices inside the red dashed circle are another active set V_a' at a different time, with the corresponding ephemeral node v_a'. Note that the two active sets may have vertices in common (one vertex in common here). All edges are incident to ephemeral nodes. Take v_a as an example. There is an edge of relationship type r_1 from v_a to event vertex v_e indicating that event v_e happens soon after the four events in V_a. In addition, there are four edges from each event node of V_a to v_a. The relationship type of such edges is based on the two events in the time precedence relationship. For instance, the relationship type of the edge from event v_i to ephemeral node v_a is r_{ie} in Figure 7, as the other end is event v_e. Likewise, the edge from v_j to v_a is of relationship type r_{je}.*

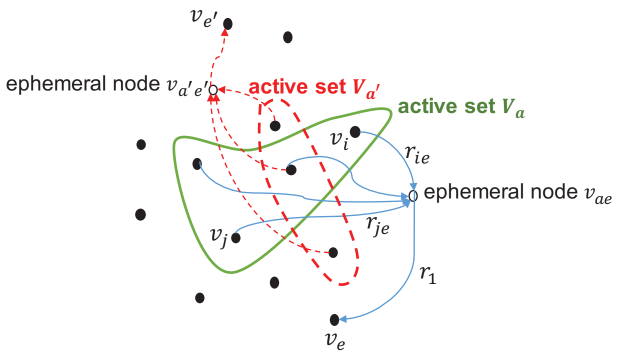


Figure 7: An activation graph

The basic idea of our solution is to perform customized knowledge graph embedding over $\mathcal{G}_\mathcal{S}$. We perform embedding over the nodes and dynamic edges with *attention* [70]. The intuition is that different events in an active set may have different *weights* (or importance) in predicting the subsequent target event. Such weights are called the *attention* parameters, and can be learned during back-propagation, such as the *stochastic gradient descent* [71] of the training process. An ephemeral node serves as a *intermediate* node to aggregate the active events in the corresponding active set.

We now introduce the graph building and embedding algorithms. Suppose we have identified n events $e_1, e_2, ..., e_n$ to follow, where the first k $(k \leq n)$ events $e_1, ..., e_k$ are the target events that we predict. The main idea is to parse the training data window of stream $\mathcal{S}$ only once and construct $\mathcal{G}_\mathcal{S}$. We show that the algorithm only needs to keep a constant portion of $\mathcal{S}$ in memory while parsing it. The r_1 and r_2 relationship-type edges to target events $e_1, ..., e_k$ are added to $\mathcal{G}_\mathcal{S}$ first—while the active set may contain any events in $e_1, ..., e_n$. We say that these edges are *target edges*.

Knowledge graph embedding is performed over such edges first. If the algorithm is performed in real time and there is still time remaining, the real-time embedding is performed over non-target edges. The intuition is that target edges are directly related to predicting target events, and are hence more relevant for learning the embedding.

Example 7. *Let us go back to the example in Figure 7. Suppose v_e is one of the k target events, then all the five edges incident to the ephemeral node v_a are target edges (e.g., the ones shown of relationship types r_{ie}, r_{je}, and r_1 in Figure 7). Our embedding will be performed over such edges first under time constraints. Now suppose v'_e is not a target event. Then the four red dashed edges incident to the ephemeral node v'_a in Figure 7 are non-target edges, which have lower priority in the online embedding.*

In order to get non-target edges on demand for embedding after the one pass over the stream, we perform *reservoir sampling* [72] over the r_1 and r_2 non-target edges, respectively, to get a uniformly random pool of such edges for selection as needed.

Figure 8 illustrates the flow of our algorithm that builds the graph. Our goal is to

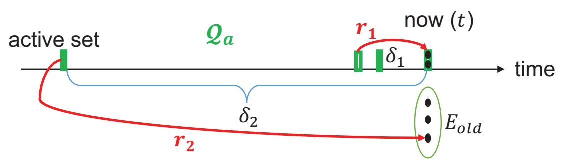


Figure 8: Illustrating the flow of BuildEventOrderGraph.

create r_1 and r_2 edges from an active set (tuple) to an event, shown as the red edges in the figure. We maintain a queue of active sets Q_a from "now" (time t, the current tuple) back in time up to length δ_2. In addition, we also maintain a set of events E_{old} that has not appeared for at least δ_2 time, shown as the green oval in Figure 8. Then, as soon as an active set in Q_a is δ_2 old, we remove it from Q_a and create r_2 edges, one to each event in E_{old}. Furthermore, when the current tuple at time t (now) joins Q_a, we also create r_1 edges from each active set within δ_1 back in time to each event in the current tuple.

We present the graph building algorithm in BuildEventOrderGraph.

Line 1 of the algorithm initializes, for each of the n events, the last time that it occurs. We will use this information to determine when an event is too old (i.e., it has not appeared in the past δ_2 window), and we will add an r_2 edge from an old active set (δ_2 earlier) to this old event. Line 2 initializes a set that stores such old events. The main loop in lines 3-16 does one pass over the stream training data.

In line 4, we get the set of active events that are true in the current tuple. Then lines 5-7 update each of these active events' last occurrence time and remove it from the old set if it is there. In line 8, we add the active set to a queue Q_a. We trim the queue when an event is over δ_2 old. Q_a is needed for adding an r_2 edge from an old active set (δ_2 earlier) to an old event (that has not appeared for long).

Lines 9-11 check if an event that does not appear in the current tuple is now old

```
Algorithm 8:
BuildEventOrderGraph ($\mathcal{S}, e_1, ..., e_k, ..., e_n$)
  Input: $\mathcal{S}$: data stream training data;
         $e_1, ..., e_k, ..., e_n$: relevant events, first $k$ are target events
  Output: event order dynamic graph $\mathcal{G}_\mathcal{S}$
1 initialize $t_1, ..., t_n$ as last occurrence times of $e_1, ..., e_n$
2 $E_{old} \leftarrow \emptyset$ //set of events that are at least $\delta_2$ old
3 for each tuple $s[t] \in$ current window of $\mathcal{S}$ do
4     $E_a[t] \leftarrow s[t] \cap \{e_1, ..., e_n\}$ //get active events
5     for each $e_i \in E_a[t]$ do
6         $t_i \leftarrow t$ //update last-occurrence time
7         $E_{old} \leftarrow E_{old} \setminus \{e_i\}$//remove from old tuple set
8     $\mathcal{Q}_a \leftarrow \mathcal{Q}_a \cup \{E_a[t]\}$ //add $E_a[t]$ into $\mathcal{Q}_a$
9     for each $e_i \in \{e_1, ..., e_n\} \setminus E_a[t]$ do
          //check if it is now old
10        if $e_i \notin E_{old}$ and $t - t_i > \delta_2$ then
11            $E_{old} \leftarrow E_{old} \cup \{e_i\}$
12    for each $E_a[t'] \in \mathcal{Q}_a$ s.t. $t - t' > \delta_2$ do
13        $\mathcal{Q}_a \leftarrow \mathcal{Q}_a \setminus \{E_a[t']\}$
14        AddR2Edges($\mathcal{G}_\mathcal{S}, E_a[t'], E_{old}$)
15    for each $E_a[t'] \in \mathcal{Q}_a$ s.t. $0 < t - t' < \delta_1$ do
16        AddR1Edges($\mathcal{G}_\mathcal{S}, E_a[t'], E_a[t]$)
17 return $\mathcal{G}_\mathcal{S}$
```

enough to be put in the E_{old} set. Lines 12-14 trim an active set from $\mathcal{Q}_a$ if it is over δ_2 old, and add the corresponding r_2 edges to each old event in E_{old}, as discussed earlier. On the other hand, lines 15-16 add the r_1 edges from each recent active set (within δ_1) to each event in the current tuple.

We next look at the AddR1Edges algorithm (AddR2Edges is similar). Each r_1 edge essentially records a relationship from an active set E_a to an event node e_i. Note that the embedding algorithm presented later will temporally add the ephemeral node v_a between E_a and e_i, as well as the ephemeral edges r_{ie} from each event node v_i in E_a to v_a (as shown in Figure 7). But for now, if the to-event is a target event (lines 2-3), we only add the triple to the set $tarE$ (target edges); others in lines 5-10 we perform the reservoir sampling [72] so the triple will be put in a fixed-size (cap) buffer $ntarR1$ (non-target r_1 edges) uniformly at random. Thus, essentially the graph building algorithm only creates *hyperedges* from an active set (multiple nodes) to a single event node.

Algorithm 9: AddR1Edges ($\mathcal{G}_\mathcal{S}, E_a, E_{to}$)

```
   Input: $\mathcal{G}_\mathcal{S}$: dynamic event order graph stream;
          $E_a$: the active set at "from" end of $r_1$ edge;
          $E_{to}$: set of event nodes at "to" end of $r_1$ edge
   Output: updated $\mathcal{G}_\mathcal{S}$
1  for each $e_i \in E_{to}$ do
2      if $i \leq k$ then
           //to a target event
3          $\mathcal{G}_\mathcal{S}.tarE \leftarrow \mathcal{G}_\mathcal{S}.tarE \cup \{(E_a, e_i, r_1)\}$ //target edges
4      else
5          if $count \leq cap$ then
               //count is total non-target $r_1$
6              $\mathcal{G}_\mathcal{S}.ntarR1[count] \leftarrow (E_a, e_i, r_1)$
7          else
8              $j \leftarrow random(1, count)$
9              if $j \leq cap$ then
10                 $\mathcal{G}_\mathcal{S}.ntarR1[j] \leftarrow (E_a, e_i, r_1)$
11 return $\mathcal{G}_\mathcal{S}$
```

Having built the event-timing knowledge graph, we now use the data stream training data to get graph embedding vectors. Due to the requirement of data stream algorithms, we extend the efficient TransE [73] knowledge graph embedding algorithm and add ephemeral nodes (with derived embedding vectors), as well as the attention parameters. We present the algorithm in StateBasedEmbedding.

In the loop of lines 1-8, we first perform the iterative training over target edges, which, as discussed earlier, have a higher priority as they directly lead to the target nodes to be predicted. Lines 9-11 are essentially the same, but work on all edges. Line 3 samples a

Algorithm 10: StateBasedEmbedding ($\mathcal{G}_\mathcal{S}$)

```
Input: $\mathcal{G}_\mathcal{S}$: dynamic event order graph stream
Output: embedding vectors for nodes and relationship types of $\mathcal{G}_\mathcal{S}$
1  loop
2      $pool \leftarrow \mathcal{G}_\mathcal{S}.tarE$ //first only use target edges
3      $S_{batch} \leftarrow sample(pool, b)$ //draw a mini-batch of size $b$
4      for $(E_a, e, r) \in S_{batch}$ do
           //loop over the original $S_{batch}$
5          $(E'_a, e', r) \leftarrow sample(S'_{(E_a,e,r)})$ //sample a corrupted triplet
6          $S_{batch} \leftarrow S_{batch} \cup \{(E'_a, e', r)\}$
7      let $f = \sum_{(E_a,e,r) \in S_{batch}} \sigma((E_a, e, r)) \cdot \sum_{i=1}^{d} [\mathbf{e}_i - \mathbf{r}_i - \frac{1}{n} \sum_{x \in E_a} \mathbf{a}_{xe}(\mathbf{x}_i + \mathbf{r}_{\mathbf{xe}_j})]^2$
8      update embeddings and attention $\mathbf{a}$ w.r.t. gradient of $f$
9  while time remains do
10     $pool \leftarrow \mathcal{G}_\mathcal{S}.tarE \cup \mathcal{G}_\mathcal{S}.ntarR1 \cup \mathcal{G}_\mathcal{S}.ntarR2$
11     do lines 3-8
```

mini-batch of triples (for mini-batch stochastic gradient descent). Similar to TransE (and most other translational distance based embedding), line 5 does negative sampling [74] by corrupting either the head or tail of a positive triple in the sample set, and line 6 includes the negative sample in the batch too.

Line 7 has the key loss function of the embedding, where the $\sigma(\cdot)$ function is the sign function that is $+1$ for a positive sample and -1 for a negative sample, and d is the dimensionality of the embedding vectors. This is similar to the L_2-distance version of TransE [73] minimizing $\|$**t - r - h**$\|_2$, except that **t** is the event node **e**, and the head **h** is replaced by the embedding of the ephemeral node v_a in Figure 7. In turn, v_a is the tails of the triples from the ephemeral edges such as r_{ie} in Figure 7. Again using the constraint **t = h + r** we derive the embedding of v_a from **x + r**$_{\mathbf{xe}}$, and does a weighted sum of them from each event node of the active set, where the weight is the attention parameter $\mathbf{a}_{xe}$.

In line 8, the algorithm does stochastic gradient descent optimization over each parameter value in each embedding vectors, including those of the event nodes, r_1, r_2, and all ephemeral edges' relationship types r_{ie} (as in Figure 7), as well as all the attention parameters $\mathbf{a}_{xe}$. Like TransE, we normalize each embedding vector to length 1, and normalize the attention parameters such that $\sum_{x \in E_a} \mathbf{a}_{xe} = n$ (where n is the total number of events).

We organize the training algorithm iterations into *epochs*, where each epoch has the number of random samples equal to the total number of training records used. Our experiments show that the convergence of the loss function value is quite fast, the details of which are in Section 4.

Making Prediction. Once we have all the embedding vectors and attention parameter values, making predictions is again based on the same loss function as in line 7, except that we do not need to use a negative sample, and there is only one triple in the sample set S_{batch}. That is, we use the loss function $f = \sum_{i=1}^{d}[\mathbf{e}_i - \mathbf{r}_i - \frac{1}{n}\sum_{x \in E_a} \mathbf{a}_{xe}(\mathbf{x}_i + \mathbf{r}_{\mathbf{xe}_i})]^2$. For example, if the current record has active set E_a, and we want to predict whether a target event e will be more likely to occur soon or to occur much later, we use the loss function above, and the relationship that results in a smaller loss function value wins.

Now, we discuss how to identify "significant" events that help predict target events. The basic idea is that we aim to find characteristic events that tend to precede target events, but not so much for other events. We resort to a metric in information retrieval called *tf-idf* (short for term frequency-inverse document frequency) [75].

For our problem, each event candidate is analogous to the "term", while context (tuples prior to the target event) before a target event occurs is analogous to the document we are interested in, and the general context of the whole stream is analogous to the text corpus.

Then there is the computation issue—how do we efficiently find the top events with the highest tf-idf with respect to each target? We first partition the set of values or value range of each attribute into *basic events*. However, just the basic events themselves may not be discriminative enough, as each basic event alone may be very common in the general stream context. Thus, we also explore the combination of two or more basic events together as a *composite event*. A composite event will have lower (or the same) tf (term frequency) in the context of target events than the individual basic events within it, but it will also have higher (or the same) idf (inverse document frequency).

Therefore, we aim to find the top-n events (either basic or composite) that have the highest tf-idf for each target event. While the number of basic events is manageable (typically a constant number of partitions times the number of attributes), there are an exponential number of composite events—it is computationally challenging to compute the tf-idf of all of them by checking the tuples prior to each target event and those prior to each tuple in the stream.

We devise a novel algorithm by using efficient bitmap operations, sampling, and A*-style pruning and bounding [76]. The main ideas are as follows. We build a bitmap $\mathcal{B}_1$ for each basic event e where each bit of $\mathcal{B}_1$ corresponds to one occurrence of the target event (say, at time t), and the bit is 1 if e occurs within time $[t - \delta_1, t)$, i.e., within δ_1 interval prior to the target event, and is 0 otherwise. This will be used to compute the tf part of tf-idf.

In the same vein, we build a bitmap $\mathcal{B}_2$ for each basic event e for its occurrence in the general stream context (i.e., every tuple). However, the problem is that there might be too many tuples in the stream, which makes $\mathcal{B}_2$ too large. Thus, we use random sampling which provides a provably accurate estimate of the idf part of tf-idf. As a matter of fact, we may do sampling for $\mathcal{B}_1$ too if the number of occurrences of the target event is too many. In the case of $\mathcal{B}_2$, we sample the stream tuples and each bit of $\mathcal{B}_2$ corresponds to a tuple that is chosen in the sample. Like $\mathcal{B}_1$, if the chosen tuple arrives at time t, the bit of $\mathcal{B}_2$ is 1 if e occurs within time $[t - \delta_1, t)$, and is 0 otherwise. The next idea is that we use A*-style aggressive pruning and bounding to efficiently search the space for top-n events in tf-idf. We next show the algorithm GetTopRelevantEvents.

In line 1, we assign an arbitrary but *fixed* order to all basic events. Lines 4-5 build the two bitmaps as discussed above. Lines 6-29 perform the A* search of top-n events with highest tf-idf. Specifically, the loop in lines 7-12 first add each basic event into the priority queue $\mathcal{Q}$. The weights set in lines 9 or 11 are used to order the items in $\mathcal{Q}$, with the root of $\mathcal{Q}$ (to be popped next) having the greatest w. Theorem 9 shows the reason for the w value, which is an upper bound of the tf-idf value of all events that can be derived from e_j.

Line 15 pops the root of $\mathcal{Q}$ each time for the event with the greatest w. In general, w is an upper bound of tf-idf. The *finalized* flag in line 16 marks that w is actually the exact tf-idf of the corresponding event e_i. Thus, if the condition in line 16 is true, event e_i's tf-idf is higher than every other event's tf-idf upper bound—which means that e_i must have the highest tf-idf among all the events not already in E_r. Then line 17 adds e_i into the set to be returned.

