FPSPARQL: A Language for Querying Semi-Structured Business Process Execution Data

Seyed-Mehdi-Reza Beheshti[1] Boualem Benatallah[1]
Hamid Reza Motahari-Nezhad[2] Sherif Sakr[1]

[1] University of New South Wales
Sydney 2052, Australia
{sbeheshti,boualem,ssakr}@cse.unsw.edu.au

[2] HP Labs Palo Alto
CA 94304, USA
hamid.motahari@hp.com

Technical Report
UNSW-CSE-TR-1103
April 2011

Abstract

Business processes (BPs) in today's enterprises are realized over multiple IT systems and services. Understanding the execution of a BP in terms of its scope and details is challenging specially as it is subjective: depends on the perspective of the person looking at BP execution. Existing business process querying and visualization tools assume a pre-defined model of BPs. However, often models and documentation of BPs or the correlation rules for process events across various IT systems do not exist or are outdated. In this paper, we present a framework and a language that provide abstractions and methods for the explorative querying and understanding business process execution from the event logs of workflows, IT systems and services. We propose a query language for analyzing event logs of process-related systems based on the two concepts of *folders* and *paths*, which enable an analyst to group related events in the logs or find paths among events. Folders and paths can be stored to be used in future analysis, enabling progressive and explorative analysis. We have implemented the proposed techniques in a graph processing engine called FPSPARQL by extending SPARQL graph query language. We present the evaluation results on the performance and the quality of the results using a number of process event logs.

1 Introduction

A business process (BP) consists of a set of coordinated tasks and activities employed to achieve a business objective or goal. In modern enterprises, BPs are realized over a mix of workflows, IT systems, Web services and direct collaborations of people. The understanding of business processes and analyzing BP execution data (e.g., logs containing events, interaction messages and other process artifacts) is difficult due to lack of documentation and especially as the process scope and how process events across these systems are correlated into process instances are subjective: depend on the perspective of the process analyst. As an example, one may want to understand the delays to the ordering process (the end-to-end from ordering to the delivery) for a specific customer, while another analyst is only considered with the packaging process for any orders in the shipping department. Certainly, one process model would not serve the analysis purpose for both situations. Rather there is a need for a process-aware querying approach that enables analysts to analyze the process events from their perspectives, for the specific goal that they have in mind, and in an explorative manner. In this paper, we focus on addressing this problem.

To enable process execution analysis, the first step is gathering and integration of process execution data in a *process event log* from various systems and services. We assume that execution data are collected from the source systems and transformed into an event log using existing data integration approaches [28]. We assume we can access the event metadata and the payload content of events in the integrated process log.

The next step is providing techniques to enable users define the relationships between process events. The various ways in which process events may be correlated are characterized in earlier work ([5] and our prior work [24]). In particular, in [24], we introduced the notion of a *correlation condition* as a binary predicate defined on the attributes of event payload that allows to identify whether two or more events are potentially related to the same execution instance of a process. We use the concept of correlation condition to formulate the relationships between any pairs of events in the log.

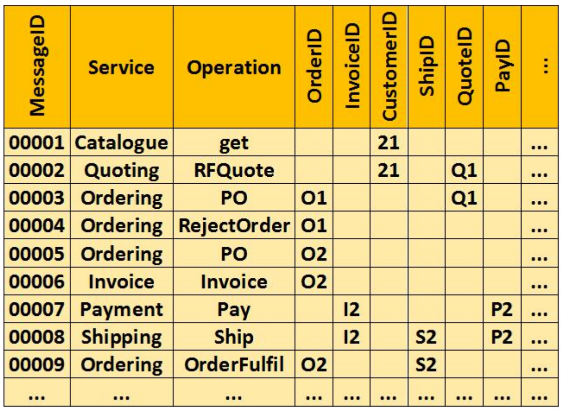
As the final step, we need abstractions and methods that enable the exploration of event relationships, and the discovery of process abstractions. In this paper, we introduce a data model for process events and their relationships and a query language to query and explore events, their relationships and possible aggregation of events into process-centric abstractions. We introduce two concepts of *folders* and *paths*, which help in partitioning events in logs into groups and paths of a graph in order to simplify the discovery of process-centric relationships (e.g., process instances) and abstractions (e.g., process models). We define a folder node as a placeholder for a group of inter-related events. Folder nodes can be nested and composed of other folders. We use a path node to represent the set of events that are related to each other through transitive relationships. These paths may lead into discovering process instances.

In summary, we present a novel framework for manipulating, querying, and analyzing event logs. The unique contributions of the paper are as follows:

- We propose a graph data model that supports typed and untyped events, and introduce *folder* and *path* nodes as first class abstractions. A folder node contains a collection of related events, and a path node represent the results of a query that consists of one or more paths in the events relationship graph based on a given correlation condition.

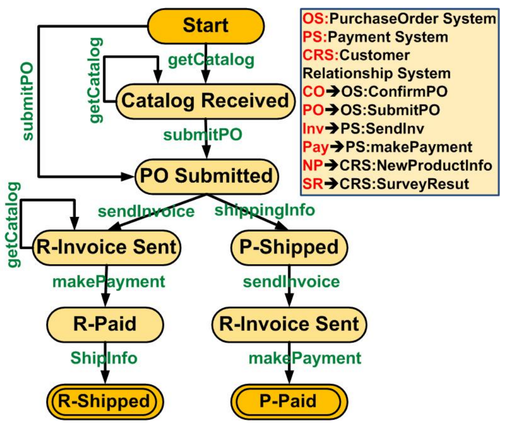
- We present a process event query language and graph-based querying processing engine called FPSPARQL, which is a Folder-Path enabled extension of SPARQL [27][1]. We use

[1]SPARQL is a declarative query language, an official W3C standard and widely used for querying and extracting information from directed-labeled RDF graphs [27].



MessageID	Service	Operation	OrderID	InvoiceID	CustomerID	ShipID	QuoteID	PayID	...
00001	Catalogue	get			21				...
00002	Quoting	RFQuote			21		Q1		...
00003	Ordering	PO	O1				Q1		...
00004	Ordering	RejectOrder	O1						...
00005	Ordering	PO	O2						...
00006	Invoice	Invoice	O2						...
00007	Payment	Pay		I2				P2	...
00008	Shipping	Ship		I2		S2		P2	...
00009	Ordering	OrderFulfil	O2			S2			...
...	...	...	...	...	...	...	...	...	...

(a) Example of SCM service interaction log.



(b) A simplified business process in SCM log for retailer service [22].

Figure 2.1: Event log analysis case study.

FPSPARQL to query and analyze events, folder and path nodes in order to analyze business process execution data.

- We describe the implementation of FPSPARQL and the results of the evaluation of the performance of the engine and the quality of results over large event logs. The evaluation shows that the engine is reasonably fast and the quality of the query results is high in terms of precision/recall.
- We provide a front-end tool for the exploration and visualization of results in order to enable users to examine the event relationships and the potential for discovering process instances and process models.

The remainder of this paper is organized as follows: We present a case study on process event logs in section 2. In section 3 we give an overview of the query language and data model. We present the FPSPARQL query language in section 4. In section 5 we show how we use the query language for analyzing the case study process log. In section 6 we describe the query engine implementation and evaluation experiments. Finally, we discuss related work in section 7, before concluding the paper with a prospect on future work in Section 8.

