Plan-Based Semantic Enrichment of Event Streams

Kia Teymourian and Adrian Paschke

Freie Universität Berlin, Institute for Computer Science, AG Corporate Semantic Web
{kia,paschke}@inf.fu-berlin.de

Abstract. Background knowledge about the application domain can be used in event processing in order to improve processing quality. The idea of semantic enrichment is to incorporate background knowledge into events, thereby generating enriched events which, in the next processing step, can be better understood by event processing engines. In this paper, we present an efficient technique for event stream enrichment by planning multi-step event enrichment and processing. Our optimization goal is to minimize event enrichment costs while meeting application-specific service expectations. The event enrichment is optimized to avoid unnecessary event stream enrichment without missing any complex events of interest. Our experimental results shows that by using this approach it is possible to reduce the knowledge acquisition costs. [1]

Keywords: #eswc2014Teymourian

1 Motivation

The fusion of background knowledge with data from an event stream can help the event processing engines to know more about incoming events and their relationships to other related resources. The usage of background knowledge in event processing requires reasoning on domain knowledge in order to be able to detect complex events based on the domain background information.

Typically there is a trade-off between the high expressiveness of the used background knowledge, which leads to higher levels of computational complexity, and the efficiency and scalability needed in real-time event processing. In this paper, we address the problem of a hybrid approach - expressive reasoning on external background knowledge for usage in high-performance real-time event processing. We propose an approach for knowledge-based event processing using semantic enrichment of event streams (section 2). The main optimization goal of our approach is the detection of events based on reasoning on huge amounts of domain background knowledge. We present a method for planning the event enrichment process [2] in order to optimize the load on the external knowledge base for knowledge acquisition (section 3).

[1] This work has been partially supported by the "InnoProfile-Transfer Corporate Smart Content" project funded by the German Federal Ministry of Education and Research (BMBF) and the BMBF Innovation Initiative for the New German Länder-Entrepreneurial Regions.

[2] see [7] for an overview on different event processing functions

We describe our approach in the context of the use case scenario of a high level stock market monitoring system. In today's world economy, companies are highly interconnected and depending on each other. They require, for example, raw materials, share distribution channels and markets, have affiliations or simply reside in the same areas. Such information yields a valuable source for knowledge-based complex event processing and can be leveraged in order to empower event processing with semantic technologies in order to grasp underlying relationships and utilize them in the course of the event processing.

Let's consider the case that a company X produces products and requires the raw materials which are procured by another company Y. The company Y is on the other hand financed by company Z. The relation between these companies defines the complex event pattern which can only be extracted from the background knowledge. An example of a pattern for complex events is the case that three stock ticks, respectively the associated companies, exhibit a specific relation in the background knowledge, and the relation spans a connection between some resources in the background knowledge. The complex event can be specified based on the company business relations and the event correlations of the stock market events. Another example of a pattern for complex events is the case that a market broker might define a detection pattern like: *select stocks when the stock price of the three companies decrease in sequence within 10 min where the first company demands for its products special computer chips and the second company produces these chips using raw materials which is supplied by the third company.* For this kind of event detection, it is required to have background knowledge about the application domain while monitoring the real-time event stream and integrating the knowledge to the real-time data stream.

2 Semantic Enrichment of Event Streams

Previously, we have proposed a new approach for semantic enabled complex event processing (SCEP) [11,12]. We proposed that the usage of background knowledge about events and other related concepts can improve the quality of event processing. We proposed to use an external *Knowledge Base (KB)* which provides background knowledge (conceptual and assertional, T-Box and A-Box of an ontology) about the events and other non-event resources. We also include a DL-reasoner on the top of the external knowledge base so that we can reason on the externally stored knowledge.

Our event model is adopted from the event models in the state of the art event processing approaches like DistCED [8]. *An Event is a tuple of* $\langle \bar{a}, t_s, t_e \rangle$ *where* $\bar{a}$ *is a multiset of fields* $\bar{a} = (a_1, ..., a_n)$*, and is defined by the schema* S. The t 's are temporal values representing the different happening times of the event, the start t_s and end timestamps t_e of the event (timestamps can also be defined as a sequence of timestamps). For example an event in stock market applications has the fields *(name, price, volume, timestamps)*, e.g., *(IBM, 80, 2400, 10:15, 10:15)*, where the start and end time of this event are the same, because it is an instantaneous event. An *Event* can also be considered as a set of attribute values $\langle \bar{av}, t_s, t_e \rangle$ where $\bar{av}$ is a multiset of attribute value tuples $\bar{av} = ((a_1, v_1), \ldots, (a_n, v_n))$. For the above

example we have:
(((name,IBM), (price,80), (volume,2400)) , 10:15, 10:15)

We assume that one or more attributes of events are in relation to resources in the KB (such as individuals, concepts, roles and sentences). It is possible to ask the KB and retrieve background knowledge about the attributes of events. For example the stock market symbol is linked to the company resource in the background knowledge so that knowledge about the company can be extracted.