Lines 19-27 expand the current event e_i by one more basic event e_j, where e_j is after all the basic events in e_i according to the order in line 1. Lines 23-26 set the upper bound value w in the same way as lines 8-11. Line 28 updates the w of the already popped out e_i to its exact tf-idf value, and marks it as *finalized* before putting it back into $\mathcal{Q}$. Theorem 9 shows the correctness of the algorithm.

```
Algorithm 11: GetTopRelevantEvents ($\mathcal{S}, e_x, n$)
  Input: $\mathcal{S}$: data stream;
         $e_x$: a target event;
         $n$: number of events to retrieve
  Output: top $n$ events with the highest tf-idf
1  $E_b \leftarrow a$ fixed order of basic events in a tuple of $\mathcal{S}$
2  while one pass of $\mathcal{S}$ do
3      for $e_j \in E_b$ do
4          build bitmap $\mathcal{B}_1(e_j)$, bit $i$ indicates if $e_j$ occurs in $[t - \delta_1, t)$, where $t$ is the
             $i-$th occurrence time of $e_x$
5          build bitmap $\mathcal{B}_2(e_j)$, bit $i$ indicates if $e_j$ occurs in $[t - \delta_1, t)$, where $t$ is
             occurrence time of $i-$th tuple in sample of $\mathcal{S}$
6  initialize priority queue $\mathcal{Q}$ of events
7  for $e_j \in E_b$ do
8      if $|\mathcal{B}_1(e_j)| < \frac{|\mathcal{S}|}{e}$ then
9          $w \leftarrow |\mathcal{B}_1(e_j)| \cdot \log \frac{|\mathcal{S}|}{|\mathcal{B}_1(e_j)|}$
10     else
11         $w \leftarrow \frac{|\mathcal{S}|}{e} \cdot \log e$
12     add $e_j$ into $\mathcal{Q}$ with weight $w$
13 $E_r \leftarrow \emptyset$ //result to be returned
14 while $|E_r| < n$ do
15     pop $(e_i, w)$ from $\mathcal{Q}$
16     if $e_i$ is marked finalized then
17         $E_r \leftarrow E_r \cup e_i$
18         continue
19     for each $e_j \in E_b$ after all basic events in $e_i$ do
20         $e'_i \leftarrow e_i \cap e_j$
21         if $|\mathcal{B}_1(e'_i)| = 0$ then
22             continue
23         if $|\mathcal{B}_1(e'_i)| < \frac{|\mathcal{S}|}{e}$ then
24             $w \leftarrow |\mathcal{B}_1(e'_i)| \cdot \log \frac{|\mathcal{S}|}{|\mathcal{B}_1(e'_i)|}$
25         else
26             $w \leftarrow \frac{|\mathcal{S}|}{e} \cdot \log e$
27         add $e'_i$ into $\mathcal{Q}$ with weight $w$
28     $w \leftarrow |\mathcal{B}_1(e_i)| \cdot \log \frac{N}{|\mathcal{B}_2(e_i)|}$ //$N$ is number of bits in $\mathcal{B}_2(e_i)$
29     mark $e_i$ finalized and add $e_i$ into $\mathcal{Q}$ with weight $w$
30 return $E_r$
```

Theorem 9. *The* GetTopRelevantEvents *algorithm returns the correct top-n events with the highest tf-idf. In particular, the w bound calculated in lines 23-26 of the algorithm for event e'_i is an upper bound of the tf-idf of all the events that can be derived from e'_i by adding basic events into it.*

Proof. Let the tf value of an event e_i be c, i.e., e_i appears in c places where the target event appears. To ensure the correctness of the A* pruning in the algorithm, its w value must be an upper bound of the tf-idf of all events that can be obtained by extending e_i—we call such events the *descendants* of e_i. Let $1 \leq x \leq c$ be the tf of such a descendant (whose tf can only be smaller than or equal to e_i's). Then $f = x \log \frac{N}{x}$, where N is the total size of the stream, is an upper bound of the tf-idf of this descendant. To get an upper bound among all such descendants, we need to find the maximum value of $f = x \log \frac{N}{x}$ for an integer $1 \leq x \leq c$. By taking the derivative of f over x, we get that if $c < \frac{N}{e}$, the upper bound is just $c \log \frac{N}{c}$; otherwise the upper bound is $\frac{N}{e} \cdot \log e$, where e is the the base of the natural logarithm. This exactly corresponds to the w assignment in lines 23-26. $\square$

3.3.2 Critical Event Monitoring

In this section, we focus on our last problem which is critical event monitoring. We devise an approach based on the relational view of the multistream $\mathbb{S}$, and construct a dynamic knowledge graph $\mathcal{G}$. Then we perform representation learning over $\mathcal{G}$. After that, we learn a probabilistic state machine pattern for a given critical event, which in turn is used to monitor and foresee critical events.

We first describe how to obtain the dynamic knowledge graph $\mathcal{G}$ from the multistream $\mathbb{S}$. We convert a stream of tuples into several types of nodes and edges as described below.

Table 1: Notation summary.

Symbol	Meaning
$\mathbb{S}, \mathcal{G}$	multistream and the dynamic knowledge graph
r_a	relationship from key node to an event node of attribute a
S_{KE}	a set of key-event relationships (as r_a above)
σ_t	state-holder node for tuple t
σ^*	critical state node
r_e	relationship from event node e to a state-holder node
a_e	attention associated with event node e
r_t	followed-by relationship between two state-holder nodes
S_T	a set of followed-by relationships (r_t above)
κ	the number of states (clusters) in the pattern model
$\mathcal{M}, \boldsymbol{P}$	state-machine model and its transition matrix

For convenience, in Table 1, we summarize the notation for symbols used more than once in the paper. By convention, we use boldface letters (e.g., $\boldsymbol{r_t}$ and $\boldsymbol{\sigma}^*$) to represent the corresponding embedding vectors.

Entity Nodes. A *basic event* is a predicate over an attribute that holds true. Depending on the application, a particular value or a set/interval of values that an attribute may bear is such a predicate, and is treated as an *event entity node* in the dynamic knowledge graph $\mathcal{G}$. For example, Figure 9 illustrates a snippet of the dynamic knowledge graph that is generated from the Dutch hospital data stream. While the actual dataset has many more attributes, we only list three of them here for clarity: A_1 event name, A_2 diagnostic code, and A_3 age. Each attribute may have a number of discrete basic events, each of which is an entity node of $\mathcal{G}$, corresponding to a black solid node in Figure 9. For instance, two of the entity nodes from A_1 (event name) are "outpatient consultation" and "o2 saturation", two of the entity nodes from A_2 (diagnostic code) are M16 and 823, and A_3 (age) is discretized into three entity nodes “young”, “old”, and “middle”. The rationale behind the event entity nodes is that we perform embedding at this attribute-value level so that an aggregate embedding of a subgraph of them will be the tuple embedding, from which we eventually derive the states.

Key Nodes and Attribute Relationships. We next add an auxiliary node for each tuple, which we call the *key node*. For tuple t_i, let its key node be K_i (i.e., each tuple has its own key node). The key nodes are shown as the hollow nodes in Figure 9. Then for each tuple t_i, for each of its attribute A_j having event node e_j, we add an edge from key node K_i to event node e_j, shown as the dashed edges in Figure 9. We say that each of these edges has an *attribute relationship* r_a, i.e., there is a distinct relationship type r_a for each attribute a, from a tuple key to an event node of a. The reason for a key node is so that we can use a minimum number of binary relations (the r_a's above) to represent the original n-ary relation of the stream tuple, as shown in the theorem below, conducive to embedding.

Theorem 10. *The construction above, by using binary attribute relationships from key nodes to event nodes, preserves the original relationship among multiple event nodes of each tuple.*

Proof. The construction decomposes a c-ary relation among all attributes (i.e., event nodes) of the schema of the stream into c binary relations, where c is the number of attributes of the stream schema. Such a decomposition must be a *lossless-join decomposition* since all these binary relations share the key attribute of the tuple, which is the key of all the resulting relations [77]. Thus, the graph faithfully preserves the original relation information. $\square$

State-holder Nodes and Critical State Node. Later, we will create a number of

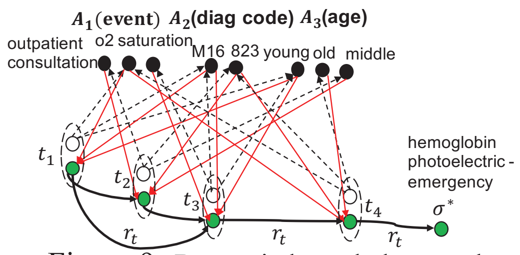


Figure 9: Dynamic knowledge graph

states, and each tuple belongs to one of the states. To that end, we create a *state-holder node* σ_t for each tuple t, illustrated as the green solid nodes in Figure 9. Moreover, from each event node e that tuple t has (as its attribute values), we create an edge from e to σ_t with relationship type r_e and an associated *attention* weight a_e [78]. These edges are shown as the red edges in Figure 9. Finally, there is a special node called the *critical state node* σ^*, which is the state of all critical tuples. In Figure 9, it corresponds to "hemoglobin photoelectric - emergency". A state-holder node essentially corresponds to a subgraph made of the event entity nodes of a tuple, and we will use it to hold the embedding of the tuple, aggregated from the event entity nodes.

Timing Relationships. Once we have the state-holder nodes for each tuple, we create timing relationship edges. If tuple t_1 precedes tuple t_2 by no more than time duration δ, then we create an edge with a "followed by" relationship type r_t from t_1's state-holder node σ_{t_1} to t_2's state-holder node σ_{t_2}. Note that the r_t relationship can also be count-based, i.e., an edge is created if t_1 is no more than δ tuples ahead of t_2. These edges are the solid bold ones in Figure 9.

A high level idea of our representation learning approach is that the latent features learned for each vertex and relationship type, together with the attention values of event-state edges, characterize the attribute correlations and the timing relationship between different tuple states. The basic idea of our learning algorithm is to treat key-attribute edges (i.e., attribute relationships) and timing edges as positive samples. Then we significantly extend TransE [73], a translation-based embedding, to also learn the attention weights of attribute-state edges. The state-holder nodes are special in that *they do not have their own embedding vectors* as other nodes do, and their vectors are derived from those of the event nodes through attention weights—i.e., such aggregate embedding is essentially for the sub-graph of event nodes that the state-holder node represents. First, we devise the loss function as follows:

$$\begin{aligned} l(\mathcal{G}) = &\sum_{(k,e,r_a)\in S_{KE}} [sgn((k,e,r_a)) \cdot \|\mathbf{k} + \mathbf{r_a} - \mathbf{e}\|^2] \\ &+ \sum_{(\sigma_1,\sigma_2,r_t)\in S_T} [sgn((\sigma_1,\sigma_2,r_t)) \cdot \|f(\sigma_1) + \mathbf{r_t} - f(\sigma_2)\|^2] \end{aligned} \tag{10}$$

where the function $f(\sigma)$ is defined as follows:

$$f(\sigma) = \begin{cases} \boldsymbol{\sigma^*}, & \sigma = \sigma^* \\ \sum_{e\in N(\sigma)} a_e(\mathbf{e} + \mathbf{r_e}), & \sigma \neq \sigma^* \end{cases} \tag{11}$$

The first sum in Equation 10 is over a sample set of key-event edges S_{KE} (of attribute relationships) that contains both positive and negative samples. Negative sampling [74] is a technique to obtain negative samples by corrupting either the head or tail of a positive triple in the sample set. For negative samples, the stochastic gradient descent will optimize in the opposite direction, which is why we have the sign function $sgn((k,e,r_a))$, which returns $+1$ if the triple is a positive sample, and -1 if it is a negative sample. The $\|\mathbf{k} + \mathbf{r_a} - \mathbf{e}\|^2$ is to optimize the triple constraint similar to TransE [73].

Similarly, the second sum is over a sample of timing edges S_T, including both positive and negative samples. $f(\sigma)$ basically returns either the embedding vector of the critical state node directly (first case), or the derived vector of the state-holder node, which is the weighted aggregate of the tail-ends of several event-state relationships, where the weights are the attentions to be learned. $N(\sigma)$ is the set of neighbor nodes of σ, i.e., the connected event nodes. Moreover, we normalize each vector to have length 1 (i.e., divided by its length), and normalize the attention values such that $\sum_{e \in N(\sigma)} a_e = 1$ for each σ (i.e., divided by the sum). We also ensure that $a_e \geq 0$.

We now present the algorithms for the representation learning. Let us start with the initialization algorithm, as shown in InitGraph, which basically creates the dynamic knowledge graph $\mathcal{G}$, and initializes the initial embedding vectors and attentions.

Algorithm 12: InitGraph ($\mathbb{S}$)

```
  Input: $\mathbb{S}$: data stream
  Output: dynamic graph $\mathcal{G}$ with initial data structures
1 for each attribute $a \in \mathbb{S}$ do
2     for each event $e \in \varepsilon(a)$ do
3         create an event node in $\mathcal{G}$ with initial embedding vector
4         attention $a_e \leftarrow \frac{1}{c}$ //$c$ is number of attributes
5         create event-state relationship $r_e$ with initial vector
6     create key-attribute relationship $r_a$ with initial vector
7 create timing relationship $r_t$ with initial vector
8 return $\mathcal{G}$, containing all data structures created
```

In line 2, $\varepsilon(a)$ represents the set of all events of attribute a. In line 3, each value of the initial embedding vector is a uniform random sample from (-1, 1), and then the vector is normalized to length 1.

Example 8. *Let us look at the example in Figure 9. Suppose at this moment, tuple t_3, which is a patient's treatment record, has just arrived. t_3 has three events, one for each* *attribute—"o2 saturation", M16, and "young". Line 3 will add three positive edges from the key node of t_3 to those event nodes, while line 4 will add the corresponding negative samples (corrupted versions). Likewise, lines 5-8 add the timing edges—the two bold edges $t_2 \rightarrow t_3$ and $t_1 \rightarrow t_3$. Line 11 samples a positive edge from S_T, and let's say it is $t_1 \rightarrow t_3$. Then line 12 will perform a stochastic gradient descent optimization over both the key-event edges and the timing edges surrounding tuples t_1 and t_3. As a result, the embedding vectors of the related event nodes "o2 saturation", M16, and "young", the vectors of the relevant relationship types, and the corresponding attentions are all updated.*

```
Algorithm 13: ContinuousRepresentationLearning ($\mathbb{S}$, $\mathcal{G}$, $w$)
  Input: $\mathbb{S}$: data stream;
         $\mathcal{G}$: current dynamic graph for $\mathbb{S}$;
         $w$: window size for dynamic embedding
  Output: updated $\mathcal{G}$ with current embedding vectors
1  for each tuple $u \in \mathbb{S}$ do
       //$u$ could be a critical tuple too
2      for each event $e$ in $u$ do
3          add $(k(u), e)$ information to $S_{KE}$, key-event set
4          add corrupted version $(k', e')$ to $S_{KE}$
5      let $u$'s substream be $s(u)$
6      for $v \in s(u)$ within time $\delta$ before $u$ do
7          add $(v, u)$ information to $S_T$, timing relation pairs
8          add corrupted version $(v', u')$ to $S_T$
9      truncate $S_{KE}$ and $S_T$ within window size $w$
10     while next tuple has not arrived do
11         sample a positive $(v, u)$ edge from $S_T$
12         update embeddings and attentions w.r.t. Eq. 10 over edges between $v$, $u$,
             and associated event/key nodes
```

We are now ready to present the main embedding algorithm in ContinuousRepresentationLearning. Note that the dynamic knowledge graph is implicit—our algorithm will take the data stream $\mathbb{S}$, and perform learning of embedding vectors of the events corresponding

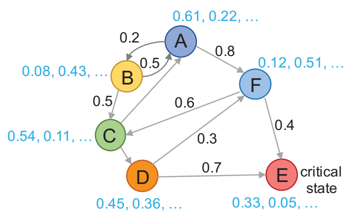


Figure 10: Probabilistic state machine (PSM) pattern

to the attribute values, as well as the derived nodes.

In line 2, each event e corresponds to an attribute value or interval of tuple u, while in line 3, $k(u)$ is the key node of tuple u. Line 4 is to perform negative sampling by changing one end (either head or tail) of a positive triple to a corrupted version—a randomly chosen value. Lines 5-8 are to add timing edges, as well as the corresponding positive and negative timing triples to S_T. This needs to be done within the substream (line 5).

We are now ready to learn the probabilistic state machine (PSM) pattern for the given critical event. In a nutshell, a PSM pattern is simply a number of *states* (each with some latent features based on embedding), one of which is the *critical state*, along with the *transition probabilities* among them. This is illustrated in Figure 10, which is a simple PSM pattern with 6 states.

The basic idea of the algorithm to learn a PSM pattern is that we first cluster the tuples in a stream window into a number of clusters, each of which will correspond to a *state* of the PSM pattern. There is also a special state, the *critical state*, corresponding to the critical event. Then the rest of the algorithm is mainly about estimating the probabilities of the "followed by" timing relationship between every pair of states, through the embedding vectors we have learned. The algorithm is shown in GenerateStateMachine.

Like before, line 3 of the algorithm is to calculate the embedding vector of the state-holder node using a weighted sum of the related event nodes, where a_e is the attention from

Algorithm 14: GenerateStateMachine (W, $\mathcal{G}$, κ)

```
Input: $W$: a finite window of stream tuples;
       $\mathcal{G}$: dynamic graph with learned embedding;
       $\kappa$: number of states
Output: a state machine as PFA
1  $D \leftarrow \emptyset$
2  for each tuple $u \in W$ do
3      $\mathbf{d} \leftarrow \sum_{e \in u} a_e(\mathbf{e} + \mathbf{r_e})$
4      $D \leftarrow D \cup \mathbf{d}$
5  cluster $D$ into $\kappa - 1$ clusters $D_1, ..., D_{\kappa-1}$
6  initialize $\mathcal{M}$ with $\kappa$ states $s_1, ..., s_\kappa$ and transition matrix $\mathbf{P}$
7  for $i \leftarrow 1, ..., l$ do
       //$l$ samples
8      for $j \leftarrow 1, ..., \kappa - 1$ do
9          pick $\mathbf{d}_{j1}$, $\mathbf{d}_{j2}$ uniformly at random from $D_j$
10     for $j \leftarrow 1, ..., \kappa - 1$ do
11         for $q \leftarrow 1, ..., \kappa - 1$ do
12             if $j = q$ then
13                 $(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2) \leftarrow (\mathbf{d}_{j1}, \mathbf{d}_{j2})$
14             if $j < q$ then
15                 $(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2) \leftarrow (\mathbf{d}_{j1}, \mathbf{d}_{q1})$
16             if $j > q$ then
17                 $(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2) \leftarrow (\mathbf{d}_{j2}, \mathbf{d}_{q2})$
18             $\mathbf{P}_{jq} \leftarrow \mathbf{P}_{jq} + 1 - \frac{\| \frac{\boldsymbol{\sigma}_1 + \mathbf{r}_t}{\|\boldsymbol{\sigma}_1 + \mathbf{r}_t\|} - \boldsymbol{\sigma}_2 \|}{2}$
19         $\mathbf{P}_{j\kappa} \leftarrow \mathbf{P}_{j\kappa} + 1 - \frac{\| \frac{\mathbf{d}_{j1} + \mathbf{r}_t}{\|\mathbf{d}_{j1} + \mathbf{r}_t\|} - \boldsymbol{\sigma}^* \|}{2}$ //going into critical state
20         $\mathbf{P}_{\kappa j} \leftarrow \mathbf{P}_{\kappa j} + 1 - \frac{\| \frac{\boldsymbol{\sigma}^* + \mathbf{r}_t}{\|\boldsymbol{\sigma}^* + \mathbf{r}_t\|} - \mathbf{d}_{j2} \|}{2}$ //out of critical state
21 $\mathbf{p} \leftarrow \frac{\mathbf{P}}{l}$
22 normalize $\mathbf{P}$
23 return $\mathcal{M}$
```

event e and $\mathbf{r_e}$ is the embedding vector (learned earlier) for the event-state relationship out of event e. Thus, after the loop in lines 2-4, D is a set of embedding vectors of the state-holders, one for each tuple in W. In line 5, we can use any clustering algorithm; for efficiency and simplicity, we use k-means clustering [79]. In line 6, the last state s_κ is reserved for the critical state σ^*.