2 Event Log Analysis: Example Scenario

Let us assume a set of web services that are interacting to realize a number of business processes. In this context, two or more services exchange messages to fulfil a certain functionality, e.g. to order goods and deliver them. The events related to messages exchanged during service conversations may be logged using various infrastructures [24]. A generic log model L represented by set of messages $L = \{m_1, m_2, ..., m_n\}$ where each message m is represented by a tuple $m_i \in A_1 \times A_2 \times ... \times A_k$ [23]. Attributes $A_1, ..., A_k$ represent the union of all the attributes contained in all messages. Each single message typically contains only a subset of these attributes and $m_x.A_i$ denotes the value of attribute A_i in message m_x. Each message m_x has a mandatory

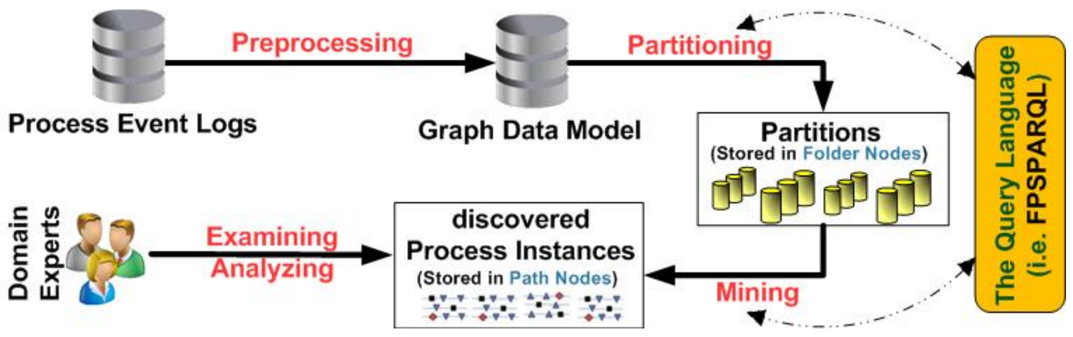


Figure 3.1: Event log analysis scenario.

attribute τ that denotes the timestamp at which the event (related to the exchange of m_x) has been recorded.

In particular, we use the interaction log of a set of services in a supply scenario provided by WS-I (the Web Service Interoperability organization), referred in the following as SCM. Figure 2.1 illustrates a simplified business process in SCM (Supply Chain Management) and an example of SCM log. The log of the SCM business service contains 4,050 messages, 14 service operations (e.g. CO, PO, and Inv), and 28 attributes (e.g. sequenceid, custid, and shipid). Figure 2.1(b) illustrates a set of processes defined in SCM scenario such as purchase order system (OS), payment system (PS), and customer relationship system (CRS).

We will use this log in the paper to demonstrate how various users use the querying framework introduced in this paper for exploring and understanding process event logs. For example, we will show how an analyst can: (i) use correlation conditions to partition SCM log (e.g. examples 1 and 2 in section 5); (ii) explore the existence of transitive relationships between messages in constructed partitions to identify process instances (e.g. examples 3 and 4 in section 5); and (iii) analyze discovered process instances by discovering a process model to understand the result of the query in terms of process execution visually (e.g. example 5 in section 5).

3 A Query Language for Analyzing Process Logs

3.1 Overview

We introduce a graph-based data model for modeling the process entities (events, artifacts and people) in process logs and their relationships, in which the relationship among entities could be expressed using regular expressions. In order to enable the explorative querying of the process logs represented in this model, we propose the design and development of an interactive query language that operates on this graph-based data model. The query language enables the users to find entities of their interests and their relationships. The data model include abstractions which act as higher level entities of related entities to browse the results as well as store the result for follow-on queries. The process events in these higher level entities could be used for further process-specific analysis purposes. For instance, inspecting entities for finding process instances, as well as applying process mining algorithms on containers having process instances for discovering process models. Figure 3.1 shows an overview of the steps in analyzing process event logs (i.e. preprocessing, partitioning, and analysis) in our framework, which is described in the following sections.

3.2 Data Model and Abstractions

We propose to model a process log as a graph of typed nodes and edges. We define a graph data model for organizing a set of entities as graph nodes and entity relationships as edges of the graph. This data model supports: (i) entities, which is represented as a data object that exists separately and has a unique identity; (ii) folder nodes, which contain entity collections. A folder node represents the results of a query that returns a collection of related entities; and (iii) path nodes, which refer to one or more paths in the graph, which are the result of a query, too. A path is the the transitive relationship between two entities. Entities and relationships are represented as a directed graph $G = (V, E)$ where V is a set of nodes representing entities, folder or path nodes, and E is a set of directed edges representing relationships between nodes.

Entities. Entities could be structured or unstructured. Structured entities are instances of entity types. An entity type consists of a set of attributes. Unstructured entities, are also described by a set of attributes but may not conform to an entity type. This entity model offers flexibility when types are unknown and take advantage of structure when types are known. We assume that all unstructured entities are instances of a generic type called *ITEM.* ITEM is similar to *generic table* in [25]. We store entities in the *entity store.*

Relationships. A relationship is a directed link between a pair of entities, which is associated with a regular expression defined on the attributes of entities that characterizes the relationship. A relationship can be *explicit*, such as *was triggered by* in '$event_1$ *wasTriggeredBy* $event_2$' in a BPs execution event log. Also a relationship can be *implicit*, such as a relationship between an entity and a larger (composite) entity that can be inferred from the nodes.

Folder Nodes. A folder node contains a set of entities that are related to each other. In other words, the set of entities in a folder node is the result of a given query that require grouping graph entities in a certain way. A folder node creates a higher level node that other queries could be executed on top of it. Folders can be nested, i.e., a folder can be a member of another folder node, to allow creating and querying folders with relationships at higher levels of abstraction. A folder may have a set of attributes that describes it. A folder node is added to the graph and can be stored in the *folder store* to enable reuse of the query results for frequent or recurrent queries.

Path Nodes. A path is a transitive relationship between two entities showing a sequence of edges from the start entity to the end. This relationship can be codified using regular expressions [2, 8] in which alphabets are the nodes and edges from the graph. We define a path node for each query that results in a set of paths. We use existing reachability approaches to verify whether an entity is reachable from another entity in the graph. Some reachability approaches (e.g. all-pairs shortest path [8]) report all possible paths between two entities. We define a path node as a triple of (V_{start}, V_{end}, RE) in which V_{start} is the start node, V_{end} is the end node and a regular expression RE. We store all paths of a path node in the *path store.*

4 Querying Process Logs

As mentioned earlier, we model process logs as a graph. In order to query this graph a graph query language is needed. Among languages for querying graphs, SPARQL [27] is a declarative query language, an official W3C standard, and based on a powerful graph matching mechanism that allows binding variables to components in the input graph. However, SPARQL does not support the construction and retrieval of subgraphs. Also paths are not first class objects in SPARQL [27, 18]. In order to analyze BPs event logs (see section 3), we propose a graph processing engine, i.e. FPSPARQL [9] (a Folder-Path enabled extension of the SPARQL), to manipulate and query entities, and folder and path nodes. We support two levels of queries: (i) Entity-level Queries: at this level we use SPARQL to query entities in the process logs; and (ii) Aggregation-level Queries: at this level we use FPSPARQL to construct and query folder nodes and path nodes.

4.1 Entity-level Queries

At this level, we support the use of SPARQL to query entities and their attributes in the process logs. SPARQL is a declarative and extendable query language which contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions in process event logs. Each pattern consists of *subject*, *predicate* and *object*, and each of these can be either a variable or a literal. We use the '@' symbol for representing attribute edges and distinguishing them from the relationship edges between graph nodes. As an example, we may be interested in retrieving a list of messages in SCM log (section 2) that have the same value on *requestsize* and *responsesize* attributes and the values for their *timestamps* falls between τ_1 and τ_2. Following is the SPARQL query for this example:

```
select ?m
where {
  ?m @type message.
  ?m @requestsize ?x.
  ?m @responsesize ?y.
  ?m @timestamp ?t.
  FILTER(?x =?y && ?t > $\tau_1$ && ?t < $\tau_2$).
}
```

In this query, variable $?m$ represents messages in the SCM log. Variables $?x$, $?y$, and $?t$ represent the value of the attributes *requestsize*, *responsesize*, and *timestamp* respectively. Finally the *filter* statement restrict the result to those messages for which the filter expression evaluates to *true*.

