Continuous Graph Pattern Matching over Knowledge Graph Streams

Syed Gillani
Univ Lyon
UJM Saint-Étienne
CNRS
Laboratoire Hubert Curien
Saint-Étienne, France
syed.gillani@univ-st-etienne.fr

Gauthier Picard
Univ Lyon
MINES Saint-Étienne
CNRS
Laboratoire Hubert Curien
Saint-Étienne, France
picard@emse.fr

Frédérique Laforest
Univ Lyon
UJM Saint-Étienne
CNRS
Laboratoire Hubert Curien
Saint-Étienne, France
frederique.laforest@univ-st-etienne.fr

ABSTRACT

Continuous Graph Pattern Matching (CGPM) is an extended version of the traditional GPM that is evaluated over Knowledge Graph (KG) streams. It comes with additional constraints of scalability and near-to-real-time response, and is used in many applications such as real-time knowledge management, social networks and sensor networks. Hence, existing GPM solutions for static KGs are not directly applicable in this setting. This paper studies continuous GPM over KG streams for two different executional models: *event-based* and *incremental*. We first propose a query-based graph pruning technique to filter the unnecessary triples from a KG event. The pruned events are materialized in a set of vertically partitioned tables. We then use a hybrid *join-and-explore* technique to further prune and finally match the triples within a KG event. Considering the on-the-fly execution of queries over pruned KG events, we use an automata-based model to guide the join and exploration process. This leads to an index-free solution optimised for streaming environments. Experimental results with both synthetic and real-world datasets confirm that our system outperforms the state-of-the-art solutions by (on average) one to two orders of magnitude, in terms of performance and scalability.

Categories and Subject Descriptors

H.2.5 [**Information Systems**]: Database Management—*Heterogeneous Databases*[Query Processing]; E.1 [**Data**]: Data Structures—*Graphs and networks*; H.2.4 [**Information Systems**]: Database Management—*Query Processing*

Keywords

Knowledge graph, stream processing, incremental evaluation, graph pattern matching

DEBS '16, June 20-24, 2016, Irvine, CA, USA

DOI: http://dx.doi.org/10.1145/2933267.2933306

1. INTRODUCTION

Stream processing systems are critical in providing content to users and allowing organizations to make faster and better decisions, particularly because of their ability to provide low latency results. However, the heterogeneous nature of streams emanating from social networks, sensor networks, financial networks, etc., makes their use and integration with other data sources a difficult and labour-intensive task. The Knowledge Graph (KG)[1] data model provides solution to these problems by lifting stream data to a semantic model, where ontologies and background information are used to enrich streams [8, 24]. This allows an easy and seamless integration, not only among heterogeneous data sources, but also between data sources and Linked Data collections [9, 35]; thus enabling a new range of "real-time" applications.

The main properties of interest in existing KG studies (including the KG streams processing) are based on one fundamental operator: Graph Pattern Matching (GPM). Given a KG G and a query graph Q, the problem of graph pattern matching is to find all the possible subgraphs of G that match the set of patterns described in Q. This problem shares the same search space with pattern matching via subgraph isomorphism [19], which is NP-Complete. In practice it can be handled with join-based or exploration-based methods [28, 39]. In dynamic streaming settings, such a problem can be described as continuous GPM (CGPM), where KG streams ($\mathcal{G}$) are matched with a query graph Q in a continuous manner.

The two main execution models for CGPM over KG streams are: *event/batch-based* and *incremental* executions [18]. Both models share the same pattern matching strategy. However, an event-based model matches each batch of graphs/edges independently, while an incremental execution model [18] incrementally matches the incoming graphs/edges by utilising already computed matches. Both models have diverse and interesting use cases ranging from complex event process (CEP) [16] over sensor networks to stream processing over social networks. For instance, an event-based model is used for CEP, where each event is processed independently: sequence or aggregate operators are used to collect a set of required values from each event.

EXAMPLE 1. *Fig. 1(a,b) presents two* KG *events (G1 and G2) from the smart grid domain. Each event denotes a house with attributes such as location, type of appliances and their power con-*

[1]Various abstractions have been defined for KGs, however, in this paper, we utilise an RDF graph model [38] for its representation (see Section 2 for more details).

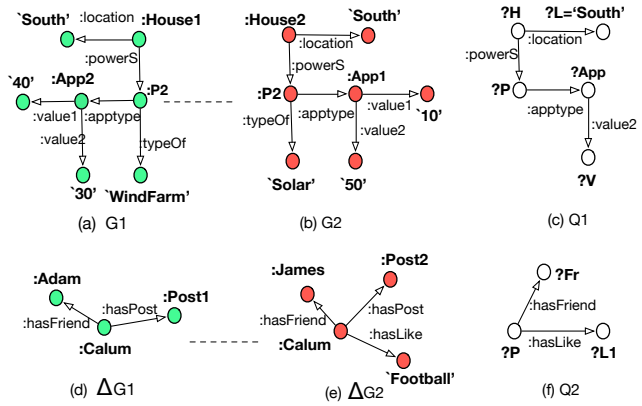

Figure 1: (a,b) KG-based events for event-based CGPM. (c) Query graph from events (a) and (b). (d,e) KG-based updates for Incremental CGPM. (f) Query graph for events (d) and (e).

sumption values. Fig. 1(c) depicts a query graph $Q1$ which when applied to streams of events ($G1$ and $G2$) finds all the subgraphs within each graph that are isomorphic to $Q1$. The execution of $Q1$ over $G2$ would result in the following matches: ((:House2,<:location>, <South>), (:House2,<:powerS>,:P2), (:P2,<:apptype>, :App1), (:App1,<:value2>,50)). The execution of $Q1$ over each KG *event has no dependency relation with the previous events. Fig. 1(d,e) presents two* KG *events ($\Delta G1$ and $\Delta G2$) from the social network domain. Each event denotes a person and its attributes such as friendship and likes, where $\Delta G2$ describes an update to the previously processed $\Delta G1$. Fig. 1(f) describes a query graph $Q2$ for such events. The execution of $Q2$ over $\Delta G1$ and $\Delta G2$ requires that the results should cover all the dependency relations between the previous matches (incremental evaluation). That is, with the arrival of a new update ($\Delta G2$ Fig. 1(e)), the query match for $Q2$ contains the following triples:*
((:Calum, <:hasFriend>, :Adam), (:Calum, <:hasFriend> :James), (:Calum, <:hasLike>, 'Football')).

Challenges. Existing solutions for both static and dynamic KGs have based their execution strategies on the *index-store-query* model. They use extensive indexing techniques to reduce the search space to match the query graphs. However, KG streams are fast changing, temporally ordered, and potentially infinite. This raises new challenges: it is prohibitively expensive to index and store dynamic KG, frequently enough for applications with real-time constraints. Therefore, the key challenges to overcome in order to achieve efficient CGPM over KG streams are as follows.

(1) *Cost of indexing in streaming settings:* Existing static and dynamic GPM solutions rely on indexing techniques; data are indexed in all possible ways allowing every execution step to be supported by some index. These indexing techniques incur super linear space, and/or super linear construction time. For example RDF3x [28] builds several clustered B+trees for all the permutations of three-columns giant table of triples with time complexity of $O(m)$, where m is the number of triples. Consider another example, the R-join [14] approach for subgraph isomorphism is based on 2-hop indexing with time complexity of $O(n^4)$ (n, number of vertices) to build such indices. In light of this, to support real-time constraints for stream processing, indexing techniques (super-linear/linear indices) may be prone to performance bottlenecks due to added latency.

(2) *On the fly computation:* The standard streaming constraint of being able to process every triple/edge only once applies naturally to KG streams. To achieve low latency, a system must be able to process data (in-memory) without having a costly storage operation [36]. This requires a light-weight data structure, which is efficient enough for write and query intensive system. Furthermore, it should be able to meet the demands of scalability due to the large size of defined windows over streams.

(3) *Incremental evaluation:* Incremental algorithms have been proven to be useful in a variety of applications [33]. They aim at incrementally generating results without incurring the expensive cost of re-evaluating everything from scratch. The field of incremental GPM is still quite fertile, and most of the solutions are based on re-evaluation strategies [8, 24]. The main reason is their dependency on the indexing techniques, thus making the re-evaluation strategy a primary choice. Few incremental GPM solutions [15, 18, 34] are presented in the context of general labelled-graphs, but they suffer from the following shortcomings: (i) most of them are approximate algorithms based on relaxed graph simulations [18, 34], (ii) they only work for small numbers of graphs within a window, and (iii) they are not directly applicable for KG model, due to its inherent complexity and multigraph nature, and show performance degradation [20].

Contributions. In this paper, to the best of our knowledge, we present the first practical solution for CGPM over KG streams that covers both event-based and incremental evaluation models. In summary, our contributions are as follows.