An event detection query is a declarative rule which defines a detection pattern for complex events and can include one or more sets of triple patterns (SPARQL basic triple patterns BGPs) to query external KBs. With the term *sQuery*, we refer to the whole event detection rule which includes sets of triple patterns and is combined with event algebra operations. We define the operational semantics for the four main event detection operations, SEQ, AND, OR and NOT from the window w (a time or count window) as follows:

$\mathsf{SEQ}(e_1, e_2)[w] = \forall(t_s^1, t_s^2)(e_1(\bar{t^1}) \wedge e_2(\bar{t^2}) \wedge t_s^1 \leq t_s^2 \wedge (e_1, e_2) \in w)$
$\mathsf{AND}(e_1, e_2)[w] = \forall(t_s^1, t_s^2)(e_1(\bar{t^1}) \wedge e_2(\bar{t^2}) \wedge (e_1, e_2) \in w)$
$\mathsf{OR}(e_1, e_2)[w] = \forall(t_s^1, t_s^2)((e_1(\bar{t^1}) \vee e_2(\bar{t^2})) \wedge (e_1, e_2) \in w)$
$\mathsf{NOT}(e_1)[w] = e_1(\bar{t^1}) \notin w$

A possible approach for the processing of events based on background knowledge is to enrich the event stream prior to the complex event detection with new derived event attributes. The Semantic Enrichment of Event Streams (SEES) is the enrichment of events with background information about them and about other possibly related concepts from the knowledge base.

The process of semantic enrichment of an event stream is illustrated in Fig. 1. A knowledge base is used by Event Mapping Agents (EMAs) to generate derived events by performing reasoning and interacting with the knowledge base. The EMAs can be replicated and deployed in parallel to achieve efficient scalability with respect to throughput. In the next step, the enriched event stream is monitored by several Event Processing Agents (EPAs). The EPAs process the enriched event stream in order to detect complex events matching the event query. The main disadvantage of semantic enrichment of events is the huge amounts of derived event data which is produced by each incoming event that needs to be processed by the final EPA to match complex queries. The raw event stream is enriched by one or many EMAs resulting in an enriched outbound event stream which is processed by a set of EPAs in order to detect complex events which can require derived events for being triggered.

An example of a complex event pattern is visualized in Fig. 2. A pattern is defined over three event instances. The query given at the top defines a connecting path between the nodes associated with the event instances e_1, e_2, and e_3. The order of occurrence of the three arbitrary event instances e_1, e_2, and e_3 is defined using the event algebra operators `SEQ` and `AND`. Thus, the sequence of e_1 followed by e_2 and e_3 is matched in case the resources referenced by e_1, e_2, and e_3 can be connected by a path corresponding to the triple statements from the query. The event detection pattern is a combination of event algebra operators for the

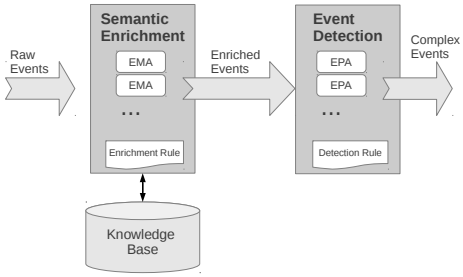


Fig. 1: Semantic Enrichment of Event Stream (SEES)

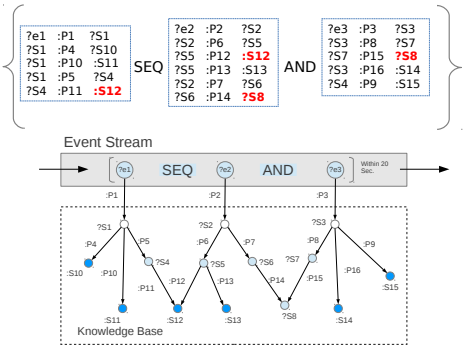


Fig. 2: Relations between Events

specification of temporal relations of events and basic triple patterns for the specification of a knowledge pattern.

A specific type of event detection queries is, if it specifies a complex event containing only one single event. We call it *star-shaped event pattern*, because of the form of attributes (triple pattern predicates) around a single event instance. This kind of event pattern detects only one single event instance from an event stream.

3 Plan-Based Semantic Enrichment

The process of semantic enrichment can be optimized to reduce the cost of event enrichment and increase the throughput of event processing by reducing the amount of raw event enrichment tasks. We propose an approach for optimization of knowledge-based event detection by using a technique for multi-step and greedy knowledge acquisition and event detection. In our approach, we define several steps for sequential enrichment and event detection. In each step the events are enriched with knowledge. The event detection engine can filter out some of the raw events based on the enriched knowledge so that only the relevant raw events are forwarded to the next step. By using this approach we can avoid the unnecessary full enrichment of all raw event instances.

The trade-off between knowledge acquisition costs (computation load on external KB and result transmission) and event processing latency are important factors for planning the execution of event enrichment and detection. Our aim is to discover a low-cost event detection plan while we meet the user-specified latency expectations so that we can reduce the polling load on the external knowledge base. One of the important constrains for generating plans is the user-specified latency expectation. We are looking for a plan which can meet this expectation and causes an acceptable load on the triplestore side.