4.2 Aggregation-level Queries

Standard SPARQL querying mechanisms is not enough to support querying needs for analyzing BP execution data based on the introduced data model in Section 3.2. In particular, SPARQL does not support folder and path nodes and querying them natively and such queries needs to be applied to the whole graph. In addition, querying the result of a previous query becomes complex and cumbersome, at best. Also path nodes are not first class objects in SPARQL [8, 18]. We extend SPARQL to support aggregation-level queries to satisfy specific querying needs of proposed data model. Aggregation-level queries in FPSPARQL include two special constructs: (a) *construct* queries: used for constructing folder and path nodes, and (b) *apply* queries: used to simplify applying queries on folder and path nodes.

Folder Node Construction. To construct a folder node (i.e. a partition in event log), we introduce the $fconstruct$ statement. This statement is used to group a set of related entities or folders. A basic folder node construction query looks like this:

$fconstruct < Folder\_NodeName >$
[*select* $?var1$ $?var2$ ... | $(Node1\ ID,\ Node2\ ID,\ ...)$]
where {
  $pattern1.\ pattern2.\ ...$
}

A query can be used to define a new folder node by listing folder node name and entity definitions in the *fconstruct* and *select* statements, respectively. Also a folder node can be defined to group a set of entities, folder nodes, and path nodes (see example 2 in section 5). A set of user defined attributes for this folder can be defined in the *where* statement. We instrument folder construction query with the *correlate* statement in order to apply a correlation condition on entity nodes (e.g. messages in service event logs) and correlated the entity nodes for which the condition evaluates to *true*. A correlation condition ψ is a predicate over the attributes of events for attesting whether two events belong to the same instance. For example, considering SCM log, one possible correlation condition is $\psi(m_x, m_y) : m_x.custid = m_y.custid$, where $\psi(m_x, m_y)$ is a binary predicate defined over the attribute *custid* of two messages m_x and m_y in the log. This predicate is true when m_x and m_y have the same value and false otherwise. A basic correlation condition query looks like this:

correlate {
  ($entity_1$, $entity_2$, $edge_1$, *condition*)
  $pattern_1.\ pattern_2.\ ...$
}

As a result, $entity_1$ will be correlated to $entity_2$ through a directed edge $edge_1$ if the *condition* evaluates to *true*. Patterns (e.g. $pattern_1$) can be used for specifying the edge attributes. Example 1 in section 5 illustrates such a query.

Path Node Construction. We introduce the *pconstruct* statement to construct a path node. This statement can be used to: (i) discover transitive relationships between two entities (e.g. by using an existing graph reachability algorithm); or (ii) discover frequent pattern(s) between set of entities (e.g. by using an existing process mining algorithm). If starting and ending nodes identified uniquely first approach will be used, otherwise the second approach will be applied. In both cases the result will be a set of paths which can be stored under a path node name. In general a basic path node construction query looks like this:

$pconstruct < Path_Node\ Name >$
$(Start\ Node, End\ Node, Regular\ Expression)$
where {
  $pattern1.\ pattern2.\ ...$
}

A regular expressions can be used to define a transitive relationship between two entities, i.e. starting node and ending node, or set of frequent patterns to be discovered. Attributes of starting node, ending node, and regular expressions alphabets (i.e. graph nodes and edges) can be defined in the *where* statement. The query applied on a folder in example 3 section 5, illustrates such a query.

Folder Node Queries. We introduce the *apply* statement to retrieve information, i.e. by applying queries, from the underlying folder nodes. These queries can apply on one folder node or the composition of several folder nodes. Our model supports the standard set operations (union, intersect, and minus) to apply queries on the composition of several folder nodes. In general, a basic folder node query looks like this:

```
[Folder Node | (Composition of Folder Nodes)]
APPLY(
 [<Entity-level Query> | <Aggregation-level Query> | <existing process mining algorithms>]
)
```

An entity/aggregation-level query can be applied on folder nodes by listing folder node or composition of folder nodes before *apply* statement, and placing the query in parenthesis after *apply* statement. We also developed an interface to support applying existing process mining algorithms on folder and path nodes. Examples 3 and 4 in section 5 illustrate such queries.

Path Analysis Queries. This type of query is used to retrieve information, i.e. by applying entity-level queries, from the underlying path node by using *apply* statement. Domain experts may use such queries in order to examine the discovered process instances. In general, a basic path node query looks like this:

```
Path_Node_Name
APPLY (
  Entity-level Query
)
```

An entity-level query can be applied on a path node by listing path node name before *apply* statement, and placing the query in parenthesis after *apply* statement. Example 5 in section 5 illustrates such a query.

5 Using FPSPARQL on SCM Scenario

In this section we show how we use FPSPARQL query language to analyze process logs. We focus on the case study presented in section 2.

Preprocessing. The aim of preprocessing of the log is to generate a graph by considering the set of messages in the log as nodes of the graph, and correlation between messages as edges between nodes. In order to preprocess the SCM log we performed the following two steps: (i) generating graph nodes: we extracted messages and their attributes from the log and formed a graph node for each message with no relations between nodes; and (ii) Generating candidate correlations: We used the correlation condition discovery technique introduced in [24] to generate a set of candidate correlation conditions that could be used for examining the relationship between process events.

Partitioning. We use the candidate conditions, in preprocessing phase, to partition the log. Identifying the interestingness of a certain way of partitioning the logs or grouping the process events is "subjective", i.e., depends on the user perspective and the particular querying goal. To cater for interestingness, we enable the user to choose the candidate conditions she is interested in to explore as a basis of relationships among events.

Example 1. Considering SCM log, the correlation condition $\psi(m_x, m_y) : m_x.custid = m_y.custid$ (where $\psi(m_x, m_y)$ is a binary predicate defined over the attribute *custid* of two messages m_x and m_y in the log.), partitions the log into a set of related messages having the same customer ID. These related messages can be stored in a folder node (e.g. custID). Correlation between these messages created automatically, e.g., messages having same value on *custid* attribute connected through a labeled edge *custid*. Figure 6.1(a) in section 6 illustrates how our tool enables users choosing the correlation condition(s) and generating FPSPARQL queries automatically. Following is the FPSPARQL query for this example.

```
fconstruct custID as ?fn
select ?m_id, ?n_id
where {
  ?fn @description 'custid=custid'.
  ?m @isA entityNode.
  ?m @type message.
  ?m @id ?m_id.
  ?m @custid ?x.
  ?n @isA entityNode.
  ?n @type message.
  ?n @id ?n_id.
  ?n @custid ?y.
  correlate{
    (?m,?n,?edge,FILTER(?x=?y && ?n_id>?m_id))
    ?edge @isA edge.
    ?edge @label "custid".
  }
}
```

In this query, the variable ?fn represent the folder node to be constructed, i.e. 'custID'. Variables $?m$ and $?n$ represent the messages in SCM log and $?m\_id$ and $?n\_id$ represent IDs of these messages respectively. Variables $?x$ and $?y$ represent the values for *m.custid* and *n.custid* attributes. The condition $?x =?y$ applied on the log to group messages having same value for *custid* attribute. The condition $?n_{id} >?m_{id}$ makes sure that only the correlation between each message and the following messages in the log are considered. The *correlate* statement will connect messages, for which the condition ($?x =?y$ && $?n_{id} > ?m_{id}$) evaluates to *true*, with a directed labeled edge $?edge$. The result will be stored in folder $custID$ and can be used for further queries.