(1) We propose a *query-based graph pruning* technique, where KG events are pruned by the structure analysis of the query graph. Thus, only the triples – within an event – that are critical for the query graph processing are stored and queried, leading to a considerable reduction in search space. The pruned set of triples from each event is materialised into a set of *vertically partitioned* (VP) tables using a novel *bidirectional multimap* data structure. Thus, query processing is performed on a set of pruned VP tables instead of the entire KG events (Section 3).

(2) We use a hybrid *join-and-explore* technique to join a set of pruned VP tables and explore the matched results. Unlike the existing algorithms that rely on sophisticated indices, we perform CGPM without using indices (except for the dictionary encoding of string to numeric IDs). To make up for the performance loss due to the lack of indexing, at each step of the algorithm, we use fast hash-joins between VP tables to prune all the triples that do not satisfy the join conditions. Hence, the exploration process to determine the matches is performed on a precise set of triples. Furthermore, we use a novel automata-based triple pattern join (TP-Join) algorithm, which guides the join and exploration process in an iterative manner: for each execution of an automaton, we make sure that it percolates only if there is enough evidence that future join operations will be required (Section 4).

(3) We extend the above mentioned approaches for incremental evaluation of CGPM. The matched results from the graph exploration process are stored in a set of matched VP tables. Thus, the pruned triples set from the newly arrived updates is matched only with the previously matched results. This greatly reduces the computation overhead of recomputing all the triples within a window for each update (Section 5).

(4) Using real-world and synthetic datasets, we experimentally verify the performance of our event-based and incremental CGPM algorithms. Our system outperforms existing systems by (on average) one to two orders of magnitude in terms of performance and scalability (Section 6).

The rest of the paper is thus organised as follows. Section 2 formulates the CGPM problems we attack in this paper. Section 3 presents our graph pruning algorithm, Section 4 describes the query processing for event-based CGPM. Section 5 presents the solution of incremental CGPM. We conducted extensive performance studies using large datasets and report our findings in Section 6. Related work is given in Section 7, and Section 8 concludes the paper.

2. PRELIMINARIES

In this section, we first present the preliminary discussion about the KG streams and query graph. Then we formulate the problem of CGPM for the event-based processing model. This problem can easily be extended for incremental CGPM.

2.1 Knowledge Graph Data model

We use W3C recommended RDF data model [38] for KGs, where data consist of a set of $\langle subject, predicate, object\rangle$ triples ($\langle s, p, o\rangle$ for short). The notations used in this paper are listed in Table 1.

DEFINITION 1 (KNOWLEDGE GRAPH). *A* knowledge graph *(KG) is a directed labelled multi-graph denoted as $G = (V, E, L, \psi)$, where V is a set of vertices, corresponding to all the subjects and objects ($s, o \in V$). $E \subseteq V \times V$ is a set of directed edges from subjects to objects. L is a set of edge and vertex labels, and $\psi : V \cup E \rightarrow L$ is a labelling function which maps a vertex or an edge to the corresponding label respectively.*

Each distinct $\langle s, p, o\rangle$ triple (t) is mapped to an edge $e \in E$ with $\psi(e) = p$. Fig. 1(d) presents a KG with two triples `(:Calum, <hasPost>, :Post1), (:Calum, <hasFriend>, :Adam)`.

DEFINITION 2 (QUERY GRAPH). *A* query graph *is a directed labelled multi-graph denoted as $Q = (V_Q, E_Q, L, \psi_Q, vars)$, where V_Q is a set of vertices, $E_Q \subseteq V_Q \times V_Q$ is a set of directed edges, L is a set of edge and vertex labels, $vars$ is a set of query variables, and $\psi_Q : V \cup E \rightarrow \{L \cup vars\}$ is a labelling function which maps a vertex or an edge to the corresponding label respectively.*

A query graph Q can be modelled as a set of triple patterns, where each triple pattern is an edge of the graph. A triple pattern is a triple that can contain variables ($vars$) in the subject, predicate or object. In the rest of the paper, we interchangeably use the terms triple pattern (tp) and an edge (E_Q) of a query graph. Moreover, in this paper, we only consider the connected query graphs and we do not consider variables at predicate position, as it is not very common in real-world query graphs, as shown in the previous study [5].

Processing a query graph Q against a KG G resolves to finding all subgraph isomorphisms (or subsequently homomorphisms) between Q and G. Let M be this GPM function, its application $M(Q, G)$ produces a resulted graph G' that is isomorphic to Q. Fig. 1(c,f) presents two query graphs. The application of the query graph $Q2$ in Fig. 1(f) on $\Delta G2$ in Fig. 1(e) results in the following set of triples: `{(:Calum, <hasFriend>, :James), (:Calum, <hasLike>, 'Football')}`.

DEFINITION 3 (KNOWLEDGE GRAPH STREAM). *A* knowledge graph stream $\mathcal{G} = \{(G^1, \tau_1), (G^2, \tau_2), \ldots, (G^n, \tau_n)\}$ *is a sequence of ordered* KGs, *where each* KG *is associated with a comparable timestamp $\tau \in TS$, where TS is a totally ordered set of timestamps.*

Table 1: Notations used in this paper

Notation	Description
G	Knowledge Graph
V, E	Set of vertices and edges of KG
t	KG triple $\langle s, p, o\rangle$, $s, o \in V$
Q	Query graph
V_Q, E_Q	Set of vertices and edges of Q
tp	triple pattern in Q
$\mathcal{G}$	KG stream
(G^i, τ_i)	KG event at time τ_i
$(\Delta G^i, \tau_i)$	KG event as an update at time τ_i
M	Subgraph matching function $M(Q, G)$
W^w	Time window of size w begins at τ_b and ends at τ_e
$\mathbb{T}_i$	Pruned vertically partitioned (VP) table
R_i	Graph structure for matched triples
A	TP-Join automaton
s	Automaton State
θ	State's transition function of the form ($\mathbb{T}$, R, J_t, J_{id})
J_t	Type of join (s-o,o-s,s-s,o-o)
J_{id}	Target join id of the triple pattern
FR	Table for the final matched triple set
$\mathbb{FT}$	Set of VP tables for incrementally matched triples

2.2 Problem Formulation

The problem formulation of event-based and incremental CGPM is described as follows.

PROBLEM 1 (EVENT-BASED CGPM). *Given (i) a query graph Q, (ii) a* KG *stream $\mathcal{G}$, and (iii) a matching function M,* Event-based CGPM *amounts to continuously compute the function $M(Q, (G^i, \tau_i))$ for each event within the* KG *streams.*

PROBLEM 2 (INCREMENTAL CGPM). *Given (i) a query graph Q, (ii) an evolving* KG *G, and $(\Delta G^i, \tau_i)$ as updates to G, such that the updates conform to a stream $\mathcal{G} = \{(\Delta G^1, \tau_1), \ldots, (\Delta G^n, \tau_n)\}$, and (iii) a matching function M,* Incremental CGPM *amounts to continuously compute the changes $\Delta M_i = M(Q, (\Delta G^i, \tau_i))$ to the matches such that,*

$$M(Q, \bigcup_{k=0}^{i-1}(\Delta G^k, \tau_k) \oplus (\Delta G^i, \tau_i)) = M(Q, \bigcup_{k=0}^{i-1}(\Delta G^k, \tau_k)) \oplus \Delta M_i$$

where operator $\oplus$ incrementally applies changes to the matched graphs.

The event-based model processes each KG event independently and only considers the new events for CGPM, while incremental evaluation model incrementally identifies changes in response to the arrival of new updates. Both models are processed over time-interval based tumbling windows (W^w) [7]. Windows are a central concept in stream processing because an application cannot store an infinite stream in its entirety. Instead windows are used to summarise the most recent set of KG events. At a time τ, a window of size $w \in \mathbb{N}^+$ begins at $\tau_b = \left\lfloor \frac{\tau - w}{w} \right\rfloor .w$ and ends at $\tau_e = \tau_b + w$. Formally, it is defined as follows,

$$W^w(\tau) = \{G | (G, \tau') \in \mathcal{G} \wedge \tau_b \leq \tau' \leq \tau_e\}$$

The size of each graph $G^i \in \mathcal{G}$, which is bounded by the defined window, has a huge impact on the query performance. As noted earlier, indexing may not be a viable solution in streaming settings. Thus, our first goal is to prune each graph-based event according to defined query graph. This consequently reduces the search space before the execution of the query graph for each event.

THEOREM 1. *The general complexity of CGPM problems for each $G^i \in \mathcal{G}$ is $O(|E_Q|.|E^i|)$, where $E^i \in G^i$ and $E_Q \in Q$*

PROOF. The proof of Theorem 1 can easily be extended from Theorem 1 in [30]. □

In the next section, we describe our graph pruning approach. Based on this, we present the TP-Join algorithm to match a set of triple patterns within a query graph against a set of pruned VP tables. Our TP-Join algorithm first executes hash-based joins on pruned VP tables to remove all the triples that would not satisfy the join conditions; it is executed using an automaton. The resulted set of candidate triples is inserted into an in-memory graph structure (adjacency lists) to verify all the matches. Note that both incremental and event-based CGPM problems share the same methodology for their graph pruning and join approaches. Thus, we first present our proposed approach for event-based CGPM problem (Problem 1) and later extend it for incremental CGPM (Problem 2).