Lines 7-21 basically use sampling and the embedding vectors to estimate the transition (i.e., "followed by") probabilities between two states. Lines 8-9 get 2 sample vectors from each cluster. Lines 13, 15, and 17 are to set a pair of sample vectors to calibrate the transition probabilities (line 18) either from a state to itself or between two states in both directions. In line 18, the numerator $\|\frac{\boldsymbol{\sigma}_1+\mathbf{r}_t}{\|\boldsymbol{\sigma}_1+\mathbf{r}_t\|} - \boldsymbol{\sigma}_2\|$ measures the distance between (normalized) $\boldsymbol{\sigma}_1 + \mathbf{r}_t$ and $\boldsymbol{\sigma}_2$ in the range [0, 2], and hence the whole increment is a probability in [0, 1] for this sample (i.e., how likely it is for σ_2 to follow σ_1). Line 21 calculates the average probabilities over the l samples above. In line 22, we normalize the probability values so that the total probability out of a state is 1. In this way, any "bold" edges that indicate strong "followed by" relationship should be discovered. For example, when j and q ($j < q$) in lines 10-11 correspond to states A and B in Figure 10, line 15 picks a pair of sample vectors from A and B's clusters, and line 18 estimates the transition probability from A to B. The average in the end after l samples is 0.2, as shown in Figure 10.

Now we use Rademacher complexity theory [80] to analyze the accuracy guarantees. Rademacher complexity [80] is a fundamental concept to study the rate of convergence of a set of sample averages to their expectations. It is at the core of statistical learning theory [81], but its usefulness extends way beyond the learning framework. The Rademacher bounds depend on the training set distribution (unlike VC-dimension based bounds [82] which are data independent), and hence can often give better bounds for specific input distributions. Moreover, it is estimated from the training set, allowing for strong bounds derived from a sample itself.

We consider a finite domain $\mathcal{D}$. Let $\mathcal{F}$ be a family of functions from $\mathcal{D}$ to $[0,1]$, and let $S = \{s_1, ..., s_n\}$ be a set of n independent samples from $\mathcal{D}$. For each $f \in \mathcal{F}$, define $m_{\mathcal{D}}(f) = \frac{1}{|\mathcal{D}|}\sum_{c\in\mathcal{D}} f(c)$ and $m_S(f) = \frac{1}{n}\sum_{i=1}^{n} f(s_i)$. Some results of Rademacher complexity theory are bounding the maximum deviation of $m_S(f)$ from $m_{\mathcal{D}}(f)$, i.e., $\sup\limits_{f\in\mathcal{F}} |m_S(f) - m_{\mathcal{D}}(f)|$. Specifically, *Rademacher variables* are defined as n independent random variables $\sigma = (\sigma_1, ..., \sigma_n)$ with $\Pr(\sigma_i = -1) = \Pr(\sigma_i = 1) = 1/2$. Then the (empirical) *Rademacher complexity* is defined as $R_{\mathcal{F}}(S) = \mathbf{E}_\sigma[\sup_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i f(s_i)]$. A key property of the Rademacher complexity of a set of functions $\mathcal{F}$ is that it bounds the expected maximum error in estimating the mean of any function $f \in \mathcal{F}$ using a sample, as we use in Theorem 11 below.

Let $\mathcal{F}$ be the family of κ^2 functions, one for each probability in the transition matrix $\mathbf{P}$ that we try to estimate. In lines 7-9, we use l samples $S = \{s_1, ..., s_l\}$ to estimate $\mathcal{F}$. The estimate for entry $f \in \mathcal{F}$ is based on the average $m_S(f)$ over S (line 21), while ideally it should be calculated from the whole space of data points $\mathcal{D}$ as $m_{\mathcal{D}}(f)$ (which is clearly infeasible). Thus, we try to bound the maximum difference of any $m_S(f)$ from $m_{\mathcal{D}}(f)$, as below.

Theorem 11. *Algorithm* GenerateStateMachine *estimates state transition probabilities based on the embedding vectors with the following guarantees. For* $\epsilon \in (0,1)$*, with probability at least* $1-\epsilon$*,* $\sup_{f\in\mathcal{F}} |m_S(f) - m_{\mathcal{D}}(f)| \leq 2R_{\mathcal{F}}(S) + \frac{\ln\frac{3}{\epsilon} + \sqrt{(\ln\frac{3}{\epsilon} + 4lR_{\mathcal{F}}(S))\ln\frac{3}{\epsilon}}}{l} + \sqrt{\frac{\ln\frac{3}{\epsilon}}{2l}}$*,* *where $R_{\mathcal{F}}(S)$ is the Rademacher complexity of $\mathcal{F}$ on S satisfying $R_{\mathcal{F}}(S) \leq \min_{r \in R^+} w(r)$, $w(r) = \frac{1}{r}\ln(\sum_{\boldsymbol{v} \in \mathcal{V}_S} \exp[\frac{r^2\|\boldsymbol{v}\|_2^2}{2l^2}])$, $\mathcal{V}_S = \{\boldsymbol{v}_{fS}, f \in \mathcal{F}\}$, and $\boldsymbol{v}_{fS} = (f(s_1), ..., f(s_l))$.*

Proof. The probability estimate in lines 18-20 of the algorithm is $1 - \frac{\|\frac{\boldsymbol{\sigma}_1 + \mathbf{r}_t}{\|\boldsymbol{\sigma}_1 + \mathbf{r}_t\|} - \boldsymbol{\sigma}_2\|}{2}$ for a "followed by" transition from state σ_1 to state σ_2. This is correct because the numerator $\|\frac{\boldsymbol{\sigma}_1 + \mathbf{r}_t}{\|\boldsymbol{\sigma}_1 + \mathbf{r}_t\|} - \boldsymbol{\sigma}_2\|$ measures the distance between (normalized) $\boldsymbol{\sigma}_1 + \mathbf{r}_t$ and $\boldsymbol{\sigma}_2$ in the range [0, 2], and hence the whole estimate is a probability in [0, 1]. Thus, our sampling framework follows Rademacher complexity, and the bounds themselves as stated in the theorem are from [83]. □

Remarks. The function w in Theorem 11 is convex and continuous in R^+, and has first and second derivatives everywhere in its domain. Hence it is possible to minimize it efficiently using standard convex optimization methods [84]. Finally, this is a data dependent bound—the $f(s_i)$ values are based on the actual data.

In order to monitor a substream, we first need to figure out what state we are in within the probabilistic state machine. This is not exactly trivial, since data streams can be noisy or dynamic. Instead of relying on a single current tuple, it is more reliable and robust to use a short sequence (e.g., ten) of the most recent tuples to gauge the optimal "consistency" with which portion of the probabilistic state machine. Intuitively, this not only tells us how suitable the PSM pattern is for the given recent history, but it also informs us of the optimal estimate of the current state within the PSM. We call it the *blend-in cost*.

In order to monitor and foresee how imminent the critical event is, once we have a good estimate of which state we are in as discussed above, we should add the second part of the cost, which is the projected future cost to reach the critical state.

Algorithm 15: AlignTrajectory ($\mathcal{H}$, $\mathcal{M}$)

```
Input: $\mathcal{H}$: sequence of states of a substream, in reverse-time order;
       $\mathcal{M}$: probabilistic state machine
Output: the minimum cost to align $\mathcal{H}$
1  create a priority queue (min heap) $\mathcal{Q}$
2  $c_m \leftarrow 0$ //optimistic future cost lower bound
3  for $s \in \mathcal{H}$ do
4      $c_m \leftarrow c_m - \log \max_i \mathbf{P}_{is}$
5  for $i \leftarrow 1, ..., \kappa$ do
       //initialize entries in $\mathcal{Q}$
6      if $i = \mathcal{H}[0]$ then
           //a match
7          add $(i, \mathcal{H}_1, c_m + \log \max_j \mathbf{P}_{ji})$ to $\mathcal{Q}$ //last entry is weight
8      else
9          add $(i, \mathcal{H}, c_m)$ to $\mathcal{Q}$
10 while $(s, H, c) \leftarrow pop(\mathcal{Q})$ do
11     if $H$ is empty then
12         return $c$
13     for $(s' \rightarrow s) \in \mathcal{M}$ do
14         if $s' = H[0]$ then
               //a match
15             add $(s', H_1, c + \log \max_i \mathbf{P}_{is'} - \log \mathbf{P}_{s's})$ to $\mathcal{Q}$
16         else
17             add $(s', H, c - \log \mathbf{P}_{s's})$ to $\mathcal{Q}$
18     add $(s, H_1, c + \log \max_i \mathbf{P}_{iH[0]} - \alpha \log \min_i \mathbf{P}_{iH[0]})$ to $\mathcal{Q}$
```

We use a history $\mathcal{H}$ of length h tuples to measure its minimum-cost match with the given PSM $\mathcal{M}$. This can be accomplished through dynamic programming over a Breadth-First-Search tree rooted at the current state. However, the numbers of choices at each state may be many and dynamic programming can be very expensive. We devise a novel A^* search [85] algorithm to judiciously compute the most promising paths and aggressively prune search paths. The algorithm is shown in AlignTrajectory.

The estimated alignment (or blend-in) cost throughout the algorithm has two portions:

a committed (i.e., decided) portion and a lower bound of the undecided (future) portion. Each tuple in $\mathcal{H}$ corresponds to a state in $\mathcal{M}$. By our design, in general, a transition from state s' to state s incurs a cost $-\log \mathbf{P}_{s's}$. Intuitively, if the transition probability is 1, then the cost is 0; the lower the probability is, the higher the cost.

Our algorithm progressively matches $\mathcal{H}$ with $\mathcal{M}$. At any point in time, the *match state* is (s, H), where s is the current state in $\mathcal{M}$, and H is the history to be matched. Since we traverse the history in reverse time order, we also traverse $\mathcal{M}$ with edges reversed.

Specifically, for each incoming neighbor state s' of s in $\mathcal{M}$, there are the following two cases:

(1) If $s' = H[0]$, where $H[0]$ is the next state in the remaining history H, then we pay the cost for $s' \to s$, which is $-\log \mathbf{P}_{s's}$, and the new match state is (s', H_1), where H_1 is the sub-history without $H[0]$ (line 15).

(2) If $s' \neq H[0]$, we pay the same cost for $s' \to s$, and the new state is (s', H). That is, we pay the cost to "float to" state s' without making progress in H (line 17).

Another plan that we should always add to the queue is that we pay the *deletion cost* to just delete $H[0]$, and the new state is (s, H_1). The deletion cost is $-\alpha \log \min_i \mathbf{P}_{i,H[0]}$, where α is a constant. That is, the deletion cost is a multiple of the maximum cost to get to $H[0]$. This is line 18.

For A^* search, the optimistic lower bound of future total cost is $\sum_{s \in H}[-\log \max_{1 \leqslant i \leqslant k} \mathbf{P}_{i,s}]$, as one has to match each state in history (skipping it would be more expensive—otherwise there would be no incentive to match). Lines 2-4 of the algorithm is to sum up this initial total cost lower bound.

To begin the A^* search, we put in the heap (i, H), for $1 \leqslant i \leqslant \kappa$ and $i \neq H[0]$, as well as $(H[0], H_1)$. The cost lower bound is the key of the MIN heap priority queue. This corresponds to lines 5-9 of the algorithm. Lines 10-18 of the algorithm repeatedly pop out the head of the priority queue and progress the search with the three possibilities as described above. In lines 11-12, when the remaining history is empty for the head of the queue (i.e., the whole path is decided), we finish the search. For example, using the PSM illustrated in Figure 10, suppose the input is $\mathcal{H} = ACDF$, in reverse time order as required. In line 6, $\mathcal{H}[0] = A$ matches one of the states (e.g., $i = 1$). Line 7 removes the lower bound for i (i.e., $-\log \max_j \mathbf{P}_{ji}$) from the total cost lower bound, and adds this match entry to the priority queue $\mathcal{Q}$. When this match entry is popped out from $\mathcal{Q}$ in line 10, $H[0] = C$ and indeed there is a transition $C \rightarrow A$ in Figure 10, which satisfies the condition in line 14, bringing the state to C in line 15 to be added back to $\mathcal{Q}$. For the cost lower bound in line 15, we remove the initial bound for $s' = C$ and add the transition cost from $s' = C$ to $s = A$. The alignment algorithm proceeds in this manner until $\mathcal{H}$ is exhausted.

Theorem 12. *Algorithm* AlignTrajectory *is correct in finding the minimum alignment cost between $\mathcal{H}$ and $\mathcal{M}$.*

Proof. There are three operations during the alignment process: (1) match a state between $\mathcal{H}$ and $\mathcal{M}$, (2) delete a state in $\mathcal{H}$ and pay the deletion cost, and (3) float to a new state (paying the state transition cost) in $\mathcal{M}$ without changing $\mathcal{H}$. The algorithm explores all three plans when expanding the search path, and maintains the optimistic lower bound of future matching cost by summing up the match cost (operation 1 above), which is indeed the minimum cost for the unmatched portion in $\mathcal{H}$. Thus, the pruning is correct. $\square$

Let us now address the second portion of the cost, which is the projected future cost to reach the critical state, based on the optimal estimate of the current state accomplished in AlignTrajectory. The algorithm outline is as follows:

1. Let each directed edge $i \to j$ of $\mathcal{M}$ have weight $-\log \mathbf{P}_{ij}$.

2. Run Dijstra's algorithm and return the shortest-path weight from the current-state node to critical-state node.

We treat this part of the problem as a shortest path problem, i.e., we find the shortest path from the current state to the critical state in the directed graph of $\mathcal{M}$, where the weight of an edge from state i to state j is $-\log \mathbf{P}_{ij}$.

Theorem 13. *The above algorithm finds the cost of the maximum probability transition path from the current state to the critical state in the probabilistic state machine pattern $\mathcal{M}$.*

Proof. The algorithm uses Dijstra's algorithm to find the shortest path where the cost of a path is the negative value of the sum of log probabilities. Since maximizing this sum (of log probabilities) is equivalent to maximizing the *product of the probabilities* (as the log function is monotonic), our algorithm finds the cost of the maximum probability path. $\square$

Remarks. We have studied two parts of the cost with the goal of estimating the imminence of the critical state. The blend-in cost intuitively characterizes the minimum cost to get to where we are right now, with respect to the learned PSM pattern, while the projected future cost indicates the upcoming cost in order to reach the critical state. The sum is the total cost of the current "trajectory" to align and reach the goal. The first part also reflects the trustworthiness of using the model to make the judgement. Thus, the total cost, serves as a score indicator for monitoring and predicting the imminence of the critical event.

We will further discuss how to select the optimal number of states. The basic idea is to compare the blend-in cost of a piece of real stream sequence with that of a “baseline” semi-random sequence in which we preserve some basic statistics as the real sequence, such as using the same set of attribute events and the same node degrees, but randomly shuffling the edges. The intuition is that a good PSM pattern $\mathcal{M}$ should maximize the blend-in cost difference between the two versions of input sequence.

We first define a baseline semi-random stream window W', which has the same number of tuples, as well as the same tuple timestamps, as our stream window W. Moreover, for each attribute, we keep the same multiset of events as in W, but shuffle them uniformly at random and place the values in the sequence of tuples in W'.

Once we build a model $\mathcal{M}$, we pick some random sequences of length h from W and W', and calculate the average consistency cost difference between them. Using this as a metric, we do gradient ascent to find the optimal number of states κ. The algorithm is in SelectStateNumber.

In line 4, the notation W'_i represents the suffix of tuple array W' starting from the i’th tuple. That is, we swap $W'[i][j]$ (the j’th attribute of the i’th tuple) with an arbitrary tuple that follows. The loop in lines 6-13 does gradient ascent optimization by building two versions of the PSM with slightly different number of states and estimating the gradient of the cost saving difference compared to the semi-random baseline sequences. Note that each random piece H or H' that we pick in line 10 must be from the same substream.

Remarks. From the construction of the algorithm, it is easy to see that the semi-

```
Algorithm 16: SelectStateNumber ($W, \mathcal{G}$ )
  Input: $W$: a finite window of $w$ stream tuples;
         $\mathcal{G}$: dynamic graph with learned embedding
  Output: optimal number of states $\kappa^*$
1 $W' \leftarrow$ copy of $W$
2 for $i \leftarrow 1, ..., w-1$ do
3     for $j \leftarrow 1, ..., a$ do
4         swap $W'[i][j]$ with attribute $j$ of a random tuple in $W'_i$
5 $\kappa \leftarrow \kappa_0$
6 while $\kappa$ has not converged do
7     $\mathcal{M}_- \leftarrow$ GenerateStateMachine $(W, \mathcal{G}, \kappa - \frac{\delta}{2})$
8     $\mathcal{M}_+ \leftarrow$ GenerateStateMachine $(W, \mathcal{G}, \kappa + \frac{\delta}{2})$
9     $f_- \leftarrow 0$; $f_+ \leftarrow 0$
10    for each of $c$ random pieces $H$ and $H'$ from $W$ and $W'$ do
11        $f_- \leftarrow f_- +$ AlignTrajectory$(H', \mathcal{M}_-) -$ AlignTrajectory$(H, \mathcal{M}_-)$
12        $f_+ \leftarrow f_+ +$ AlignTrajectory$(H', \mathcal{M}_+) -$ AlignTrajectory$(H, \mathcal{M}_+)$
13    gradient ascent update of $\kappa$ based on $\frac{\partial f}{\partial \kappa}$ estimated as $\frac{f_+ - f_-}{\delta}$
14 return $\kappa$
```

random data sequence W' has exactly the same set of nodes, the same degrees of each node, the same set of edges, the same time edges, but the event-key edges and event-state edges are randomly shuffled. Each attribute's event group has exactly one edge to each tuple's key/state.

4 EXPERIMENTS

We set up experiments by using 10 real-world datasets to compare our techniques with prior approaches and baselines. The effectiveness and efficiency of our methods were evaluated based on the results obtained.

4.1 Attribute Values Prediction

The experiments help us understand the answers to following questions: **(1)** How long are the graph/model building and vertex grouping times under real-world dynamic network datasets? How effective is our vertex grouping in preserving value correlations? **(2)** How much delay is there to learn all the parameters of our model? Is there any connection between the number of states chosen by the learning algorithm at each node and the group size parameter α ? **(3)** How accurate is our model in answering queries on unobserved values? How does it compare with two baseline algorithms? **(4)** How efficiently does our algorithm answer queries, as compared to the two baseline algorithms? How much do our optimizations improve the performance? **(5)** How effective and efficient is our proposal of cross-validation and incremental model-update?

4.1.1 Datasets and Experiment Setup

We use two real-world dynamic graph datasets as follows:

- **Hong Kong traffic data.** We get this dataset from [86], which provides a link to download very large historical traffic data of Hong Kong. For example, using an interval from 8/1/2009 to 9/22/2017 gives a dataset of 42.59 GB. The graph update frequency is on average about one report of a road every 0.38 second. A road report consists of a *timestamp*, *two ends* of the road, *region*, *type*, *saturation level*, and observed *travel speed*. The attribute we are most interested in, other than time, is the travel speed, which indicates the traffic condition.
- **New York taxi data.** This 11 GB New York city's 2013 taxi trip data contains in-

formation such as medallion number, vendor ID, pick-up and drop-off date, time and location, passenger count, trip time, and trip distance [2]. A new data update—a report about a pick-up and drop-off trip information comes in on average every 0.186 second.

Dataset	**#vertices**	**#edges**	**Average inter-update time**	**Data Size**
HK traffic	605	365,420	0.48 sec	42.59GB
NY taxi	5,654	31,962,062	0.186 sec	11GB

Table 2: Statistics of the datasets.