Example 2. Consider two folder nodes *custID* and *payID* each representing correlated messages based on correlation conditions $\psi(m_x, m_y) : m_x.custid = m_y.custid$ and $\psi(m_x, m_y) : m_x.payid = m_y.payid$ respectively. We can construct a new folder (e.g. *custID_payID*) on top of these two folders in order to group them. The *custID_payID* folder contains events related to customer orders that have been paid. Queries applied on *custID_payID* folder will be applied on all its subfolders. Example query is defined as follows.

```
fconstruct custID_payID as ?fn (custID,payID)
where {
  ?fn @description 'set of ...'.
}
```

In this example the variable ?fn represent the folder node to be constructed, i.e. *custID_payID*. This folder node contains two folder nodes and has a user defined attribute *description*. These folder nodes are hierarchically organized by *part-of* (i.e. an implicit relationship) relationships.

Mining. In this phase a query can be applied on previously constructed partitions to discover process model. As mentioned earlier, a folder node, as a result of a correlation condition, partitions a subset of the events in the log into instances of a process. The process model which the instances inside a folder represent can be discovered using one of the many existing algorithms for process mining [33], including our prior work [22].

It is possible that some folders contain a set of related process events (e.g., the set of orders for a given customer), but not process instances. It is possible for the analyst to apply a regular expression based query on the events in a folder. The regular expression may define a relationship that is not captured by any candidate correlation condition. Applying such queries on a folder node may result in forming a set of paths which can be then stored in a path node. The constructed path node can be examined by the analysts and may considered as a set of process instances. Example queries is defined as follows.

Example 3. We are interested in exploring the existence of transitive relationships between two messages with IDs '3958' and '4042', which includes at least one occurrence of a message having the value *ConfirmProduction* for the attribute *operation*. The reason was to check for the existence of this pattern in order to discover events related to a specific order. We apply this query on the folder node constructed in example 1. As a result, one path discovered. We stored the result in a path node (and named it OrderDiscovery) for further analysis. Figure 6.1(b) in section 6 illustrates the visualized result of this example generated by the front-end tool. Following is the FPSPARQL query for this example.

```
(custID)
apply(
   pconstruct OrderDiscovery
   (?startNode,?endNode,(?e ?n)* e ?msg e (?n ?e)*)
   where {
     ?startNode @isA entityNode.
     ?startNode @type message.
     ?startNode @id '3958'.
     ?endNode @isA entityNode.
     ?endNode @type message.
     ?endNode @id '4042'.
     ?n @isA entityNode.
     ?e @isA edge.
     ?msg @isA entityNode.
     ?msg @type message.
     ?msg @operation 'OrderFulfil'.
   }
)
```

In this example a *pconstruct* query applied on the folder *custID*. Variables $?startNode$ and $?endNode$ denote messages $m_{id=3958}$ and $m_{id=4042}$ respectively. Variables $?e$ and $?n$ denote any edges and nodes in the transitive relationship between $m_{id=3958}$ and $m_{id=4042}$. Finally, $?msg$ denotes a message having the value *ConfirmProduction* for the attribute *operation*. In the regular expression, parentheses are used to define the scope and precedence of the operators and the asterisk indicates there are zero or more of the preceding element.

Example 4. We are interested in exploring transitive relationship between correlated messages in *custID* folder (see example 1) having the pattern "*start* with a message having the value *produce* for the attribute *operation*, which followed by a message having the value *confirmProduction* for the attribute *operation*, and *end* with a message having the value *pay* for the attribute *operation*". As the result set of this query, 12 paths discovered. Unlike example 3, these paths have different starting and ending nodes. We stored the result in a path node (and named it ProductDiscovery) for further analysis. Following is the FPSPARQL query for this example.

```
(custID)
apply(
  pconstruct ProductDiscovery
  (?startNode, ?endNode, e ?msg e)
  where {
    ?startNode @isA entityNode.
    ?startNode @type message.
    ?startNode @operation 'Produce'.
    ?endNode @isA entityNode.
    ?endNode @type message.
    ?endNode @operation 'Pay'.
    ?e @isA edge.
    ?msg @isA entityNode.
    ?msg @type message.
    ?msg @operation 'ConfirmProduction'.
  }
)
```

In this example a *pconstruct* query is applied on the folder *custID*, in which $?startNode$ and $?endNode$ denote set of starting and ending nodes. Variable $?e$ denotes any edges in the regular expression pattern and ?msg denotes a message having the value *ConfirmProduction* for the attribute *operation.* A frequent sequence mining algorithm (developed based on a process mining method) is used to generate frequent pattern(s) based on the specified regular expression, i.e. $(?startNode \rightarrow e \rightarrow ?msg \rightarrow e \rightarrow ?endNode)$.

Example 5. Consider the path node *OrderDiscovery* constructed in example 3. We are interested to find messages in this path node, that have the keyword *Retailer* in their *binding* attributes. Following is the FPSPARQL query for this example.

```
(OrderDiscovery)
apply (
  select ?m_id
  where {
    ?m @isA entityNode.
    ?m @type message.
    ?m @binding ?b.
    Filter regex(?b,"Retailer").
  }
)
```

In this example ?*m_id* denotes message *ID*s that fall inside *OrderDiscovery* path node. The query 'retrieve the messages that have the keyword *Retailer* in their *binding* attributes" will apply on this path node.

6 Implementation and Experiments

6.1 Implementation

The query engine is implemented in Java (J2EE) and uses a relational database system (we utilized IBM DB2 as a back-end database for the experiments). As FPSPARQL core, we implemented a SPARQL-to-SQL translation algorithm based on the proposed relational algebra for SPARQL [13] and semantics preserving SPARQL-to-SQL query translation [12]. This algorithm supports *aggregate* and *keyword search* queries. We implemented the proposed techniques on top of this SPARQL engine. We developed four optimization techniques proposed in [11, 29, 12] to increase the performance of the query engine. Implementation details, include a graphical representation of the query engine, can be found in [9]. A front-end tool prepared to assist users in four steps:

Step1: Preprocessing. We have developed a workload-independent algorithm for: (i) processing and loading a log file into an RDBMS for manipulating and querying entities, folders, and paths; (ii) generating powerful indexing mechanisms (see [9] for details). We also provided inverted indexes [35] on folder store in order to increase the performance of queries applied on folders.

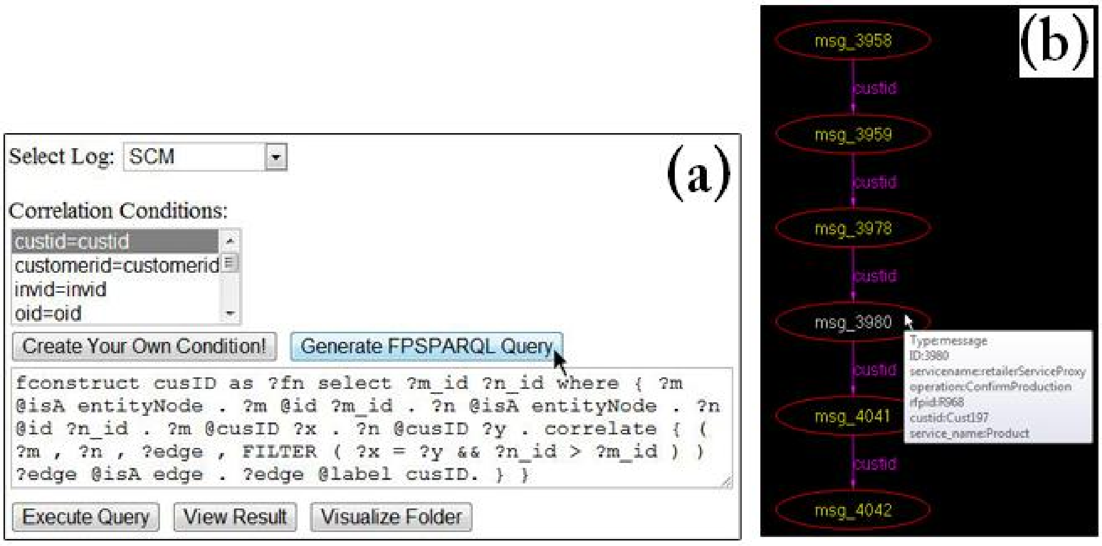
Step2: Partitioning. We provide users with a list of interesting correlation conditions based on the algorithm for discovering interesting conditions in [24]. Users may choose these correlation conditions to partition a log. In order to generate *folder construction queries*, we provide users with an interface (i.e. FPSPARQL GUI) to choose the correlation condition(s) and generate FPSPARQL queries automatically (see Figure 6.1(a)).