3. PRUNING AND MATERIALISATION

In this section, we present our KG pruning and materialisation strategy, which serves as the backbone for the query graph processing for event-based and incremental evaluation models.

Our query-based pruning technique is grounded on the concept of *view*. Given a query Q and a set of views ($\mathbb{V}$), the idea is to find another query $Q*$, such that $Q*$ is equivalent to Q, and $Q*$ only refers to views in $\mathbb{V}$ [6, 21]. This technique provides considerable advantages: given a dataset D, one can compute the answers $Q(D)$ in D by using $\mathbb{V}$ without accessing D. Motivated by these properties, we implement query-based KG pruning and vertically data partitioning. The basic idea is to prune and partition KG data into a set of vertically partitioned (VP) tables [1]. That is, a unique predicate in a query graph represents a table, where the first column contains the subject and the second column the object value of that subject. Each table, denoted as $\mathbb{T}_i$, collects all the valid triples for each triple pattern $tp_i \in Q$ and the system prunes all the "dangling" triples that would not be useful during the query graph evaluation.

OBSERVATION 1. *Given a query graph Q and a KG event (G^i, τ_i), the number of edges $|E_Q| \in Q$ is less than or equal to the number of edges $|E^i| \in G^i$.*

In general, query graphs are focused on a specific part of the graph to be matched, thus pruning the unnecessary triples that would not be used during the query processing greatly reduces the search space. Furthermore, query graphs typically involve in finding the connected components of the input KG. Thus, if one or more subjects or objects are labelled with constants, we can safely prune all the false positives from the results. This process is generally called as join-ahead pruning [32]. The pruned KG event is materialised into a set of VP tables; each VP table is assigned with a predicate and contains its associated subjects and objects.

EXAMPLE 2. *Fig. 2(a) shows a set of triples from a KG event (G^i), which describes various attributes of a house. Fig. 2(b) presents a query graph Q that is matched against (G^i, τ_i). Note that Q contains only two triple patterns (tp_1 and tp_2) with predicates <:hasWeather> and <:hasPower>. Thus, all the triples in G^i, whose predicates do not match with <:hasWeather> and <:hasPower> can safely be pruned from G^i. Furthermore, the object of tp_1 contains a constant value, i.e., 'sunny', therefore all the objects for predicate <:hasWeather> that do not match with the value 'sunny' can also be pruned from G^i. Fig. 2(c) shows the pruned and vertically partitioned triples that are used for the final query graph processing, as described in the next section.*

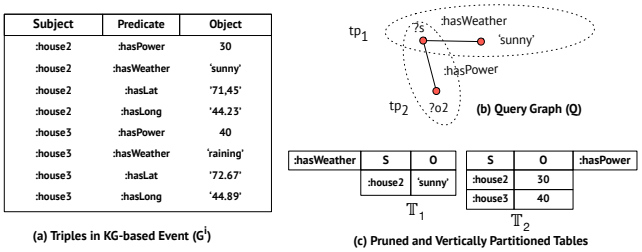

Figure 2: (a) A set of triples for a KG-based event ((G^i, τ_i)), (b) a registered query graph (Q) (c) Vertically partitioned tables after query-based pruning for query graph Q.

The materialisation of VP triples requires a data structure that can be used efficiently during the join process. Thus, we design a light-weight and write-friendly data structure called as *bidirectional multimap*. Multimaps are associative containers that associate values to keys in a way that there is no limit on the number of elements with the same key (an important property for multi-valued triples, i.e., when triples have same subject for different objects). It allows constant look-ups (considering there are no hash-collisions) by keys. However, a query graph can also be involved with object-subject (by join operator reordering) and object-object joins, thus in order to implement the linear joins, we use bidirectional multimaps. Furthermore, we have implemented two different kinds of hashing for our data structure: 32-bit and 128-bit MurmurHash [4]. They are efficiently used depending on the expected size of the input KGs and use cases. It greatly reduces the probability of hash-collisions.

Algorithm 1 Graph Pruning and Materialisation

```
1:  G^i ← {t_1, t_2, ..., t_i}                          ▷ G^i ∈ (G^i, τ_i)
2:  tp_j ∈ Q ← Triple pattern
3:  d ← Dictionary                                      ▷ see Dictionary Encoding
4:  T_j := empty VP table
5:  for each triples t ∈ G^i do
6:      if pred(tp_j) = pred(t) then                    ▷ predicate comparison
7:          t' := d.encode(t)                           ▷ dictionary encoding
8:          T_j := T_j ∪ {t'}
9:      end if
10: end for
11: for each triples t' ∈ T_j do
12:                                                     ▷ subject and object comparison
13:     if sub(tp_j) ∉ vars(tp_j) and sub(tp_j) ≠ sub(t') then
14:         T_j := T_j \ {t'}
15:     end if
16:     if obj(tp_j) ∉ vars(tp_j) and obj(tp_j) ≠ obj(t') then
17:         T_j := T_j \ {t'}
18:     end if
19: end for
```

Algorithm 1 describes the pruning and materialisation of a KG event. Given a triple pattern $tp_j \in Q$ and an event (G^i, τ_i), it returns a table $\mathbb{T}_j$ containing a subset of the triples in G^i. It uses two subroutines. The first one (lines 5-10) iterates over all the triples $t \in G^i$ and compares the predicate of a triple pattern (tp_j) and a triple $t \in G^i$. If the match yields to true (line 6), it first encodes the value of t (encode(t, d)) by using a dictionary (see below for details about dictionary encoding). The encoded triple t' is added into the table $\mathbb{T}_j$ (line 8). The second subroutine (lines 11-19) iterates over the values in the table $\mathbb{T}_j$. It first checks if the subject/object of tp_j is a variable or a constant (line 13,16). In

the case of constant subject/object, it simply compares their values and prunes the triples that do not satisfy it. Note that Algorithm 1 describes the materialisation of a single table; each table is disjoint to others, thus the pruning and materialisation process can take the advantages of multicore processing to parallelise the computation.

Dictionary Encoding. Encoding triples by using a dictionary results in space reduction, and increases the performance by performing arithmetic comparisons instead of string comparisons. While most of the existing dictionary encoding techniques focus on static data [28, 39], we implement a dynamic in-memory dictionary for KG streams. The basic idea is to use two different kinds of dictionaries (associations between the numeric and textual IDs): *persistent* and *adaptive*. The persistent dictionary encodes the query graph data, while the adaptive one encodes the data in the VP tables. Both of these dictionaries are synchronised in a way that they use the same encoding for the same strings. Thus, encoded values within the persistent dictionary are not affected by the removal operations on the adaptive dictionary. The encoded values within an adaptive dictionary are removed, (i) after processing each event (for event-based CGPM), or (ii) after the expiration of the window (for incremental CGPM). Leveraging this, the size of the encoded values does not explode linearly with the size of the KG streams.

4. EVENT-BASED CGPM

In this section, we present our algorithm for event-based query graph processing over a set of pruned and materialised VP tables. We extend this algorithm in the next section for the incremental query graph processing.

Existing KG processing techniques that are based on a vertical partitioning approach require indices, such as B+-trees and k^2-tree [1, 3]. Therefore, we employ a hybrid join-and-explore approach to match a query graph with an event without the aid of indexing. Our approach is based on two steps: (1) join the set of tables produced after executing Algorithm 1, and remove all the triples that do not satisfy the join conditions in the query graph, (2) insert the resulted triples in graph structures (adjacency lists-based) and explore them to produce the final matched graphs.The use of the second step enables the identification of the triples that cannot be part of the final matched results (in case of chain or cyclic queries). This process is usually implemented by the use of indices. Thus, at each step we reduce the search space by removing the "dangling" triples that are not part of the final matches. In order to aid the join and graph exploration process, we use an automata-based algorithm.

4.1 TP-Join Automaton

Pattern matching using automata models has been studied for several decades [2, 31], where patterns are usually pre-processed into a finite automaton (non-deterministic or deterministic). Thus, the automaton allows to determine the matched patterns by a single scan of the input. Considering the on-the-fly nature of the KG streams processing, we opt to use an automata-based TP-Join algorithm. Our design choice for this model is influenced by the following factors: (1) finite automata (especially the non-deterministic version) are highly expressive, and are closed under union, intersection and Kleene closure [31]. Therefore, the solution for basic graph pattern matching can be easily extended to more complex patterns. (2) The automata model presents a favourable choice in dynamic and incremental evaluation settings [10]. Considering these properties, we use a finite automata-based design (TP-Join automata) to model the set of triple patterns $tp \in Q$ on a set of automaton states. Each state is associated with an edge, which is labelled with a transition rule, i.e., a join condition between the triple patterns.