Some statistics of the datasets are summarized in Table 2. We implement all the algorithms presented in the paper, as well as two baseline methods for comparison, which we will describe below. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

4.1.2 Experimental Results

In the first set of experiments, we examine our graph and model building, including the vertex grouping algorithm. For the HK traffic data, the *travel speed* on each road segment is the attribute value of our interest, which is dynamic over time and indicates the traffic condition of the roads. We discretize the speed value into 0 (below 10km/hr), 1 (between 10km/hr and 20km/hr), ..., 6 (above 60km/hr). Thus, each road is a *vertex*, and there is an *edge* between two vertices if the roads they correspond to intersect.

For the NY taxi data, we partition the whole area into 0.0075 latitude by 0.01 lon-

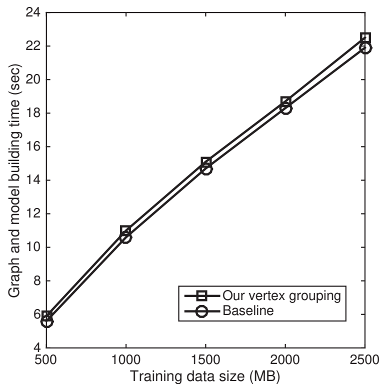


Figure 11: Graph/model building time

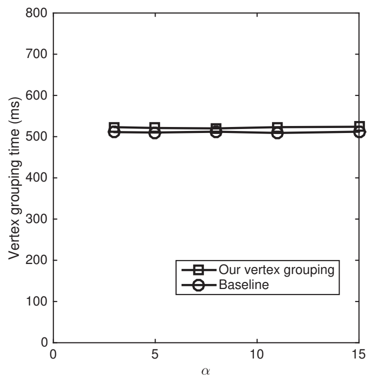


Figure 12: Vertex grouping time vs.α

gitude (i.e., about 0.5 mile by 0.5 mile) grids, and each grid area is a vertex. The attribute value in question is the number of pick-ups at the grid area during the time step. We discretize this count to 0-4 by partitioning it into five intervals. An edge is added between two grid areas (vertices) that are neighbors.

In Figure 11, we show the graph and model building time for various training data sizes from 500MB to 2.5 GB, using the NY taxi data (HK traffic data gives us a similar trend in result). As discussed, general partition algorithms such as [66] do not meet our needs. While the group size balance has been proven, to evaluate the effectiveness of our algorithm in preserving value correlations within a group, we design a baseline that, like in previous work [65], does a breadth-first-search and ensures a balanced group size α without explicitly optimizing intra-group value correlation. We can see that even for large training data sizes, the graph and model building of our algorithm takes only 10's or 20's seconds, and grows linearly with the training data size. It is only slightly slower than the baseline algorithm. This result is consistent with our analysis in Theorem 2, which shows that our vertex grouping algorithm has an $O(|E| + |V|)$ amortized cost.

In Figure 12, using 30 hours of the HK traffic data, we examine the performance of

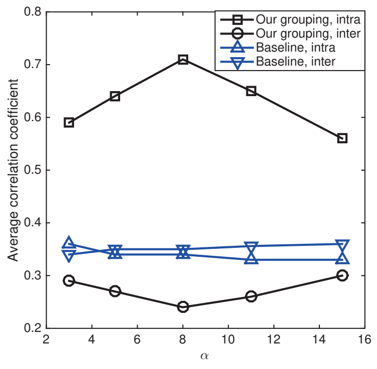


Figure 13: Correlation coefficients vs.α

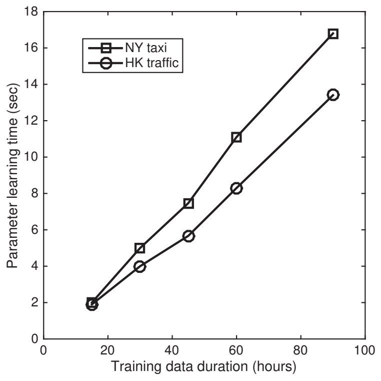


Figure 14: Parameter learning time

the vertex grouping algorithm alone under different group-size parameter α values ranging from 3 to 15, again compared with baseline algorithm. Our vertex grouping algorithm is slightly slower than the baseline, and the speed is independent of α. Again, this is consistent with Theorem 2 which indicates that the amortized cost is not a function of α.

Furthermore, in Figure 13, we test the vertex grouping result under different group-size parameter α values using the NY taxi dataset, by calculating the average (absolute value of) Pearson correlation coefficients [15] of pairs of vertices connected by edges either inside each group partition (marked as “intra” in Figure 13), or across two group partitions (marked as “inter”). We also compare the result with the baseline method as described above. It is clear that the correlations of vertices connected by intra-group edges are much higher than those of inter-group edges, under our grouping algorithm.

This is because our algorithm systematically ensures both high correlations within each group and the balance of group sizes in the range of $[\alpha, 2\alpha - 1)$. This is not the case with the baseline algorithm, which results in more or less equal division of edges with respect to vertex correlation. In addition, the difference between intra- and inter- group edges is most significant when α is a certain value (8 in our setting). When α is either too

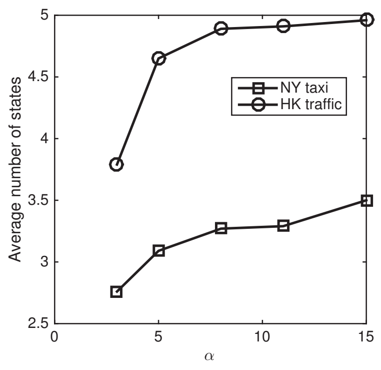


Figure 15: Average # of states vs.α

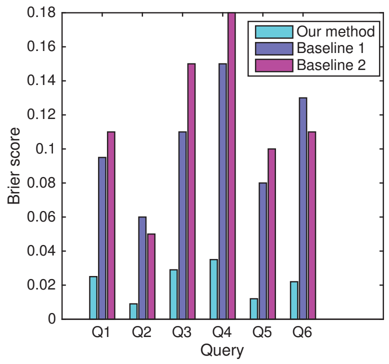


Figure 16: Brier scores/errors (HK traffic)

small or too large, more fraction of low-correlation edges must be put inside a group, as all vertices must be grouped with their neighbors by the algorithm. As shown later, α also has an effect on model parameters such as the number of states of a node.

In the next set of experiments, we study the model parameter learning algorithm (AllParameterLearning). Figure 14 shows the running times for both datasets, varying the training data duration from 15 hours to 90 hours. We can see that they are both linear functions of the size of training data, with slightly different rates of change. Moreover, the parameter learning times are acceptable, ranging from 2 seconds to 17 seconds.

In Figure 15, we further look at the result of this learning, in the average number of states of each node of the model, while varying the group size parameter α. For both datasets, the average number of states per node increases as group size increases. The reason is that, for a larger group size, the parent node in our model, as a latent variable, must be able to characterize more complex co-evolvement of the child nodes as a group– therefore more "states" are needed. As it turns out, on average more states are needed for the travel speed attribute in the Hong Kong traffic dataset than the number of taxi pick-ups in a grid area of the New York taxi dataset. In subsequent experiments, we use a default

$\alpha = 5$ unless otherwise specified.

In the next set of experiments, we examine the query result accuracy, as well as the performance of answering queries. We design six query templates considering variations such as the density of observed data, whether the queried vertex is among the one observed (but at a different time) or an unobserved one, and whether the queried time is within the interval $T = [t_b, t_e]$ of the observed vertices, or precede or succeed T. The query generation templates are as follows:

(Q1) Perform a random walk for 10 time steps from an arbitrary vertex; observe one vertex at each time step; query an observed vertex at an unobserved time within T.

(Q2) Same as Q1, except that we observe *three* vertices at each time step (i.e., walk through more vertices/edges).

(Q3) Same as Q1, except that we query an unobserved vertex to be seen at time $t_e + 1$ (for $T = [t_b, t_e]$, i.e., the next vertex to be traversed if the random walk were to continue).

(Q4) Same as Q3, except that we query the vertex at time $t_e + 3$, i.e., two more time steps later.

(Q5) Same as Q2, except that we query an unobserved vertex at time $t_b - 1$ (for $T = [t_b, t_e]$, i.e., the preceding vertex if the random walk started a time step earlier).

(Q6) Same as Q5, except that we query an unobserved vertex at time $t_b - 3$.

Recall that our query processing algorithms return a probability distribution as a query answer. Fortunately, both of our two real-world datasets have observed values as the ground truth for us to evaluate the accuracy of query answers. Simply using the difference between the expected value of the distribution and the ground truth value is undesirable, as that does not take the uncertainty (i.e., the "spread") of the query answer into consideration. Inspired

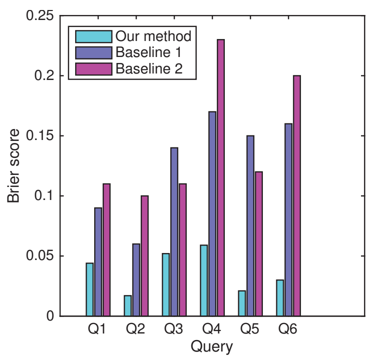


Figure 17: Brier scores/errors (NY taxi)

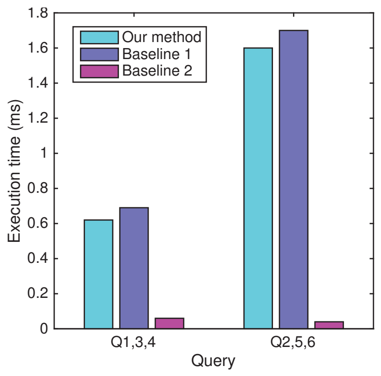


Figure 18: Execution time (HK traffic)

by the Brier score [87] which works only for *nominal* attributes, we define a variant of it that is more suitable for *ordinal* values, as in our case. Our Brier score variant is defined as $\frac{\bullet \sum_i p_i(v_i - o)^2}{d_m^2(o)}$, for a query result distribution $\{(v_i, p_i)\}$ and the ground truth value o, where $d_m(o)$ is the maximum possible distance to o. For example, if the whole value range is $[0, 4]$, the ground truth is $o = 2$, and the query result is $\{(2, 0.5), (3, 0.5)\}$, then the Brier score variant is $\frac{0.5 \times 1^2}{2^2} = 0.125$. Thus, it is always a value in $[0, 1]$, with 0 being the most accurate, considering both the *location* and *spread* of a distribution.

In Figure 16, using the Hong Kong traffic dataset, we show the average Brier scores of the results of six groups of queries Q1 to Q6 generated from the six query templates as described above. Figure 17 is likewise for the New York taxi dataset. We compare against two baselines. Baseline 1 is as described earlier, doing BFS for vertex grouping, but otherwise is the same as our approach in learning parameters and answering queries. Baseline 2 is the previous state-of-the-art algorithm as described in [88] that does a combined Kernel Regression and k-NearestNeighbor with $k = 10$. We can see that our method has small Brier scores (errors) for both datasets, significantly smaller than the two baseline approaches. Among the six types of queries, Q2, Q5, and Q6 have smaller errors due to the fact that they have three times of the observed vertices, i.e., more evidence that sheds light to the queried values. Moreover, in general, the farther away in time is the queried vertex from the observed vertex interval T, the more error is incurred (e.g., Q1 within T, Q3 at $t_e + 1$, and Q4 at $t_e + 3$). This is because when belief propagation to the query vertices has to go through more unobserved nodes, greater errors are accumulated.

Baseline 1 has larger errors because its vertex grouping is rather random, which makes its model inferior to ours, even though it uses the same parameter learning and query processing algorithms. Baseline 2 is less accurate than our method because it is not as sophisticated as our model in capturing the value correlations near and far in the graph topology and over time. Finally, we observe that the New York taxi dataset tends to give slightly greater errors in general than the Hong Kong traffic dataset.

Then we measure the query execution times of the three approaches, as shown in Figure 18 for the Hong Kong traffic dataset, and in Figure 19 for the New York taxi dataset. Query groups 1, 3, and 4 have the same number of observed (evidence) vertices, thus having almost identical execution times, and we show them together; similar for query groups 2, 5, and 6. The figures show that, for both datasets, our method is slightly faster than baseline 1 in answering queries, while baseline 2 is much faster than both of them—baseline 2 has a significantly smaller computation overhead. Baseline 1 is slightly slower than our method because the vertices in each group are less correlated; hence a higher entropy is present in each group of child nodes, causing their parent node to have more states on average, slowing down the inference during query processing. Query groups 2, 5, and 6 are slower than 1, 3, and 4 since the former query groups involve more observed vertices, which require more time in the first pass belief propagation (from observed leaves to root). In addition, we

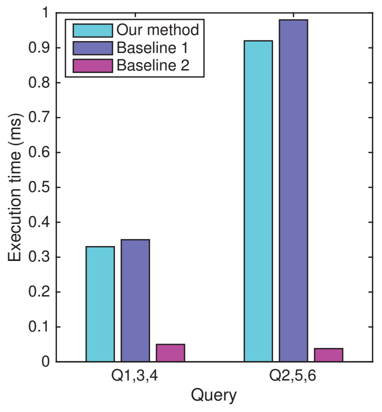


Figure 19: Execution time (NY taxi)

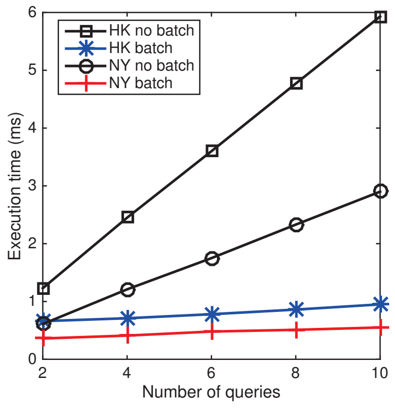


Figure 20: Query batch optimization

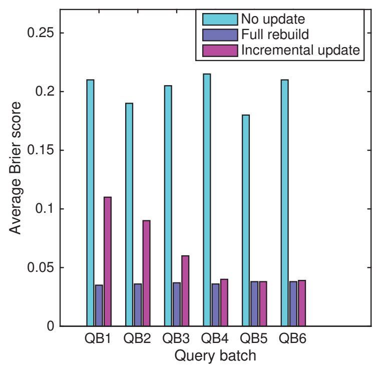


Figure 21: Effect of incremental update

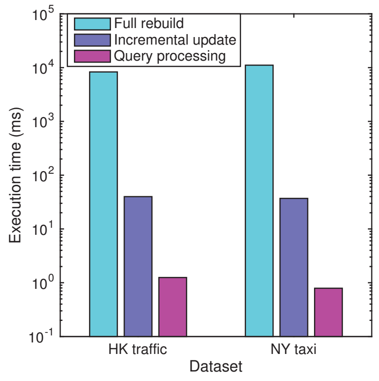


Figure 22: Overhead of incremental update

observe from Figures 16 and 17 that our method takes longer time for the Hong Kong traffic dataset. This is because, as indicated in Figure 15, the average number of states per node is higher for the Hong Kong traffic dataset, which makes the BeliefPropagationOverBridges algorithm slower.

In Figure 20, we further evaluate the benefit of the refinements/optimizations. We only test the query batch technique because the other two refinements involve belief propagation over fewer vertices, and are obviously faster. Figure 20 shows the execution times under various query batch sizes (i.e., the number of queries in a batch) We can see that the batch versions considerably improve the performance due to the sharing of the belief/message propagations among the queries in a batch.

Our last set of experiments examine the effectiveness and efficiency of the incremental model update algorithm described in Section 3.1.3. We first arbitrarily change the parameters of our model, causing the model to be unsuitable. Then we run six query batches QB1 to QB6 in a sequential order, each containing 12 queries, two from each query templates described earlier. We compare three versions: (1) no model update, and always use the unsuitable model, (2) a full rebuild of model first, before answering the six query batches, and (3) run CrossValidationPartialUpdate while answering each query batch in order. In Figure 21, we measure the average Brier scores (i.e., result errors). The no model update version incurs the most errors, while a full rebuild gives small errors. Our incremental update algorithm achieves accuracy improvement as it answers each batch of queries in order. This is because it partially updates the model for the portion of the model parameters relevant to the current queries. The query result accuracy improves as this update progresses, eventually approaching the level achieved by a full model rebuild in the beginning.

Figure 22 shows the execution times of the relevant operations in the approaches above. Note that the y-axis is in logarithm scale. We can see that a full model rebuild is two orders of magnitude slower than an incremental update, which only accesses a small part of the model. We also show this cost side-by-side with the query processing cost of a batch of queries described above, which is the smallest among the three operations. The combined results in Figures 19 and 20 show that the incremental update approach is practical and useful in adapting the model for answering queries.

4.1.3 Summary of Experiment Results

The comprehensive experimental results in this section indicate that our vertex grouping is fast and produces grouping that preserves high vertex correlations within groups. The model building and parameter learning algorithms are efficient and result in models that can answer queries accurately. Factors that affect query result accuracy include the density of observed evidence vertices in the graph and over time, as well as the distance of query vertex/time from the observed vertices. Finally, our proposal of incremental model update is effective and efficient in rebuilding models and improving result accuracy.

4.2 Unobserved Links/relationships Prediction

In this section, we show that our cracking index is very effective in accomplishing our goal for answering two types of important queries of virtual knowledge graphs—top-k entity queries and aggregate queries.

4.2.1 Datasets and Experiment Setup

We use three real world knowledge graph datasets: **(1) Freebase data.** Freebase [89] is a large collaborative knowledge base, an online collection of structured data harvested from many sources, including individual, user-submitted wiki contributions. Google's Knowledge Graph was powered in part by Freebase. The dataset we use is a one-time dump through March 2013. **(2) Movie data.** This is another popular knowledge graph dataset, which describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service [90]. Entities include users, movies, genres, and tags. Ratings are made on a 5-star scale, with half-star increments (0.5 stars to 5.0 stars). We create two relationships for ratings: a user "likes" a movie if the the rating is at least 4.0 (in the range between 0 and 5.0); a user "dislikes" a movie if the rating is less than or equal to 2.0. There are also relationships "has-genres" and "has-tags". **(3) Amazon data.** This dataset [91] contains product reviews and metadata from Amazon, including 142.8 million reviews spanning May 1996 to July 2014. The review rating scale ranges from 1 to 5, where 5 denotes the most positive rating. Nodes represent users and products, and edges represent individual ratings. We create relationships "likes" and "dislikes" in the same way as movie data. In addition, the data contains "also viewed" and "also bought" relationships.