Step3: Mining. We provide users with templates to generate regular expressions and use them in path queries. We have developed an interface to support applying existing process mining algorithms on folder nodes. We developed a regular expression processor which supports optional elements (?), loops (+,*), alternation (—), and grouping ((...)) [8]. We provide the ability to call existing process mining algorithms for path node queries (see [9] for details).

Step4: Visualizing. We provided users with a graph visualization tool for the exploration of results (see Figure 6.1(b), i.e. the discovered process model for the query result in example 3). Users are able to view folders, paths, and the result of queries in a list and visualized format. This way, event relationships and candidate process instances can be examined by the analyst.

6.2 Datasets

We carried out experiments on three datasets: (i) SCM: This dataset introduced in the case study in section 2; (ii) Robostrike. This is the interaction log of a multi-player on-line game service. The log contains 40,000 messages, 32 service operations, and 19 attributes; and (iii) PurchaseNode. This process log was produced by a workflow management system supporting a purchase order management service. The log contains 34,803 messages, 26 service operations, and 26 attributes. Details about these dataset can be found in [34].



```
fconstruct cusID as ?fn select ?m_id ?n_id where { ?m
@isA entityNode . ?m @id ?m_id . ?n @isA entityNode . ?n
@id ?n_id . ?m @cusID ?x . ?n @cusID ?y . correlate { (
?m , ?n , ?edge , FILTER ( ?x = ?y && ?n_id > ?m_id ) )
?edge @isA edge . ?edge @label cusID. } }
```

```
Type:message
ID:3980
servicename:retailerServiceProxy
operation:ConfirmProduction
rfpid:R968
custid:Cust197
service_name:Product
```

Figure 6.1: Screenshots of FPSPARQL GUI: (a) The query generation interface in FPSPARQL, and (b) The discovered process model for the query result in example 3.

6.3 Evaluation

We evaluated the performance and the query results quality using SCM, Robostrike, and PurchaseNode process logs.

Performance. The performance of the queries assessed using *query execution time* metric. The preprocessing step took 3.8 minutes for the SCM log, 11.2 minutes for the Robostrike log, and 9.7 minutes for the PurchaseNode log. For the partitioning step, we constructed 10 folders for each process log (i.e. SCM, Robostrike, and PurchaseNode). These folders selected according to provided list of interesting correlation conditions by the tool. Figure 6.2(a,b, and c) shows the average execution time for constructing selected folders for each log. For the mining step, we applied path node construction queries on each constructed folder. These path queries generated by domain experts who were familiar with the process models of proposed process logs. For each folder we applied one path query. As the result, the set of paths for each query were discovered and stored in path nodes. Figure 6.2 (d,e, and f) shows the average execution time for applying constructed path queries on the folders for each log. We ran these experiment for different sizes of process logs.

Quality. The quality of the results is assessed using classical *precision* metric which defined as the percentage of discovered results that are actually interesting. For evaluating the interestingness of the result, we ask domain experts who have the most accurate knowledge about the dataset and the related process to: (i) codify their knowledge into regular expressions that describe paths through the nodes and edges in the folders; and (ii) analyze discovered paths and identify what they consider relevant and interesting from a business perspective. The quality evaluation applied on SCM log. Five folders constructed and three path queries applied on on each folder. As a result 31 paths discovered, examined by domain experts, and 29 paths (precision=93%) considered relevant.

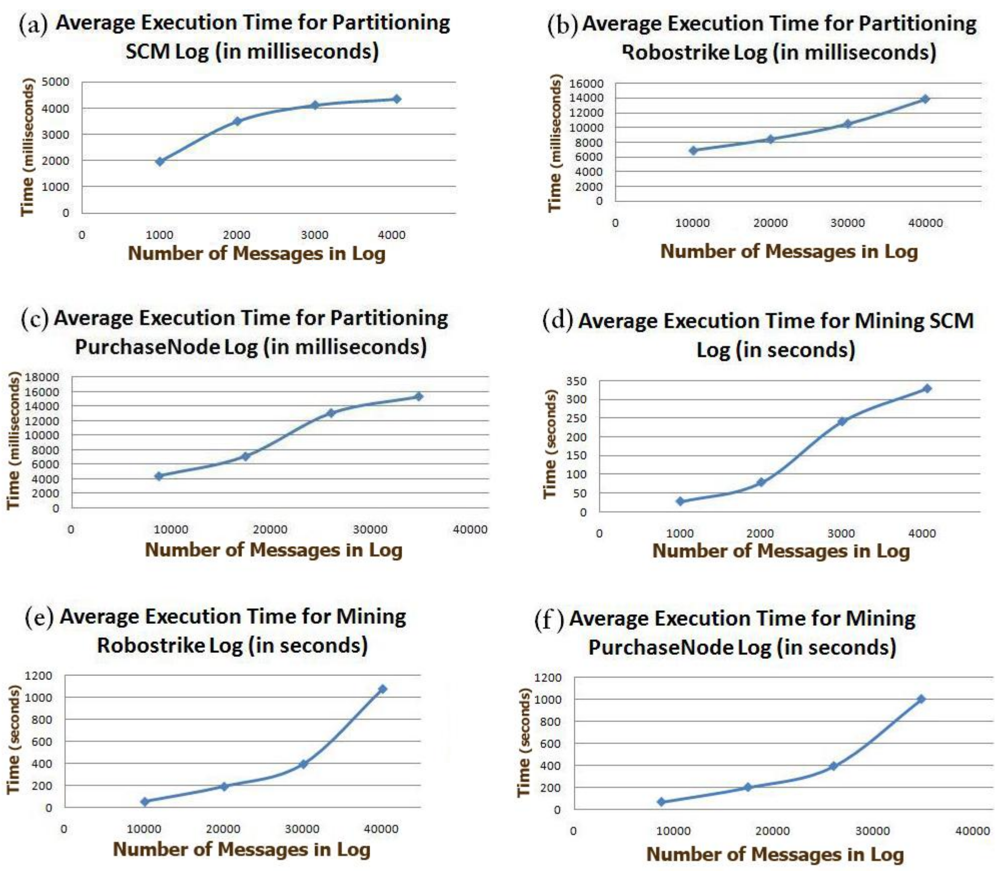


Figure 6.2: The performance evaluation results of the approach on three datasets, illustrating: (i) the average execution time for partitioning: (a) SCM log, (b) Robostrike log, and (c) PurchaseNode log; and (ii) the average execution time for mining: (d) SCM log, (e) Robostrike log, and (f) PurchaseNode log;

6.4 Discussion

We evaluated our approach using different types of process event logs, i.e. PurchaseNode (a single-process log), SCM (a multi-service interaction log), and Robostrike (a complex logic of a real-world). The evaluation shows that the approach is performing well (also in [9] we evaluated the performance of the FPSPARQL query engine compared to one of the well-known graph databases, the HyperGraphDB, which shows the great performance of the FPSPARQL query engine). As illustrated in Figure 6.2 we divide each log into regular number of messages and ran the experiment for different sizes of process logs. The evaluation shows a polynomial (nearly linear) increase in the execution time of the queries in respect with the dataset size. Based on the lesson learned, we believe the quality of discovered paths is highly related to the regular expressions generated to find patterns in the log, i.e. generating regular expressions by domain experts will guarantee the quality of discovered patterns.

