Continuous Graph Pattern Matching over Knowledge Graph Streams

Syed Gillani, Gauthier Picard, and Frédérique Laforest, Laboratoire Hubert Curien, UMR CNRS 5516, and Institute Henri Fayol, EMSE

1. INTRODUCTION

In the recent years, there has been growing interest in *Knowledge Graph* (KG) [1] structured data *streams*: arising in social networks, sensor networks, financial transaction networks, to name a few. These semantically enriched KG streams enable the extraction of contextual information using background static information and ontologies.

The main properties of interest in these studies (including static KG processing) are based on one fundamental parameter: Graph Pattern Matching (GPM). Given a KG and a query graph Q, the problem of graph pattern matching is to find all the possible subgraphs of KG that match a set of patterns described in Q. This problem shares the same search space with pattern matching via subgraph isomorphism [Garey 1990], which has proven to be a NP-Complete problem, and in practice can be solved with join based or exploration based methods [Neumann and Weikum 2010; Zou 2014]. In dynamic stream setting this problem can be described as continuous GPM (CGPM), where each KG-based event is matched continuously with a query graph.

Existing solutions for both static and dynamic KGs based their execution strategies on the index-store-query model, where extensive indexing techniques are utilised to reduce the search space to match query graph patterns. However, KG streams are massive (e.g., terabytes in volume), temporally ordered, fast changing, and potentially infinite. This raises new challenges; it is prohibitively expensive to index and store dynamic KGs frequently enough for applications with real-time constraint. Worse still, there are two different execution models used on KG streams; event-based CGPM and incremental CGPM. We describe these model with an example in Fig.1.

Event-based CGPM: Fig. 1(a,b) presents two KG-based events emanating from a house, while Fig. 1(c) presents a query graph for such events. For each execution of the query graph, all the attributes are present in each event and they are continuously matched to provide the final results. The application of such a model is prominent in systems designed for complex event processing, where each event is processed independently. Sequence or aggregate operators are used to collect a set of required values from each event.

Incremental CGPM: Fig. 1(d,e) presents two KG-based events emanating from a social network. Fig. 1 (f) describes a query graph for such events. Note that in contrast to the event-based model, in this setting the entire graph of interest is no longer available all the time, but individual edges/triples or set of edges/triples are received, and are matched with the query graph. The main aim of incremental CGPM is to minimise the recomputation by utilising already computed results [Fan et al. 2011]. Its main application is usually in general stream processing or reasoning systems [Le-Phuoc and Dao-

[1] Various abstractions have been defined for KGs, however, in this paper, we utilise an RDF graph model for its representation. Hence, it is a multigraph with a set of *triples* ($\langle subject, predicate, object \rangle$) (*triple patterns* for a query graph), see Section 2 for more details.

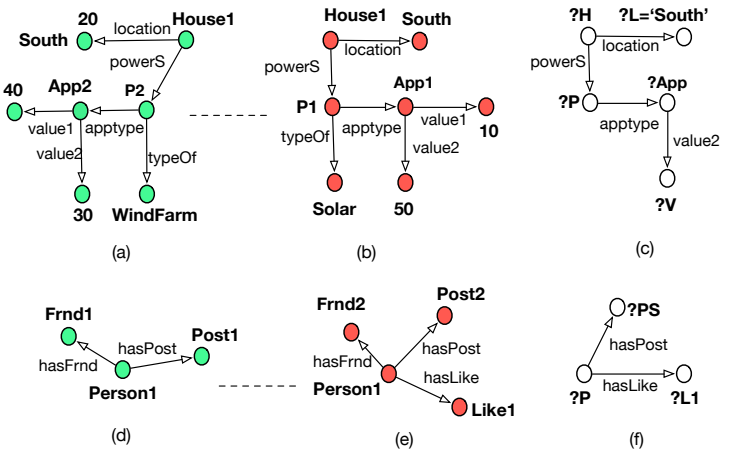

Fig. 1: (a,b) KG-based Events for event-based CGPM. (c) Query graph from events a and b . (d,e) KG-based updates for Incremental CGPM. (f) Query graph for events d and e.

Tran 2011; Barbieri and Braga 2009; Choudhury and Holder 2015].

Challenges: The key challenges to overcome in order to achieve efficient CGPM over KG streams are as follows.
(1) *Cost of indexing in streaming settings:* As noted above, existing static and dynamic GPM solutions heavily rely on extensive indexing techniques to reduce the search space; data are indexed in all possible ways allowing every execution step to be supported by some index. These indexing techniques incur super linear space, an/or super linear construction time. For example RDF3x [Neumann and Weikum 2010] builds several clustered B+trees for all permutation 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 [Cheng and Yu 2008] approach for subgraph isomorphism is based on 2-hop indexing, with the time complexity of $O(n^4)$ (n, number of vertices) to build such indices. Thus, these techniques are quite expensive and unrealistic to use in a system with millions of updates per second.
(2) *On the fly computation:* The standard streaming constraint of being able to process every triple/edge only once (i.e., on-the-fly processing) 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 [Stonebraker et al. 2005]. This requires a light-weight data structure, which is efficient enough for write and query intensive system. Furthermore, it should be able to meets the demands of scalability due to the large volume of the streams. For example, a social network containing 10^7 users, can produce a large number of distinct edges in the order of 10^{14}.
(3) *Incremental Evaluation:* Incremental algorithms have proven useful in a variety of applications [Ramalingam and Reps 1989]. However, the field of incremental GPM is still quite fertile and only few solutions have been proposed [Fan et al. 2011; Saha 2007; Choudhury and Holder 2015]. In order to reduce the complexity measures, most solutions use, (i) a relaxed technique *graph simulation* [Milner 1989] for pattern matching, or (ii) indexing techniques (B+ trees) are utilised to incrementally match the new and old triples [Le-Phuoc and Dao-Tran 2011]. Furthermore, most of these solutions are based on a triple stream model and thus, cannot be directly used for a KG model, where each KG-based event contains a set of triples.

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 partitioning* technique, where KG-based events are partitioned by structure analysis of the query graph. The partitioned graph is materialised into a set of view using a novel *Bidirectional MultiMap* data structure. Thus, query processing is performed on a set of view instead of the whole graph. In our approach, no indexing technique is used. To make up for the performance loss due to lack of indexing, we efficiently employ the set of views, and distribute the query graph into a set of subquery graphs, and finally implement join ordering between a set of patterns, based on the cardinality of the views. (Section 3)

(2) Considering the on-the-fly requirement, we propose a novel automata-based triple pattern join (TP-Join) algorithm. The basic idea is to map a set of patterns for a query graph on a set of automaton states, where the automaton guides the join process in an iterative manner. This provides an adaptive and incremental join procedure, and can be extended for much more complex patterns. Furthermore, for each execution of an automaton, we make sure that it percolates only if there is enough evidence that future join operations would be required. (Section 4)

(3) We extend the above mentioned approaches to enable incremental evaluation of CGPM using a lazy evaluation strategy. That is, the join operations between query graph patterns are initiated only if all the patterns have a non-empty set of mappings in materialised views. This reduces the number of unnecessary joins and enables the incremental evaluation of new matches, while considering the joins between the already matched set of triples. (Section 5)

(4) Using real-world and synthetic KGs, we experimentally verify the performance of our event-based and incremental CGPM algorithms. Our system outperforms existing in-memory graph processing engines and triple-based stream processing systems by many orders of magnitude, in terms of performance and scalability. (Section 6)

To the best of our knowledge, this work is among the first efforts to develop a unified framework for processing both event-based and incremental CGPM over KG streams. Our proposed techniques are flexible enough to be easily adopted into a wide variety of applications, such as Complex Event Processing, stream reasoning and Top-K computation over KG streams.

2. PRELIMINARIES

In this section we present the preliminary discussion about the KGs and query graphs, and later extend it to describe the problem statements. We use RDF [ref f] model for KGs, where data, a set of $\langle subject, predicate, object\rangle$ triples ($\langle s, p, o\rangle$ for short) , can be represented as a graph according to the following definition.

Definition 2.1. (Knowledge Graph). A Knowledge Graph is a directed labelled multigraph denoted as $G = (V, E, L)$, where V is a set of vertices, corresponding to all the subjects and objects ($s, o \in V$) of triples. $E \subseteq V \times V$ is a set of directed edges from subjects to objects, corresponding to predicates. L is a labelling function which maps from a vertex or an edge to the corresponding label set or label respectively.

Each edge $e = (v_i, v_j) \in E$ denotes a directed relationship from a vertex v_i to v_j and it has a label denoted as $L(e)$, where (v_i, e, v_j) represents a KG triple t. Multiple edges can have the same label. Fig. 1(a,b) presents two KGs, where a labelled-edge `powerS` represents a directed relationship between (`House1`,`P1`) and (`House1`,`P2`).

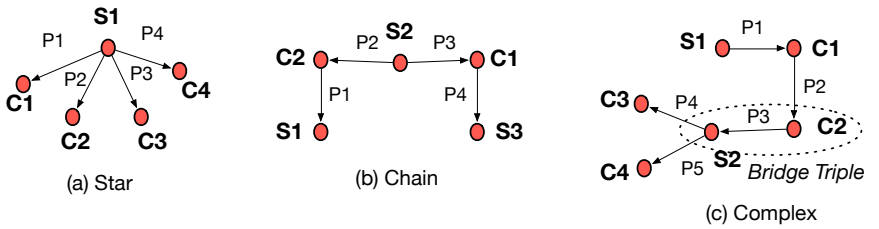

Fig. 2: (a,b,c) Query graph patterns (star, chain and complex).

A query graph Q can be modelled as a set of triple patterns. Each triple pattern tp is a triple that contains variables in the subject, predicate or object. Then Q can be denoted as $Q = \{tp_1, tp_2, .., tp_m\}$, where $tp_i (1 \leq i \leq m)$ is a triple pattern. Triple patterns are usually connected by shared subject or object and a join occurs on their shared subjects or objects. Thus, the exemplary types of joins in query graph are subject-subject join (s-s join), subject-object join (s-o join) and object-object join (o-o join). In this paper, we only consider the connected query graphs and we do not consider predicate join, which is not very common as shown in previous study [Arias and Fernández 2011]. Based on the above mentioned joins, the two main types of patterns in a query graph are : (i) star-shaped (s-s join), (ii) chain-shaped (s-o join). These patterns are used in a combination to build more complex patterns, as shown in the Fig. 2(a,b,c).