We implement all the algorithms in Java. We also use the graph embedding code from the authors of [92], a high-dimensional index PH-tree from the authors of [93], and the H2-ALSH code from [94] for comparisons. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

4.2.2 Experimental Results

In the first set of experiments, we use the Freebase data to compare a few approaches. Two of these approaches are our main cracking index method and the top-k split-choice index build method. One baseline approach is what one would do without our work—answering the top-k entity queries without using an index by iterating over all possible entities. The second baseline uses a state-of-the-art high-dimensional index, called PH-tree [93], to index the high-dimensional (50 or 100 dimensions) embedding vectors directly, without transforming them to $\mathbb{S}_2$. Another baseline approach goes a step further by using

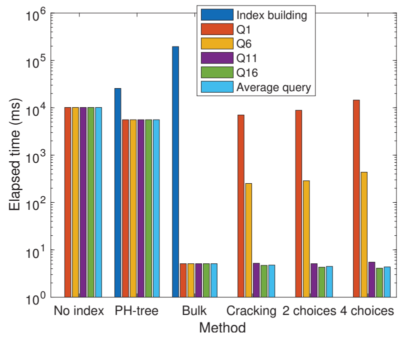


Figure 23: Elapsed time (Freebase)

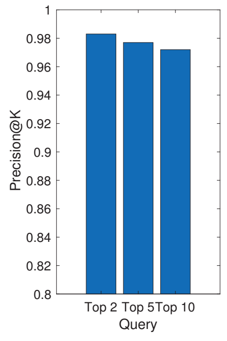


Figure 24: Accuracy (Freebase)

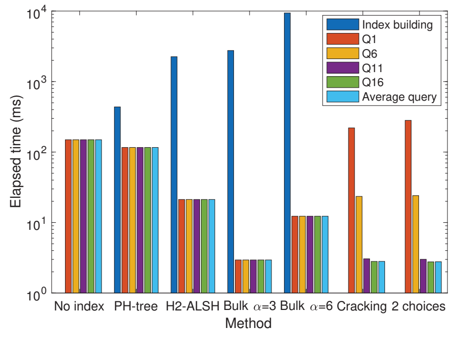


Figure 25: Elapsed time (movie)

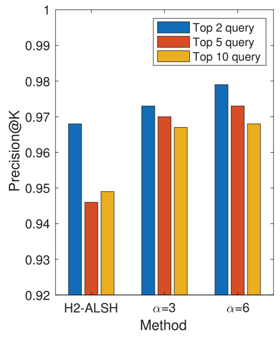


Figure 26: Accuracy (movie)

an R-tree index by bulk-loading it, without our cracking index techniques. The results are the average of at least ten runs.

Note that for general knowledge graphs, such as the Freebase data, we cannot use the H2-ALSH scheme [31] because it can only work with one relationship type—H2-ALSH is basically a locality sensitive hashing mechanism working with collaborative filtering. Later we will use other datasets to compare against H2-ALSH.

In Figure 23, we examine the execution times of the approaches described above over the Freebase data. For the top-k split-choice index build method, there is a parameter of how many choices to take into account at each split. In the figure, we show the results when this parameter is 2 or 4, respectively (i.e., the last two groups of bars). Recall that our cracking index methods do not have offline index building, but start to shape the index

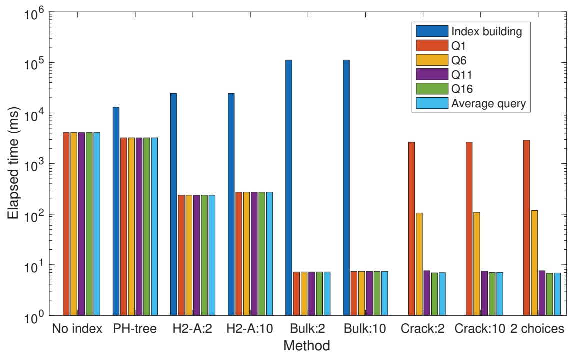


Figure 27: Elapsed time (Amazon)

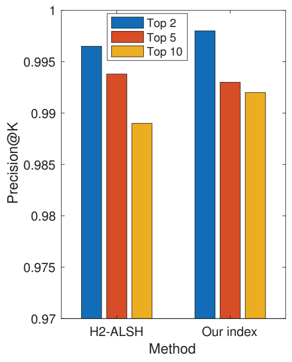


Figure 28: Accuracy (Amazon)

when queries arrive online. Hence we examine how the response time changes over the initial sequence of queries. In order to systematically explore as much as possible the space of queried embedding vectors (e.g., $\mathbf{h} + \mathbf{r}$) in $\mathbb{S}_2$, for each query we either (1) randomly choose a head entity and a relationship and query the top-k tail entities, or (2) randomly choose a tail entity and a relationship and query the top-k head entities. In Figure 23, we measure the index building time (if any), as well as the execution times of the 1st, 6th, 11th, and 16th queries, respectively—to evaluate how the response time evolves.

In Figure 24, we also examine the accuracy of these methods with respect to the no-index method (as the ground truth). Since the no-index method is our base, we study the accuracy loss from using an approximate index (due to the transform from the embedding space $\mathbb{S}_1$ to $\mathbb{S}_2$). We use the *precision@K* metric, which is commonly used in information retrieval [95]. In our context, precision@K is the precision of the top-k result tuples when treating the top-k tuples under the no-index method as the ground truth. From Figure 24 we can see that the precision@K of all these indexing methods are high on average (at least 0.965).

We next move on to the movie data. Here we query the top-k movies that a particular person likes or dislikes (but these facts are not in the training data). We note that, even with this dataset, the closest previous work, H2-ALSH [94], still does not fully handle it, since H2-ALSH can only take into account one relationship type (say, "like") when doing the collaborative filtering and building H2-ALSH. However, other relationship types such as "dislike" has the opposite semantic meaning and would help the prediction of "like" too. Thus, our method is a more holistic approach for virtual knowledge graphs. Nonetheless, we run H2-ALSH as well for a single relationship type to observe its performance.

We show the execution time results in Figure 25, where we also compare the two parameter choices of the dimensionality of the $\mathbb{S}_2$ (and hence the index), $\alpha = 3$ versus $\alpha = 6$. We can see that the index building time of H2-ALSH is slightly faster than bulk-loading R-trees when the dimensionality $\alpha = 3$. However, H2-ALSH's query processing time is much longer, partly due to the fact that it is not a hierarchical structure, and each bucket could still be very large. Furthermore, we see that, when the index dimensionality is higher such as $\alpha = 6$, there are significant overheads both in bulking loading (building) the index and in query processing. This is because these indices have a harder time with higher dimensionalities such as 6, as overlap regions tend to be much higher. An example of query results is that a particular person (with id "176299")'s top-k "like" movie list includes "152175, Ghosts (1997), Horror", "156903, The Waiting (2016), Thriller", and "3457, Waking the Dead (2000), Drama|Thriller", where the first field is the movie ID, the second field is the movie name (and year), and the third field is the genre(s) of the movie. It seems that this person likes the thriller/horror type of movies.

In Figure 26, we also report the accuracy. The H2-ALSH numbers are based on the report from running the code of the authors [31], comparing to its no-index case. We

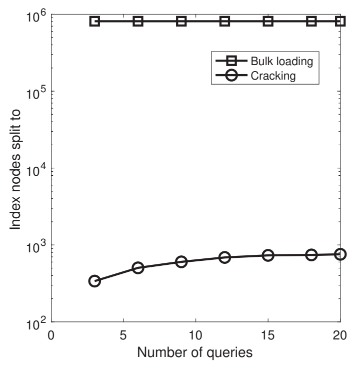


Figure 29: Nodes vs. queries (FB)

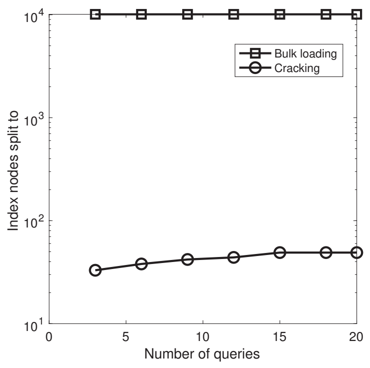


Figure 30: Nodes vs. queries (movie)

can see that the precision@K of all these approaches are quite high (at least 0.945), and our cracking index methods are slightly more accurate. This is partly due to the way our embedding space transform preserves the distance.

In the next set of experiments, we use the Amazon dataset. The performance result is shown in Figure 27. Here we also examine the overhead when we vary the "k" parameter as in top-k results. In particular, we compare two cases $k = 2$ (labeled "H2-ALSH: 2" in Figure 27) and $k = 10$ (labeled "H2-ALSH: 10"). We see that increasing k from 2 to 10 has a slight impact on the performance of H2-ALSH, but has little or no impact on the performance of our index approaches. This is because this change of number of result tuples likely still has retrieved data points within the same index node.

One interesting and important phenomenon when we compare Figure 27 with Figure 25 is that as we increase the dataset size (in particular, the number of entities)—as the case in going from movie dataset to Amazon dataset, the increase in query processing overhead is much higher for H2-ALSH than for our indexing approaches. Our query processing time is one order of magnitude faster than H2-ALSH for the movie dataset and two orders of magnitude faster for the Amazon dataset. Our method scales better due to its overall tree-

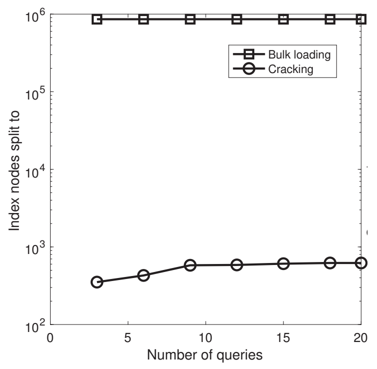


Figure 31: Nodes vs. queries (Amazon)

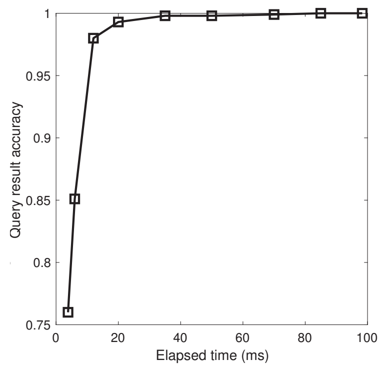


Figure 32: COUNT queries (FB)

structure index (unlike the flat buckets of LSH) with a cost logarithmic of the data size. We measure the accuracy in precision@K of these approaches in Figure 28. The comparison result is similar to those of other datasets.

In the next experiments, we compare the number of index nodes split into between a bulk loaded index and our cracking index. The results are shown in Figure 29 for the Freebase dataset, in Figure 30 for the movie dataset, and in Figure 31 for the Amazon dataset. We show the node numbers after different number of initial queries. We can see that, for all three datasets, the cracking and uneven index has a very small fraction of node splitting than the full bulk-loaded index. This is because the search space is highly uneven and the queried space is only a small fraction. Furthermore, we observe that the convergence of node number is very fast—typically after around 10 queries.

We next study approximately answering aggregate queries. Recall that there is a tradeoff between sample size (execution time) and query result accuracy. We first examine COUNT queries using the Freebase dataset, i.e., the expected count of tail entities given the head and relationship. The result is in Figure 32. The tradeoff between execution time and accuracy (compared to accessing all data points up to a probability threshold 0.01) is

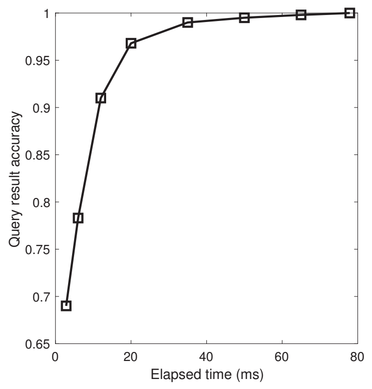


Figure 33: AVG queries (movie)

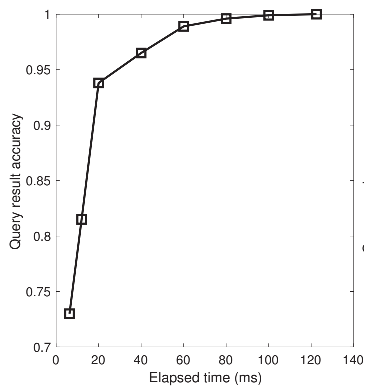
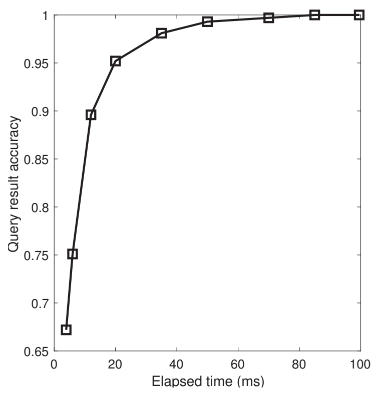


Figure 34: AVG queries (Amazon)

clearly seen here. The accuracy is measured as $1 - \frac{|v_{returned} - v_{true}|}{v_{true}}$, where $v_{returned}$ and v_{true} are query returned aggregate value and the ground truth aggregate value (from accessing all points until a probability threshold is reached), respectively.

Figure 33 shows the result of AVG queries using the movie dataset. The attribute being aggregated is the year of the movie—i.e., the query returns the average year of the movies that a particular user likes. We see a similar tradeoff as in Figure 32. When the execution time (hence the number of closest data points visited) reaches a certain value, the accuracy stays at a high level. This is because the data points are in a decreasing order of probability; thus the entities that are visited later have smaller probabilities and hence have lower weight in query result. We then use the Amazon dataset and measure the AVG queries. We add an attribute to each product entity called "quality", which is the average rating this product has received (based on all existing ratings of the product). Then we query the average quality of all the products that a particular user likes. The result is shown in Figure 34. Compared to the movie dataset, the Amazon dataset takes slightly longer time to get to high accuracy due to the much larger size of the Amazon data.

Finally, we examine the MAX/MIN queries. For the Freebase dataset, we add an



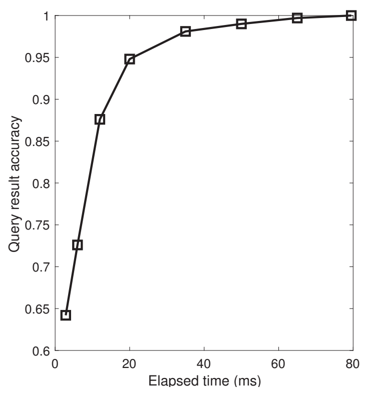


Figure 35: MAX queries (Freebase)

Figure 36: MIN queries (movie)

attribute *popularity* to each entity that is the number of related entities (i.e., in-degree plus out-degree)—indicating how popular an entity is. We issue a query to return the maximum popularity of the target entity set. For the movie dataset, we issue a query which returns the minimum year (i.e., the oldest age) of a movie among all the ones that a particular user would like. The result of these two queries are displayed in Figure 35 and Figure 36, respectively. We can see that MAX and MIN queries show a similar tradeoff between performance and accuracy as we have observed for other aggregate queries.

4.2.3 Summary of Experiment Results

The cracking index methods do not have offline index building time, and it takes a little longer for first query (but still 30 to 40 times faster than bulk-loading), with the query processing time slightly shorter than that of the bulk-loaded index. 2 or 3 choice node-split methods provide a tradeoff between a slight increase of initial queries' processing time and getting better performance over the long run. The previous work H2-ALSH cannot handle multiple relationships as in a knowledge graph. Nonetheless when restricted to only one relationship type, our cracking index methods have much smaller overhead for query processing while providing similar or slightly better accuracy; moreover, our methods scale better for larger datasets. The cracking indices only split a tiny fraction of nodes than a full bulk-loaded index, and is very compact and efficient. Lastly, our approximate aggregation methods can obtain high accuracy after a short processing time of the initial data points closest to the query center. Our analysis provides a theoretical guarantee.

4.3 Event Timing Prediction and Critical Event Monitoring

We perform a systematic empirical study that demonstrates the high accuracy of our predictions, and the high efficiency of the graph-building and training algorithms for event timing prediction problems. We also perform a systematic evaluation over four real-world datasets in different domains and compare against two baseline approaches for critical event monitoring problems.

4.3.1 Event Timing Prediction Datasets and Experiment Setup

(1) Diabetes data. This is the AIM-94 dataset provided by Michael Kahn, MD, PhD, Washington University in St. Louis [96]. It records 70 diabetes patients' event logs (multi-attribute streams) for about 5 to 9 months including insulin dose (regular, NPH, and Ultra-Lente), blood glucose at various times, hypoglycemic symptoms, meal ingestion amount, and exercise activity amount. **(2) Mobile system data.** This dataset is collected and used by Banerjee et al. in an ACM UbiComp paper [97]. The data contains traces of battery usage data for 60 laptops. In addition to battery usage, the multi-attribute streams also contain data on CPU utilization, disk space, on-AC status, Internet connectivity, and idle time (based on keyboard events) for the laptop users. **(3) Oil well data.** This dataset from Brazil [98, 99].

Prediction of undesirable events in oil wells is critical. It records various measurement events in oil wells, as well as a number of undesirable real events. The measurement events include pressure events at Permanent Downhole Gauge (PDG), temperature events at Temperature and Pressure Transducer (TPT), pressure events at TPT, among many others. The dataset is over 5 GB and has about 1 tuple per second. **(4) NY taxi data.** The trip data of this dataset is about 30 GB, containing the information of all taxi trips in the New York City in 2013 [2]. It has 14 attributes, including medallion, hack license, vendor ID, pick-up date/time, drop-off date/time, pick-up longitude/latitude, drop-off longitude/latitude, trip time, and trip distance.

We implement all the algorithms presented in this paper in Java. In particular, we extend the code of TransE [73] to handle active states, ephemeral nodes, and the attention mechanism.In addition, we have also implemented three most relevant baseline methods for comparisons: (1) sequential association rule mining, the Generalized Sequential Pattern (GSP) algorithm [11], (2) event prediction with Bayesian and Bloom filters in ICPE'13 [100], and (3) kernel-SVM [101], marked as SAR, ICPE, and KSVM in our upcoming figures, respectively. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

4.3.2 Event Timing Prediction Experimental Results

Based on the attributes of each of the two datasets, we define a set of *basic events*, as well as a set of *target events*. For the diabetes data, there are 16 basic events, each of which is out of a single attribute, including the second (or more) insulin injection of the day, a significantly increased blood glucose measurement, hypoglycemic symptoms, more-than-usual meal ingestion, and less-than-usual exercise activity. Out of the basic events, we define the target events that a user may be interested in, such as a blood glucose measurement significantly higher (or lower) than the most recent measurements, and the hypoglycemic symptoms.

Likewise, for the mobile system dataset, there are 16 basic events on individual attributes, including battery being near-empty, being connected to the Internet, and so on. We also define target events such as CPU usage of 95% or more, and idle time of at least 20 seconds. For the diabetes data, we set δ_1 to be 6 hours and δ_2 to be 80 hours, while for the mobile system application, these two time intervals should be much shorter to be useful, and we set δ_1 to be 10 minutes and δ_2 to be 1 hour.