The user query can be preprocessed and separated into several subqueries. We generate a plan for stepwise processing of the generated subqueries so that we can pre-filter the raw event stream to reduce the cost of event enrichment. In each step, we check only a part of the user query. If any of the subqueries cannot

be matched, the whole query is not matched and the EPA sends the event to the event sink.

The input for the optimization problem are the user query *sQuery*, the raw event stream (including event types) and some heuristics about the external KB. We are looking for an optimized execution plan of the user query with acceptable latency and costs (computation, materialization and network transmission costs).

The user-given query *sQuery* includes a graph G_{user} (the given triple pattern combined with event algebra operations) which can be pre-processed and rewritten to several subgraphs G_{SUBs} so that we have $G_{user} = \{G_{SUB1} \cup \ldots \cup G_{SUBn}\}$. The G_{user} is matched if and only if all of its subgraphs G_{SUBs} are matched.

The first step of our approach is to split the G_{user} into several main subgraphs G_{Events} based on nodes which represent the raw events and are separated by the event algebra operations (e.g., `AND`, `SEQ`). For each event we have a tree structured graph (a cycle-free directed graph) which has one of the raw events as its root. We also mark the nodes which are in the intersection of G_{Events}.

In the second step, we divide the G_{Events} based on its tree structure. By starting from its roots (the events) we traverse the tree to its leafs and divide the tree to its branches so that we generate several sub-graphs G_{SUBs}. A sub-graph is generated for each path branch from the root to one of the leafs. For each raw event type we also generate all of the joint possibilities of subgraphs (based on the event operator). At the end of this step we have a multi-set of graphs MS_{GSUBs}.

For example for event pattern e_1 shown in Fig. 2, we will generate the subgraph set like { {(?e1, :p1, ?s1)}, {(?s1, :p4, :s10)}, {(?e1,:p1,?s1), (?s1, :p10, :s11)}, {(?e1, :p5, ?s4), (?s4,:p11,:s12)}, {(?e2, :p2, ?s2), (?s2, :p6, ?s5), (?s5, :p12, :s12)}, {(?e2, :p2, ?s2), (?s2 , :p7, ?s6), (?s6, :p14, ?s8)}, ... }.

We expand the subgraphs to their semantically similar patterns. For example, if the leaf nodes of subgraphs are bounded resources, we check them against the knowledge in KB to find if they are connected to other resources through the *sameAS* predicate (The *sameAS* predicate gives two resources exact the same semantic meaning). If we can find new resources through the *sameAS* predicate, we generate a new sub-graph G_{SUB} with the new resource and add it to the multi source. We also follow up the type hierarchy of resources (e.g., rdf:type) and add new upper resources in the type hierarchy. We also check the properties of the graph, whether we can find any subProperties of them, and add the new graphs to the multi-set. In this way, it is possible to take the semantics of resources (and their relations) into account for the calculation of sub-graphs.

Planning of Event Processing in Multi-Steps: In each processing step the required knowledge for one of the attributes is enriched to the raw event stream. If an event instance can be matched to one of the attributes it is forwarded to the following step, until all of the attributes can be matched.

For the planning of subqueries, we need to estimate the matching probability of subgraphs. We expect that queries with a high number of results are more likely to be matched. We assume that we can have some statistics about the external knowledge bases, so that we can estimate the enrichment cost based on the collected heuristic data about external KBs. We consider the following statistics about the KB: the total number of existing triples in the KB for each of

the predicates (N_p) and the total number of triples stored in the KB ($N_{Triples}$) are known . These statistics are used to optimize the search process for an acceptable plan because the search space is exponential to the number of subgraphs and number of processing steps.

Furthermore, we do the following assumptions: all of the subgraphs with only a single triple pattern have a bounded predicate and all of the other subgraphs have at least one triple pattern with a bounded predicate. Any updates on the knowledge base can only minimally change the above statistics about the KB, and the statistics can be recalculated in the future time-intervals.

We expect that queries with a high number of results are more selective (materialization costs) and cause high load. For the subqueries with two or more triple patterns, the computation of joins will cause more computation load than the subgraphs with a single triple pattern.

The calculation of the reasoning costs on the KB is highly complex and depends on the complexity of the reasoning algorithm and the reasoning rules. As we need only an estimation of the costs for each subgraph, we define an estimation factor for each of the predicates. Based on the reasoning rules, each of the predicates can activate a different set of rules which will cause different computation costs. For example, based on the reasoning level, and complexity of the used rules the predicates like *"rdf:type"* or *"owl:sameAs"* can activate different reasoning rules than the other predicates like *"owl:intersectionOf"*. We call this factor the reasoning factor $F_{reasoning}$ and assume that we can define for each of the predicates a reasoning factor, a number between 0 and 1. For example, the predicates like *"rdf:type"* or *"owl:sameAs"* have the highest factors. We assume that we can define this factor manually, by looking at the chains in the reasoning rules. We consider the reasoning factor for object properties that are not explicitly defined in the reasoning rules as zero and the data predicates have also a reasoning factor of zero. The *estimated matching probability factor of predicates* F_p^{EM} is calculated by the following function: $F_p^{EM} = 2/(F_{reasoning} * N_p/(N_{Triples} - N_p) + N_p/N_{Triples})$