7 Related Work

We discuss related work in two categories: (i) business processes and (ii) graph query languages.

7.1 Business Processes

In recent years, querying techniques for BPs received high interest in the research community. Some of existing approaches for querying BPs [4, 6, 15, 16, 30] focused on querying the definitions of BP models. They assume the existence of enterprise repository of business process models. They provide business analysts with a visual interface to search for certain patterns and analyze and reuse BPs that might have been developed by others. These query languages are based on graph matching techniques. BP-QL [6] and [15] are designed to query business processes expressed in BPEL. BPMN-Q [4, 30] and VQL [16] are oriented to query generic process modeling concepts.

Process mining techniques [33, 10, 34] represent another perspective for querying business processes. The main concern of these approaches is to reverse engineer the definitions of business process models from execution logs of information system components. Moreover, depending on how much details the log gives, they can provide statistics about many aspects of the business processes such as: the average duration of process instances or average resource consumptions.

Querying running instances of business processes, represents another flavor of querying BPs [7, 20, 26]. These approaches can be used to monitor the status of running processes and trace the progress of execution. BP-MON [7] and PQL [20] can be used to discover many problems such as: detecting the occurrence of deadlock situations or recognizing unbalanced load on resources. Pistore et. al. [26] proposed an approach, i.e. based on BPEL specification, to monitor the composition and execution of Web services. These approaches assume that the business process models are predefined and available. Moreover, they presume that the execution of the business processes is achieved through a business process management system (e.g BPEL).

In our approach, understanding the processes, i.e. in the enterprise, and their execution through exploration and querying event logs is a major goal. The focus is also on scenarios where processes are implemented over IT systems, and there is no up-to-date documentation of process definition, its execution and/or no data on the the correlation rules for process events into process instances, which is often the case in today's environments.

7.2 Graph Query Languages

Recently, several research efforts show a growing interest in graph models and languages. A recent book [1] and survey [3] discuss a number of graph data models and query languages. Some of existing approaches for querying and modeling graphs [14, 3, 19] focused on defining constraints on nodes and edges simultaneously on the entire object of interest. These approaches do not support querying nodes at higher levels of abstraction.

Some query languages for graph [14, 19] are focused on the uniform treatment of nodes and edges and support queries that return subgraphs. BiQL [14] supports a closure property on the result of its queries meaning that the output of every query can be used for further querying. HyperGraphDB[19] supports grouping related nodes in a higher level node through performing special purpose APIs. Compared to BiQL and HyperGraphDB, in our work folders and paths are first class abstractions (graph nodes) and can be defined in a hierarchical manner, over which queries are supported.

SPARQL [27] is a declarative query language, an W3C standard, for querying and extracting information from directed labeled RDF graphs. SPARQL supports queries consisting of triple patterns, conjunctions, disjunctions, and other optional patterns. However, there is no support for querying grouped entities. Paths are not first class objects in SPARQL [27, 18]. In FPSPARQL, we support folder and path nodes as first class entities that can be defined at several levels of abstractions and queried. In addition, we provide an efficient implementation of a query engine that support their querying.

8 Conclusion

In this paper, we presented a data model and query language for querying and analyzing business processes execution data. The data model supports structured and unstructured entities, and introduces folder and path nodes as first class abstractions. Folders allow breaking down an event log into smaller clearer groups. Mining folder nodes may result in discovering process-centric relationships (e.g., process instances) and abstractions (e.g., process models) which can be stored in path nodes for further analysis. The query language, i.e. FPSPARQL, defined as an extension of SPARQL to manipulate and query entities, and folder and path nodes. We have developed an efficient and scalable implementation of FPSPARQL. To evaluate the viability and efficiency of FPSPARQL, we have conducted experiments over large event logs.

As future work, we plan to design a visual query interface to support users in expressing their queries over the conceptual representation of the event log graph in an easy way. Moreover, we plan to make use of interactive graph exploration and visualization techniques (e.g. storytelling systems [31]) which can help users to quickly identify the interesting parts of a event log graph. We are also interested in the temporal aspects of process graph analysis, as in some cases (e.g. provenance[1]) the structure of the graph may change rapidly over time.

Bibliography

[1] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data.* Springer Publishing Company, Incorporated, 2010.

[2] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending sparql with regular expression patterns. *J. Web Sem.*, 7(2):57–73, 2009.

[3] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40:1:1–1:39, February 2008.

[4] A. Awad. BPMN-Q: A Language to Query Business Processes. In *EMISA*, 2007.

[5] Alistair P. Barros, Gero Decker, Marlon Dumas, and Franz Weber. Correlation patterns in service-oriented architectures. volume 4422, pages 245–259, 2007.

[6] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes with BP-QL. *Inf. Syst.*, 33(6), 2008.

[7] C. Beeri, A. Eyal, T. Milo, and A. Pilberg. Monitoring Business Processes with Queries. In *VLDB*, 2007.

[8] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. ICSE'10, pages 125–134, 2010.

[1]Provenance [21] is the process of recording events happening in digital environments which generates the documented history of information items' creation.

[9] Seyed-Mehdi-Reza Beheshti, Sherif Sakr, Boualem Benatallah, and Hamid Reza Motahari-Nezhad. Extending SPARQL to support entity grouping and path queries. Technical report, unsw-cse-tr-1019, University of New South Wales, 2010.

[10] R. Bose and W. Aalst. Context Aware Trace Clustering: Towards Improving Process Mining Results. In *SDM*, 2009.

[11] Artem Chebotko, Shiyong Lu, Xubo Fei, and Farshad Fotouhi. Rdfprov: A relational rdf store for querying and managing scientific workflow provenance. *Data Knowl. Eng.*, 69(8):836–865, 2010.

[12] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving sparql-to-sql translation. *Data Knowl. Eng.*, 68(10):973–1000, 2009.

[13] R. Cyganiak. A relational algebra for SPARQL. Hpl-2005-170, HP-Labs, 2005.

[14] Anton Dries, Siegfried Nijssen, and Luc De Raedt. A query language for analyzing networks. CIKM'09, pages 485–494, NY, USA, 2009. ACM.

[15] R. Eshuis and P. Grefen. Structural Matching of BPEL. In *ECOWS*, 2007.

[16] C. Francescomarino and P. Tonella. Crosscutting Concern Documentation by Visual Query of Business Processes. In *BPM Workshops*, 2008.

[17] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. Fast and accurate estimation of shortest paths in large graphs. CIKM'10, pages 499–508, 2010.

[18] David A. Holl, Uri Braun, Diana Maclean, and Kiran kumar Muniswamy-reddy. Choosing a data model and query language for provenance. IPAW'08, 2008.

[19] Borislav Iordanov. Hypergraphdb. WAIM'10, pages 25–36, 2010.

[20] M. Momotko and K. Subieta. Process Query Language: A Way to Make Workflow Processes More Flexible. In *ADBIS*, 2004.

[21] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. Mcgrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. IPAW'08, pages 323–326, Berlin, Heidelberg, 2008. Springer.