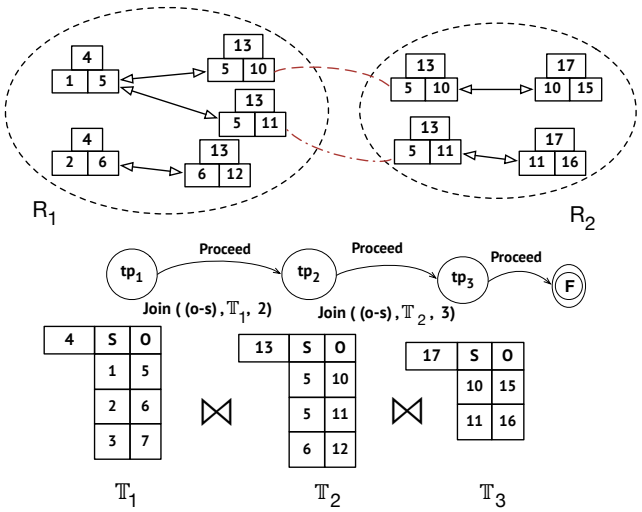

Figure 3: Automata-based join execution and exploration of matched triples

DEFINITION 4 (TP-JOIN AUTOMATON). *Given a query graph Q and a set of VP tables $\mathbb{T}$, the TP-Join automaton A is a tuple (S, E_A, θ, s_i,s_f), where S is a set of states, $E_A \subseteq S \times S$ is a set of edges, each labelled with a transition rule θ, $s_i \in S$ is a start state, $s_f \in S$ is a final state, such that: (i) each triple pattern $tp \in Q$ is represented by a state $s_k \in S$, (ii) transition rules θ are of the form ($\mathbb{T}$, R, J_t, J_{id}), where $\mathbb{T}$ is a set of VP tables generated from the Algorithm 1, R is a set of graph structures to store the intermediate join results, J_t is the type of the join between the triple patterns (s-s, s-o, o-s, o-o), J_{id} is the target join ID of the triple pattern.*

The execution of a TP-Join automaton is based on two operators named as `Join` and `Proceed` (see Fig. 3). The `Join` operator uses the transition rules mapped at edges and implements the required join methods with the appropriate triple patterns. The objective of the `Proceed` operator is to check the status of the `Join` operator, i.e., if the `Join` operator produces a non-empty intermediate result set then the automaton transits to next state, otherwise it terminates the execution of the automaton with no match. The core functionality of `Join` operators is implemented using hash joins. After the joining process, the `Proceed` operator is executed in the reverse direction from the last to the first state to collect the final set of triples from the graph-structure R, which captures all the intermediate join results. The choice of an automata model to evaluate query graphs provides another important feature called as *percolation*. The TP-Join algorithm percolates, if at some point there is enough evidence to join a new triple pattern. Otherwise, it stops its execution without computing the expensive joins that are not required.

EXAMPLE 3. *Let us consider a query graph with tp_1(`?x, <p1>, ?y`)), tp_2(`?y, <p2>, ?z`) and tp_3(`?z, <p3>, ?k`). $tp_1 \bowtie tp_2$ has o-s join (J_t) on variable `?y`, $tp_2 \bowtie tp_3$ has o-s join on variable `?z`. Fig. 3 shows the mapping of these triple patterns on states s_1, s_2 and s_3 respectively. The tables $\mathbb{T}_1$, $\mathbb{T}_2$ and $\mathbb{T}_3$ describe the pruned and VP triples for the three triple patterns; the triple values are encoded to integers by utilising a dictionary (after Algorithm 1). Starting from the state s_1, we first implement the o-s hash-join between $\mathbb{T}_1$ and $\mathbb{T}_2$, and results are added in a graph structure R_1: triple (`1,4,5`) is joined with triples (`5,13,10`) and (`5,13,11`), and triple (`2,4,6`) is joined with (`6,13,12`). The join between the tables produces the result, thus the automaton proceeds to the next state. At state s_2, an o-s hash-join is implemented with $\mathbb{T}_2$ and*

$\mathbb{T}_3$ and the matches are added in R_2; triple (5,13,10) is joined with (10,17,15) and triple (5,13,11) is joined with (11,14,16). After conducting the join process, we start the exploration process from the last state's (s_3) graph structure R_2 till the first state's (s_1) graph structure R_1. This process is similar to depth first search (DFS). We first explore the joined triples from (10,17,15) to reach (1,4,5), and then from (11,17,16) to reach (1,4,5). The final set of explored triples provides the matched result for an event.

Algorithm 2 TP-Join algorithm

```
1: 𝕋 := {𝕋_1, 𝕋_2, ..., 𝕋_k}, k = |E_Q|
2:                                   ▷ set of pruned VP tables, each for a state
3: A := {s_1, s_2, ..., s_{k+1}}
4:                                   ▷ an automaton with mapped triple patterns
5: R := {R_1, R_2, ..., R_{k-1}}
6:                                   ▷ set of graph structures to store the joined result
7: FR ← {∅}
8:                                   ▷ a three column table for the final matched triple set
9: for each s ∈ A do
10:   s_join := getJoinState(s, A)
11:   𝕋_s := hashJoin(𝕋_s, 𝕋_{s_join})
12:   if 𝕋_s ≠ ∅ then
13:     R_s := Insert(𝕋_s)
14:     Proceed to the next state
15:   else
16:     Terminate the execution
17:   end if
18: end for
19: for vertex v ∈ R_{k-1} do        ▷ Iterate over graph structure for the last state (s_{k-1})
20:   FR = explore(R, v, FR)         ▷ M(Q, G^i) := FR
21:   output the triples in FR
22: end for
23:
24: procedure explore(R, v, FR)
25: label v as discovered
26: FR := FR ∪ {v}
27: for edges u from v, where u, v ∈ R do
28:   if u is not labelled then
29:     explore(R, u, FR)
30:   end if
31: end for
```

4.2 Execution of TP-Join Automaton

Algorithm 2 describes the execution of the TP-Join algorithm. It starts its execution after Algorithm 1, only if $\mathbb{T}_i \neq \varnothing$ for all $tp_i \in Q$. It has two main steps.

1. *Automaton Traversal* (lines 7-15). Triple patterns $tp \in Q$ are mapped onto a set of states in A, where each tp_i is associated with a VP table $\mathbb{T}_i$ and a graph structure R_i. Each state $s \in A$ is traversed to conduct the hash-joins between the associative tables. First, the algorithm gets the joined state s_{join} for the current state (line 10) using mapped J_{id}. For example, in Fig. 3 tp_1 has o-s join with tp_2 at state s_1. The algorithm then executes the hash-join between the current state's table $\mathbb{T}_s$ and joined state's table $\mathbb{T}_{s_{join}}$ (line 11). If such join produces a result, joined triples are inserted into the graph structure, and the automaton proceeds to the next state (lines 12-13); otherwise it terminates its execution.

2. *Matched Result Exploration* (lines 19-31). The algorithm initiates the exploration process, if the automaton reaches the final state s_f after the execution of all the joins. The graph structure associated with the last state is first selected (line 19); as the joined triples propagated to the last state are among the ones with the complete match. Therefore, we select the last state's graph structure R_{k-1} (line 19) and execute a depth first search (DFS) on all of its vertices. For each $v \in R_{k-1}$, its neighbouring triples are found and the process executes in a recursive manner (lines 24-31). The final result set FR contains all the matched triples for a query graph. Note that we only take into account the triples set in FR, whose triples contain all the defined distinct predicates in Q.

Analysis. The order of each hash-join between VP tables is sorted according to the size of the tables: the smallest table is selected to iterate over other (join reordering). It takes linear time $O(|\mathbb{T}_i|)$ for each hash-join. The exploration process is usually executed on a relatively small set of triples; all the useless triples are pruned during the execution of Algorithm 1 and hash-joins. The general complexity of the exploration process is $O(|V_f|(E_R + V_R))$, where V_f ($f = k-1$, see Algorithm 2) is the number of vertices in the final state's graph structure R_f, E_R and V_R are the number of edges and vertices in R.

Let us notice that the use of KG-based events enables another interesting feature, where static background knowledge bases can be used to semantically enrich the events. For example, an event containing the power usage information of a house can be processed with the historical power consumption of the house, thus computing the predictive behaviour of future events. VP tables can be utilised to partition such historical events and we can employ them within the TP-Join algorithm.

5. INCREMENTAL CGPM

In this section, we extend our event-based CGPM algorithm to support incremental CGPM over KG streams. As discussed in Section 3 (Problem 2), an incremental match problem takes a query graph Q, a set of graph updates from a stream $\mathcal{G}$, where each update contains a set of triples. It finds new matches $M(Q, (\Delta G^i, \tau_i)) = \Delta M_i$, while considering the old ones. That is, whenever a new update $(\Delta G^i, \tau_i)$ arrives, we would like to solve the following problems: (1) incrementally locate new matches, while considering the old ones, (2) efficiently update the old matches with the arrival of new updates.

To address these problems, we extend our TP-Join algorithm to consider the old matches during join operations. Note that the graph pruning and VP approach applies in the same manner to this problem.