Filter Functionality Estimation of Subgraphs: In the multi-step SCEP the throughput of an event stream in each step is highly depending on the rate of events matched in the previous step. Subgraphs with less results are good filters for event detection, because they are less likely to be matched. In the case that they are used at the beginning of multi-step event enrichment and detection, the rate of the events in the following step can be highly reduced. A *filter functionality factor* is introduced for each of the subgraphs. This factor is calculated based on the *estimated matching probability factor of the predicate* (F_p^{EM}) and the graph structure properties of the user query. The subgraph marks a specific part of the graph pattern of a user query which can have different properties, e.g., if the subgraph is positioned in the leaf of a tree structure, or if it is in the intersection of subgraphs (see Figure 2). The intersection nodes are nodes on the graph where event operations divide the graph. We define two factors for the graph properties of the subgraph, F_{Leaf} is 2 if final leaf of G_{SUB} is bounded and 1 if not. F_{Inter} is 2 if G_{SUB} includes intersection nodes and 1 if not.

Sometimes in a user query a subgraph is repeated in several places on the graph pattern. In this case this subgraph might have a better filtering functionality

for the event detection than the other subgraphs. We consider this effect with the factor for repetition of subgraphs: $F_{Repetition} = N_{Repetition}/N_{total}$

$N_{repetition}$ is the repetition count number of the G_{SUB} in the multi-set MS_{GSUBs} (how many times a subgraphs appears in the whole graph pattern).

We estimate the cost of different subgraphs included in the query based on some heuristics of the stored data on KB (like shown example data in Table 1) so that we can compare the subgraphs and organize the sequence order of processing in multi-step SCEP. Based on the estimated cost we can calculate the saving cost by changing the order of subgraphs. The filter functionality of each subgraphs is calculated by the following equation: $F_{Filter} = (F_{Leaf} * F_{Inter} * F_{Repetition} * max(F_p^{EM}))/AvgNumberOfResults$

One of the heuristics about the used predicates in BGPs is how many answer triples have the triple pattern in average ($AvgNumberOfResults$), e.g., a triple with *dbpedia-owl:location* has how many triple results in average.

Transmission Cost: If the query to the external KB has several results, the EMA can retrieve them and transmit them to the enrichment base node. In the case that the user query includes BGPs with RDF data properties, the result of such triple pattern has only one single literal as result. If the BGP includes an object property it can have several result resources as results (URIs). For our cost estimation we can count the number of result items (resources or literals).

Based on the order of subgraphs in an execution plan and number of processing steps, different latencies and loads can be generated. The total load of a plan is estimated with the total number of queries sent to the external KB, and the total number of transmitted results within a time window (for a user query).

Algorithm 1: Algorithm for Selecting of Execution Plan for Subgraphs

Data: MS_{GSUBs} a multiset of subgraphs
Data: $latency_{expected}$ user specified latency expectation
Result: $plan$: an execution Plan for enrichment and detection
$FirstStepGraphs = selectFirstStepGraphs(MS_{GSUBs})$;
$SelectedGraph = getFirstElement(sort(FirstStepGraphs, F_{Filter}))$;
while *hasNextPlan* **do**
  $NextStepGraph = MS_{GSUBs} \backslash SelectedGraph$;
  $plan = (SelectedGraph, NextStepGraph)$;
  $(latency_{current}, load_{current}) = execute(plan, Time_t)$;
  if $(latency_{current} \leq latency_{expected} \wedge load_{current} \geq load_{previous})$ **then**
    $SelectedGraph = getFirstElement(sort(NextStepGraph, F_{Filter}))$;
    $MS_{GSUBs} = MS_{GSUBs} \backslash (SelectedGraph \cup FollowingStep)$;
    Add a new processing step, if all plans for the number of steps are searched ;
    $load_{previous} = load_{current}$;
  else
    return $PLAN$;
  end
end

Our approach for searching an acceptable plan is presented in Algorithm 1. We sort the list of subgraphs based on their estimated filter functionality factor F_{Filter}. For the generation of an execution plan, we use this list as initial execution plan. Our algorithm is a greedy algorithm which starts with an initial plan. To avoid

the exponential search effort for testing the costs of all possible plans, we start with an estimated plan (a plan that might have an acceptable cost and latency) and then run several iterations with other plans which might improve the latency and load, until we find an acceptable plan for the user query.

Our algorithm starts by using a two-step processing plan. If the average latency is acceptable and the subgraph set has more elements, then a new processing step is added. This process is continued until the latency of the overall system is greater than the user expected latency.

We monitor the latency and the total result transmission for a time period, if the requirements can not be satisfied, then we change the execution plan until we have one of the acceptable plans. If the latency is under the threshold of the user expectation, we change the plan to check if we can reduce the caused load on the external KB. In the case that the load is acceptable for the external KB, we can accept the current plan as our execution plan.

4 Evaluation

We have implemented a prototype of our multi-step approach and its algorithms in Java. We use the OpenRDF framework[3] to process the triple patterns and send them as SPARQL queries to an external triplestore. For the event detection steps, we used the Esper[4] event processing engine. In our experiments, we forwarded the output stream of event enrichment to the event detection step so that they can build up the multi-step processing steps. To cleanly separate the impact of our approach from the underlying implementation and configuration choices, we compared the evaluation metrics with each other on the same implementation setup and used abstract performance metrics. We compared different transmission costs of different approaches on the same implementation and data setups. To separate the impact of specific data on our experimental results, we executed the experiments on different event streams, knowledge bases and queries.