The semantics of the joins can be defined by using a mapping terminology. A mapping μ is a partial functions $\mu : var \rightarrow (V \cup E)$, where var is a set of variables. We call $\mu(?v)$ the variable binding of μ for $?v$. Abusing notation, for a triple pattern tp we call $\mu(tp)$ the triple that is obtained by substituting the variables in $\mu(tp)$ according to μ. The domain of μ, $dom(\mu)$, is the subset of var where μ is defined. Then, the join between the mappings of two triple patterns tp_i and tp_j can be defined as,
$map(tp_i) \bowtie map(tp_j) = \{\mu_i \cup \mu_j \mid \mu_i \in map(tp_i),\ \mu_j \in map(tp_j)\}$.

Consider the triple patterns tp_1 and tp_2 of query graph in Fig. 1(c), the join between tp_1 and tp_2 mappings, extracted from the KG presented in Fig. 1(a) is as, $map(tp_1) \bowtie map(tp_2) = \{$(`House1,location,South`)$\}, \{$(`House1, powerS,P1`)$\}$. In the rest of the paper, we interchangeably use the term triple and triple pattern mappings for a query graph.

Definition 2.2. (Query Match). A query graph Q is a match to a KG G = (V, E, L), if the application of Q on G, denoted as (Q(G)) produces a graph $G' = (V', E', L')$, which is isomorphic (or subsequently homomorphic) to G. Formally there exist an injection function $F : V' \rightarrow V$, such that, (1) $\forall v' \in V'$, $L'(v') \subseteq L(F(v'))$ and (2) $\forall (u', v') \in E'$, $(F(u'), F(v')) \in E$ and $L'(u, v) = L(F(u'), F(v'))$.

The query match problem is to find all the distant subgraph isomorphisms of a query graph in a data graph.

The execution of the query graph in Fig. 1(c) on two KGs in Fig. 1(a,b) produces two matched subgraphs, such as $G'^1 = \{$(`House1,location,South`), (`House1,powerS,P2`), (`P2,apptype,App2`), (`App2,value2,30`)$\}$ and $G'^2 = \{$(`House1,location,South`), (`House1,powerS,P1`), (`P1,apptype,App1`), (`App1,value2,50`)$\}$.

2.1 Problem Analysis

In this section, we present the analysis of continuous graph pattern matching and its two variants; event-based and incremental CGPM over KG streams

Definition 2.3. (Continuous Graph Pattern Matching). Given a query graph Q, a KGs stream $\mathcal{G}$ and a function M. The CGPM is that, for each $G^i \in \mathcal{G}$, the function $M(Q, G^i)$ returns a graph G'^i, such that all the conditions hold in Definition 2.

This problem can easily be extended to event-based CGPM with an addition of temporal sequence.
Problem 1: (Event-based CGPM). This problem can be generally described as, given a query graph Q, a function M and a sequence of KGs $\{(G^1, \tau_1), (G^2, \tau_2), \ldots, (G^n, \tau_n)\}$ (ordered by timestamps), where each KG-based event is associated with a timestamp τ_i. Then for each of such graphs, function $M(Q, G^i)$ returns a graph (G'^i, τ_i), such that all the conditions hold in Definition 2.

Problem 2: (Incremental CGPM). The problem of incremental CGPM is somewhat different and can be described as, given a query graph Q, consider an evolving KG G, and $(\Delta G^i, \tau_i)$ as updates to G, such that the updates conform to a stream $\mathcal{G} = \{(\Delta G^1, \tau_1), (\Delta G^2, \tau_2),$
$\ldots, (\Delta G^n, \tau_n)\}$. Each of these graph updates $(\Delta G^i, \tau_i)$ contains a set of triples and is associated with a timestamp τ_i. Then incremental CGPM is to continuously compute changes $\Delta M_i = M(Q, (\Delta G^i, \tau_i))$ to the matches by considering all the conditions in Definition 2, and $M(Q, \bigcup_{k=0}^{i-1}(\Delta G^k, \tau_k) \oplus (\Delta G^i, \tau_i)) = M(Q, \bigcup_{k=0}^{i-1}(\Delta G^k, \tau_k)) \oplus \Delta M_i$, where operator $\oplus$ incrementally applies changes to the matched graphs.

As opposite to the event-based model that processes each KG-based event independently, an incremental evaluation model incrementally identifies changes in response to the arrival of new updates in the stream. With loss of generality, in both processing models we consider the time-interval based window, i.e., for a current time τ and given the time-interval window, the match contains the graphs with timestamps $\tau' \leq (|w| - \tau)$. Thus, our streaming model is based on tumbling windows.

THEOREM 2.4. *The general complexity of CGPM problems is PTime or O($|Q|.|G^i|$), where $G^i \in \mathcal{G}$.*

Proof: See Appendix A.

The size of each graph $G^i \in \mathcal{G}$ has a huge impact on the query performance. As noted earlier, indexing is not a viable solution in streaming settings, thus our first goal is to partition and prune each graph, and consequently to reduce the search space to match triple patterns $tp \in Q$. In the next section, we describe our graph partitioning and pruned view materialisation approach. Based on this approach, we present the TP-join algorithm to match a set of triple patterns within a query graph (Section 4). Note that both CGPM and event-based CGPM problems share the same methodology for their solutions. Thus, we first present our proposed approach for event-based CGPM problem (Problem 1) and extend it for incremental CGPM (Section 5).

3. GRAPH PARTITIONED VIEW MATERIALISATION

In this section, we first provide our basic partitioning strategies and data structure which serve as the backbone for the two CGPM models discussed above. The two main points to consider for KG perprocessing are: (1) scalability due to the large size of the streams, (2) reducing the search space without using extensive indexing.

In traditional relational systems, scalability challenges have been successfully addressed by partitioning the data collection and by processing queries/subqueries in a parallel or distributed environment. Additionally, the concept of views has also been floated in relational query optimisation to minimise the search space. Given a query Q and a set $V = \{\mathsf{V}_1, \mathsf{V}_2, \mathsf{V}_3, \ldots, \}$ of views, the idea is to find another query $Q*$, such that $Q*$ is equivalent to Q, and $Q*$ only refers to views in V [Armbrust and Liang 2013; Halevy 2000]. This technique provides considerable advantages, where given a dataset D, one can compute the answers $Q(D)$ in D by using V without accessing D. Motivated by these properties, and in order to make up for the performance loss due to the lack of indexing, we implement

query-based partitioned views. Each of these views contains an exclusive set of triples that corresponds to the properties of the triple patterns defined in the query graph. Note that, a set of vertices is denoted by V, while a set of views is denoted by V.

3.1 Query-based Graph Partitioning

The basic idea of partitioning graph data into a set of views is to collect all the valid triple mappings for each $tp_i \in Q$ and prune all the "dangling" triples that would not be useful during the query graph evaluation. The important properties that can be exploited for such pruning and partitioning strategies are as follows.

PROPOSITION 3.1. *Given a query Q and a graph G^i, the cardinality of triple patterns in Q is , $|E_Q| \in Q \leq |E_G| \in G^i$ to the number of edges in G^i .*

The set of edges $E \in G^i$ is a multiset instead of a set, therefore multiples edges may have same vertices. The cardinality of directed edges $|E|$ represents the total number of edges $E \in G^i$. Before describing the second property, we first define the filter constraint.

Definition 3.2. (Filter Constraint). For $tp \in Q$, a mapping $\mu \in map(tp)$, var as set of variables in Q, and if b is a boolean expression (value constraint) and $var(b) \in var(Q)$, then a filter constraint for tp can be defined as,

F(map(tp), b) = $\{\mu \mid \mu \in map(tp) \ \wedge \ \mu(b)\ is\ true\}$.

The type of the filter constraint can be described inductively as follows.

—*If $x \in var(tp)$, and b is of the form $x = c$, $x \neq c$, $x > c$, $x < c$, where c is constant, then such $F(map(tp), b)$ is described as an atomic filter constraint.*

—*If $x \in var(tp_i)$ and $y \in var(tp_j)$, where $tp_i,\ tp_j \in Q$ and b is of the form $x = y$, $x \neq y$, $x > y$, $x < y$, then $F((map(tp_i) \bowtie map(tp_j)), b)$ is described as complex filter constraint.*

Triple patterns usually use the filter constraints in Q to get more customised results. For example, in Fig. 3(b), an atomic filter constraint (`?o1 = sunny`) is used to get all the houses with the weather condition as `sunny`.

PROPOSITION 3.3. *The cardinality of $|F(map(tp), b)| \leq |map(tp)|$.*

Example 1: Consider Q and G^i in Fig. 3(b,a). The mapping of tp_1 is $map(tp_1) = \{$`(house2, sunny), (house1, raining)`$\}$. By pushing the atomic filter constraint during the graph partitioning phase $F(map(tp_1), b)$, where b is $?o1 =$ `sunny`, we can get rid of the mapping $\{$`(house1, raining)`$\}$ that would not produce any matched result during the join operation of mappings.

Based on these properties, the query-based partitioned views can be defined as follows.

Definition 3.4. (Query-based Partitioned Views). Given Q and G^i, the query-based partitioning over G^i is to divide all the triples in G^i into m disjoint views $\{V_1, V_2, \ldots V_m\}$, where each V_i contains an exclusive subset of G^i that complies to the property 1 and property 2 (if there exists a filter constraint).

The use of query-based graph partitioning enables the optimal partitioning and pruning of input KG and makes sure that only relevant triples/edges are used for the query evaluation. Each view is continuously updated for each KG-based event, and thus containing a set of triples at time τ_i. Note that our naive partitioning technique may contains triples that may not produce results during query evaluation. However, in practice $|Q| << |G^i|$, thus it still provides efficient performance (as described in the experimental section) in streaming settings – without incurring the heavy cost of indexing. Next we describe our in-memory data structure for views.

3.2 Bidirectional MultiMap Data Structure

An important challenge to keep up with the write-intensive stream applications is to design a light-weight data structure, which not only provides lower bounds of insertion but also can be utilised in an efficient manner during query evaluation. That is, it should provide a linear subject-object and object-subject joins over a set of views. To address this challenge, we develop a *Bidirectional MulitMap* data structure. Traditionally, MultiMaps are associative containers that contain values to keys in a way that there is no limit on the number of elements with the same key (an important property of KG with multiple same subjects). It allows constant look-ups (considering there are no hash-collisions) by keys. However, chain-shaped query graphs generally involve in object-subject joins. Thus to implement linear object-subject joins, we designed a Bidirectional MultiMap.