[22] Hamid R. Motahari-Nezhad, Régis Saint-Paul, Boualem Benatallah, and Fabio Casati. Deriving protocol models from imperfect service conversation logs. *IEEE Trans. on Knowl. and Data Eng.*, 20:1683–1698, December 2008.

[23] Hamid Reza Motahari-Nezhad, Boualem Benatallah, Régis Saint-Paul, Fabio Casati, and Periklis Andritsos. Process spaceship: discovering and exploring process views from event logs in data spaces. *PVLDB*, 1(2):1412–1415, 2008.

[24] Hamid Reza Motahari-Nezhad, Régis Saint-Paul, Boualem Benatallah, and Fabio Casati. Event correlation for process discovery from web service interaction logs. *Accepted in VLDB Journal*, 2010.

[25] Beng Chin Ooi, Bei Yu, and Guoliang Li. One table stores all: Enabling painless free-and-easy data publishing and sharing. CIDR'07, pages 142–153, 2007.

[26] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and Monitoring Web Service Composition. In *AIMSA*, 2004.

[27] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf (working draft). Technical report, W3C, March 2007.

[28] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[29] Sherif Sakr and Ghazi Al-Naymat. Relational processing of rdf queries: a survey. *SIGMOD Rec.*, 38(4):23–28, 2009.

[30] Sherif Sakr and Ahmed Awad. A Framework for Querying Graph-Based Business Process Models. In *WWW*, 2010.

[31] Arjun Satish, Ramesh Jain, and Amarnath Gupta. Tolkien: an event based storytelling system. *Proc. VLDB Endow.*, 2:1630–1633, August 2009.

[32] Silke TriBl and Ulf Leser. Fast and practical indexing and querying of very large graphs. SIGMOD '07, pages 845–856, NY, USA, 2007. ACM.

[33] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47:237–267, November 2003.

[34] L. Wen, J. Wang, W. Aalst, B. Huang, and J. Sun. A novel approach for process mining based on event types. *J. Intell. Inf. Syst.*, 32(2), 2009.

[35] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. WWW '08, pages 387–396, USA, 2008.

Appendix: FPSPARQL Queries

In this section we present a number of FPSPARQL query samples used in SCM dataset evaluation. For each query expressed in English, we construct a FPSPARQL query and its equivalent SQL query, generated by FPSPARQL-to-SQL translation algorithm.

Query 1. Partition SCM log into a set of related messages having the same customer ID, i.e. correlation condition $\psi(m_x, m_y) : m_x.custid = m_y.custid$.

FPSPARQL:

```
fconstruct cusID as ?fn
select ?m_id ?n_id
where {
 ?m @isA entityNode .
 ?m @id ?m_id .
 ?n @isA entityNode .
 ?n @id ?n_id .
 ?m @cusID ?x .
 ?n @cusID ?y .
 correlate {
  ( ?m , ?n , ?edge , FILTER ( ?x = ?y && ?n_id > ?m_id ) )
  ?edge @isA edge .
  ?edge @label cusID.
 }
}
```

SQL:

```
INSERT INTO NULLID.FolderStore (FolderId , Subject , Predicate , Object)
SELECT 'Folder1' as FolderId , m_id As Subject, 'e1' As Predicate,
n_id As Object From (SELECT r.m_id AS m_id , r.n_id AS n_id
FROM (Select es1.Subject AS m, es2.Object AS m_id, es3.Subject AS n,
es4.Object AS n_id, es5.Object AS x, es6.Object AS y
From NULLID.FilteredEntity es1, NULLID.FilteredEntity es2,
NULLID.FilteredEntity es3, NULLID.FilteredEntity es4,
NULLID.FilteredEntity es5, NULLID.FilteredEntity es6
Where es1.predicate = '@class' AND es1.object = 'entityNode' AND
es3.object = 'entityNode' AND es2.predicate = '@id' AND
es3.predicate = '@class' AND es5.predicate = '@custid'
AND es1.subject = es2.subject AND es1.subject = es5.subject
AND es1.object = es3.object AND es1.object <> '' AND
es3.subject = es4.subject AND es3.subject = es6.subject AND
es2.predicate = es4.predicate AND es5.predicate = es6.predicate AND
(es5.object = es6.object AND es4.object > es2.object)) AS r);
INSERT INTO NULLID.EntityStore (Subject , Predicate , Object)
VALUES ('e1' , '@class' , 'edge');
INSERT INTO NULLID.EntityStore (Subject , Predicate , Object)
VALUES ('e1' , '@label' , 'custid');
```

```
INSERT INTO NULLID.EntityStore (Subject,Predicate,Object)
SELECT DISTINCT 'Folder1' as Subject , '@name' as Predicate ,
'custID' as Object  FROM NULLID.FolderStore
WHERE FolderId = 'Folder1';INSERT INTO NULLID.EntityStore
(Subject,Predicate,Object) SELECT DISTINCT 'Folder1' as Subject ,
'@class' as Predicate , 'folderNode' as Object
FROM NULLID.FolderStore WHERE FolderId = 'Folder1';
```

Query 2. Partition SCM log into a set of related messages having the same order ID and order reference ID , i.e. correlation condition $\psi(m_x, m_y) : m_x.oID = m_y.OrdRef$.

FPSPARQL:

```
fconstruct orderRefID as ?fn
select ?m_id ?n_id
where {
 ?m @isA entityNode .
 ?m @id ?m_id .
 ?n @isA entityNode .
 ?n @id ?n_id .
 ?m @oID ?x .
 ?n @OrdRef ?y .
 correlate {
  ( ?m , ?n , ?edge , FILTER ( ?x = ?y && ?n_id > ?m_id ) )
  ?edge @isA edge .
  ?edge @label orderRefID.
 }
}
```

SQL:

```
INSERT INTO NULLID.FolderStore (FolderId , Subject , Predicate , Object)
SELECT 'Folder1' as FolderId , m_id As Subject, 'e1' As Predicate,
n_id As Object From (SELECT r.m_id AS m_id , r.n_id AS n_id FROM
(Select es1.Subject AS m, es2.Object AS m_id, es3.Subject AS n,
es4.Object AS n_id, es5.Object AS x, es6.Object AS y From
NULLID.FilteredEntity es1, NULLID.FilteredEntity es2,
NULLID.FilteredEntity es3, NULLID.FilteredEntity es4,
NULLID.FilteredEntity es5, NULLID.FilteredEntity es6 Where
es1.predicate = '@class' AND es1.object = 'entityNode' AND
es3.object = 'entityNode' AND es2.predicate = '@id' AND
es3.predicate = '@class' AND es5.predicate = '@oID' AND
es6.predicate = '@OrdRef' AND es1.subject = es2.subject AND
es1.subject = es5.subject AND es1.object = es3.object AND
es1.object <> '' AND es3.subject = es4.subject AND
es3.subject = es6.subject AND es2.predicate = es4.predicate AND
(es5.object = es6.object AND es4.object > es2.object)) AS r);
INSERT INTO NULLID.EntityStore (Subject , Predicate , Object)
```

```
VALUES ('e1' , '@class' , 'edge');INSERT INTO NULLID.EntityStore
(Subject , Predicate , Object) VALUES ('e1' , '@label' , 'orderRefID');
INSERT INTO NULLID.EntityStore (Subject,Predicate,Object)
SELECT DISTINCT 'Folder1' as Subject , '@name' as Predicate ,
'orderRefID' as Object FROM NULLID.FolderStore WHERE FolderId = 'Folder1';
INSERT INTO NULLID.EntityStore (Subject,Predicate,Object)
SELECT DISTINCT 'Folder1' as Subject , '@class' as Predicate ,
'folderNode' as Object FROM NULLID.FolderStore WHERE FolderId = 'Folder1';
```

Query 3. Partition SCM log into a set of related messages having the same cust ID and rfp ID , i.e. correlation condition $\psi(m_x, m_y) : m_x.custid = m_y.custid \wedge m_x.rfpid = m_y.rfpid$.