For the oil well data, we define the increases and decreases of each measurement attribute as basic events, and each type of undesirable events as target events. For the NY taxi data, we partition the latitude and longitude ranges of NYC into 8-by-8 grids. As mentioned earlier, we define the ratio between trip time and trip distance as the trip's *delay score*. We then define the target events as the average delay score of all trips within 5 minutes at a grid area that we are interested in is above (or below) a threshold—top (or bottom) 1/4 of the whole history at that area. However, delays scores are expensive to obtain, as they require the statistics of all trips going into or coming out of an area. The idea is to use easy-to-observe simple statistics of some grid areas (other than the target areas), such as the incoming/outgoing taxi counts within 5 minutes. These events will help us predict he target events. For both oil well and NY taxi datasets, we set the δ_1 of 1/3 of the target events to be 5 minutes, 1/3 to be 10 minutes, and 1/3 to be 15 minutes. Furthermore,

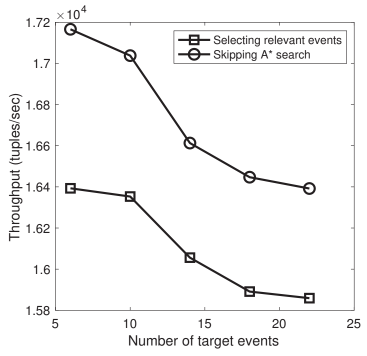


Figure 37: Learning events (mobile)

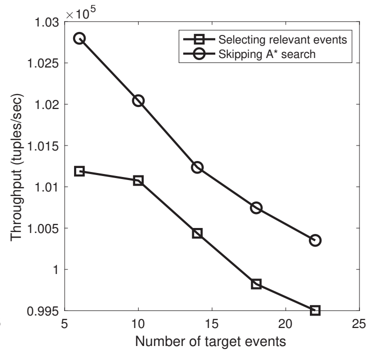


Figure 38: Learning events (oil well)

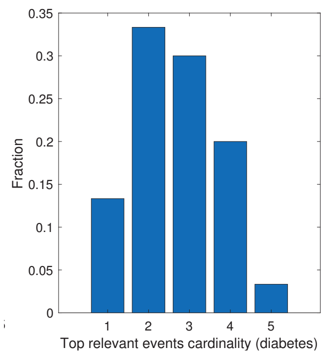


Figure 39: Event cardinality (diabetes)

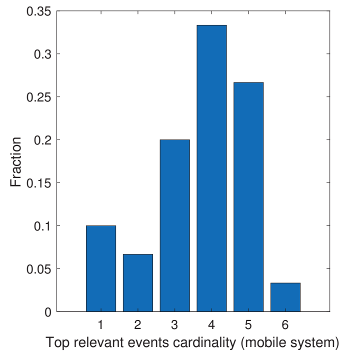


Figure 40: Event cardinality (mobile)

we set the δ_2 of the oil well data to be 3 hours, and the δ_2 of the NY taxi data to be 1 hour. In this section, unless otherwise specified, we set the default number of relevant events n to be 150 for the diabetes and mobile system data, 300 for the oil well data, and 400 for the NY taxi data.

In the first set of experiments, we evaluate the GetTopRelevantEvents algorithm that retrieves the most relevant events (w.r.t. the target events) which are any possible combinations of the basic events. Since our algorithm is a one-pass stream algorithm, to evaluate the processing speed, we adopt the conventional approach of measuring the *throughput* of the algorithm, i.e., how many stream tuples it can handle per second.

We first run the GetTopRelevantEvents algorithm using the diabetes dataset and varying the number of target events. The results are shown in Figure 37 for the mobile system dataset, and in Figure 38 for the oil well dataset (the results of other two datasets show similar trends and are omitted). In order to understand the impact of the A* search to performance, we also run a version of the algorithm skipping the A* search part only. We can see that A* search slightly decreases the throughput by a small percentage, for both datasets. Moreover, the throughput slightly decreases as the number of target events increases. This is because the algorithm needs to proportionally handle more events and candidate intermediate events.

Then we examine the distribution of the discovered top relevant events, in what we call the *cardinality*, which is the number of basic events that a discovered relevant event comprises. The cardinality distribution of the top events is shown in Figure 39 for the diabetes dataset, and in Figure 40 for the mobile system dataset. We can see that, for the diabetes data, the top relevant events have the highest fraction of cardinality 2 (the next one is 3), while cardinality 4 has the highest fraction for the mobile system data.

The above result reflects a tradeoff in the cardinality of a relevant event. Having too few basic events gives a higher "tf" part of the tf-idf, since individual basic events are more likely to appear in the context of target events; however, that would also result in a lower idf as it also appears frequently in the general stream context. In the other extreme, a very-high-cardinality event will have a greater idf but a very low tf.

In the next set of experiments, we examine the performance of building event-order graphs. Our algorithm BuildEventOrderGraph is again a one-pass stream algorithm, and we use throughput to uniformly measure the performance. The results are shown in Figure 41 for the diabetes data and Figure 42 for the NY taxi data. Recall that BuildEventOrderGraph does not include all edges that lead to *non-target* nodes, but keeps a reservoir sample of

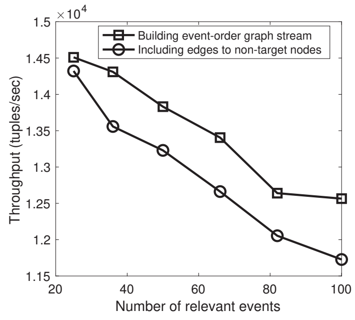


Figure 41: Building graph (diabetes)

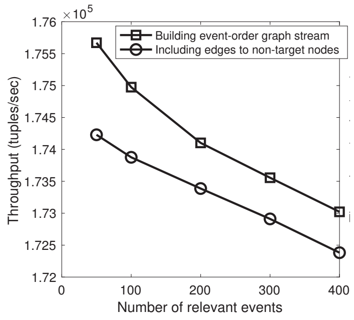


Figure 42: Building graph (NY taxi)

them. We compare the performance with a variant of the algorithm that includes in the graph all edges to the non-target nodes as well.

Figures 41 and 42 show that the performance slightly decreases as the number of relevant events increases. However, since we are dealing with the top events that are relevant to the target events, the number of such events does not need to be high to achieve an equivalent prediction accuracy, as found in our subsequent experiments. Including all edges to non-target nodes also slightly decreases the performance for building the graph, compared to discarding many such edges but only keep a uniformly random sample as in the reservoir sampling. Another interesting fact is that the throughput with the NY taxi data is in general higher than the diabetes data. This is because we aggregate the tuples of every five minutes to obtain the events of the NY taxi data, and hence the overall throughput is higher.

Our next set of experiments is concerned with a key step, which is to train the embedding vectors and the attention parameters of our event-order graph. We first show the throughput performance of the training, as shown in Figure 43 for the oil well data and Figure 44 for the NY taxi data. Recall that our pipeline of building dynamic graph and learning embedding is dynamic and incremental over each sliding window of size w. For training, this means that we keep sampling *(active set, r_1 or r_2, target event)* triples (and the

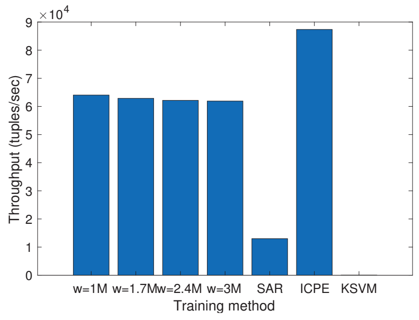


Figure 43: Training model (oil well)

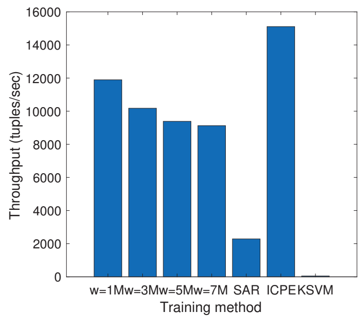


Figure 44: Training model (NY taxi)

associated negative samples) from the current window and performing stochastic gradient descent (SGD) until convergence. Thus, we measure the throughput of training with varying window sizes, ranging from 1M tuples to 3M tuples per window for the oil well dataset and 1M to 7M tuples for the NY taxi dataset. Later on we will also examine the impact of window size on prediction accuracy.

In addition, we also examine the training speed of the three baseline methods, SAR, ICPE, and KSVM. Note that, unlike our method which incrementally updates the embedding over sliding windows, the training methods of baseline ones are not continuous but work in batch—but we still report them in the form of throughputs for comparison.

Figures 43 and 44 show that the training throughput slightly decreases as we increase window size w (but still remains high). This is because, as mentioned above, our incremental embedding samples the triples in the current window and performs SGD until convergence. Increasing w may slightly decrease convergence speed because more tuples will likely exhibit more variable latent features; however, this variability tends to be less as we further increase the window size as it approaches more global stability. In fact, as shown in the experiments later, a larger window size does not necessarily translate to better prediction accuracy after some point.

Among the three baseline methods, sequential association rule mining (SAR) is slower than our method, while ICPE is faster and KSVM is the slowest in training. ICPE is faster due to its simplistic data structures and algorithms; however, as shown later, its prediction accuracy is the worst. Furthermore, off-the-shelf classification methods such as ICPE and KSVM are not designed for predicting the timing of *future* events in a future tuple—instead, they are designed for predicting the unknown class attribute in the *current* tuple. To use ICPE or KSVM, we have to couple the current tuple with the r_1 or r_2 relationship to a target event in a *future* tuple.

Note that, as shown in Figure 51 later, the actual prediction using the learned embedding vectors only incurs simple calculation and has a negligible cost—less than 10 microseconds per prediction. Since continuous training is the bottleneck of the stream processing pipeline, we find that the overall system throughput for graph building, embedding training, and event prediction is about the same as that from embedding training, which ranges from over one thousand to sixty thousand tuples per second for the four datasets. Such high efficiency makes possible preventive and predictive interventions, as well as predictive complex event processing (e.g., under resource constraints) [102–104], as discussed.

We then look into the *loss function* values (i.e., the f function values in line 7 of StateBasedEmbedding) after each *epoch*, which is defined as a uniform random sample of edges of the same size as the total number of edges in the graph, combined with an equal number of negative sample edges, following the terminology in the TransE algorithm [73] that we adopt and extend. We show the results in Figure 45 for the diabetes data and Figure 46 for the mobile system data (the other two datasets show a similar convergence).

From Figures 45 and 46, we can see that, interestingly, the loss function value sharply

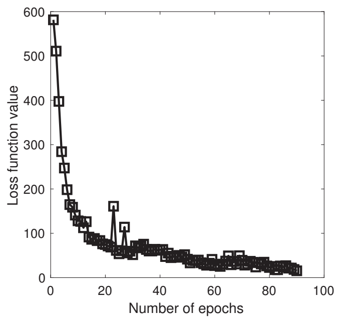


Figure 45: Loss function (diabetes)

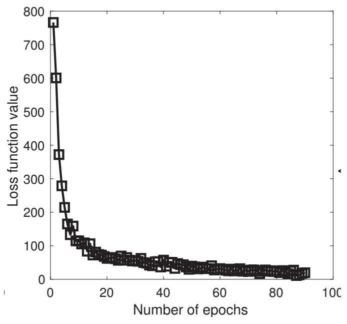


Figure 46: Loss function (mobile)

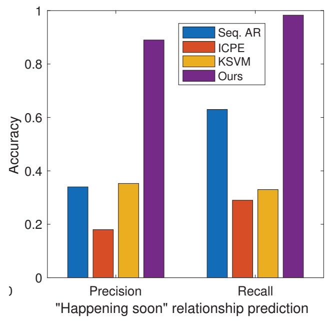


Figure 47: Accuracy (mobile, r_1)

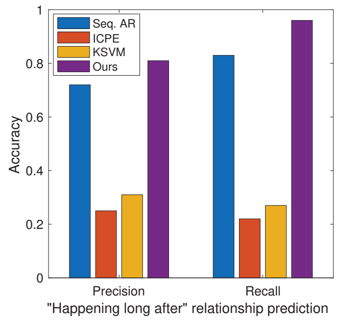


Figure 48: Accuracy (mobile, r_2)

decreases after each of the initial epochs of the embedding, where we perform stochastic gradient descent optimization over each value in the embedding vectors and the attention values for each ephemeral edge between an event node in the active set and a target event node. After 40 to 50 epochs, the loss function values in both figures level off, and converge at the lowest by around 80 to 90 epochs. The convergence is slightly faster with the mobile system dataset.

After observing the training process, we next examine the event prediction accuracy. The prediction tests are run over the period after the current window. We predict both the r_1 relationship ("happening soon") and the r_2 relationship ("happening long after"). We show the results in Figures 47 and 48 (for r_1 and r_2, respectively) for the mobile dataset, and in Figures 49 and 50 for the oil well dataset.

Since our work is to predict discrete events over multiple-attribute data streams in

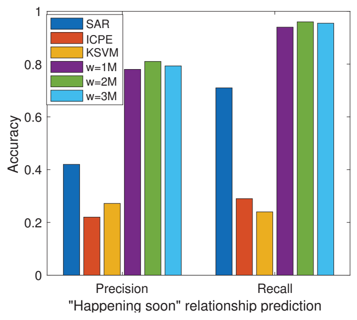


Figure 49: Accuracy (oil well, r_1)

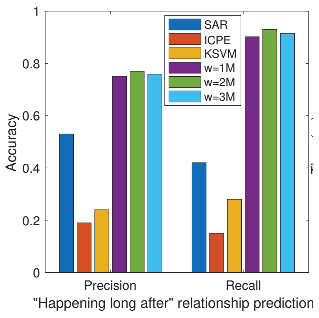


Figure 50: Accuracy (oil well, r_2)

real time, the closest previous work is mining sequential association rules [11] and using the rules to predict. We first mine all the rules that have the target events (to be predicted) on the right hand side, for both r_1 and r_2. Then for a target event e to be predicted using an active set A of events, we identify all the rules that have e on the right hand side, and compute the similarity between its left hand side and the active set A—combining this and the confidence of the rules gives us the prediction of r_1 or r_2. As discussed earlier, ICPE and KSVM are not designed for predicting the timing of *future* events (in a future tuple), but for predicting the unknown class attribute in the *current* tuple. To use ICPE and KSVM, we have to extend the current stream tuple with its r_1 or r_2 relationship with a target event in a *future* tuple, treating it as the class attribute. We need to add one class attribute for each target-event and r_1 or r_2 combination.

For accuracy, we measure both *precision* and *recall* [11]. We first parse the test data (a window of stream after the training period) and for each distinct active set of events, we record the set of target events that have r_1 or r_2 relationships with it. Using this as the ground truth, we calculate the precision and recall values when using our trained embedding vectors and attention values for prediction. First, Figures 47 and 48 over the mobile dataset show that our method is much more accurate than the three baseline methods for this problem, achieving good precision and recall values ranging from around 0.8 to nearly 1. ICPE and KSVM are not designed for this future event timing prediction problem, and they do not capture very well the timing relationships of event co-occurrence, following by a short interval, and following by a large interval. The accuracy of ICPE is the worst due to its simplistic data structures and algorithms.

In Figures 49 and 50 with the oil well dataset, we further show the impact of sliding window size on prediction accuracy. We find that, while initial increase of window size can improve precision and recall accuracy, further increase beyond a certain point actually decreases the accuracy. This is because the more recent data is more accurate for training the *current* model; data more ancient in the history may distract the training process in learning the current latent features.

Last but not least, we observe that, interestingly, the recall accuracy values are in general slightly higher than the precision values, for both r_1 and r_2 relationships. The reason is as follows. Just as knowledge graphs are generally *incomplete* [5], what we observe in the test window as "ground truth" data is generally incomplete (i.e., the "fact" just has not happened yet). Therefore, our method's *recall* value is relatively higher, since we (almost always) correctly tell that those relationships in the ground truth should be there. But since the "ground truth" (test data window) may miss some real targets for a relationship, while those targets may be correctly returned by our model, the *precision* of our method using the "ground truth" slightly suffers.

Finally, we implement the computation of the data-dependent accuracy bounds using Rademacher complexity, where we set the ϵ parameter to be 0.05. This involves solving

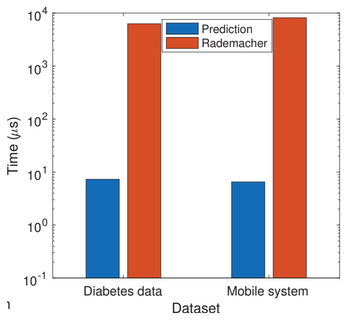


Figure 51: Prediction and bounds

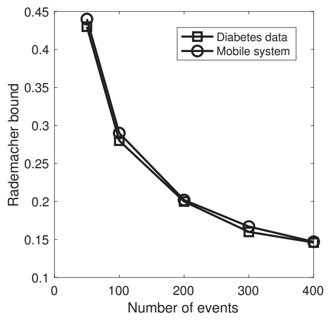


Figure 52: Rademacher bounds

a convex optimization problem to get $\min_{r \in R^+} w(r)$. We implement gradient descent for that purpose. In Figure 51, we show the overhead of making a prediction and that of computing the Rademacher bound side by side for diabetes and mobile system data (the other two datasets have very similar results). We can see that making a prediction is very fast (as the model of embedding is already trained), in a few *microseconds*, while computing the Rademacher bound is longer (mostly due to the gradient descent optimization)—but it is still only a few *milliseconds*. In Figure 52, we show the computed average Rademacher bounds as a function of the number of event nodes used. The error bound decreases as the sample size increases, and the average bounds from the two datasets are close. Note that these bounds are theoretical guarantees and tend to be more conservative. As shown earlier, in practice, we often get better accuracy. Moreover, the bound here is the error in predicting the exact probability. In practice, it often suffices to make relative judgements, e.g., which target event between the two is more likely to happen soon, or will this event more likely happen soon or long after now, which can be based on exact probabilities but are more robust to exact-probability errors.

4.3.3 Summary of Event Timing Prediction Experiment Results

The experimental results show that learning the top relevant events for a set of target events using A* search is quite efficient, and the cardinality (number of basic events) of the top events typically ranges from 2 to 5. We have also evaluated the efficiency of building the event-order graphs and of training the embedding vectors and attention values. We have clearly observed the fast convergence of the loss function value after around 50 epochs during the training. The overall system throughput for graph building, embedding training, and event prediction ranges from over one thousand to sixty thousand tuples per second for the four datasets. Using our trained model for event timing relationship (r_1 or r_2) prediction is generally accurate, with precision and recall values ranging from around 0.7 to nearly 1, much higher than those of the three baseline methods, some of which are off-the-shelf classification methods not specifically designed for our future event timing prediction problem.

4.3.4 Critical Event Monitoring Datasets and Experiment Setup

We use the Oil well data [98, 99] and NY Taxi data [2] same as in event timing prediction. Additionally, we add **(1) Hospital data**. This dataset is a real-life event log taken from the Dutch Academic Hospital [13]. The log contains events related to the treatment and diagnosis steps for a heterogeneous mix of patients with cancer pertaining to the cervix, vulva, uterus, and ovary. Attributes include event name, treatment code, diagnosis, date and time, specialism code, activity code, department/lab, and age. The dataset is about 73 MB. **(2) Grocery data**. This is an online grocery shopping dataset [105] that contains information about orders, products, aisles, and departments. The attributes include order ID, user ID, day of the week, hour of the day, days since prior order, product ID, order added to cart, whether this is an re-order of the product for the user, product name, aisle, and department. The preprocessed dataset is 3.6 GB.