For our experiments we needed two kind of test datasets; the event data stream (dynamic data part) and the background knowledge base. We used in our experiments both synthetic and real-world data sets.

Event Stream Dataset: We have used an event stream which simulates the event stream of a stock market exchange. We use a list of 500 companies (S&P500), each event is the price change of a company in stock market. The event stream is generated by randomly selecting one of the companies in the list and sending the event object to the stream. Each event instance includes a string for stock symbol, a string for stock name, an integer for stock latest price, an integer for stock last volume and a string for the link URL (the URL links the event instances to the relevant resources in the KB, e.g. a stock event to the appropriate company). We use the synthetically generated stream to be able to conduct performance and cost experiments.

Background Knowledge Dataset: As background knowledge we have used a complete mirror of DBpedia (version 3.4). Each of the event instances includes

[3] OpenRDF http://www.openrdf.org/

[4] `http://esper.codehaus.org`, version 4.6.0

a URL which maps to a DBpedia resource. By using this link, the SCEP system can extract background knowledge. To each event instance a payload is added which is a key-value set of the enriched attributes and the extracted value for the attribute.

Experiment Setup: As our evaluation metrics do not depend on the run-time environment, the hardware setup [5] and configurations used in the experimentation do not impact our evaluation results, due to the reason that we compare the different approaches to each other. However, we mention some of the results of event processing in some of the experiments, like the performance of the event processing or the latency of detection complex events.

4.1 Evaluation of Multi-Step Processing

We conducted several experiments with different types of event detection queries to investigate the effect of multi-step event enrichment and detection. The main effects that we investigated are the overall performance, the detection delay time (the total transit time of events identified as complex event) and the overall load on the external knowledge base (number of queries to the KB, number of transmitted results).

Star-Shaped Event Patterns: One of the pattern types used for event enrichment and detection is the simple star-shaped event patterns. We conducted several experiments and changed the number of BGPs in the event enrichment queries and measured the average processing performance, the latencies of detection of complex events and the transmission costs. Our experiments have been done on queries which include 2 up to 7 BGPs. The number of BGPs specified also the number of processing steps, i.e., a query with 3 BGPs is processed in maximal 3 steps. A complete list of our queries is listed on this URL.[6] The SPARQL queries which we used in our experiments on star-shaped patterns are sQ2 to sQ7 for event enrichment (including 2 to 7 BGPs) and the Esper queries eQs2 to eQs7 for event detection. Table 1 shows the relevant statistics about the predicates. The column *"Result"* is the average number of results for a BGP with this predicate.

The comparison of processing performance of single-step processing with two-step and multi-step approaches for different star-shaped patterns are presented in Fig. 3. We conducted different experiments with queries including 2 BGPs up to 7 BGPs. In single-step processing we enriched each event instance with the results of the complete query, i.e., sending the query as a whole to the KB and enrich the results to the event stream. In two-step processing we used the first BGP (with predicate *dbpedia-owl:location*) for the first processing step and the rest of query in the following second step, e.g., for a query with 7 BGPs, 1 BGP in first step and 6 in the following step. In multi-step processing, we extended the processing steps

[5] In our experiments we use one single instance of the Virtuoso triple store, version 06.01.3127. It is installed on a host (Intel Xeon CPU E31245 @ 3.30GHz) with 8 GB RAM and Ubuntu Linux 12.04. The event mapping agents and the event processing agents (EPAs) are installed on a separated host (i7-2600 CPU @ 3.40GHz) with 16 GB RAM. Each of the processing agents are different java threads on the same host.

[6] List of our queries `http://download.teymourian.de/scep-queries.html`

Table 1: Distribution of RDF Predicates used in our Queries in DBpedia Dataset

No.	Predicate	Numbers	% of KB	Results
1	*dbpedia-owl:location*	219880	0.076%	2
2	*dbpedia-owl:industry*	31047	0.011%	2
3	*dbpedia-owl:numberOfEmployees*	12425	0.004%	1
4	*dbpprop:products*	11899	0.004%	2
5	*dbpedia-owl:subsidiary*	2663	0.001%	1
6	*rdf:type*	11085199	3.849%	3
7	*dcterms:subject*	13606126	4.724%	4
Others Predicates		263044482		
No. Triples in the KB		288013721		

to the number of existing BGPs in the user query, i.e., for a query with 7 BGPs we have 7 processing steps. The performance of the single-step processing approach is continuously reduced with the number of BGPs as illustrated in Fig. 3. We observed that the performance of two step processing and multi-step processing are very close to each other and they significantly differ from the performance of single step processing.