The design of our data structure is based on three hash tables. In table 1, we store all the (key,$value$) pairs using each entire key-value pair as the key. With each such pair, we store the count which identifies an ordinal number for this value associated with this key, starting from 0. For example, if the keys were (4, Alice), (4, Bob), and (4, Eve), then (4, Alice) might be pair 0, (4, Bob) pair 1, and (4, Eve) pair 2, all for the key, 4. In table 2, we store all the unique keys. For each key, we store a pointer to the array, a_k, that stores all the key-value pairs having a key, stored in order by their ordinal values from table 1. With the record for a key, we also store n_k, the number of pairs having that key, i.e., the number of key-value pairs in a_k. For example, the key 4 from earlier example will contain a list 0, 1, 2. In table 3, we store unique values from Table 1 and a pair key-index, where index describes the location of the value in the list in Table 2. For instance, from the earlier example, Table 3 will contain bob as key and (4,0) as a value. This results in constant time complexity for look-up, add and remove operations. Furthermore, we have implemented three different kinds of hashing, MD5, SHA1 and SHA256, depending on the expected size of the input graph and use cases. Note that, the use of SHA256 results in a negligible probability of collisions (at least 2^{128}), thus providing constant look-ups.

3.3 Triple Encoding and View Materialisation

In previous sections, we describe the procedure of partitioning the graph according to query structure and materialising views using Bidirectional MultiMaps. In this section, first we describe the encoding of triples to be stored in the views. Second, we describe the algorithm for materialising a set of views V .

Triple Encoding. Encoding triples by utilising 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 [Neumann and Weikum 2010; Zou 2014], we implement a synchronised in-memory dictionary for dynamic 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 only encodes the data in the views. Both of these dictionaries are synchronised in a way that both use the same encoding for same strings. Thus, persistent encoded values of a query graph are not effected if (1) an adaptive dictionary is cleared after the data from the set of views is processed or if (2) the temporal window expires. Leveraging this, the size of the encoded values does not explode linearly with the size of the streams for longer temporal windows.

View Materialisation. As noted earlier, we partition each KG-based event according to the Properties 1 and 2, and materialise the resulted triples in a set of views, each for a $tp_i \in Q$. Algorithm 1 describes the materialisation of a view. Given a triple pattern $tp_i \in Q$ and and an event G^i, it returns a view V_i containing a subset of triples in G^i. It uses two subroutines. The first one (lines 5-8) iterates over all the triples $t \in G^i$ and compares the predicate $pred_t p$ of a triple pattern tp and $pred_t$ of a triple

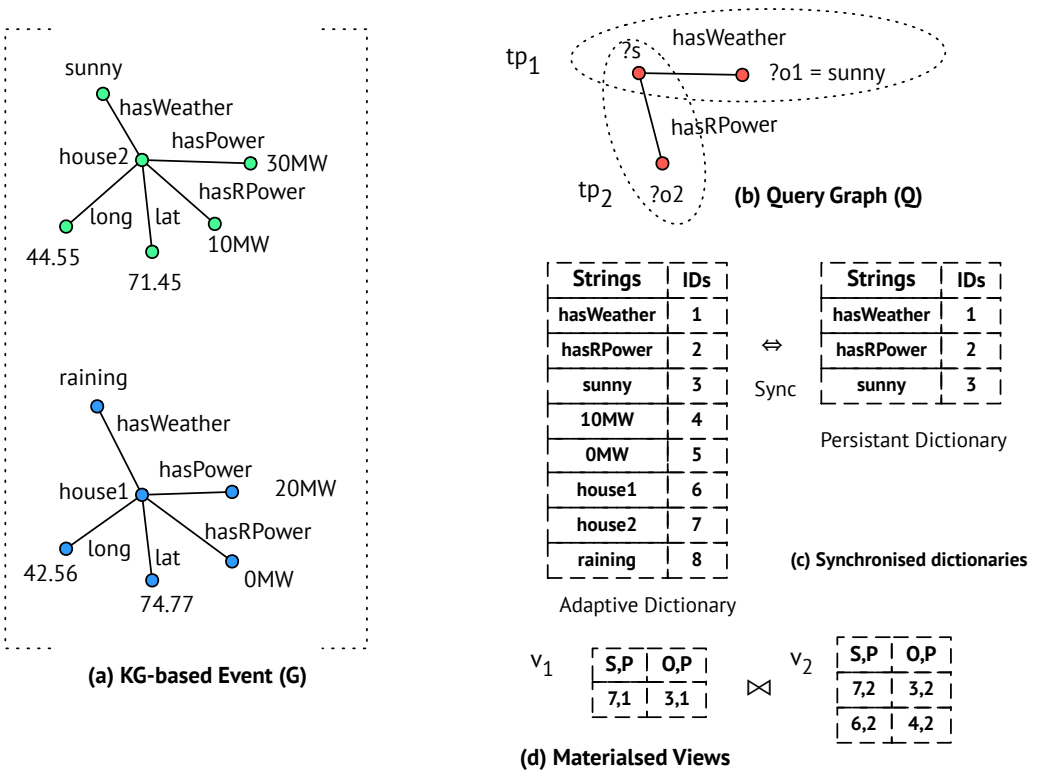

Fig. 3: (a) KG-based event. (b) Query graph for CGPM, (c) synchronised dictionary for encoding triples and triple patterns. (d) Materialised view for tp_1 and tp_2 for the query graph Q.

$t \in G^i$. If the match yields to true (i.e., $pred_{tp_i} = pred_t$), it first encodes the value of t ($\texttt{Encode}(t, d_{adap})$) by utilising the adaptive dictionary. Then the encoded triple t' is added into the view $\mathsf{V_i}$ (line 8). The second subroutine (lines 9-14) takes as an input the same tp_i and iterates over the values in view $\mathsf{V_i}$. It first checks if the subject/object of tp_i is a variable or a constant. In the case of constant subject/object, it simply compares sub_{tp_i} and sub_t. If $sub_{tp_i} \neq sub_t$ it removes $t' \in \mathsf{V_i}$ from $\mathsf{V_i}$ (line 12). Then it checks if tp_i is assigned with an atomic filter constraint (either for subject or object) (line 13). If so, it uses that filter constraint b along-with the binary operators $(>, <, =, \neq)$ to further prune t' from $\mathsf{V_i}$ (line 14). Finally $\mathsf{V_i}$ contains a set of materialised mappings that satisfies all the conditions defined in tp_i.

Note that Algorithm 1 describes the materialisation of a single view. Each view is disjoint to others, thus, the process of partitioning and materialisation of views can take advantages of multicore processing to parallelise the computation.

THEOREM 3.5. *Algorithm 1 produces a feasible set of views $V = \{\mathsf{V_1}, \mathsf{V_2}, \ldots, \mathsf{V_m}\}$, such that the query graph Q can be answered using V and $Q(V) = Q(G^i)$.*

Example 2: Consider Q in Fig. 3(b), the object of tp_1 contains an atomic filter condition (`?o1 = sunny`). The KG G^i in Fig. 3(a) contains two different instances of the house and its relative attributes. The strings in the query graph are mapped into numeric IDs using a persistent dictionary (Fig. 3(c)). During the first part of the algorithm, predicates (`(hasWeather, hasRPower)`) in Q are used to partition the graph G^i, and pruned triples are materialised in views $\mathsf{V_1}$ and $\mathsf{V_2}$. Triples in materialised views are

Algorithm 1 View Materialisation

Input: Triple Pattern tp_i, Knowledge Graph G^i, Filter constraints b
Output: Materialised view $\mathsf{V_i}$

$G \leftarrow \{t_1, t_2, \ldots, t_i\}$ ▷ Input KG-based event
$tp_i \leftarrow \langle sub, pred, obj \rangle$ ▷ Triple Pattern
$d_{adap} \leftarrow$ adaptive Dictionary
$\mathsf{V_i} \leftarrow$ empty view
while *triples* $t \in G$ **do**
  if $pred_{tp} = pred_t$ **then** ▷ predicate comparison
    $t' :=$ `Encode`(t, d_{adap}) ▷ adaptive dictionary encoding
    $\mathsf{V_i} := \mathsf{V_i} \cup \{t'\}$
while triples $t' \in \mathsf{V_i}$ **do**
  ▷ subject and object comparison
  if $sub_{tp_i} \notin var(tp_i)$ and $sub_{tp_i} \neq sub_{t'}$ **then**
    $\mathsf{V_i} := \mathsf{V_i} \setminus \{t'\}$
  else if $sub_{tp_i} \in var(tp_i)$ and $sub_{tp_i} \in var(b)$ and $F(sub_{t'}, b) \neq sub_{t'}$ **then** ▷ atomic filter comparison
    $\mathsf{V_i} := \mathsf{V_i} \setminus \{t'\}$
  ▷ Same If block for the objects

encoded using an adaptive dictionary (Fig. 3(c)). As there are two predicates in Q, triples only with those predicates are encoded in the dictionary. The second part of the algorithm utilises an atomic filter constraint for tp_1 and further prunes the unwanted triples from the materialised views. The final set of partitioned triples for each triple pattern is stored in our data structure (Bidirectional Multimap) (Fig. 2(d)). After executing the joins between triple patterns (described in next section), these encoded triples are cleared from the adaptive dictionary.

3.4 Query Decomposition and Join ordering

In this section, we describe few preprocessing steps on the query graph before evaluating it using a set of views. A query graph consists of two main patterns, namely chain and star-shaped. Other complex patterns can be formed by using them in combination. Inspired from traditional query optimisation approaches [Gubichev 2014; Stocker and Seaborne 2008], we decompose a query graph into a set of subquery graphs $Q = \{SQ_1, SQ_2, \ldots, SQ_k\}$. Each of subquery graph contains a set of triple patterns corresponding to a chain or star-pattern. Such decomposition enables a concurrent execution of subquery graphs. If two subquery graphs are connected by a shared variable, then a join between them produces the final result. Additionally, if any subquery graph produces an empty intermediate result set, it provides a non-refutable evidence that there is no need to process the joins between remaining subquery graphs.

Join Ordering. The joins between a set of triple patterns are commutative and associative [Pérez et al. 2009]. Let $Tp \in Q$ be a set of triple patterns, each sharing a join variable with other triple patterns, then the problem of ordering a set of triple patterns consists of finding an ordered set Tp', such that for two triple patterns tp_i and tp_j, tp_i precedes tp_j iff $|map(tp_i)| \leq |map(tp_j)|$. An efficient join ordering results in smaller intermediate joined results leading to the lower cost of future join operations. In general static settings, this problem is solved through (i) cardinality estimation [Gubichev 2014] of mappings for triple patterns $tp \in Q$, (ii) dynamic programming algorithms or greedy heuristics [Moerkotte 2006; Neumann and Weikum 2010] to implement join ordering plans of lower cost. These techniques are useful, if an extensive indexing is used for the KG, and could be quite expensive in streaming settings – considering the lack of indexing. For example if there are 10 triple patterns

$tp \in Q$, then there can be 10! optimal join ordering plans. In our algorithm, we materialised a set of views for each triple patterns, thus the cardinality measures of each view can be used to implement the join ordering between a set of triple patterns. This property also highlights our approach of materialising views before processing joins for a query graph Q.