Contrary to the event-based computation, matches emerge slowly during incremental evaluation. Usually we find partial matches for each update and then incrementally process the remaining matches. The execution of partial matches can result in unnecessary computation of join operations: future triples received in the tables may invalidate the partial results. Thus, we employ a lazy evaluation strategy to reduce the number of unnecessary joins and exploration-based computations. The basic idea is that when the graph pruning does not produce any triple for a $tp_i \in Q$, we defer the join between the triple patterns and only process the joins with the already matched triple set – if it is not empty.

5.1 Incremental Execution of TP-Join Automaton

Algorithm 3 describes the details of our incremental TP-Join algorithm. It has the following steps:

1. *Automaton Traversal* (lines 8-13). As noted earlier, we use a lazy evaluation strategy to implement the joins between the

Algorithm 3 Incremental TP-Join algorithm

1: $\mathbb{T} := \{\mathbb{T}_1, \mathbb{T}_2, \ldots, \mathbb{T}_k\}, k \leq |E_Q|$
2: ▷ set of pruned VP tables, each for a state
3: $A := \{s_1, s_2, \ldots, s_{k+1}\}$
4: ▷ an automaton with mapped triple patterns
5: $R := \{R_1, R_2, \ldots, R_{k-1}\}$
6: ▷ set of graph structures to store the joined result
7: $\mathbb{FT} := \{\mathbb{FT}_1, \mathbb{FT}_2, \ldots, \mathbb{FT}_k\}$
8: **if** $|A| = |E_Q| - 1$ **then**
9: | *Same as Algorithm 2 (lines 9 - 18)*
10: | **for each** *vertex* $v \in R_b$ **do**
11: | | $\mathbb{FT} = \texttt{explore}(R, v, \mathbb{FT})$
12: | **end for**
13: **else**
14: | **for each** $s \in A$ **do**
15: | | $\mathbb{FT}_{join} := \texttt{getJoinTable}(s, \mathbb{FT})$
16: | | $\mathbb{T}_s := \texttt{hashJoin}(\mathbb{T}_s, \mathbb{FT}_{join})$
17: | | **if** $\mathbb{T}_s \neq \varnothing$ **then**
18: | | | $\mathbb{FT}_{join} := \mathbb{FT}_{join} \cup \mathbb{T}_s$
19: | | | *Output the new matches in* $\mathbb{T}_s$
20: | | **end if**
21: | | Proceed
22: | **end for**
23: **end if**

VP tables. We initiate the automaton traversal only if for all $tp_i \in Q$, $\mathbb{T}_i \neq \varnothing$. Once $|A| = |E_Q| + 1$, i.e., all the triple patterns with $\mathbb{T}_i \neq \varnothing$ are mapped onto the automaton. In this case, the automaton traversal is the same as described in Algorithm 2 (lines 9-18). However, to cache the matched results for a given window, we use a final VP tables set $\mathbb{FT}$ (lines 10-12). This enables the incremental join procedure with new KG updates. Note that, during incremental evaluation, the matches $M(Q, (\Delta G^i, \tau_i))$ emerges with time. Therefore, if a certain update does not match with the query graph Q, it may match with future updates: unlike Algorithm 2, the incremental evaluation strategy requires to keep for a given window the old matches which are not matched in a given execution of the joins. Hence, if an automaton reaches the final state s_f with all the matched triples, we only remove the matched triples – after the exploration process – from each VP table $\mathbb{T}_i$. Furthermore, there are no duplicate triples due to our bidirectional multimap data structure.

2. *Incremental Automaton Traversal* (lines 13-23). For incremental evaluation, there can be cases, when $|A| \neq |E_Q| + 1$ and for some $tp_i \in Q$, $\mathbb{T}_i \neq \varnothing$, and the triples in $\mathbb{T}_i$ may belong to the previously matched results in $\mathbb{FT}$. Thus, for incremental automaton traversal, we use the already matched result set $\mathbb{FT}$ to get the associated VP table of the state and implement the hash-join between them (lines 14-17). If the join produces the result, we directly add the matched results in $\mathbb{FT}_{join}$ from $\mathbb{T}_s$ (line 16). Note that $\mathbb{FT}$ contains all the previously matched triples. Thus, in incremental case we do not require the exploration process. The final table list $\mathbb{FT}$ stores all the matched triples for a given window; if the window expires, all the matched triples are removed from it.

THEOREM 2. *Incremental CGPM produces the same results as the re-evaluation based CGPM within a window.*

PROOF SKETCH. Let tp_1, tp_2 be two triple patterns with a join on their attributes. Let $\mathbb{T}_1^i$, $\mathbb{T}_2^i$ be two materialised VP tables for tp_1 and tp_2 at time i respectively. We show in Algorithm 3 that at time $t = 1$ the matched triples are updated in $F\mathbb{T}_1$ and $\mathbb{FT}_2$, if $\mathbb{T}_1^i \bowtie \mathbb{T}_2^i \neq \varnothing$. Thus, at $t = 1$, $\mathbb{FT}$ only contains the triples that

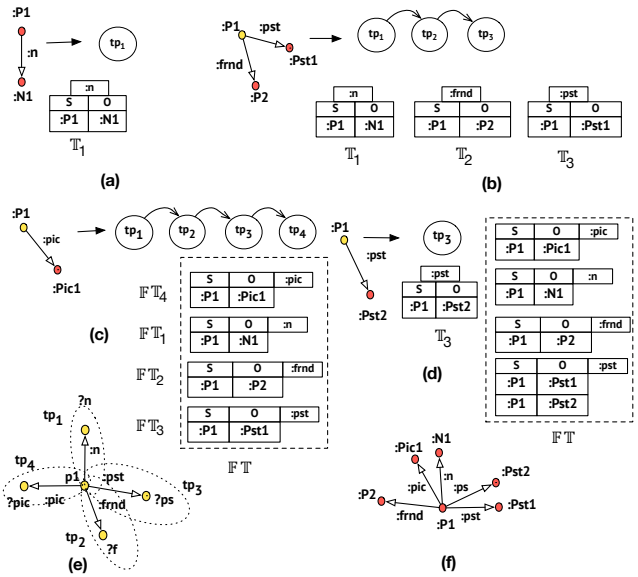

Figure 4: (a,b,c,d) Three updates from a KG stream, that resulted in respective automaton formation for the query graph in (e) and materialisation VP tables V. (d) Final matched graph

satisfied the join conditions of tp_1 and tp_2. At $t = 2$, if $\mathbb{T}_1 \neq \varnothing$ and $\mathbb{T}_2 = \varnothing$, we join $\mathbb{T}_1$ with $\mathbb{FT}_2$; and if the join produces the matched triples, they are inserted in $\mathbb{FT}_1$. Thus, during each execution of the algorithm, we capture the valid joined triples of $tp \in Q$ and the incremental evaluation produces the same result as the re-evaluation. It is quite trivial to extend the conclusion for a set of triple patterns $tp \in Q$. □

Analysis. Similarly to Algorithm 2, it takes linear time $O(|\mathbb{T}_i|)$ for each hash-join, and $O(|V_b|(E_R + V_R))$ $(b = k - 1)$ for the exploration process. In order to implement the joins with the final table $\mathbb{FT}$, we rearrange the join operators and it still takes $O(|\mathbb{T}_i|)$ for each hash-join with $\mathbb{FT}$.

EXAMPLE 4. *Consider the query graph Q shown in Fig. 4(e) and the four* KG *updates in Fig 4(a,b,c,d) with their respective VP tables. For space constraints we only show simple triple-based updates and values of $\langle s, p, o \rangle$ are not encoded. Each update results in the construction of an automaton state. However, due to our lazy evaluation strategy, updates are joined once for each $tp_i \in Q$, $\mathbb{T}_i \neq \varnothing$, i.e., with the arrival of an update (`:P1, <:pic>, :Pic1`) (as shown in Fig. 4(c)). At this point, all the hash-joins are processed, and after the exploration process, all the matched triples are added into $\mathbb{FT}$. In Fig 4(d) a new update arrives (`:P1, <:pst>, :Pst2`): it cannot be joined with other VP tables, thus we initiate the incremental join process with $\mathbb{FT}$. As the join is successful, we insert the new update into $\mathbb{FT}$. The final matched* KG *with Q – after the last update – is shown in Fig. 4(f).*

6. EXPERIMENTAL EVALUATION

In this section, we present the performance evaluation of the event-based and incremental CGPM algorithms presented in Sections 4 and 5. We use an extensive set of queries and datasets from famous benchmarks. Overall, our experimental evaluations provide support to the following claims:

1. The use of hybrid join-and-exploration approach provides significant performance improvements, while indexing-based solutions spend considerable amount of time in data loading and index creations, thus perform poorly in dynamic streaming settings.

2. The incremental processing of KG streams for a given window avoids overheads of recomputing all the matches from scratch, thus outperforms re-evaluation based solutions by on average one to two orders of magnitude.