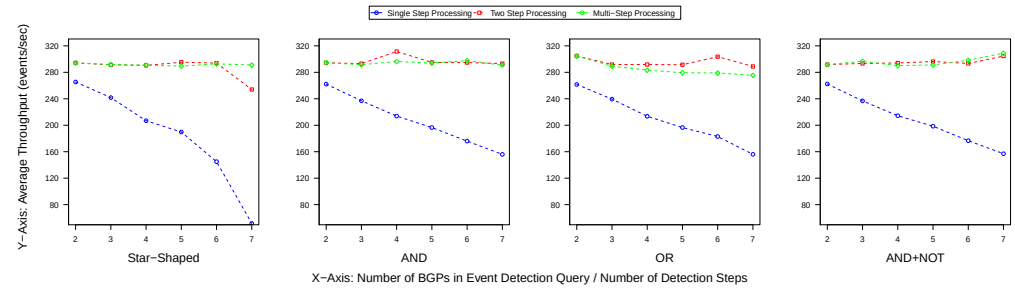


Fig. 3: Performance Comparison of Multi-Step Processing

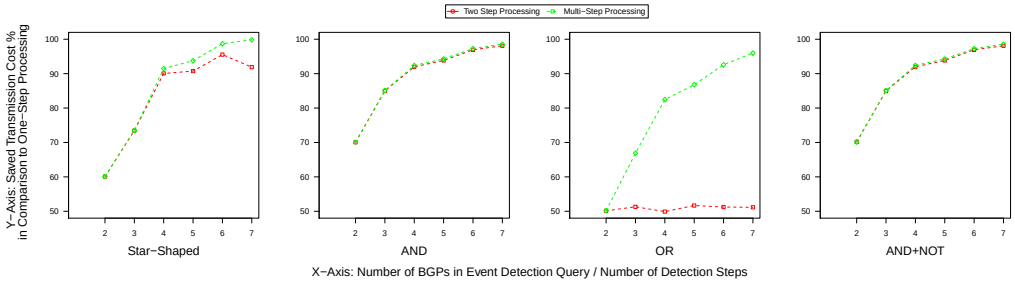


Fig. 4: Saved Transmission Costs in Comparison to Single-Step Processing

We sent 50000 raw events (50k events) through the system and counted the total number of transmitted results (No. of transmitted RDF resource or literals) from the knowledge bases to the event detection point. The saved transmission cost in comparison to the single-step processing is shown in Fig. 4. As it is shown,

we can significantly save costs if we do two-step or multi-step processing. For a query with two BGPs we can save up to 60% of transmission costs and for a query with 6 or 7 BGPs up to 90% of costs. One interesting observation is that the difference of transmission cost saving between two-step processing and multi-step processing (in our case up to 7-steps) are very close to each other. When we add more steps beyond the second step we do not save much more of processing costs. However, the existing small difference of cost reduction between two-step and multi-step is depending on the query used for the first step. In each of the

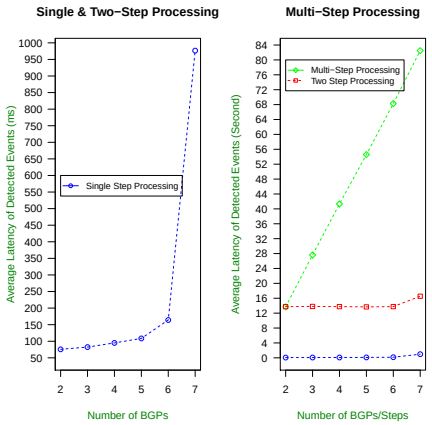


Fig. 5: Average Latency of Detected Complex Events in Multi-Step Processing (Star-Shaped Query)

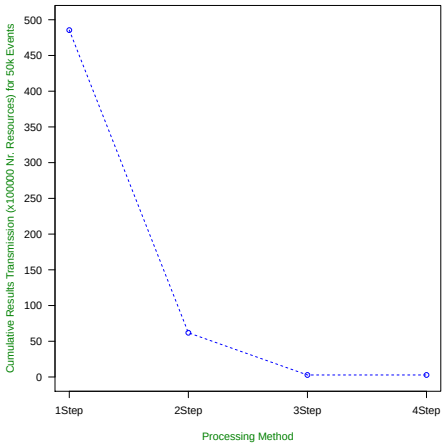


Fig. 6: Cumulative Transmission Cost for Different Steps using AND OP

processing steps based on the filter functionality factor of queries, a large amount of events are filtered out and only a subset of them are forwarded. As expected, the forwarding of events to the next step causes a delay for the detection of complex events. Fig. 5 shows the comparison of latencies of different approaches (single, two and multi-step), it only shows the latency of detected of complex events (not the latency of the dropped events), i.e., the time difference between event generation and detection of complex event in the final stage. The single step processing has the lowest latency due to the fact that the events are processed in single step.

With the comparison of two Figures 3 and 4 we can argue that the two-step processing has a good balance between the performance, latency and the generated transmission costs. The single step processing might not be suitable for use cases with a high throughput event stream, and causes a high load on the KB, but has an acceptable latency when the complex events are detected and the time between event capturing and event detection is very low. The two-step processing has high performance and saves transmission costs, but the event detection latency should be in the expected range for an specific use case.