The three main steps of our join ordering technique are as follows, (1) cluster the query graph into a set of subquery graphs according to the chain and star-patterns, (2) order triple patterns within each subquery graph according to the cardinality of mappings in each view (after executing Algorithm 1), (3) order the execution of joins within each subquery graph according to their combined cardinality of mappings. Note that the join ordering is performed to process each $G^i \in \mathcal{G}$. However, the clustering of subquery graphs is performed during query compilation.

4. CONTINUOUS QUERY EVALUATION

In this section, we present our algorithm to evaluate joins between a set of triple patterns $tp \in Q$, while utilising a set of views V materialised from the input KG G^i. Several approaches [Halevy 2000; Cyganiak 2005; Sakr and Elnikety 2012] have been proposed for compiling query graphs, including core SPARQL algebraic and relational algebraic plans. However, these algebraic plans (SPARQL or relational) are based on already indexed and stored data. Thus, are not suitable for dynamic graphs, where a query plans have to adapt according to each event. Therefore, we introduce an automata-based approach for compiling and executing query graphs.

TP-Join Automata. Pattern matching using automata models has been studied for several decades [Aho and Corasick 1975; Pettersson 1992]. Generally, patterns are pre-processed into a finite automaton (non-deterministic or deterministic). Thus, allowing the automaton to determine the matched patterns by a single scan of the input. Considering the on-the-fly nature of KG stream 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 expressible, and are closed under union, intersection and Kleene closure [Pettersson 1992]. 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 [Björklund 2009].

Considering these properties, we implemented a finite automata-based technique to evaluate joins between triple patterns. The ordered set of $Tp' \in Q$ is modelled as a set of automaton states. Each state is associated with an edge, which is labelled with a transition rule, i.e., join conditions between triple patterns.

Definition 4.1. (TP-Join Automaton). Given a query graph Q with an ordered set of triple patterns Tp', the TP-Join automaton A is a tuple (S, E, θ, s_i,s_f), where S is a set of states, $E \subseteq S \times S$ 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 (V, R, J_t, J_{id}), where V is set of views generated from Algorithm 1, R is a set of views to store the intermediate join results, J_t is the type of the join between the triple patterns (s-s, s-o, o-s), 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. 4). The `Join` operator uses the transition rules mapped at edges and implements the required join method with the appropriate triple patterns. The objective of the `Proceed` operator is to check the status of the `Join` operator,i.e., if (1) the `Join` operator produces a non-empty intermediate joined result set or if (2) it has already been executed (as described later) then the automaton transits to next state, otherwise it terminates the execution of the automaton with no match. The core functionality of `Join`

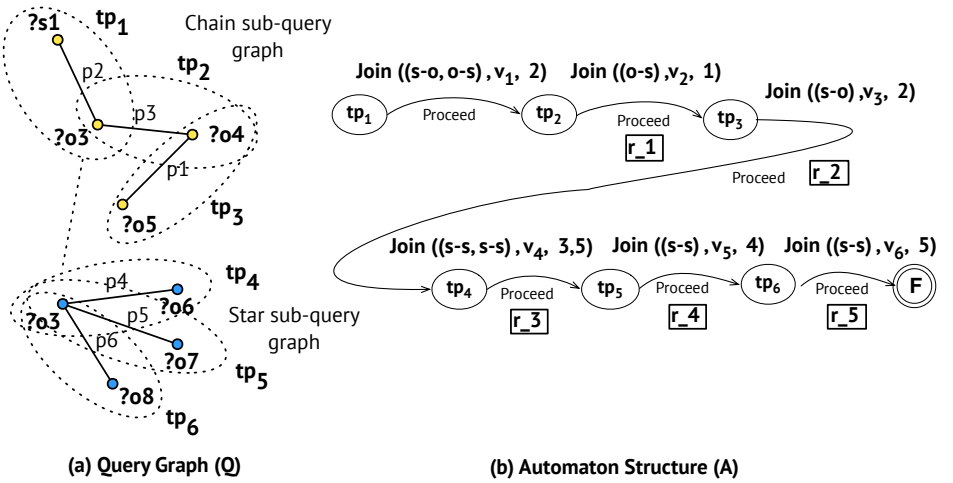

Fig. 4: (a) A query graph containing chain and star-shaped triple patterns. (b) The transformation of query graph into an automaton for TP-Join algorithm

operators is implemented using hash and sort-merge joins. Both are two well-known join algorithms with good performance [Kim 2009]. However, hash join can only handle equi-joins (atomic filter, simple s-o, o-s, s-s, o-o joins), while merge-sort joins can handle non-equi-joins (complex filters, see Definition 6). Therefore, we use both types of joins, where equi-joins are handled using hash-merge joins, while sort-merge joins are used non-equi-joins.

Another important property of using an automata model to evaluate query graphs is percolation. The TP-Join algorithm has to percolate: if at some point there is not enough evidence to join a new triple pattern, the algorithm should stop. If this happens before or after significant number of triple patterns have joined, the algorithm fails. If the algorithm does percolate, it has to percolate correctly, as an incorrect join of mappings may percolate to other incorrect mappings in further steps. This could possibly lead to cascades of errors while wasting computing resources.

Algorithm. Algorithm 2 outlines the overall execution of the TP-Join algorithm. It executes only if the triple pattern mappings (i.e., triples) in a view $\mathsf{V}_\mathsf{i} \neq$ for all $tp_i \in Q$. It has the following steps:

`SortandCombine` (lines 1-4). For each $tp_i \in Q$, the algorithm maps each tp_i i.e., star and chain-shaped to the corresponding automaton. As noted earlier, we cluster the query graph into subquery graphs, where triple patterns within each subquery graph are ordered according to the cardinality of the materialised views. Once each automaton is constructed, the algorithm takes the union of states in each automaton and adds A_{chain}, A_{star} in final automaton A (line 4). Note that this procedure is executed for each execution of the TP-Join algorithm, thus providing a dynamic join-ordering technique.

Automaton Traversal (lines 6-13). Each state $s \in A$ in an automaton is traversed to conduct the joins between the set of views V. First, the algorithm locates the joined state s_{join} of the current state (line 8). For example, in Fig. 4(b) tp_1 mapped at state 1 has a join with tp_2 mapped at state 2. Note that due to the join ordering, the join state for chain-shaped triple patterns can be at any other position than previous state; however, for star shaped subquery graphs, it is the previous state (except for first state). It then executes the `Join` procedure between the view V_s of current state and intermediate joined result view $r_{s_{join}}$ of joined state ($\mathsf{V}_{s_{join}}$ for first state) (line 9). If the result of the `Join` produces

Algorithm 2 TP-Join algorithm

Input: Set of triple patterns $tp \in Q$, set of views V (Algorithm 1)
Output: Matched graph , $M(Q, G^i) = G'^i$

```
1:  for each tp ∈ Q do
2:      A_chain, A_star := Add(tp)
3:                                              ▷ add tp to the corresponding Automaton
4:  A := SortAndCombine(A_chain, A_star)
5:  r ← an empty merged triples view
6:  for each s ∈ A do
7:      s_join := GetJoinState(s, A)
8:      V' := Join(V_s, r_{s_join}, s, A)
9:      if V' ≠ then
10:         r_s := r_s ∪ {V'}
11:         Proceed to next state
12:     else
13:         Terminate the execution of A
14:
15: Procedure JOIN
Input: Views l_s, l_{s_join}, current state s, complex filter b ∈ s, triple pattern tp_i ∈ s, automaton A.
Output: Merged joined mappings
16:     V'' := HashMergeJoin(l_{s_join}, l_s)                ▷ use hash-merge join
17:     if var(tp_i) ∈ var(b) then
18:                                  ▷ if current state's tp_i has complex filter condition
19:         s'_join := GetJoinState(s, A)
20:                                  ▷ search the state having complex filter join to state s
21:         return SortMergeJoin(V'', r_{s'_join})       ▷ V_{s'_join} if |r_{s'_join}| =
22:     else
23:         return V''
```

$|V'| \neq$ (line 10), then joined mappings are merged into r_s and automaton `Proceeds` to next state (line 12). Otherwise it terminates the execution, as further joins are not necessary (line 16). Note that the `Join` procedure is executed only once for each triple pattern. If a triple pattern is already joined with other triple patterns the algorithm skips it during automaton traversal. For example, in Fig. 4(a) tp_1 joins with tp_2 and when the automaton is at state 2 to implements the join between the views of tp_2 and tp_1, it skips to next state, as tp_2 mappings are already joined and merged in r_1.

Procedure `Join` (lines 15-23). The `Join` procedure is invoked to implement the hash/sort merge joins. It takes two sets of views l_s (current state) and $l_{s_{join}}$ (target joined state). It first finds the smallest of the two to determine the order of the hash-merge join. Then, depending on the join condition, it implements s-s, o-s or s-o joins and merges the results in V'' using `HashMergeJoin` (line 16). As noted earlier, we use the sort-merge join if a triple pattern is defined with a complex filter condition. Thus, our algorithm first checks if the triple pattern $tp \in s$ at the current state contains any complex filter condition (line 17). If so, it locates the join state s'_{join}, with which the current state has $o - o$ join (`GetJoinState`) (line 19). Note that only star-shaped triple patterns can have complex filter conditions. Finally, the algorithm implements the join between V'' and $r_{s'_{join}}$ ($V_{s'_{join}}$ if $|r_{s'_{join}}| =$) using `SortMergeJoin` (line 21). The resulted merged mappings set is used to determine the execution of next join operations in the automaton A.

Analysis. The correctness of the algorithm 2 is assured by the following. (1) `SortandCombine` cor-

rectly orders each subquery graph according to the cardinality of the views $\mathsf{V_i}$ of each $tp_i \in Q$. Thus, it ensures that the smaller intermediate joined triple pattern mappings are produced. (2) The `Join` procedure initiated for each $s \in A$ correctly joins the triple pattern mappings according to the defined transition rules. It merges the resulted joined mappings that are used by future join operations. (3) The `Proceed` operator ensures that the algorithm percolates only if there exists an injective function between the two sets of views used for `Join` procedure, i.e. $\mathsf{V_i} \bowtie \mathsf{V_j} \neq$.

It takes $O(|Tp|)$ for `SortandCombine` and in practice $|Tp| << |G^i|$.The `Join` procedure takes $O(|\mathsf{V_i}|)$ due to the bidirectional hashing. Although, in case of complex filters, there is an extra cost of sorting and an additional join operation $O(|\mathsf{V_i}|\ log\ |\mathsf{V_i}| + |\mathsf{V_i}|)$.