6.1 Experimental Settings

Metrics. For the event-based CGPM, we vary the size of each event to evaluate its effect on the performance. Each event is evaluated independently and the size of the window does not have any influence on the system's performance. Therefore, using push-based semantics, we evaluate each event as soon as it enters the system and process its matches. Note that if aggregate operators are used, the window size is required for event-based evaluation. However, the use of such operators is outside the scope of this paper.

The incremental evaluation-based strategy refers to previously matched events. Therefore, we vary the size w of the window (W^w) and evaluate the events that are within the tumbling window, i.e., the time stamp τ of all the processed events should be within the boundaries of the window, $\tau_b \leq \tau \leq \tau_e$.

Datasets. We use a real-world dataset and a synthetic one:

1. The NY Taxi Dataset[2] is a publicly available real-world dataset with total of 50 million taxi related events. Each event contains 17 different measurement values for taxi fares, locations, trip distance, trip time etc. We mapped each event into a KG of 24 triples, where each predicate describes the type of measurement. In total, the dataset contains more than 1 billion triples. This dataset was also used in the last year's DEBS grand challenge [22].

2. The Social Network Benchmark (SNB) [17] is a synthetic dataset containing social data distributed into streams of GPS, posts, comments, photos, and static data contain the users profiles. It contains information about the persons, their friendship network and content data of messages between persons, e.g, posts, comments, likes etc. We generated a total of 50 million triples that contain data for 30,000 users. The distribution of the dataset is described in Table 2. This dataset is also used for this year's DEBS grand challenge[3].

Query Graphs. We designed and reused multiple different types of query graphs for the above mentioned datasets.

1. We generated three different query graphs (NY-Q1, NY-Q2 and NY-Q3) for the NY taxi dataset by using different types of joins (*s-s, s-o, o-s, o-o,* etc.) for triple patterns. These query graphs show the relationships between taxi fare information, location, trip times etc,. For example, NY-Q1 finds the trip-type, trip-distance and total trip-time for *s-s* joined triple patterns on each taxi event. These query graphs are presented in Appendix A.

2. For the SNB dataset, we randomly generated three different types of query graphs by varying the types of joins and data selectivity. These query graphs are based on the use cases described in the benchmark, and describe the relationships between posts, comments, forums and persons. For example SNB-Q1 retrieves the post, its creator and its tag name for SNB events. These query graphs are presented in Appendix B. In order to compare the systems based on a triple stream model, we also reused the query graphs from LSBench[4] [8]. It is also a social network benchmark. However, the SNB dataset presents a more connected graph structure with a larger number of attributes compared with LSBench. Thus we can introduce more complex events for LSBench queries. These query graphs are presented in Appendix C.

Configurations. All the experiments were performed on an Intel Xeon E3 1246v3 processor with 8MB of L3 cache, and we report averages over 10 runs. The system is equipped with 32GB of main memory and a 256Go PCI Express SSD. It runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u05.

6.2 Analysis of Event-based CGPM

In this set of experiments, we look at the performance and scalability measures of event-based CGPM called as "E-CGPM" in the following. Performance is measured in terms of throughput in graphs per second: it is equivalent to informing the time taken by the given amount of data. Let the throughput be g graphs per second. If a certain dataset contains n graphs, then it takes $\frac{n}{g}$ seconds to answer the query graph over the dataset. For the comparison analysis, we use three widely known in-memory triple stores (Jena [25], Sesame [11], RDFox [27]) and a RDF stream processing (RSP) engine: CQELS [24]. Note that although in-memory triple stores are not optimised for dynamic settings, using them will give us insight into the performance bottlenecks of indexing solutions in dynamic settings. Furthermore, some of these in-memory systems, such as Jena, are also used as an underlying graph matching component for RSP systems. We only used one RSP engine (CQELS) for comparison, as it outperforms C-SPARQL [8] and others, as shown in previous studies [24].

Fig. 5(a) shows the performance of NY-Q1 on the NY Taxi dataset; each event is processed, matches are produced and then discarded from the systems. As described earlier, each event is independent to another, thus we process the number of graphs from few thousands up to 50 millions. We do not show the results of query graph NY-Q2 and NY-Q3 for NY Taxi data, as they provided similar performance measures: due to the smaller sized graphs, the query optimisation for each system does effect the query performance. Our system outperforms others – about an order of magnitude faster than Jena – due to its lower latency measures and fast hash-joins. Jena and Sesame, as compared to other systems, performed better due to their light-weight indexing and storage structure; both Jena and Sesame use property tables for data storage and a triple table indexing technique. RDFox provides an optimised and scalable indexing and storage technique – a single triple table for storage and a variant of B+-tree for indexing – but it fails to perform better than Jena and Sesame.

CQELS – an RDF triple streams system – does not support graph-based events, thus we modified its window parameter by using a count-based tumbling window for 24 triples. That is, the registered query is applied on the 24 triples of each event and then the event is discarded. CQELS is optimised for streaming settings but its performance measures are lower than Jena due to the following reasons: (1) it is based on a triple stream model, thus caching a whole graph within a window results in the re-evaluation of all

[2] http://chriswhong.com/open-data/foil_nyc_taxi/
[3] http://debs2016.org/
[4] https://code.google.com/archive/p/lsbench/

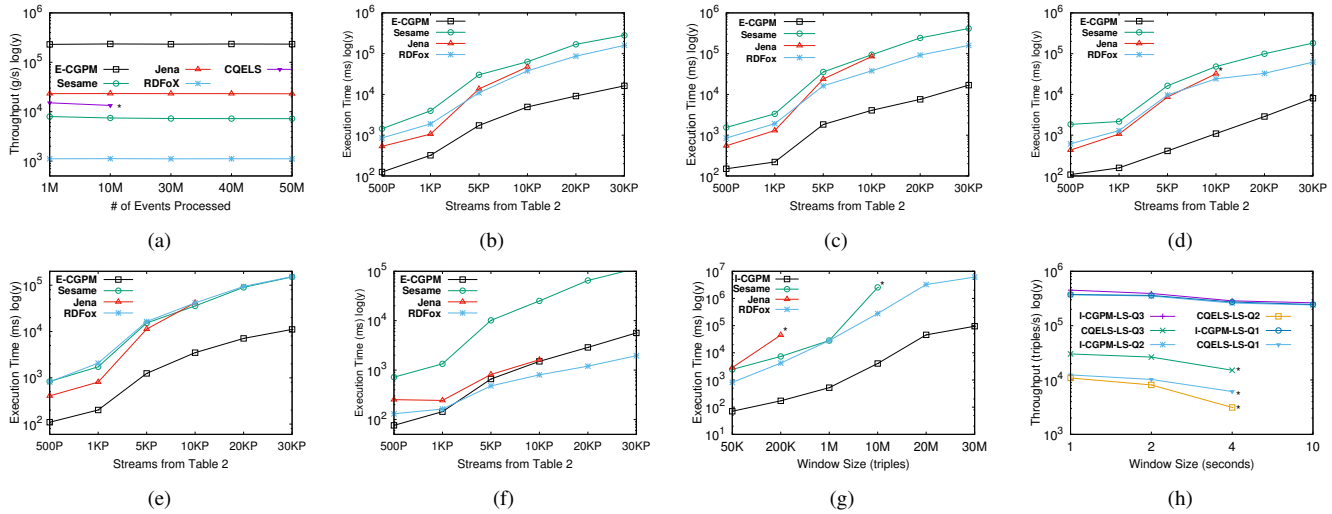

Figure 5: (a) Performance measures of NY-Q1 for event-based CGPM on the NY Taxi dataset. (b,c,d) Performance measures of SNB-Q1, SNB-Q2 and SNB-Q3 for event-based CGPM on the SNB dataset. (e,f) Latency measures and query time of SNB-Q1 for event-based CGPM on the SNB dataset. (g) Performance of SNB-Q3 for incremental CGPM on the SNB dataset. (h) Performance measures of LSBench query graphs LS-Q1, LS-Q2 and LS-Q3 for incremental CGPM on the SNB dataset. ("*" shows query time-out.)

Table 2: *Dataset distribution for large-scale CGPM, Min and Max describes the range of no. of triples for each event of SNB streams*

Dataset(streams)	Min (triples/event)	Max (triples/event)
500P	783	148K
1KP	2340	397K
5KP	217K	301K
10KP	50K	805K
20KP	115K	1.9M
30KP	145K	3.2M

the triples within the window for each triple-based event; (2) it uses a B+-tree to index all the triples within a window, resulting in extra overheads of index creation and updates; (3) it uses a static dictionary approach (for mapping strings to numeric IDs), and uses disk storage if the size of the dictionary reaches a certain threshold, resulting in high I/O costs. Our system enjoys the benefits of its index free solution and our bidirectional multimap structure guarantees linear joins for all kinds of hash-joins. The exploration of the matched graphs is performed on a small set of results; thus is a low computation process for NY-Q1, NY-Q2 and NY-Q3 of NY Taxi dataset. Furthermore, we refresh our dictionary after the expiration of each window (in this case after each event), thus the memory barriers do not explode for a large number of events.