Different Effects of Event Operators We conducted experiments for analysing the effects of event algebra operators on multi-step event processing. We made different experiments for OR, AND, SEQ, and NOT operators. [7]

Star-Shaped Event Patterns: For the evaluation of execution plans for the star-shaped event pattern, we use the same event enrichment and detection queries as used for previous experiments. In our experiment we consider a multi-step event enrichment and detection for a query with 7 different subgraphs (7 BGPs) which are processed in 7 processing steps. The cumulative KB result transmission for four different execution plans is presented in Fig. 7a. Our experiments shows that the Plan-3 (7,6,1,2,4,3,5) has the maximum of the KB results transmitted and the Plan-2* (5,3,4,2,1,6,7) has the minimum result transmission (selected by our algorithm).

AND and SEQ Operators: As previously described, the processing of AND and SEQ can be handled in a sequential process, and every single event instance can be checked for the possible matching in sub-graphs/sub-events. The performance and cost reduction is very similar to star-shaped event patters. The cost reduction differs from the cost reduction effect of the OR operator. The cost reduction can only be compared with the single step processing and not with the OR operator.

OR Event Algebra Operation: The OR operation has an impact on the topology of the multi-step event processing, due to the nature of the OR operator. As we can see the performance is significantly higher than the single-step processing and very close to multi-step processing. The transmission cost reduction shows that the cost reduction for two-step processing is mostly around 50% in comparison to the single-step processing and it significantly increases with the multi-step approach. However, the average latency for detection of complex events is increased with the usage of the multi-step approach.

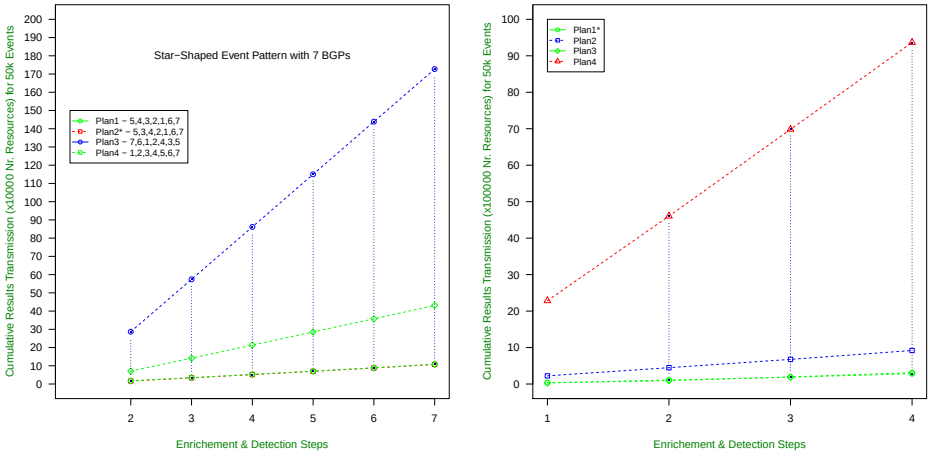
NOT Operator: We used the NOT operator together with an AND operator due to the fact that only a single NOT operator can only change the detection topography in the multi-step processing. The result of our experiments shows that the performance and cost reduction of the (AND+NOT) operator is very similar to the AND operator.

Multi-Step Planning: We compare the performance, enrichment and transmission costs of different plans provided by our algorithm (marked with *) with some of the randomly selected plans. We evaluate the proposed planning algorithm for different SCEP query types, Star-Shaped and combination with different event operators. The optimization of execution plans also has its effects on the performance of the overall system. The Throughput performance for the Plan-2 is about 270 events/s and for the Plan-3 (the worst plan) is 310 events/sec. One acceptable plan is the two-step processing plan in which the first step filters the high rate of raw events and in the following step the rest of the extracted query

[7] The queries `http://download.teymourian.de/scep-queries.html` for the event enrichment are the same SPARQL queries sQ2 to sQ7 and sQorm2 to sQorm7. For the event detection based we used eQopm1 to eQopm7 and eQorm1 to eQorm7 (changed based on event operators).

subgraphs are matched. Thereby, we can improve the throughput and latency of event processing, and reduce the processing and transmission costs.

Effect of Operators on Planning: We consider a query which includes two event component parts which are combined with an event operator. In our first experiment we use the AND operator and setup 4 different processing steps. The table 2 shows the different predicates used in the 4 processing steps for event enrichment and detection, in step 4 the AND operator is applied.



(a) Plans for Star-Shaped Pattern (b) Plans for AND Operator

Fig. 7: Cumulative Transmission Costs for different Plans

The Figure 7b shows the cumulative transmission of results with different plans. Plan-4 shows one of the maximum cost plans and plan-1 one of the optimal plans, any other plans may have a cost in this range. The Figure 6 shows the comparison of commutative transmission costs in the case that we apply a different number of processing steps. We observe a high cost reduction from single step processing to 4 step processing. The Table 3 shows the Plan-2* (generated by our algorithm) and 3 randomly selected processing plans.