We implement all the algorithms presented in this paper in Java. In particular, we significantly extend the code of TransE [73] to work with our loss function in Equations (1) and (2) with the attention mechanism. In addition, we compare against **two baseline approaches**. **(1)** The first one is IL-Miner [106], which is closest to our work. **(2)** From the machine learning literature, we find that the state-of-the-art *early prediction* of sequential events and time series, particularly in the domains of our dataset applications—hospital emergent events [107], market sales [108], taxi demand [109], and oil well emergencies [110]—is all based on the Long Short Term Memory (LSTM) model [111]. We implement such an approach for each dataset domain. For this baseline we use Python 3.6.9, Keras version 2.3.1 with Tensorflow version 1.14.0 backend. The Keras LSTM has a hidden layer size of 500 and a learning rate of 0.001, using the Adam optimizer [112] with the mean absolute error (MAE) loss function. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

4.3.5 Critical Event Monitoring Experimental Results

In the first set of experiments, we evaluate and verify the efficiency and quality of our continuous representation learning algorithm. For the hospital dataset, each individual patient's data is a substream, and there are 1143 substreams. We select a few event names that

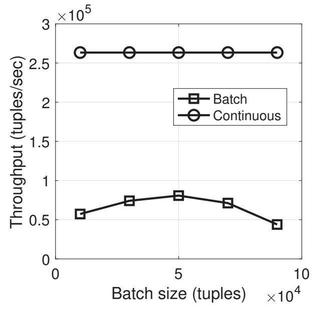


Figure 53: Embed. throughput (hospital)

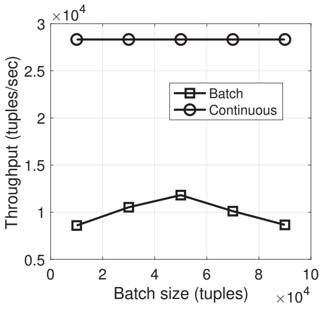


Figure 54: Embed. throughput (grocery)

have the Dutch word "spoed" in them (translated to "emergency" in English; this dataset is in Dutch) as *critical events*, such as "kalium vlamfotometrisch - spoed" (i.e., "potassium flame photometric - emergency" in English) and "haemoglobine foto-electrisch - spoed" (i.e., "hemoglobin photoelectric - emergency" in English). For the grocery dataset, each customer's data is a substream, and we arbitrarily pick buying "Organic Strawberries" and buying "Organic Whole Milk" as critical events (which may be of interest to the producers of those products). Likewise, for the NY taxi dataset, each taxi's trip data is a substream. We partition the latitude and longitude ranges of NYC into 8-by-8 grids, and treat dropping off or picking up at a particular geographic (grid) area as a critical event. Our algorithm incrementally updates existing embedding vectors as tuples arrive. The throughput of the stream system for handling this should be high, especially if the stream is stable and there is little change in embedding. To verify this, we compare against the "throughput" of learning on the *first batch* of stream tuples. We first use the hospital data with the result in Fig. 53. For initial batch learning, we vary the batch size in the number of stream tuples. The system throughput is measured under the convergence of the stochastic gradient descent optimization in line 12 of ContinuousRepresentationLearning. The learning is very efficient, with throughput 3 to 5 times of that of the initial batch embedding. For initial batch learning,

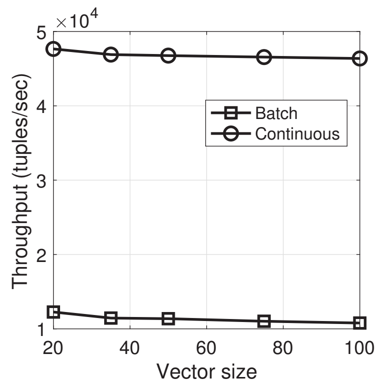


Figure 55: Embed. throughput (taxi)

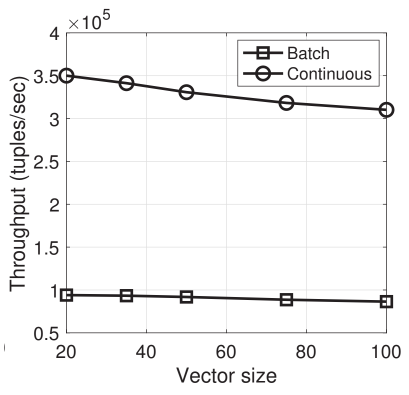


Figure 56: Throughput (3W)

as batch size increases, first the throughout slightly increases and then slightly decreases. The intuition is that a larger batch, when the underlying latent features (of embedding) are consistent, will be more efficient in terms of number of tuples per unit time; however, as batch size further increases, the latent feature values may change and the learning would have more overhead, thus causing the throughput to decrease.

We also repeat the same experiment with the grocery dataset and the NY taxi dataset. Fig. 54 shows the result for the grocery data (the results for the taxi and 3W data show a similar trend and is omitted). We can see that it has a similar pattern as the hospital data, except that the throughput difference between continuous and initial batch learning is slightly less—around 3 times. The intuitive reason is that people's grocery shopping trajectories are more diversified and dynamic, compared to the treatment procedures of cancer patients in the hospital data. Thus, continuous embedding needs to keep adjusting to the latent feature (i.e., vector) changes. We then also vary another parameter, the size of embedding vectors, ranging from 20 to 100, and show the throughput results of the representation learning in Fig. 55 for the NY taxi data, and in Fig. 56 for the 3W dataset. For initial-batch learning, the batch sizes are chosen that give the highest throughputs. First, we see that the through-

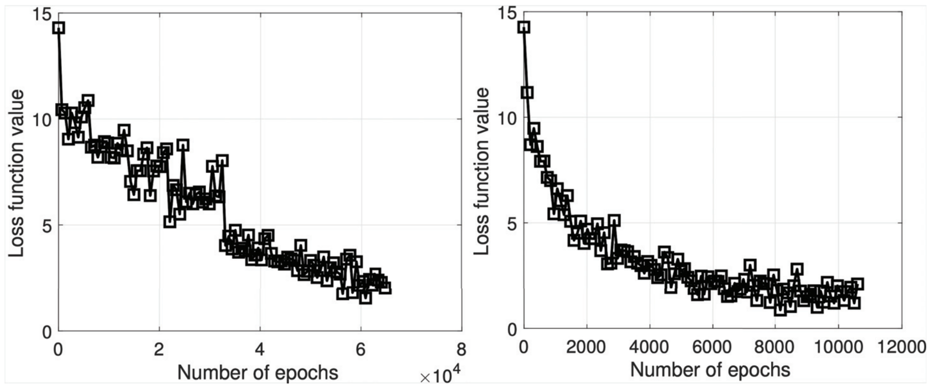


Figure 57: Loss function value (grocery, NY taxi)

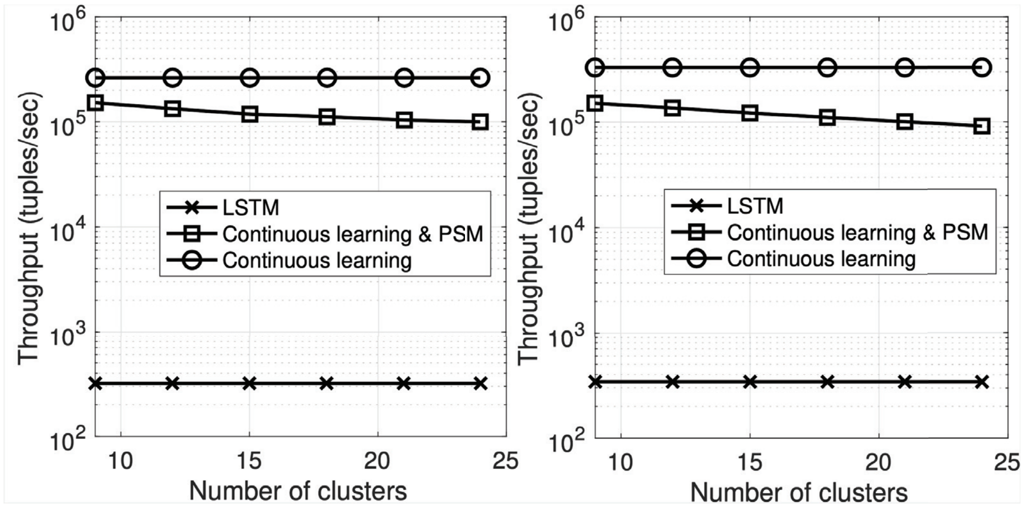


Figure 58: Learning throughput (hospital, 3W)

put difference between continuous learning and initial-batch learning is 4 to 5 times for the NY taxi data. Second, in both figures, the throughputs slightly decrease as the size of embedding vectors increases. This is because more overhead is incurred for doing stochastic gradient descent over larger vectors. Next we evaluate the quality of our continuous learning. One direct measurement is to examine how the loss function value in Equation (1) that we optimize changes during the run of the algorithm (we will also do other evaluations shortly, after we learn the probabilistic state machine patterns). Fig. 57 shows the result for the grocery data and the NY taxi data. Here, we define the notion of an *epoch* as: (1)

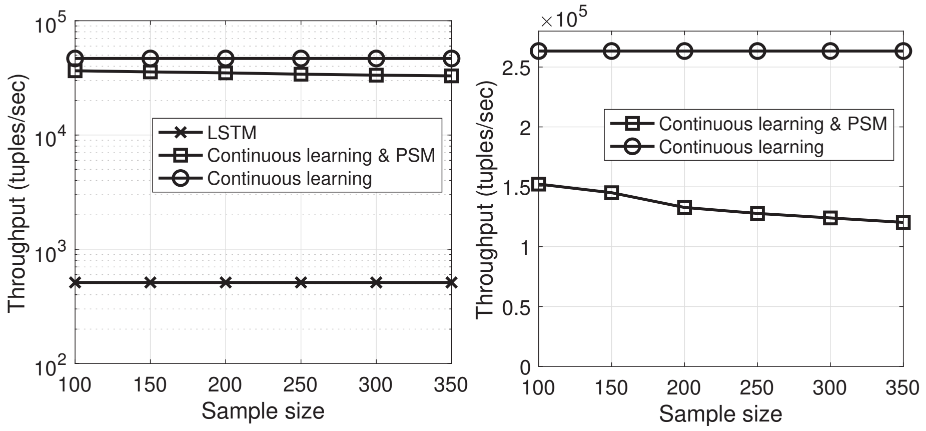


Figure 59: Throughput (NY taxi, hospital)

Sample a timing edge with relationship r_t between two tuples t_1 and t_2 uniformly at random from the graph; (2) Update the embeddings and attentions with respect to Equation 10 for all edges and nodes of t_1 and t_2, including attribute relationships and entity nodes (along with the corresponding negative samples). From the figures, it is clear that the loss function value of Equation 10 decreases until it converges at a low value.

We first vary the number of clusters (while fixing the sample size parameter $l = 200$, the results of which are shown in Fig. 58 for the hospital dataset and the 3W dataset.

We compare two versions: continuous representation learning and PSM building as described above, versus continuous representation learning only (without building PSM). There is some overhead associated with building PSM patterns for critical events. Nevertheless, the overall system throughput is still quite high. Moreover, when the number of clusters parameter increases, the overall throughput slightly decreases. This is because the GenerateStateMachine algorithm has to iterate over more clusters and learn a larger state transition matrix. We will examine the optimization of the number of cluster parameter shortly below.

In addition, we compare against the throughput of training the baseline LSTM model

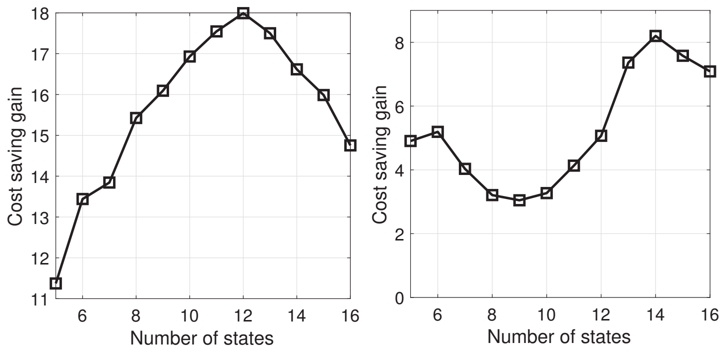


Figure 60: Cost saving gains (hospital, taxi)

for early prediction. As in previous work [107], the model is trained with data lagging the input sequence by X time units, where X is the amount of time we wish to predict the critical events in advance. The throughput of continuous LSTM training is also displayed in Fig. 58. We can see that throughput of our model training is over two orders of magnitude higher than that of the LSTM model. This is expected as the training of a deep learning model is much more computationally expensive, and it is indeed one of our motivations for using a multi-relational graph representation learning method with significantly less overhead for real-time fast streams.

Then we vary the sample size parameter l, while fixing the number of clusters to 12; Fig. 59 shows the result for the NY taxi dataset and the hospital dataset. Again, the overall throughput of combining continuous representation learning and continuous PSM learning is still quite high. When the sample size l increases, the system throughput slightly decreases since GenerateStateMachine needs to go through more iterations to estimate the transition probabilities. For the NY taxi data, in Fig. 59, we compare against the throughput of the continuous training of the LSTM model, which also shows that the throughput of our model training is about two orders of magnitude higher than that of the LSTM model.

We now study the selection of the number of clusters/states in the PSM patterns for critical events. The result is shown in Fig. 60 for the hospital dataset and the NY taxi dataset. As in the SelectStateNumber algorithm, for a sequence W in the real data stream, we create a baseline sequence W' for comparison that maintains the same set of events (nodes), the same degrees of each node, the same set of edges, and the same time edges, but the event-key and event-state edges are randomly shuffled.

For a random piece of length 10 in W and W', respectively, we calculate their costs by calling AlignTrajectory using the PSM we build with a specific number of states (clusters). We display the cost saving gains compared to using the baseline sequence W' under different numbers of states in the figures. From Fig. 13 we can see that the optimal number of states is 12 for the hospital dataset. For the NY taxi dataset, Fig. 60 shows that the optimal number of states is 14. In addition, the figure shows that there can be more than one peak point in the function. Note that although it is possible for a gradient ascent algorithm to settle at a local optimum, there are standard solutions to this problem, e.g., by using a variable step length in the iterations.

Interestingly, even though the clustering of states is based on the latent features in the embedding vectors, we find that the resulting states have distinctive semantics and interpretation. Specifically, for the hospital data, the PSM states correspond to treatment activities and diagnoses. For instance, one state corresponds to a condition of gamma glutamyl transpeptidase in the clinical chemistry lab. The learned *attentions* inform us of the importance (weights) of each attribute in a tuple in forming the states. For example, we find that "Treatment code" and "Diagnosis" attributes are more important (i.e., greater attentions) than the "Age" attribute. For 3W data, the states are clusters of oil well measurement con-

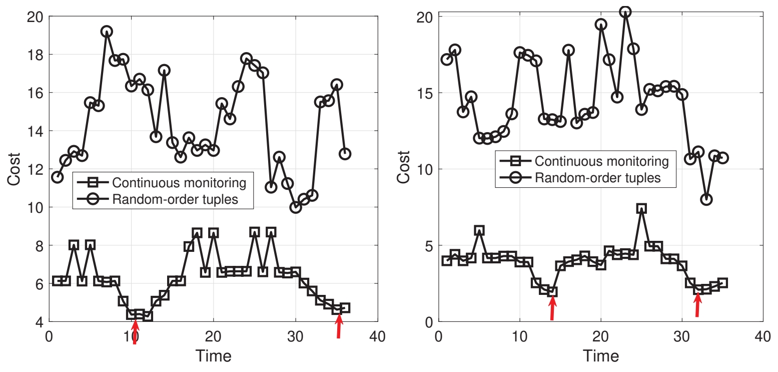


Figure 61: Cost over time (hospital) Cost over time (grocery)

ditions, with “temperature at TPT” and “pressure at JUS-CKGL” much more critical than “pressure at PDG” and “QGL” based on the learned attentions.

Likewise, for the grocery dataset, the PSM states indicate different categories of purchases. For instance, one state corresponds to beverages including various types of coffee, tea, and cream. From the learned *attentions*, we find that the “day of the week” (of the order) attribute and the “days since prior order” attribute have little importance compared to more descriptive attributes such as “product name” and “aisle”. For the taxi dataset, we find that a PSM state often corresponds to a grid area under our partition (as described earlier), such as the fifth longitude interval and the fourth latitude interval in one state. In addition, “trip time” and “trip distance” attributes tend to have much more weights than the “period of the day” attribute or whether the tuple is a pick-up or a drop-off. In all, PSM is an *interpretable* pattern structure for continuous monitoring and prediction of critical events, which is one of the major differentiators from a deep learning model.

Having learned the probabilistic state machine patterns for critical events, we now use them to monitor the critical events in the streams as discussed earlier. We also compare against the most relevant previous work, IL-Miner [106]. Recall that we compute the sum of two costs (i.e., consistency cost with stream history and projected future cost to reach the critical state) as the overall cost that indicates how far away the current stream state is away from the critical event. We display the result for the hospital dataset in Fig. 61 for the critical event "kalium vlamfotometrisch - spoed", and the result for the grocery dataset for the critical event of buying "Organic Strawberries" in Fig. 61. The substream ID of the sequence shown in Fig. 61 (hospital data) is 420, and the ID of the substream shown in Fig. 61 (grocery data) is 596. The small red arrows in the figures point to the locations in the curves where the critical events happen.

In the figures, we also show the comparison with a baseline sequence which has the same set of tuples but where the order of the tuples is randomly shuffled. Interestingly, we can see that, even though the baseline sequence has exactly the same set of tuples, its overall cost is much higher than and well separated from the real sequence that we continuously monitor. This is because of the first part of the cost—consistency cost with stream history, since the randomly shuffled sequence will destroy the state transition statistics learned in our PSM patterns. The second part of the cost—projected future cost to reach the critical state would not be significantly different for individual tuples even though they are reordered, as a tuple will be mapped to a state in the PSM pattern. This also demonstrates the necessity of having the first part of the cost. Figure 61 shows that we can nicely monitor and foresee the critical events by observing the fluctuations of the overall cost value. A critical event happens at the valleys of the curve. Depending on the applications, the corresponding planning actions, preventive measures, or interventions can be performed.

We now take a closer look and examine the accuracy of imminence monitoring. Recall that the LSTM baseline model is trained by lagging the input sequence X time units,

Table 3: Accuracy of various methods.

data	**LSTM prec.**	**LSTM' prec.**	**ILM. prec.**	**ours prec.**	**LSTM rec.**	**LSTM' rec.**	**ILM. rec.**	**ours rec.**
hosp.	0.93	0.87	0.98	0.90	0.94	0.89	0.36	0.93
3W	0.69	0.62	–	0.65	0.80	0.73	–	0.82
taxi	0.83	0.78	–	0.82	0.96	0.88	–	0.92

where X is the amount of time we wish to predict the critical events in advance. Thus, we define the accuracy metrics as follows. (1) *Recall*: Amongst all the true occurrences of the critical event, what fraction of them is correctly predicted by the model at any time in the interval $[(1-\delta)X,(1+\delta)X]$ prior to the event? (2) *Precision*: When the model makes a positive prediction, how often does the critical event actually happen in the time interval $[(1-\delta)X,(1+\delta)X]$ after the prediction?

We show the results in Table 3. The X values for the hospital, 3W, and taxi datasets are 7 minutes, 16 minutes, and 30 minutes, respectively, and we use $\delta = 0.15$. For our method, we determine the cost thresholds corresponding to time point X from the average value in the training data. We can see that the accuracy of our PSM model is competitive with or slightly worse than the LSTM model. Since the LSTM baseline model is trained by lagging the input sequence a pre-determined number of time units, we also examine the accuracy when there is a mismatch between the early prediction time interval parameter X and training time interval (we use $0.85X$), as shown in the LSTM' columns, which indicate a decrease of both precision and recall values.