In the next set of experiments, we use the SNB dataset and queries to measure the performance and scalability of different systems. The main objective of these experiments is to determine how different systems scale by varying the number of triples within each event. We generated multiple distributions/streams of the SNB dataset, by changing the number of persons for one year worth of data. We divided the data into a set of events, where each event contains 1 week worth of data. Thus, each event contains a large number of triples, but there are few numbers of total events. The distribution details of the dataset are provided in Table 2. Events with variable sizes are processed, matches are produced and then discarded from the system. Note that again we process each event independently (for E-CGPM), thus the window size does not have any effect on the performance. The elapsed time for each query graph was measured with warmed up cache by using the static SNB dataset (around 50K triples). We do not use CQELS for this set of experiments, as it performed poorly for larger graphs.

We report the total execution time for SNB-Q1, SNB-Q2 and SNB-Q3 in Fig. 5(b,c,d). SNB-Q1 contains all the triple patterns with *s-s* joins (i.e., a star-shaped pattern) and low selectivity measures, thus it returns a large number of matches. Our system scales linearly and smoothly (no large variations are observed). It outperforms others by taking the advantage of fast hash-joins and a simple explorations process: in a star-shaped query graph (i.e., only with *s-s* joins), the exploration process is simplified as all the triples share the same subject. Furthermore, our query-based pruning technique discards many useless triples before the execution of joins. Jena and Sesame, which perform better for smaller events resulted in time-outs with the increase in the size of an event; their light-weight indexing fails for large numbers of triples and their structure becomes quite dense. RDFox performs better for larger events due to its parallel and lock-free architecture and one big table indexing (six columns triple table) technique. However, it resulted in high latency measure due to indices, as shown in Fig. 5(e). The query graph SNB-Q2 encompasses high selectivity measures as compared to SNB-Q1 and contains a combination of *s-s* and *o-s* joins, resulting into a tree-like pattern. It produces less number of matches for smaller events. However, the number of matches grows exponentially with the increase in event size. This results in an expensive exploration process for our system. However the lower latency values compensates for it. Jena and Sesame perform in a similar fashion to SNB-Q1 and does not scale well with the increase in the number of matches. RDFox, as compared to them, still proved to be the winner; the parallel and lock free architecture of RDFox enables parallel join operations. The query graph SNB-Q3, as compared to SNB-Q1 and SNB-Q2, contains very high selectivity measures, thus only a handful of matches are produced. Here, our query-based pruning algorithm kicks-in and prunes most of the triples in each event without executing the join or exploration procedures. Hence, resulting in an increase throughput by reducing the number of join and exploration procedures. Other systems prune the unwanted triples during the join operations, resulting in higher insertion and computation times.

In order to demonstrate the difference between our indexing-free system and indexing-dependent systems, we report the latency measures and query time for SNB-Q1 in Fig. 5(e,f). From earlier observations, our system takes less time to load triples from each event, as no indexing is used and the graph pruning strategy discards many triples before the join process. However, the query time is lower for RDFox with its complex indexing technique, with high insertion/indexing time.

6.3 Analysis of Incremental CGPM

In this section, we bring all the above mentioned systems together for a side-by-side comparison using an incremental evaluation strategy. We use the SNB dataset for this set of experiments and generate queries for SNB and LSBench. For both sets of queries, we vary the window size and compare the performance of each system. SNB queries are evaluated on the KG updates (and compared with existing triple stores) that are incrementally matched with triples within each window; each update is of variable length from 100-1000 triple sets. The LSBench queries are evaluated on triple streams (and compared with the RSP systems), i.e., each update contains one triple that is incrementally matched with the rest in the window.

We report the results of SNB-Q3 for the SNB dataset in Fig. 5(g) with different window sizes; for the sake of brevity, we show the number of triples in each window. Note that due to the space limitation and similarity of performance measures from the earlier experiments, we do not show the results of SNB-Q1 and SNB-Q2. Our system (denoted as "I-CGPM") shows superior performance by incrementally processing the matches. That is, triples in updates are matched with each other only if for each $tp_i \in Q$, $\mathbb{T}_i \neq \varnothing$. If this condition does not evaluate to true, then the newly arrived updates are incrementally matched with $\mathbb{FT}$. This reduces the overheads of unnecessary hash-joins and re-evaluation of the entire window. Jena and Sesame perform better for smaller windows but they result in time-outs, as their respective underlying storage structures become quite dense for a large number of triples. Furthermore, the sizes of intermediate results keep on increasing with each new update, thus the re-evaluation of the same data results in extensive overheads. RDFox, due to its parallel architecture, indexing and storage structure performs reasonably well as compared to Jena and Sesame. But it suffers from high latency measures for larger windows.

We evaluate three additional queries of LSBench using the SNB dataset to better cover the RSP engines with the triple stream model. We use triple streams for this set of experiments, where each event contains one triple. The results of LS-Q1, LS-Q2 and LS-Q3 are reported in Fig. 5(h). Due to the simplicity of the triple stream model, LSBench queries, and the incremental evaluation strategy, our system shows similar performance measures for all queries. To elaborate, LS-Q3 is highly selective and there are less matches and less join operations. Therefore our system and CQELS perform much better on LS-Q3. However, CQELS, contrary to our system, does not prune the unwanted triples before starting the join operations; hence our system results in superior throughput. CQELS performance degrades on less restrictive query graphs (LS-Q1 and LS-Q2), where a larger number of matches are produced within a defined window. Thus, for each new triple update it re-evaluate all the matches from the triples set within a window: the information about the partial and complete matches is not stored and utilised for future match operations. Furthermore, it results into time-outs for larger windows, as it continuously updates its B+-tree indices and uses a static dictionary approach. In our system, on the contrary, any change made by the newly arrived triples to the query graph matches is stored, and information about the partial and total matches are incremental updated where needed.

7. RELATED WORK

GPM over KGs is a routine process and a fair number of KG storage systems have been developed in the past decade, such as Jena [25] Sesame [11], RDF3x [28], Gstore [39], RDFox [27] and SW-store [1]. These systems can be broadly classified into two categories: (i) graph-based storage with exploration-based querying, and (ii) tabular-based storage with join-based querying. The exploration-based [39] approaches are inspired from general subgraph matching algorithms [37]; subgraph matching is performed by searching the graph with some pruning strategies: Ullman [37] and Cordella *et al*. [29] are the pioneers in this field. The join-based techniques [1, 11, 25, 27, 28] have taken their roots from the relational data management systems, i.e., by modelling KG data in tabular structures, along-with their respective indices. Most of the above mentioned approaches eschew the mapping to relational databases, and use efficient indexing techniques to accelerate the querying process. GPM with join and exploration-based approaches is implemented through multi-way joins, and to support joins, indices must be used. In comparison, we advocate two important design principles. First, we argue that in order to truly scale and meet the real-time response requirements of CGPM, we should minimise or remove the dependency over indices. Second, we argue that, as query graphs are registered before the arrival of data, the query graph structure can be used to prune the unnecessary triples.

The field of processing dynamic and streaming KGs is still quite fertile and few solutions address this problem. These solutions can be classified into two categories, general labelled graphs-based solutions, and RDF stream processing (RDF) solutions. Both of these categories differ around their data models and use cases. For instance, RSP solutions use each RDF triple as an event, and utilise background knowledge-bases and ontologies to enrich matched events. Thus, due to the model similarity RSP systems are the most closely related to our work with some important distinctions, such as our system employs incremental evaluation and uses a set of RDF triples as an event model. We first review the general labelled-graph streaming systems.

The works on subgraph isomorphism for dynamic general labelled-graphs are classified into two categories: repeated pattern search, and incremental algorithms. The work by Fan *et al*. [18] presents an incremental algorithm for graph pattern matching. However, their solution to subgraph isomorphism is based on the repeated search strategy. Chen *et al*. [13] propose a feature structure called the node-neighbour tree to search multiple graph streams using a vector space approach. They relax the exact match requirement and their system requires significant preprocessing on the graph streams. [26] presents a solution to support continuous ego-centric queries in a dynamic graph. However, their solution is primarily focused on aggregate queries. [15] presents a subgraph selectivity approach to determine subgraph search strategies. It uses a subgraph-tree structure to decompose the query graph into smaller subgraphs, which is responsible for storing partial results. However, it only supports simple path-based queries and is optimised for homogeneous graphs by using an edge stream model. Our work is distinguished by its focus on the more complex RDF data model that supports a set of triples for each event. Furthermore, we use a join-based technique to incrementally join triples within a window, which avoids the repeated search to determine the candidate graphs for incremental evaluation.