Remarks. The use of KG-based events enables an interesting feature, where an external static *background knowledge base* can be used to semantically enrich the events. For example, an event containing the information for the power usage can be processed with the historical power consumption of the house to compute the predictive analysis for future events. Our data structure can be utilised to partition such historical events (according to predicates) and we can employ them within the TP-Join algorithm.

Example 3: Recall the query graph Q that is mapped to an automaton A in Fig. 4(a,b). A is constructed through `SortandCombine`, and is traversed as follows. It starts from the first state with mapped triple pattern tp_1. The join ID of tp_1 is 2, thus it invokes the `Join` procedure for o-s join with views V_1 and V_2. If the `Join` procedure produces a non-empty set $\mathsf{V}_1 \bowtie \mathsf{V}_2 \neq$, then the automaton proceeds to next state (with $r_1 = \mathsf{V}_1 \bowtie \mathsf{V}_2$). The next state contains tp_2, as its join operation has already been executed (during the first iteration of the automaton), the automaton `Proceeds` to next state with the same merged mappings set r_1. tp_3 at state 3 implements the join with the merged mapping set and its view $r_2 = r_1 \bowtie \mathsf{V}_3$. If $r_1 \bowtie \mathsf{V}_3 \neq$, then automaton `Proceed` to next state. tp_4 is a bridge triple pattern (with s-o and o-s joins). Thus, it implements the s-s join with the mappings r_2 from the previous state. The whole process continues and r_5 contains all the merged mappings of $Q(G^i)$, considering that the automaton reaches the final state s_f.

5. INCREMENTAL CGPM

In this section we extend our event-based CGPM algorithm to support incremental CGPM over KG streams. As discussed in Section 2 (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, such that $M(Q, \bigcup_{k=0}^{i-1}(\Delta G^k, \tau_k) \oplus \Delta M_i$. That is, whenever a new update $(\Delta G^i, \tau_i)$ arrives, we would like to solve the following problems. (1) Finding matches incrementally, while considering the old ones. (2) Efficiently updating 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. Graph partitioning and view materialisation is treated in a same manner as discussed in Section 3. However, the use of a dictionary for triple mappings is somewhat different. As noted earlier, the incremental evaluation persists the former matched results. Thus, we persist the encoding of such matched mappings in an adaptive dictionary till the end of the defined window.

5.1 Incremental TP-Join Algorithm

Contrary to the event-based computation, matched triple patterns emerge slowly during incremental evaluation. Usually we find the 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, as future triples, received in the views may invalidate the partial results. Thus, we employ

Algorithm 3 Incremental TP-Join algorithm

Input: Set of triple patterns $Tp \in Q$, set of views V, set of combined merged matched mappings f
Output: Set of matched $G'^i \in \mathcal{G}$

```
1: for each tp ∈ Q do
2:     if |V_tp| ≠ then
3:         A_chain, A_star := Add(t_p)
4:                                          ▷ add t_p to corresponding Automaton
5: A := SortAndCombine(A_chain, A_star)
6: if |A| = |Tp| then
7:     r ← an empty merged triples view
8:     for each s ∈ A do
9:         s_join := GetJoinState(s, A)
10:        V' := Join(V_s, r_{s_join}, s, A)        ▷ V_{s_join} if ID(s_join) = 0
11:        if V' ≠ then
12:            r_s := r_s ∪ V'
13:            Proceed to next state
14:        else
15:            V' := Join(V_s, f, s, A)
16:                                  ▷ join with already merged matched mappings set f
17:            if c ≠ then
18:                f := f ∪ V'
19:            Terminate Automaton A
20:    if r_{s_f} ∈ A ≠ then                    ▷ if A reaches to the final state s_f
21:        f := f ∪ r_{s_f}
22:        for each s ∈ A do
23:            V_s := V_s \ r_{s_f}
24: else
25:    if f ≠ then
26:        for each s ∈ A do
27:            if V_s ≠ and newMapping_{V_s} := true then
28:                ▷ If there are some new triples in V_s after the execution of Algorithm 1
29:                V' := Join(V_s, f, s, A)
30:                if V' ≠ then
31:                    f := f ∪ V'
```

a lazy evaluation strategy to reduce the number of unnecessary joins computations. In incremental CGPM, the use of lazy evaluation strategy for $M(Q, (\Delta G^i, \tau_i))$ can increase the query performance by deferring the joins between triple patterns until for each $tp_i \in Q$, $|\mathsf{V_i}| \neq$ There can be numerous cases during incremental evaluation, when the process of view materialisation does not produce any mapping for a certain $tp_i \in Q$. Thus, lazy evaluation makes sure that the joins between the triple patterns execute, only if there is enough evidence that TP-Join algorithm can reach to the final state of the automaton.

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

`SortandCombine` (lines 1-5). As noted earlier in Algorithm 2, we implement the join ordering between $tp \in Q$ according to the cardinality of the set of views V. However, for incremental evaluation, we can have fewer triple patterns with $|\mathsf{V_i}| \neq$ as compared to the total number of $tp \in Q$. Thus, for each execu-

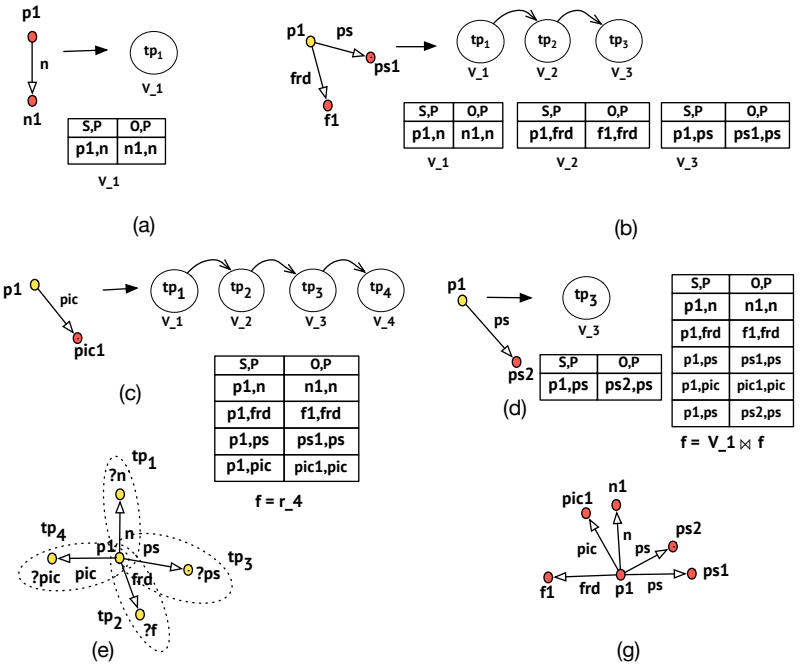

Fig. 5: (a,b,c,d) Three updates from KG stream, that resulted in respective automaton formation for the query graph in (e) and materialisation of a set of views V. (d) Final matched graph

tion of the automaton, we only order the triple patterns with $|\mathsf{V_i}| \neq$ (line 2). The advantage of this step is described later.

Automaton Traversal (lines 8-19). We use lazy evaluation for automaton traversal. In order to implement this, we initiate the algorithm traversal, i.e, execution of joins between $tp \in Q$ only if for each $tp_i \in Q$ $|\mathsf{V_i}| \neq$. Once $|A| = |Tp|$, i.e., all the triple patterns are mapped onto the automaton states, the `Join` procedure is initiated (same as to the Algorithm 2) between triple patterns according to the join IDs (lines 9-10). However, for incremental evaluation, there may be cases when if the join between two triple patterns cannot produced any result (line 14), the triple pattern mappings of the currently active state can be the part of an already computed set of matched mappings f (from previous executions of the automaton). Thus, we implement the join between the triple pattern mappings (view) of current state $\mathsf{v_s}$ and the already computed set of mappings f (lines 15-18).

Union of Matches (lines 20-23). Intuitively, during incremental evaluation, the matches between updates $(\Delta G^i, \tau_i)$ and query graph Q, i.e., M$(Q, (\Delta G^i, \tau_i))$ emerges with time. If a certain update does not match with the query graph Q, it may match with the future updates. Thus, unlike Algorithm 2, the incremental evaluation requires to keep the old mappings that are not matched in a particular execution of the joins. Hence, if an automaton reaches the final state s_f with all the matched mappings r_{s_f} for each $tp_i \in Q$ (line 23). We only remove the matched mappings from each view $\mathsf{v_i}$ (line 26). Furthermore, at this point, the algorithm also takes the union of the already merged matched mappings set f of all $tp \in Q$ and the newly acquired merged matched set of mappings. Hence, making a consistent set of all the matches to this point. Note that, due to our Bidirectional Multimap data structure there are no duplicate mappings.

Join with former Matches (lines 24-31). From Fig. 5(e), it can be seen that there could be cases, when if for some $tp_i \in Q$, $\mathsf{v_i} \neq$ and $t \in \mathsf{v_i}$ may belong to an already matched graph G'^i (tp mappings $\{(\texttt{p1,ps,ps2})\}$ belong to previous matched results in Fig. 5). Therefore, if $|A| \neq |TP|$ we use an already matched merged mappings set f to join the triple patterns using non-empty views. Note this process is initiated only if the view materialisation of the new input graphs G^j results in addition of new mappings in the view $\mathsf{v_i}$.

THEOREM 5.1. *Incremental CGPM produces the same results as the re-evaluation based GPM.*

Proof Sketch: Let tp_1, tp_2 be the two triple patterns with a join on their attributes. Let V_1^i, V_2^i be the two materialised views for tp_1 and tp_2 at time i respectively. We show in Algorithm 3 that at time $t = 1$ for both triple patterns that are joined on their attributes (s or o), the matched mappings are updated in the intermediate mappings sets r_1 if $\mathsf{V}_1^1 \bowtie \mathsf{V}_2^1 \neq$. If $|r_1| \neq$, the intermediate merged mappings are added to the final mapping sets f_1. Thus, at $t = 1$, f_1 only contains the triple pattern mappings that satisfied the join conditions of tp_1 and tp_2 from V_1^1, V_2^1. At $t = 2$, we first join the new materialised views V_1^2, V_2^2 and if $\mathsf{V}_1^2 \bowtie \mathsf{V}_2^2 =$, we join them with previously computed merged joined mappings f_1. Thus, during each execution of the algorithm, we capture the valid mappings of triple patterns and incremental evaluation produces the same result as the re-computation. It is quite trivial to extend the conclusion for a set of triple patterns $tp \in Q$.