Table 2: Execution Plan-1

Steps	Enrichment Predicates	Matching Predicates
Step-1	5,3	(5 OR 3)
Step-2	4,2	(4 OR 2)
Step-3	1,6	(1 OR 6)
Step-4*	7	((5,4,1,6) AND (3,2,6,7))

Table 3: Four Enrichment Plans

Plans	Step-1	Step-2	Step-3	Step-4
Plan-1	5,4	3,2	1,6	7
Plan-2*	5,3	4,2	1,6	7
Plan-3	7,6	1,2	4,3	5
Plan-4	1,2	3,4	5,6	7

The effect of the *SEQ Operator* is very similar to the AND operator with the small difference that first event component of the complex event should happen before the other event components. And the effect of the OR operator is very similar to the shown star-shaped pattern. Since a major assumption in our approach for the semantic enrichment of events is that the knowledge bases in the use case is huge, the experiment with the proposed framework with respect to the different sizes and complexity of KBs can only effect the performance of the CEP system but it does not effect our greedy algorithm for the planning of event enrichment and detection.

5 Related Work

Margara et al. [6] provide a survey on event processing and data stream processing systems. The event processing approaches can be categorized in rule-based and non-rule-based approaches. Our plan-based event processing approach extends the research results from the previous work on plan-based event processing [1,9]. These systems deal with planning the event acquisition to reduce the network transmission costs. In our approach, we have to optimized the load and transmission costs of knowledge acquisition.

Several stream reasoning languages [4] and processing approaches [2,10] haven been proposed. Bolles et. al. propose StreamSPARQL [4] for extending SPARQL for the propose querying RDF streams. It enables window-based and event-based windowing on RDF streams. Barbieri et al. propose Continuous SPARQL (C-SPARQL) [3] as a language for continuous query processing and Stream Reasoning. Stream reasoning approaches like [13] are proposed for reasoning on RDF streams, and are not designed for fusion of background KBs and event streams.

CQELS [10] is a query processor for unified query processing over both Linked Stream Data and Linked Data. ETALIS [2] is a rule-based stream reasoning and complex event processing (CEP) system. ETALIS is implemented in Prolog and uses a Prolog inference engine for event processing. ETALIS provides EP-SPARQL as a language for complex events and stream reasoning.

The differences of our approach with the RDF streaming approach are: 1. Some of the RDF stream reasoning systems may also use 'static' reference knowledge along with RDF streams, but the amounts of static reference knowledge is limited to the main memory of the reasoner, because they have to include the knowledge into the reasoner memory. In our approach the reasoning is delegated to a highly optimized external reasoner. We assume that the external KB is a highly scalable triple store with a scalable reasoner (a distributed triple store and reasoner). The event mapping engine can query the external knowledge base and activate the reasoner for the external knowledge. The result of the reasoning is then enriched to the event stream and forwarded for the event pattern matching in the following event detection phase. 2. In our approach the event stream is not mapped to an RDF stream. The event stream is based on an event model, e.g., a name/value pair stream, as it is usual in most of the event processing use cases.

6 Discussion

We have shown that our approach for planning of multi-step event enrichment and detection can be optimized so that we can avoid as much as possible the full stream enrichment to optimize the knowledge acquisition costs. One main conclusion of our work is that within the user event processing latency expectation it is possible to plan the enrichment and detection steps so that the knowledge acquisition costs can be reduced. One future optimization of our work would be to optimize the query planning algorithms by considering the intermediate joins of event detection graphs.

References

1. Mert Akdere, Ugur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1:66–77, August 2008.
2. Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. ETALIS: Rule-Based Reasoning in Event Processing. In Sven Helmer, Alexandra Poulovassilis, and Fatos Xhafa, editors, *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 99–124. Springer Berlin / Heidelberg, 2011.
3. Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for c-sparql queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452, New York, NY, USA, 2010. ACM.
4. Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming sparql extending sparql to process data streams. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 448–462, Berlin, Heidelberg, 2008. Springer-Verlag.
5. François Bry, Adrian Paschke, Patrick Th. Eugster, Christof Fetzer, and Andreas Behrend, editors. *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*. ACM, 2012.
6. Alessandro Margara and Gianpaolo Cugola. Processing flows of information: from data stream to complex event processing. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 359–360, New York, NY, USA, 2011. ACM.
7. Adrian Paschke, Paul Vincent, Alexandre Alves, and Catherine Moxey. Tutorial on advanced design patterns in event processing. In Bry et al. [5], pages 324–334.
8. Peter R. Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 62–82, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
9. Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.
10. Martin Serrano, Danh Le Phuoc, Maciej Zaremba, Alex Galis, Sami Bhiri, and Manfred Hauswirth. Resource optimisation in iot cloud systems by using matchmaking and self-management principles. In Alex Galis and Anastasius Gavras, editors, *Future Internet Assembly*, volume 7858 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2013.
11. Kia Teymourian and Adrian Paschke. Semantic rule-based complex event processing. In *RuleML 2009: Proceedings of the International RuleML Symposium on Rule Interchange and Applications*, 2009.
12. Kia Teymourian, Malte Rohde, and Adrian Paschke. Fusion of background knowledge and streams of events. In Bry et al. [5], pages 302–313.
13. Onkar Walavalkar, Anupam Joshi, Tim Finin, and Yelena Yesha. Streaming Knowledge Bases. In *Proceedings of the Fourth International Workshop on Scalable Semantic Web knowledge Base Systems*, October 2008.