It should be noted that, unlike our PSM model or IL-Miner, the model from LSTM is not an *interpretable pattern*, but merely an *opaque model*. Furthermore, LSTM models are trained with pre-determined fixed X values and are not flexible for *continuous* imminence monitoring as PSM is. Finally, as shown in Fig. 58 and Fig. 59, PSM models typically render two orders of magnitude higher throughputs than LSTM models, which is important for high-speed real-time stream monitoring.

We now compare with the closest previous work, IL-Miner [106]. One issue is that one has to know the strict pattern as specified in a simple language in order to locate the places in the sequence to train and learn the pattern, which, however, precludes the need to learn the pattern, had one already known it. If one does not know the pattern, as is the case in our datasets, but only knows the markers of critical events in the training data, then the *strictly-common* pattern, as required by IL-Miner, is always more complex than can be expressed in the simple language of IL-Miner. As a result, we cannot find any common pattern for the critical events in all three datasets. By contrast, our pattern language (PSM) is stochastic and much more flexible and powerful. Just for the sake of experiment, during pattern-learning using IL-Miner for the hospital data, we try each individual patient *separately* and only use a small number of occurrences of the critical event and only a small subset of the attributes (i.e., event name, diagnosis code, and diagnosis), we are able to find three *different patterns* for three of the patients, respectively. Note that we are not able to find any common patterns if we use more attributes. Then averaging over those three patients, we measure the precision and recall as shown in Table 3 (the ILM. columns). The low recall is due to its overly restrictive pattern language that cannot generalize well to unseen occurrences of critical events. By contrast, the results of our method and LSTM are over all patients and a complete set of attributes.

4.3.6 Summary of Critical Event Monitoring Experiment Results

The experimental results show that our continuous representation learning for data streams is efficient and effective in obtaining a converged loss function value that incorporates node aggregation and the attention mechanism. Moreover, we can also efficiently and continuously learn the probabilistic state machine patterns for given critical events. The PSM patterns are effective in the imminence monitoring of critical events. Our method has high precision and recall values. The utility and effectiveness are also demonstrated from comparisons with baseline approaches that are the closest previous work, namely IL-Miner and LSTM models.

5 CONCLUSIONS

We have seen graph streams increasingly common in big data analytics today. One important phenomenon, however, is that, graph streams are typically a partial presence of the whole dynamic graph state. Consequently, it is important to be able to answer queries on unobserved variables in the graph, given the observed ones. To address this, we introduce VTeller, a probabilistic graphical model with latent variables that captures both variable correlations and time-variant patterns. We also present algorithms for vertex grouping, model learning, and query inference based on dynamic Bayesian networks and belief propagation, as well as a non-intrusive incremental model update algorithm. Our experiments on real-world datasets demonstrate the effectiveness and efficiency of our approach.

To efficiently and accurately answer top-k entities and aggregate queries over a virtual knowledge graph, we propose an incremental index scheme based on knowledge graph embedding. We transform embedding vectors to a lower-dimensional space for indexing and prove a tight bound on the accuracy guarantees. Our query processing algorithms and analysis of result accuracy show that our cracking index is very efficient in answering queries with accuracy guarantees, performing only a small fraction of the node splits compared to a full index.

To predict event timing information in multi-attribute data streams, we introduce a novel approach that uses a knowledge graph to represent event timing relationships and an ”active state” to characterize state information. With embedding vectors containing latent features, we achieve high precision and recall values in predicting event timing, outperforming a baseline approach. Our training uses only a small amount of streaming data and is suitable for data stream applications. Lastly, we address the problem of discovering subtle and complex event patterns prior to critical events from training data streams. Our approach transforms an arbitrary multi-stream into a dynamic knowledge graph, preserves relation information through lossless join decomposition, and employs attention mechanisms and embedding pooling/aggregation for embedding. We propose algorithms for learning probabilistic state machine patterns and a cost model for monitoring critical events and optimizing the pattern model parameter. Our experiments demonstrate the effectiveness and advantages of our method over previous work.

6 RECOMMENDATIONS

Part of the work in this dissertation is presented in our previous publications, and we refer to [113–116] it for more details.

In our work, we focus on three types of significant information to be predicted and queried in large dynamics graph data–attribute values, unobserved links/relationships, and event timing/critical events. There are several potential areas of future work that could be explored.

For the proposed approach of using probabilistic graphical models with layers of latent variables for answering queries on unobserved variables in graph streams, future work could focus on improving the efficiency and scalability of the model by exploring techniques such as parallelization and approximation methods. Additionally, exploring the potential of incorporating additional graph properties, such as edge weights, could lead to more accurate query results. Another direction for predicting event timing information in multi-attribute data streams, future work could focus on evaluating the robustness of the model to noisy or missing data, as well as exploring the potential of using deep learning techniques, such as recurrent neural networks, may be an avenue to consider.

Overall, there is significant potential for future work in these research directions, and continued exploration and development could lead to valuable advancements in the field of big data analytics.

7 REFERENCES

[1] Google inside search, 2018. Available at https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html.

[2] New york taxi data, 2017. Available at http://chriswhong.com/open-data/foil_nyc_taxi/.

[3] Maya Rotmensch, Yoni Halpern, Abdulhakim Tlimat, Steven Horng, and David Sontag. Learning a health knowledge graph from electronic medical records. *Scientific Reports*, 7, 2017.

[4] Qingyao Ai, Vahid Azizi, Xu Chen, and Yongfeng Zhang. Learning heterogeneous knowledge base embeddings for explainable recommendation. *Algorithms*, 11, 2018.

[5] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104, 2016.

[6] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[7] Streambase systems, 2019. Available at https://www.crunchbase.com/organization/streambase-systems\#section-overview.

[8] Oracle complex event processing: Lightweight modular application event stream processing in the real world. *An Oracle White Paper*, 2009.

[9] Ann E. Goebel-Fabbri, Nadine Uplinger, Stephanie Gerken, Deborah Mangham, Amy Criego, and Christopher Parkin. Outpatient management of eating disorders in type 1 diabetes. *Diabetes Spectrum*, 22(3):147–152, 2009.

[10] Jan G. de Gooijer and Rob J. Hyndman. 25 Years of IIF Time Series Forecasting: A Selective Review. (05-068/4), June 2005.

[11] Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. *Introduction to Data Mining (2Nd Edition)*. Pearson, 2nd edition, 2018.

[12] Nikos Giatrakos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. Complex event recognition in the big data era. *Proceedings of the VLDB Endowment*, 10(12):1996–1999, 2017.

[13] RP Jagadeesh Chandra Bose and WMP van der Aalst. Analysis of patient treatment procedures: The BPI challenge case study. *BPM reports*, 1118, 2011.

[14] Jason W. Osborne. Dealing with missing or incomplete data: Debunking the myth of emptiness. In *Best Practices in Data Cleaning: A Complete Guide to Everything You Need to Do Before and After Collecting Your Data*, 2013.

[15] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson Addison Wesley, 2006.

[16] J. Honaker and G. King. What to do about missing values in time-series cross-section data. In *American Journal of Political Science*, volume 54, April 2010.

[17] Nicholas Horton and Ken Kleinman. Much ado about nothing: A comparison of missing data methods and software to fit incomplete data regression models. *The American Statistician*, 61(1):79–90, February 2007.

[18] Z. Ghahramani and M. Jordan. Supervised learning from incomplete data via an em approach. In *NIPS*, 1993.

[19] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stan Zdonik. The case for predictive database systems: Opportunities and challenges. In *CIDR*, 2011.

[20] Songyun Duan and Shivanath Babu. Processing forecasting queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 711–722. VLDB Endowment, 2007.

[21] Myunghwan Kim and Jure Leskovec. The network completion problem: Inferring missing nodes and edges in networks. In *Proceedings of the Eleventh SIAM International Conference on Data Mining*. SIAM, 2011.

[22] Amol Deshpande, Lise Getoor, and P. Sen. Graphical models for uncertain data. In Charu Aggarwal, editor, *Managing and Mining Uncertain Data*, pages 77 – 77. Springer, 2009.

[23] Daisy Zhe Wang, Eirinaios Michelakis, Minos Garofalakis, and Joseph M. Hellerstein. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. *Proc. VLDB Endow.*, 1(1):340–351, August 2008.

[24] Amol Deshpande, Carlos Guestrin, and Samuel Madden. Using probabilistic models for data management in acquisitional environments. *CIDR*, pages 317–328, 01 2005.

[25] Yasuko Matsubara, Yasushi Sakurai, and Christos Faloutsos. Autoplait: Automatic mining of co-evolving time sequences. SIGMOD '14, pages 193–204, 2014.

[26] Shai Fine, Y. Singer, and N. Tishby. The hierarchical hidden Markov model: Analysis and applications. *Machine Learning*, 32:41–62, 1998.

[27] Mert Akdere, Ugur Çetintemel, and Eli Upfal. Database-support for continuous prediction queries over streaming data. *Proc. VLDB Endow.*, 3(1-2):1291–1301, September 2010.

[28] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

[29] Charu C. Aggarwal, Yao Li, Philip S. Yu, and Ruoming Jin. On dense pattern mining in graph streams. *Proc. VLDB Endow.*, 3(1-2):975–984, September 2010.

[30] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. Event pattern matching over graph streams. *Proc. VLDB Endow.*, 8(4):413–424, December 2014.

[31] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. SIGMOD '16, pages 1481–1496, 2016.

[32] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal of Semantic Web Information Systems*, 2009.

[33] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. 2014.

[34] B. Shaw and T. Jebara. Structure preserving embedding. 2009.

[35] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. 2014.

[36] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. 2016.

[37] Y. Jia, Y. Wang, H. Lin, X. Jin, and X. Cheng. Locally adaptive translation for knowledge graph embedding. 2016.

[38] L. Yao, Y. Zhang, B. Wei, Z. Jin, R. Zhang, Y. Zhang, and Q. Chen. Incorporating knowledge graph embeddings into topic modeling. 2017.

[39] A. Bordes, J. Weston, R. Collobert, and Y. Bengio. Learning structured embeddings of knowledge bases. 2011.

[40] G. Ji, S. He, L. Xu, K. Liu, and J. Zhao. Knowledge graph embedding via dynamic mapping matrix. 2015.

[41] M. Y. C. Z. Muhao Chen and Yingtao Tian. Multilingual knowledge graph embeddings for cross-lingual knowledge alignment,. 2017.

[42] Y. Zhao, Z. Liu, and M. Sun. Representation learning for measuring entity relatedness with rich information. 2015.

[43] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. 2007.

[44] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. 2013.

[45] Shashi Shekhar and Sanjay Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.

[46] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, 1986.

[47] I. Fodor. *A survey of dimension reduction techniques*. Center for Applied Scientific Computing, Lawrence Livermore National, Technical Report, 2002.

[48] E. Bingham and H. Mannila. Random projection in dimensionality reduction. 2001.

[49] A. Rajaraman and J. Ullman. *Mining of Massive Datasets*. 2010.

[50] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of a theorem of Johnson and Lindenstrauss. *Random Structures and Algorithms*, 22, 2003.

[51] Qiang Huang, Guihong Ma, Jianlin Feng, Qiong Fang, and Anthony K. H. Tung. Accurate and fast asymmetric locality-sensitive hashing scheme for maximum inner product search. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '18, pages 1561–1570, New York, NY, USA, 2018. ACM.

[52] Jan G. De Gooijer and Rob Hyndman. 25 years of iif time series forecasting: A selective review. *SSRN Electronic Journal*, 01 2005.

[53] P. Brockwell and R. Davis. *Introduction to Time Series and Forecasting.(2Nd Edition)*. 2002.

[54] R.L. Eubank. *A Kalman filter primer*. 01 2005.

[55] Moshe Vardi. Fundamentals of dependency theory. *Foundations and Trends in Theoretical Computer Science - FTTCS*, 01 1987.

[56] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 647–658, New York, NY, USA, 2004. ACM.

[57] Paul Brown and Peter J. Haas. Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *VLDB*, 2003.

[58] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[59] Louis Woods, Jens Teubner, and Gustavo Alonso. Complex event detection at wire speed with fpgas. *Proc. VLDB Endow.*, pages 660–669, September 2010.

[60] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[61] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[62] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. 01 2009.

[63] J. Rissanen. Automatica. In *Modeling by shortest data description*, volume 14, pages 465–658, 1978.

[64] Judea Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the Second AAAI Conference on Artificial Intelligence*. AAAI Press, 1982.

[65] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Graph Partitioning*. arXiv:1311.3144, 2013.

[66] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 120–124. ACM, 2004.

[67] N. Alon and J. Spencer. *The Probabilistic Method*. New York: Wiley, 1992.

[68] Peter J. Denning. The locality principle. *Communication Networks and Computer Systems*, 2006.

[69] A. A. Borovkov. *Mathematical Statistics*. CRC Press, 1999.

[70] John Boaz Lee, Ryan A. Rossi, Sungchul Kim, Nesreen K. Ahmed, and Eunyee Koh. Attention models in graphs: A survey. *CoRR*, 2018.

[71] Jeannette Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. 1952.

[72] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, pages 37–57, March 1985.

[73] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc., 2013.

[74] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, 2013.

[75] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[76] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Ed.)*. Prentice Hall, 2009.

[77] KVSVN Raju and Arun K Majumdar. Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems. *ACM Transactions on Database Systems (TODS)*, 13(2):129–166, 1988.

[78] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, ICIR, 2018.

[79] Tapas Kanungo, David Mount, Nathan Netanyahu, Christine Piatko, Ruth Silverman, and Angela Wu. An efficient k-means clustering algorithm analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:881–892, 07 2002.

[80] Vladimir Koltchinskii. Rademacher penalties and structural risk minimization. *IEEE Transactions on Information Theory*, 47, 2001.

[81] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, 1999.

[82] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, 16(2):264, 1971.

[83] Matteo Riondato and Eli Upfal. Abra: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. *ACM Transactions on Knowledge Discovery from Data*, 0, 2018.

[84] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[85] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[86] Hong kong traffic data, 2017. Available at `https://data.gov.hk/en-data/dataset/hk-td-tis-traffic-speed-map`.

[87] G. W. Brier. Verification of Forecasts Expressed in Terms of Probability. *Monthly Weather Review*, 78:1, 1950.

[88] James Haworth and Tao Cheng. Non-parametric regression for space-time forecasting under missing data. *Computers, Environment and Urban Systems*, 36:538–550, 2012.

[89] Freebase data, 2013. Available at `https://developers.google.com/freebase/`.

[90] Movielens data, 2017. Available at `https://grouplens.org/datasets/movielens/latest/`.

[91] Amazon data, 2019. Available at `http://jmcauley.ucsd.edu/data/amazon/`.

[92] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, pages 2787–2795. Curran Associates, Inc., 2013.

[93] Tilmann Zaschke, Christoph Zimmerli, and Moira C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. 2014.

[94] Qiang Huang, Guihong Ma, Jianlin Feng, Qiong Fang, and Anthony K. H. Tung. Accurate and fast asymmetric locality-sensitive hashing scheme for maximum inner product search. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '18, pages 1561–1570, New York, NY, USA, 2018. ACM.

[95] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[96] Diabetes data, 2019. Available at `http://archive.ics.uci.edu/ml/datasets/Diabetes`.

[97] Nilanjan Banerjee, Ahmad Rahmati, and Mark D. Corner1. Users and batteries: Interactions and adaptive energy management in mobile systems. 2007.

[98] Oil wells dataset, 2020. Available at `https://github.com/ricardovvargas/3w_dataset`.

[99] Ricardo Emanuel Vaz Vargas, Celso Jos□ Munaro, Patrick Marques Ciarelli, Andr□ Gon□alves Medeiros, Bruno Guberfain do Amaral, Daniel Centurion Barrionuevo, Jean Carlos Dias de Ara□jo, Jorge Lins Ribeiro, and Lucas Pierezan Magalh□es. A realistic and public dataset with rare undesirable real events in oil wells. *Journal of Petroleum Science and Engineering*, 181:106223, 2019.

[100] Miao Wang, Viliam Holub, John Murphy, and Patrick O'Sullivan. Stream-based event prediction using bayesian and bloom filters. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, page 217–228, New York, NY, USA, 2013. Association for Computing Machinery.

[101] Chan Nam, Man Yiu, and Hou Leong. Karl: Fast kernel aggregation queries. pages 542–553, 04 2019.

[102] Syed Gillani, Abderrahmen Kammoun, Kamal Singh, Julien Subercaze, Christophe Gravier, Jacques Fayolle, and Frederique Lafores. Pi-cep: Predictive complex event processing using range queries over historical pattern space. *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2017.

[103] L. Fulop, A. Beszedes, G. Toth, H. Demeter, L. Vidacs, and L. Farkas. Predictive complex event processing: a conceptual framework for combining complex event

processing and predictive analytics. *Proceedings of the Fifth Balkan Conference in Informatics*, 2012.

[104] Zheng Li and Tingjian Ge. History is a mirror to the future: Best effort approximate complex event matching with insufficient resources. *Proceedings of the VLDB Endowment (PVLDB journal) and 43rd International Conference on Very Large Data Bases (VLDB 2017)*, 2017.

[105] The instacart online grocery shopping dataset, 2017. Accessed from https://www.instacart.com/datasets/grocery-shopping-2017.

[106] Lars George, Bruno Cadonna, and Matthias Weidlich. IL-Miner: Instance-level discovery of complex event patterns. *Proceedings of the VLDB Endowment*, 10(1):25–36, 2016.

[107] Josef Fagerstrom, Magnus Bang, Daniel Wilhelms, and Michelle S. Chew. LiSep LSTM: A machine learning algorithm for early detection of septic shock. *Scientific Reports, Nature Research*, 9(15132), 2019.

[108] Kasun Bandara, Peibei Shi, Christoph Bergmeir, Hansika Hewamalage, Quoc Tran, and Brian Seaman. Sales demand forecast in e-commerce using a long short-term memory neural network methodology. In *International Conference on Neural Information Processing*, 2019.

[109] J. Xu, R. Rahmatizadeh, L. Boloni, and D. Turgut. Real-time prediction of taxi demand using recurrent neural networks. *IEEE Transactions on Intelligent Transportation Systems*, 19(8):2572–2581, 2018.

[110] Yadigar N. Imamverdiyev and Fargana J. Abdullayeva. Condition monitoring of equipment in oil wells using deep learning. *Advances in Data Science and Adaptive Analysis*, 12(1), 2020.

[111] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[112] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[113] Yan Li, Tingjian Ge, and Cindy Chen. Vteller: Telling the values somewhere, sometime in a dynamic network of urban systems. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, page 577–586, New York, NY, USA, 2018. Association for Computing Machinery.

[114] Yan Li, Tingjian Ge, and Cindy Chen. Online indices for predictive top-k entity and aggregate queries on knowledge graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1057–1068, 2020.

[115] Yan Li, Tingjian Ge, and Cindy Chen. Data stream event prediction based on timing knowledge and state transitions. *Proc. VLDB Endow.*, 13(10):1779–1792, jun 2020.

[116] Yan Li and Tingjian Ge. Imminence monitoring of critical events: A representation learning approach. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1103–1115, New York, NY, USA, 2021. Association for Computing Machinery.