RSP systems [8, 12, 23, 24] inherit the processing model of DSMSs [7], but consider a semantically annotated data model, namely RDF triple streams. C-SPARQL [8] is among the first contributions, and is often cited as a reference in this field. It supports timestamped RDF triples and employs push-based semantics:

queries are continuously updated with the arrival of new triples. It uses Jena [25] as an underlying system for GPM. The CQELS RSP system [24] is considered as the most optimised system of the lot. It offers a new language, and provides various optimised execution strategies. Similar to C-SPARQL, it adopts DSMS's processing model and windowing operators for each distinct stream. However, it is based on a pull-based execution strategy, where the evaluation of the RDF triples is triggered periodically. Both of the above mentioned systems and various others [12, 23] are an extension of general stream processing systems, where RDF data are consumed as triples. Their query processing within a window is based on the re-evaluation strategy: with the arrival of new triples, all the triples within the window are re-evaluated with the registered query. Furthermore, indices such as B+trees are used to index triples within a defined window. We argue that the use of incremental evaluation can greatly reduce computation duties and improve the performance of the system, as shown in our experimental studies.

8. CONCLUSION

In this paper, we studied the problem of CGPM over KG streams using event-based and incremental evaluation models. We propose a query-based graph pruning technique to enable join-ahead pruning of unnecessary triples. We use a bidirectional multimap data structure to materialise a set of pruned VP tables. These VP tables are then processed using fast hash-join, thus further removing the unnecessary triples. The final pruned set of triples is explored using a graph structure. We named this technique as join-and-explore and it is index-free. We used an automata-based model to guide the join and exploration process. We extended these techniques to enable the incremental evaluation of KG streams, i.e., by joining the new updates only with the matched triples within a window. Our experimental results show good performance improvements as compared to existing indexing and re-evaluation-based solutions. Our future work includes extending our system to support sliding windows, and extending operators for our automata model, such as sequencing, Kleene-closure and negations. This will enable CEP over KG streams.

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. In *Commun. ACM*, pages 333–340, 1975.

[3] S. Álvarez-García, N. R. Brisaboa, M. A, and G. Navarro. Compressed vertical partitioning for full-in-memory RDF management. *CoRR*, abs/1310.4954, 2013.

[4] A. Appleby. Murmurhash3 finalizer. version 19/02/15. https://github.com/aappleby/smhasher.

[5] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. *CoRR*, abs/1103.5043, 2011.

[6] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, pages 625–636, 2013.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *SIGMOD-PODS*, pages 1–16, 2002.

[8] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *WWW*, pages 1061–1062, 2009.

[9] C. Bizer. The emerging web of linked data. In *IEEE Intelligent Systems*, pages 87–92, 2009.

[10] H. Björklund, W. Gelade, and W. Martens. Incremental XPath evaluation. In *ACM Trans. Database Syst.*, pages 29:1–29:43, 2009.

[11] J. Broekstra, A. Kampman, and F. v. Harmelen. Sesame: a generic architecture for storing and querying RDF and RDF schema. In *ISWC*, pages 54–68, 2002.

[12] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pages 96–111, 2010.

[13] L. Chen and C. Wang. Continuous subgraph pattern search over certain and uncertain graph streams. In *IEEE Trans on Know. and Data Eng.*, pages 1093–1109, 2010.

[14] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[15] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *EDBT*, pages 157–168, 2015.

[16] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. In *ACM Comput. Surv.*, volume 44, pages 15:1–15:62, June 2012.

[17] O. Erling, A. Averbuch, A., M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*, pages 619–630, 2015.

[18] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, pages 925–936, 2011.

[19] M. R. Garey. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[20] A. Gubichev and M. Then. Graph pattern matching: Do we have to reinvent the wheel? In *GRADES*, pages 8:1–8:7, 2014.

[21] A. Y. Halevy. Theory of answering queries using views. In *SIGMOD Rec.*, pages 40–47, Dec. 2000.

[22] Z. Jerzak and H. Ziekow. The debs 2015 grand challenge. In *DEBS*, pages 266–268, 2015.

[23] S. Komazec, D. Cerri, and D. Fensel. Sparkwave:continuous schema-enhanced pattern matching over RDF data streams. In *DEBS*, pages 58–68, 2012.

[24] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC*, pages 370–388. 2011.

[25] B. McBride. Jena: Implementing the RDF model and syntax specification. In *SemWeb*, pages 23–28, 2001.

[26] J. Mondal and A. Deshpande. EAGr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. pages 1335–1346, 2014.

[27] Y. Nenov, R. Piro, B. Motik, I. Horrocks, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab. RDFox: A highly-scalable rdf store. In *ISWC*, pages 3–20, 2015.

[28] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. In *VLDB*, pages 91–113, 2010.

[29] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large

graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, Oct. 2004.

[30] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *ACM Transac. on Database Systems*, pages 16:1–16:45, New York, NY, USA, 2009.

[31] M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In *International Conference on Compiler Construction*, pages 258–270, 1992.

[32] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, pages 406–421, 2012.

[33] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *POPL*, pages 502–510, 1989.

[34] D. Saha. An incremental bisimulation algorithm. In *FSTTCS*, pages 204–215, 2007.

[35] J. Sequeda and O. Corcho. Linked stream data: A position paper. In *SSN09*, pages 148–157, 2009.

[36] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. In *SIGMOD Rec.*, pages 42–47, 2005.

[37] J. R. Ullmann. An algorithm for subgraph isomorphism. In *J. ACM*, pages 31–42, 1976.

[38] D. Wood, M. Lanthaler, and R. Cyganiak. RDF 1.1 concepts and abstract syntax. In *W3C Recommendation, Technical Report*, 2014.

[39] L. Zou, M. T. Ozsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gStore: a graph-based SPARQL query engine. In *VLDB*, pages 565–590, 2014.

APPENDIX

A. NY TAXI QUERIES

NY-Q1.

```
SELECT ?trip ?tripType ?tripdis
WHERE  {
    ?trip rdf:type           ?tripType.
    ?trip :trip_distance     ?tripdis.
    ?trip :pickup_datetime   ?pickupdt.
    ?trip :dropoff_datetime  ?ddt.
    ?trip :mta_tax           ?tax.
    ?trip :trip_time_in_secs ?triptime.
}
```

NY-Q2.

```
SELECT ?trip ?fare_amount ?info
WHERE {
    ?trip    :hasTaxiInfo     ?info.
    ?info    :hasHack_license ?lics.
    ?trip    :hasPickupLoc    ?pickUPL.
    ?pickUpL :hasPickupLat    ?pUPLong.
    ?trip    :hasFareInfo     ?finfo.
    ?finfo   :hasFare_amount  ?amount.
}
```

NY-Q3.

```
SELECT ?trip ?pickupdt
WHERE {
    ?trip :hasTaxiInfo      ?info.
    ?info :hasHack_license  ?lics.
    ?trip :hasPickupLoc     ?pickUpL.
    ?trip :pickup_datetime  ?pickupdt.
    ?trip :dropoff_datetime ?dropoffdt.
}
```

B. SNB QUERIES

SNB-Q1.

```
SELECT ?post ?creator ?loc
WHERE {
    ?post rdf:type           snvoc:Post.
    ?post snvoc:id           ?id.
    ?post snvoc:creationDate ?date.
    ?post snvoc:hasCreator   ?creator.
    ?post snvoc:hasTag       ?tag.
    ?post snvoc:isLocatedIn  ?loc.
}
```

SNB-Q2.

```
SELECT ?forum ?meb ?interest
WHERE {
    ?forum snvoc:title       ?title.
    ?forum snvoc:hasMember   ?meb.
    ?meb   snvoc:hasPerson   ?p.
    ?p     snvoc:hasInterest ?interest.
}
```

SNB-Q3.

```
SELECT ?post ?creator ?id
WHERE {
    ?cmnt    snvoc:hasCreator   ?creator.
    ?creator snvoc:speaks       ?lang.
    ?cmnt    snvoc:isLocatedIn
        <http://dbpedia.org/resource/Nicaragua>.
    ?cmnt    snvoc:replyOf      ?post.
    ?post    snvoc:hasCreator   ?id.
}
```

C. LSBENCH QUERIES

LS-Q1.

```
SELECT ?post1 ?user ?post2
WHERE {
    ?post1 snvoc:content ?cont.
    ?post1 snvoc:hasCreator ?user.
    ?post2 snvoc:content ?cont2.
    ?post2 snvoc:hasCreator ?user.
}
```

LS-Q2.

```
SELECT ?post ?person1 ?person2
WHERE {
    ?person1  snvoc:likes     ?p.
    ?p        snvoc:hasPost   ?post.
    ?p2       snvoc:hasPost   ?post.
    ?person2  snvoc:likes     ?p2.
    ?person1  snvoc:knows     ?know.
    ?know     snvoc:hasPerson ?person2.
}
```

LS-Q3.

```
SELECT ?post1 ?user1 ?user2
WHERE {
    ?post1 snvoc:hasCreator  ?user.
    ?post1 snvoc:content     ?cont.
    ?know  snvoc:hasPerson   ?user.
    ?user2 snvoc:knows       ?know.
    ?user2 snvoc:hasInterest
        <http://dbpedia.org/resource/Charles_Dickens>.
}
```