Analysis. Similarly to Algorithm 2, it takes $O(|Tp|)$ to order triple patterns and $O(|\mathsf{V_i}|)$ for `Join` procedure. For the joins between previous matched mappings f (line 14), our hash-merge join rearranges the join operators (mappings sets), thus it takes either $O(|\mathsf{V_i}|)$ or $O(|f|)$. However, as we remove all the matched mappings from each view $\mathsf{V_i}$, it takes extra $O(|A||\mathsf{V_i}|)$.

Example 4: Recall the query graph Q in Fig. 5(e) and four updates from KG stream in Fig. 5(a,b,c,d). 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. Due to our lazy evaluation technique, updates are joined once if for each $tp_i \in Q$ $|\mathsf{V_i}| \neq$. i.e., when a triple `(p1, pic, pic1)` arrives in Fig. 5(c). The final joined merged mappings are added in f. In Fig 5(d), when a new update arrives, although we cannot perform the joins due to lazy evaluation, but as it is the part of older matches, we execute its join with f (lines 25-31 in Algorithm 3). As the join is successful, the final matched graph (shown in Fig. 5(g)) contains all the matched updates.

In effect, our algorithm builds the semantics and the topology of a KG by continuously and incrementally processing the updates in the KG streams, while maintaining the older matches.

6. EXPERIMENTAL EVALUATION

In this section we experimentally verify the effectiveness and efficiency of the proposed algorithms for both event-based and incremental CGPM.

6.1 Experimental Settings

Datasets. We use the following four datasets for our tests.

(1) *NY Taxi Dataset* [ref a], a publicly available real-life 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.

(3) *Linked Stream Benchmark (LSBench* [Barbieri and Braga 2009; ref d]), a synthetic dataset contain-

ing social data distributed into streams of GPS, Post, Comments, photos, and static data contain the users profiles. We generated various distributions of the dataset, that include triples from 1 million to 10 millions for the streaming part and around 500K for the static graphs.
(2) *Social Network Benchmark (SNB)*, a recently published [Erling and Averbuch 2015] synthetic dataset. This is again a social network dataset, but it contains much richer attributes and is consolidated into a single stream. The dataset 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 distributions of the dataset is described in Table 2.
(4) *LUBM* [Guo et al. 2005], a synthetic dataset provided by Lehigh university. It contains information about the universities, their registered students, courses, publications, professors etc,. It is widely used by the semantic web community for benchmarking triple stores. We generated a dataset of 150 universities, with a total of 24 million triples.
Query Graphs. We designed and reused multiple query graphs for the above mentioned datasets. The details are as follows.
(1) *NY taxi Queries*: We generated three different query graph describing the event data in star-shaped (Q1), chain-shaped (Q2) and complex-shaped (Q3) queries. These query graphs show the relationships between taxi fare information, location, trip times etc,. For example, Q1 is to find the trip type, trip distance, pick-up datetime, pick-up location, drop-off location, trip-fare, trip tax and total trip time for each taxi event. Each query graph contains 6 triple patterns, without any filter constraints.
(2) *LSBench Queries*: We used their first three query graphs (Q1, Q2, Q3) as described in the benchmark. They queries match the user's post related patterns. Other queries are focused on aggregate operations, and are thus out-of-the scope of this paper. The nature of the queries can be referenced from [ref d].
(3) *SNB Queries*: We randomly generated three different query graphs (star (Q1), chain (Q2) and complex (Q3)) from the use cases described in the benchmark. These query graphs describe the relationships about posts and persons. For example Q1 is to find all the posts and their ids, creation date, creators, tags, locations, full name of the locations, content, by a person. All of them contains 8 triple patterns with no filter constrains.
(4) *LUBM Queries*: For the LUBM dataset, OpenRDF has published a list of queries [Guo et al. 2005]. But many of these queries are simple 2-triple pattern queries or they are quite similar to each other. Hence we chose 7 representative queries out of this list as described in [Atre and Chaoji 2010]. Details on the used queries are available at [ref c]

Implementation. We have implemented our algorithms in Java and compared it with other systems in two different settings: event-based and incremental-based. For our comparison, we use two different types of system, (1) triple-based stream processing systems, (2) in-memory RDF graph processing systems. The first category of systems includes CQELS [Le-Phuoc and Dao-Tran 2011], while the second one includes, Apache Jena [Wilkinson 2006], Sesame [ref b] and RDFox [Motik and Nenov]. Please note that, although, in-memory triple stores are not optimised for dynamic settings, we opted to use them based on the following points. (1) the lack of any baseline system that uses a KG processing model (set of triples) for CGPM. (2) To show the performance degradation, even in case of light-weight indexing techniques. (3) Some of these in-memory system, such as Jena are also being utilised as an underlying system for CQELS and C-SPARQL [Barbieri and Braga 2009]. For triple-based systems, we only used CQELS as it outperforms others by many orders of magnitude [ref d]. Furthermore, [Choudhury and Holder 2015] provides a recent solution for incremental GPM, but it was not available for comparative analysis. Each of our tests is limited to 10 hours and we report averages over 50 runs.

All the experiments were performed on Intel Xeon E3 1246v3 processor with 8MB of L3 cache. The system is equipped with 128GB of main memory and a 256Go PCI Express SSD. The system runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u05. The code for all the tests including our system is available at this link [ref c].

6.2 Analysis for Event-based CGPM

Performance Analysis: In this set of experiments, we use throughput to measure the performance (graphs per second) of event-based CGPM (denoted as E-CGPM in Fig. 6). This is equivalent to informing the time taken for a given amount of data. Let graph per second be the throughput for processing a graph stream $\mathcal{G}$. If a certain dataset contains n graphs, then it takes $n/graphsseconds$ to answer the query graphs over the dataset. The details of the experiments on different datasets is as follows.

NY Taxi Dataset: For this dataset, the size of KG streams varied from few thousands graphs all the way up to 50 million (50M) graphs (around 1 billion triples). The throughput results of all the systems are shown in Fig. 6 for the query graph Q1. We do not show the results of query graph Q2 and Q3, as they provided similar performance measure, due to the smaller sized graphs and thus, the query optimisation for each system does not make any difference. In general, our system performs much better than other systems (about 10 orders of magnitude faster than Jena). Jena and Sesame, when compared to other systems performed better due to their light-weight indexing and storage structure. RDFox, which provides a more optimised indexing and query techniques, fails to perform better than Jena and Sesame. This is because, as analysed in Sections 3, extensively indexing dynamic KG streams with high frequency, results in longer insertion times. We modified the window mechanism of CQELS to cache a whole graph for each event, such that it can handle a whole graph as an input. However, it performs poorly and results in time-outs due to the following reasons. (1) It is based on the triple model, thus caching a whole graph within a window results in the re-evaluation of the query (using B+ tree indexing) for each triple-based input, thus it quickly runs out of memory due to the large number of matches returned. (2) It uses a static dictionary approach to map strings to numeric IDs and utilises disk storage if the size of the dictionary reaches to a certain threshold, resulting in high I/O cost. Our implementation of event-based CGPM performs in a steady manner even for large sized streams as a result of the adaptive dictionary, and memory barriers do not explode for large windows.

LSBench Dataset: Fig. 7 presents the results on the LSBench dataset (10 million triples), where each KG-based event contains one post (10-14 triples). We use Q1 and extended with the more triple patterns (x-axis in Fig. 7 shows the number of triple patterns), other queries are tested for incremental analysis. In order to provide the comparative measures, we use our system with (denoted as E-CGPM) and without (denoted as E-CGPM*) the graph partitioning and the view materialisation phase (Algorithm 1). From Fig. 7, the use of graph partitioning highly influenced the join processing and thus, results in much higher performance. In case of E-CGPM*, each triple pattern uses a whole graph to join with another and thus results in degradation of performance. Another important result that can be seen from Fig. 7 is that E-CGPM* outperforms the indexing based solution Jena, thus validating our earlier claims. However, if a large graph is used as an input indexing based solution would perform better than E-CGPM*.

Scalability Analysis: Another important factor for evaluating CG-
PM is how the performance scales with the size of each event. We evaluated the scalability of event-based CGPM by using SNB generated dataset. We generated

multiple distributions of data depending on the number of persons for one year worth of data. We divided 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 details of the dataset dis-

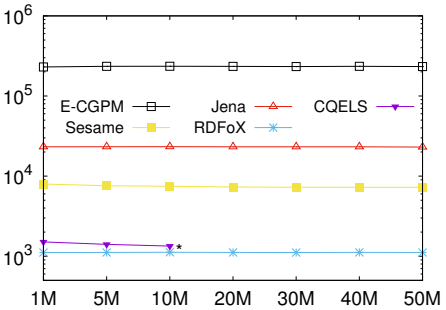

Fig. 6: Throughput (graphs/second, log-scale) Q1 on New York Taxi Dataset

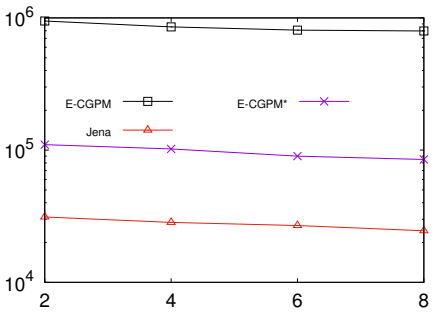

Fig. 7: Throughput (graphs/second, log-scale) Q1 on the LSBench dataset (x-axis # of triple patterns)

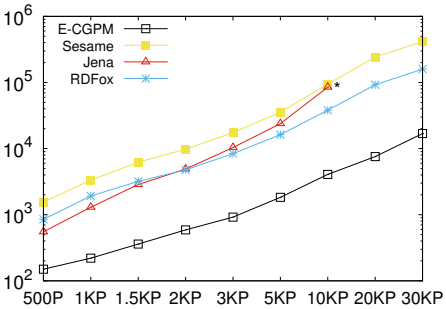

Fig. 8: Total Execution Time (ms, log-scale) Q1 on the SNB dataset

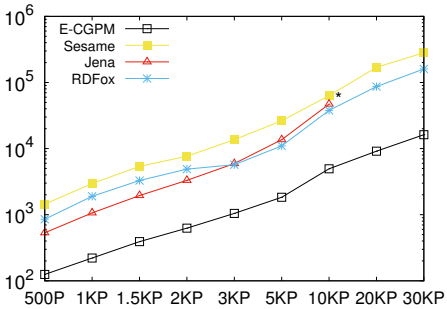

Fig. 9: Total Execution Time (ms, log-scale) Q2 on the SNB dataset

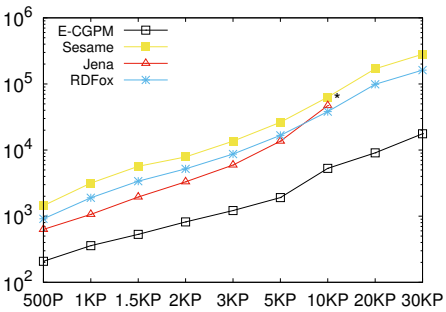

Fig. 10: Total Execution Time (ms, log-scale) Q3 on the SNB dataset

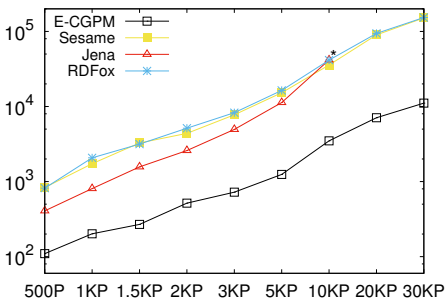

Fig. 11: Total Insertion Time (ms, log-scale) Q1 on the SNB dataset

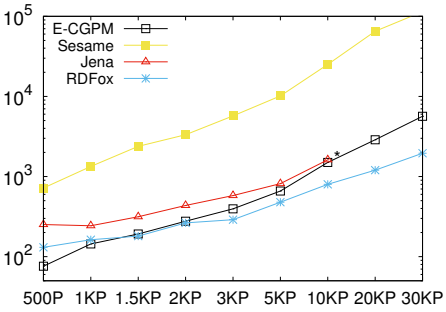

Fig. 12: Total Query Time (ms, log-scale) Q1 on the SNB dataset

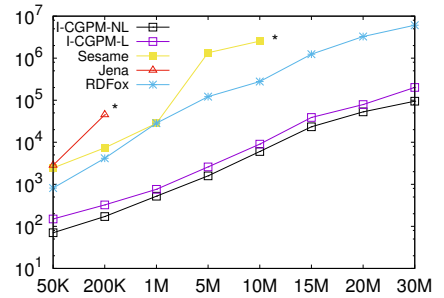

Fig. 13: Total Execution Time (ms, log-scale) Q1 (incremental) on the SNB dataset

Table I. : Dataset distribution for Large-scale CGPM, Min and Max describes the range of no. of triples in each event

Dataset	Min (triples/event)	Max (triples/event)	Total (triples/stream)
500P	783	148K	0.29M
1KP	2340	397K	0.68M
1.5KP	3328	792K	1.2M
2KP	5352	988K	1.8M
3KP	12848	990K	3.2M
5KP	217K	301K	5.7M
10KP	50K	805K	13.5M
20KP	115K	1.9M	30.9M
30KP	145K	3.2M	51.5M

tribution are described in Table 2. The elapsed time for each query graph was measured with warmed up cache by using the static SNB dataset (around 50K triples).

The total execution time (insertion and querying) for the three query graphs are shown in Fig. 8,9,10. Even with the increase in numbers of triples in each event, our system scales linearly and smoothly (no large variations are observed) with the size of each event and outperforms others, about 5 orders of magnitude faster for smaller event graphs and about 9 for larger graphs. The reasons are as follows. (1) Due to our query-based graph partitioning technique our system discards many useless triples before joins and does not apply any extensive indexing. This results in lower insertion time and smaller number of triples for each triple pattern. (2) The use of join ordering, according to the cardinality of materialised views, results in smaller intermediate joined results. (3) Bidirectional Multimaps allows linear joins for all the three different types of the query graphs (hash-merge subject-object, object-subject and object-object joins). We observed similar results for all query graphs for our system. However, other systems have slight variations in their execution times due to their internal optimisations. Specially, the chain-shaped query graph results in index scan merge join (subject-object) and thus have a variable join processing time. Jena which performs better for smaller events results in time-outs as the size of an event increases to around 1 million triples. Thus, its light-weight indexing fails for large numbers of triples and its 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) architecture. However, it results in high insertion times. The experimental results presented in Fig. 11,12 (insertion and query time for star-shaped query graphs) show that the indexing techniques result in longer insertion times. Our query-based data partitioning technique and light-weight data structure results in much lower insertion times. Although the querying time for our system increases with the size of events, but it compensates for it with lower insertion time. The query time for RDFox is lower for larger event graphs due to its storage and indexing structures.

6.3 Analysis for Incremental CGPM

In this set of experiments, we (1) compare the efficiency of our incremental CGPM (denoted as I-GPM-L in Fig. 13) against re-evaluation based GPM. (2) We present the performance comparison of our lazy evaluation with non-lazy evaluation (non-lazy incremental CGPM denoted as I-CGPM-NL). (3) We describe the effect of complex and atomic filters on the execution time of query graphs. (4) We compare the performance of our system with triple-based systems and, (5) we use the LUBM dataset and queries to evaluate I-CGPM. The details of the experiments are as follows.

SNB Dataset: We use the SNB benchmark and a star-shaped query as described above for experiments (1,2,3) and varied the window sizes (different from stream size) from 50K triples up to 30M triples, where each graph update is of variable length (10-50 triples set). The results in Fig. 13 provide the following insights. (1) Incremental CGPM scales linearly with the size of the window, while

Table II. : Evaluation – LUBM dataset for 24 million triples (time in seconds)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
100	226	97	189	76	27	36	204
1000	206	83	178	61	25	28	190

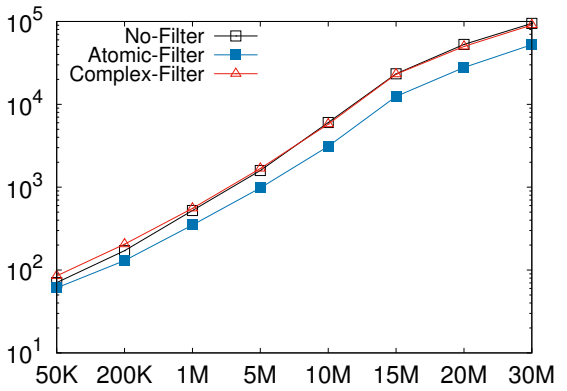

Fig. 14: Total Execution Time (ms, log-scale) Q1 on the SNB dataset

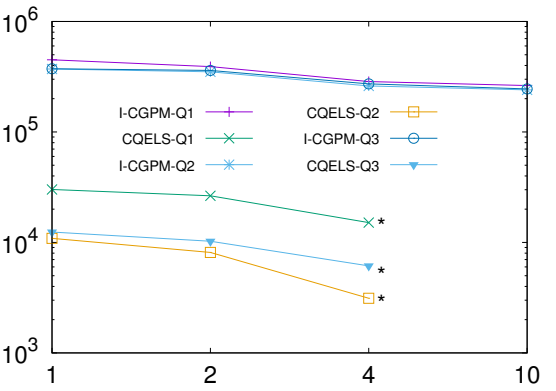

Fig. 15: Throughput analysis (g/s,log-scale) Q1,Q2,Q3 on the LSBench dataset

others scales super-linearly. It also outperforms other by about 24 orders of magnitude for smaller windows and 64 for larger. The reason is that we incrementally process the matches for each incoming update, while others re-evaluate matches from scratch, which becomes a bottleneck for larger windows (i.e., 20M and 30M windows). 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. RDFox, due to its parallel architecture, indexing and storage structure perform reasonably well as compared to Jena and Sesame. But it suffers from high insertion costs for larger windows. (2) The lazy evaluation of incremental CGPM performs better than non-lazy evaluation due to the fewer number of joins.

The effects of filter constraints are compared in Fig. 14. Clearly, the use of an atomic filter results in fewer triples in materialised views, as a consequence there are fewer intermediate joined triples and also fewer joins – because only a specific type of triples is allowed. The use of complex filter constraints results in a similar execution time as compared to no filter constraints. This is because the use of complex filter results in extra object-object joins. However, it also prunes more triples during the triple patterns join operations, resulting in smaller intermediate results, and thus evening out the effect of extra join operations.

LsBench Dataset: We use a triple-based stream to execute Q1, Q2 and Q3 on LSBench dataset. Results in Fig. 15 provide the following insights. (1) CQELS does not scale well on larger window sizes. The reason is the same as described earlier, it utilises a B+ tree indexing structure to incrementally match the triple patterns defined on a set of streams. Additionally, the use of background knowledge further increases the response time (Q2,Q3) due to joins on static and streaming triples. (2) Our system outperforms CQELS for all the queries. The use of graph partitioning and view materialisation prunes unnecessary triples without incurring the cost of joins. Furthermore, our dictionary approach does not encode all the triples in the stream. Only the relevant triples that are materialised and used in the matched sets are encoded. This greatly reduces the memory consumptions and increases the performance. Furthermore, the use of query-based graph partitioning for background knowledge only

materialises the static triples required to join with the streaming one. However, the join of static and streaming triples is expensive and thus result in slight degradation for Q3.

LUBM dataset: In order to show the performance of our system on a standard triple store benchmark, we use the LUBM dataset (25 million triples) and stream graphs in two configurations: each graph contains 100 or 1000 triples. Tables 2 describes both triple insertion, materialization and query processing time. Despite, the fact that our system is optimised for dynamic in-memory settings, it performs considerably well on a dataset utilised for static triple stores. As described in [Guo et al. 2005], in-memory static triple stores result in query time-outs even for a smaller (5 million triples) LUBM dataset. Thus, our data structure is efficient enough to scale on a window of 25 millions triples and even more. Furthermore, traditional in-memory or disk-based solutions concede a considerable amount of time during triple insertion and indexing. Although, our system may not scale well to billions of triples (for incremental CGPM), but in streaming settings it shows good scalability measures, and such huge window sizes are not practical for streams. Other insights from Table 2 are as follows, (1) Q1,Q3 and Q7 are cyclic structures, bear high input and selectivity and results in larger number of triples in the materialised views. Thus, they have longer execution times. (2) Q5 and Q6 with atomic filter constraints result in less number of triples in materialised views. (3) Q2 and Q4 bear large input and high selectivity but contain simple patterns, thus results in fewer joins.

7. RELATED WORK

In recent years, there have been significant efforts to speed up the query graph processing over KG data by developing specialised and novel systems. Many systems model KG data as a tabular structure (one giant table, property tables, vertical partitioning). RDF3x [Neumann and Weikum 2010] uses one giant table to store data in one three-column table and thus enabling the manipulation of triples in a uniform manner. However, it requires heavy computation to handle self-joins. One remedy proposed is to use several clustered B+tress for all permutations of three columns. Jena [Wilkinson 2006] exploits multiple-property tables, while BitMat [Atre and Chaoji 2010] uses a 3-dimenstional bit cube, so it can also support 2D matrices of (s,o), (p,o), and (ps). Trinity.RDF [Zeng and Yang 2013] is a sub-system of a distributed graph processing engine, where triples are stored in Trinity's key-value store and are indexed using separate tables. Several graph stores [Zou 2014; **?**] support KG data in their native graph storage and implement specialised indexing to accelerate the query processing. gstore [Zou 2014] performs graph pattern matching using the filter-and-refinement strategy and finds promising subgraphs using the VS+-tree index. All of these techniques use extensive indexing to store and query data.

CGPM is quite a fertile area of research and there are few solutions presented in the context of general labelled graphs. [Chen 2010] proposed techniques using a feature-based structure called as node-neighbour tree to search multiple graph streams using a vector space approach. This technique uses a relaxed form of GPM and incurs significant processing overheads. [Mondal 2014] proposed a solution to support ego-centric queries in dynamic graphs and is primary focused on aggregate queries. [Gao and Zhou 2014] presented a vertex centric approach for query processing on dynamic graphs build on top of Apache Giraph. They decompose the query graph into identical sub-direct acyclic graphs (DAG). The DAGs are then traversed to identify source and sink vertices. Although they address many challenges for processing dynamic graphs, their approach is only optimised for general labelled graphs. [Choudhury and Holder 2015] provided 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. Finally the partial results are incrementally joined by search in the tree. It is based on an edge-stream model, where each edge is assigned with a timing attribute. Similarly, [Chunyao Song et al. 2014] presents a solution based on an edge-stream model.

They have extended the dual simulation algorithm, a number theoretic signature method and graph colouring algorithm to incrementally match incoming edges with timing constraints. As described earlier, none of theses techniques are efficient for the KG model. To apply general sub-graph isomorphism algorithms, KG has to be transformed into labelled graphs and may result in performance degradation as compared to the specialised KG-based system. Furthermore, most of these solutions are based on edge-stream model, i.e., they use one timing function on every edge of the graph within a stream. This is not practical in many use cases, where usually each event contains a set of edges and vertices. Another class of systems [Barbieri and Braga 2009; Le-Phuoc and Dao-Tran 2011] uses the RDF triple model on the top of general stream solutions to process continuous queries over RDF streams. In these system each incoming triple is matched with a triple pattern registered for a particular stream and are optimised for aggregate queries. Their query processing within a window is based on re-evaluation and they can only handle very small graphs. Whereas our solution is based on a triple-set (or edge-set) model, where a query graph is processed on a set of triples either using event-based one-time execution or incremental evaluation of a set of edge-sets. Furthermore, W3C standardising RSP community group [ref e] is still working on various models and their initial recommendations includes the use of triple-set streams for RDF.

8. CONCLUSION

Generally, GPM is optimised for static KGs and extensive indexing techniques have been used to accelerate the joins or exploration of KG data. However, with the emergence of new applications, KGs arrive in the form of streams. CGPM on streaming KGs raises further new challenges and existing approaches can no longer be applicable in this setting. In this paper, we propose novel algorithms for two main computation models for CGPM, i.e., event-based CGPM and incremental CGPM. We propose a light-weight data structure and query-based graph partitioning technique to materialise a set of views for each KG-based event or update. We then propose an automata-based TP-Join algorithm with a series of optimisations. It is based on the dynamic nature of the KG streams and adopts accordingly (join ordering, automata execution). It percolates only if there is enough evidence that the processing of future joins will produce matched graphs. Furthermore, we extend our TP-Join algorithm to support the incremental evaluation of query graph matches over KG streams. We next performed extensive experiments using real-life and synthetic datasets in order to show the superiority of our algorithms on traditional in-memory and triple-based streaming systems.

A. PROOF OF THEOREM 1

We use the Evaluation problem from [Pérez et al. 2009] to construct a Evaluation+ problem, where given an RDF graph-based event G^i, a query graph Q consisting of basic graph pattern matching (BGP) i.e., ***AND*** and ***FILTER*** operators from SPARQL (see [Pérez et al. 2009]) and a mapping μ, we need to verify that $\mu \in Eval(Q, G^i)$, where $Eval(Q, G^i)$ is a function that returns a set of mappings for the matched query graph Q (see Definition 2.2). Now we can construct a PTime algorithm to solve the Evaluation+ problem for a query graph with operators AND and $FILTER$ from SPARQL (BGP). The algorithm is defined recursively on the query graph Q and returns true of $\mu \in Eval(Q, G^i)$ and false otherwise. The three different clauses that can be used in the algorithm are, (1) if $Q := tp$ is a triple pattern, we return true iff $\mu \in Eval(tp, G^i)$, (2) if $Q := Q_1\ AND\ Q_2$, where Q_1 and Q_2 comply to the clause (1), then we recursively checks if $\mu \in Eval(Q_1, G^i) \wedge \mu \in Eval(Q_2, G^i)$, (3) if $Q = Q_1\ FILTER\ b$, for a filter constraint b (see definition 3.2), then it returns true iff $\mu \in Q_1 \wedge b \models \mu$. From these set of cases, we can easily see that the algorithm runs in polynomial time, i.e., the execution of $Eval$ function. Now for the stateful queries as described for SPASEQ, the input also contains a set of mappings with Ω form a super-query. Thus, for all the above clause, we need to check if the mapping $\mu \in \Omega$, for example

if $Q := tp$, $\mu_1 \in Eval(tp, G^i)$, $\mu_2 \in \Omega$, then it returns to true iff $\mu \in (\mu_1 \cup \mu_2)$, where $\mu_1 \sim \mu_2$, i.e., they are compatible to each other. Therefore, we have an extra cost of determining if the mapping also belongs to the set of mappings obtained from the super-query, $O(|\Omega|.|Q|.|G^i|)$.

REFERENCES

http://chriswhong.com/open-data/foil$_nyc_taxi$/.

http://rdf4j.org/.

https://bitbucket.org/syd12/contunuousgpm/overview.

https://code.google.com/p/lsbench/.

https://www.w3.org/community/rsp/.

http://www.w3.org/RDF/.

Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (June 1975), 333–340. DOI:http://dx.doi.org/10.1145/360825.360855

Mario Arias and Javier D. Fernández. 2011. An Empirical Study of Real-World SPARQL Queries. *CoRR* abs/1103.5043 (2011). http://arxiv.org/abs/1103.5043

Michael Armbrust and Eric Liang. 2013. Generalized scale independence through incremental precomputation. (2013), 625–636. DOI:http://dx.doi.org/10.1145/2463676.2465333

Medha Atre and Chaoji. 2010. Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *WWW*. 41–50. DOI:http://dx.doi.org/10.1145/1772690.1772696

Davide Francesco Barbieri and Braga. 2009. C-SPARQL: SPARQL for continuous querying. In *WWW*. 1061–1062.

Henrik Björklund. 2009. Incremental XPath Evaluation. In *ICDT*. DOI:http://dx.doi.org/10.1145/1514894.1514915

Lei Chen. 2010. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. *IEEE Trans* (2010), 1093–1109. DOI:http://dx.doi.org/10.1109/TKDE.2010.67

Jiefeng Cheng and Yu. 2008. Fast Graph Pattern Matching. In *ICDE*. DOI:http://dx.doi.org/10.1109/ICDE.2008.4497500

Sutanay Choudhury and Lawrence B. Holder. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. *EDBT* (2015).

Chunyao Chunyao Song, Tingjian Ge, and Chen. 2014. Event Pattern Matching over Graph Streams. *PVLDB* (2014), 413–424. http://www.vldb.org/pvldb/vol8/p413-ge.pdf

R. Cyganiak. 2005. *A relational algebra for SPARQL*. Technical Report. Digital Media Systems Laboratory, HP Laboratories Bristol.

O. Erling and A. Averbuch. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. 619–630.

Wenfei Fan, Jianzhong Li, and Luo. 2011. Incremental Graph Pattern Matching. In *SIGMOD*. 18. DOI:http://dx.doi.org/10.1145/1989323.1989420

Jun Gao and Chang Zhou. ICDE, 2014. Continuous pattern detection over billion-edge graph using distributed framework. 556–567. DOI:http://dx.doi.org/10.1109/ICDE.2014.6816681

Michael R. Garey. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

Andrey Gubichev. EDBT, 2014. Exploiting the query structure for efficient join ordering in SPARQL queries. 439–450. DOI:http://dx.doi.org/10.5441/002/edbt.2014.40

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.* 3, 2-3 (2005), 158–182. http://dblp.uni-trier.de/db/journals/ws/ws3.html#GuoPH05

Alon Y. Halevy. 2000. Theory of Answering Queries Using Views. *SIGMOD Rec.* 29, 4 (Dec. 2000), 40–47. DOI:http://dx.doi.org/10.1145/369275.369284

Changkyu Kim. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB* (2009), 1378–1389. http://www.vldb.org/pvldb/2/vldb09-257.pdf

Danh Le-Phuoc and Dao-Tran. 2011. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *ISWC*. DOI:http://dx.doi.org/10.1007/978-3-642-25073-6_24

R. Milner. 1989. In *Foundations of Software Technology and Theoretical Computer Science*. Prentice Hall.

Guido Moerkotte. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products. VLDB Endowment, 930–941.

Jayanta Mondal. 2014. EAGr: Supporting Continuous Ego-centric Aggregate Queries over Large Dynamic Graphs. *SIGMOD* (2014). http://arxiv.org/abs/1404.6570

Boris Motik and Yavor Nenov. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *AAAI 2014*.

Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB* (2010), 91–113. DOI:http://dx.doi.org/10.1007/s00778-009-0165-y

Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* (2009). DOI:http://dx.doi.org/10.1145/1567274.1567278

Mikael Pettersson. 1992. A Term Pattern-Match Compiler Inspired by Finite Automata Theory.. In *CC* (2009-10-15) *(LNCS)*, Vol. 641. http://dblp.uni-trier.de/db/conf/cc/cc92.html#Pettersson92

G. Ramalingam and Thomas Reps. 1989. A Categorized Bibliography on Incremental Computation. In *POPL*. 502–510.

Diptikalyan Saha. 2007. An Incremental Bisimulation Algorithm. In *FSTTCS*.

Sherif Sakr and Elnikety. 2012. G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs *(CIKM)*. 10. DOI:http://dx.doi.org/10.1145/2396761.2396806

Markus Stocker and Seaborne. 2008. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *WWW*. DOI:http://dx.doi.org/10.1145/1367497.1367578

Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. DOI:http://dx.doi.org/10.1145/1107499.1107504

Kevin Wilkinson. 2006. Jena Property Table Implementation. In *SSWS*.

Kai Zeng and Yang. 2013. A distributed graph engine for web scale RDF data. In *PVLDB*. 265–276.

Lei Zou. 2014. gStore: a graph-based SPARQL query engine. *VLDB* (2014). DOI:http://dx.doi.org/10.1007/s00778-013-0337-7