FPSPARQL:

```
fconstruct cusID_rfpID as ?fn
select ?m_id ?n_id
where {
?m @isA entityNode .
?m @id ?m_id .
?n @isA entityNode .
?n @id ?n_id .
?m @cusID ?x1 .
?n @cusID ?y1.
?m @rfpID ?x2.
?n @rfpID ?y2 .
correlate {
( ?m , ?n , ?edge , FILTER ( ?x1 = ?y1 && ?x2 = ?y2 && ?n_id > ?m_id ) )
?edge @isA edge . ?edge @label cusID_rfpID. } }
```

SQL:

```
INSERT INTO NULLID.FolderStore (FolderId , Subject , Predicate , Object)
SELECT 'Folder1' as FolderId , m_id As Subject, 'e1' As Predicate,
n_id As Object From (SELECT r.m_id AS m_id , r.n_id AS n_id FROM
(Select es1.Subject AS m, es2.Object AS m_id, es3.Subject AS n,
es4.Object AS n_id, es5.Object AS x1, es6.Object AS y1, es7.Object AS x2,
es8.Object AS y2 From NULLID.FilteredEntity es1, NULLID.FilteredEntity es2,
NULLID.FilteredEntity es3, NULLID.FilteredEntity es4,
NULLID.FilteredEntity es5, NULLID.FilteredEntity es6,
NULLID.FilteredEntity es7, NULLID.FilteredEntity es8
Where es1.predicate = '@class' AND es1.object = 'entityNode' AND
es3.object = 'entityNode' AND es2.predicate = '@id' AND
es3.predicate = '@class' AND es5.predicate = '@cusID' AND
es7.predicate = '@rfpID' AND es1.subject = es2.subject AND
es1.subject = es5.subject AND es1.subject = es7.subject AND
es1.object = es3.object AND es1.object <> '' AND es3.subject = es4.subject
AND es3.subject = es6.subject AND es3.subject = es8.subject AND
es2.predicate = es4.predicate AND es5.predicate = es6.predicate AND
es7.predicate = es8.predicate AND (es5.object = es6.object AND
es7.object = es8.object AND es4.object > es2.object)) AS r);I
NSERT INTO NULLID.EntityStore (Subject , Predicate , Object)
VALUES ('e1' , '@class' , 'edge');INSERT INTO NULLID.EntityStore
(Subject , Predicate , Object) VALUES ('e1' , '@label' , 'cusID_rfpID');
INSERT INTO NULLID.EntityStore (Subject,Predicate,Object)
SELECT DISTINCT 'Folder1' as Subject , '@name' as Predicate ,
'cusID_rfpID' as Object FROM NULLID.FolderStore WHERE FolderId = 'Folder1';
INSERT INTO NULLID.EntityStore (Subject,Predicate,Object)
SELECT DISTINCT 'Folder1' as Subject , '@class' as Predicate ,
'folderNode' as Object FROM NULLID.FolderStore WHERE FolderId = 'Folder1';
```

Query 4. Partition SCM log into a set of related messages having the same pay ID or ship ID , i.e. correlation condition $\psi(m_x, m_y) : m_x.payID = m_y.payID \lor m_x.shipID = m_y.shipID$.

FPSPARQL:

```
fconstruct payidORshipid as ?fn
select ?m_id ?n_id
where {
 ?m @isA entityNode .
 ?m @id ?m_id .
 ?n @isA entityNode .
 ?n @id ?n_id .
 ?m @payID ?x1 .
 ?n @payID ?y1 .
 ?m @shipID ?x2 .
 ?n @shipID ?y2 .
 correlate {
  ( ?m , ?n , ?edge ,
  FILTER ( (?x1 = ?y1 || ?x2 = ?y2) && ?n_id > ?m_id ) )
  ?edge @isA edge . ?edge @label payidORshipid.
 }
}
```

SQL:

```
INSERT INTO NULLID.FolderStore (FolderId , Subject , Predicate , Object)
SELECT 'Folder1' as FolderId , m_id As Subject, 'e1' As Predicate,
n_id As Object From (SELECT r.m_id AS m_id , r.n_id AS n_id FROM
(Select es1.Subject AS m, es2.Object AS m_id, es3.Subject AS n,
es4.Object AS n_id, es5.Object AS x1, es6.Object AS y1, es7.Object AS x2,
es8.Object AS y2 From NULLID.FilteredEntity es1, NULLID.FilteredEntity es2,
NULLID.FilteredEntity es3, NULLID.FilteredEntity es4,
NULLID.FilteredEntity es5, NULLID.FilteredEntity es6,
NULLID.FilteredEntity es7, NULLID.FilteredEntity es8
Where es1.predicate = '@class' AND es1.object = 'entityNode' AND
es3.object = 'entityNode' AND es2.predicate = '@id' AND
es3.predicate = '@class' AND es5.predicate = '@payID' AND
es7.predicate = '@shipID' AND es1.subject = es2.subject AND
es1.subject = es5.subject AND es1.subject = es7.subject AND
es1.object = es3.object AND es1.object <> '' AND es3.subject = es4.subject
AND es3.subject = es6.subject AND es3.subject = es8.subject AND
es2.predicate = es4.predicate AND es5.predicate = es6.predicate AND
es7.predicate = es8.predicate AND ((es5.object = es6.object OR
es7.object = es8.object) AND es4.object > es2.object)) AS r);
INSERT INTO NULLID.EntityStore (Subject , Predicate , Object)
VALUES ('e1' , '@class' , 'edge');INSERT INTO NULLID.EntityStore
(Subject , Predicate , Object) VALUES ('e1' , '@label' , 'payidORshipid');
INSERT INTO NULLID.EntityStore (Subject,Predicate,Object)
SELECT DISTINCT 'Folder1' as Subject , '@name' as Predicate ,
'payidORshipid' as Object FROM NULLID.FolderStore WHERE FolderId = 'Folder1';
INSERT INTO NULLID.EntityStore (Subject,Predicate,Object)
SELECT DISTINCT 'Folder1' as Subject , '@class' as Predicate ,
'folderNode' as Object FROM NULLID.FolderStore WHERE FolderId = 'Folder1';
```

Query 5. Apply the query 'discovering transitive relationships between two messages with IDs '3958' and '4042', which includes at least one occurrence of a message having the value *ConfirmProduction* for the attribute *operation*' on the folder 'cusID' constructed in Query1.
FPSPARQL:

```
  (cusID)
  apply(
     pconstruct OrderDiscovery
     (?startNode,?endNode,(?e ?n)* e ?msg e (?n ?e)*)
     where {
       ?startNode @isA entityNode.
       ?startNode @type message.
       ?startNode @id '3958'.
       ?endNode @isA entityNode.
       ?endNode @type message.
       ?endNode @id '4042'.
       ?n @isA entityNode.
       ?e @isA edge.
       ?msg @isA entityNode.
       ?msg @type message.
       ?msg @operation 'OrderFulfil'.
     }
  )
```

SQL:
A graph reachability algorithm used to discover transitive relationships between nodes. We developed an interface to support various graph reachability algorithms [1] such as Transitive Closure, GRIPP, Tree Cover, Chain Cover, Path-Tree Cover, and Shortest-Paths [17]. In general, there are two types of graph reachability algorithms [1]: (1) algorithms traversing from starting

vertex to ending vertex using breadth-first or depth-first search over the graph, and (2) algorithms checking whether the connection between two nodes exists in the edge transitive closure of the graph. Considering $G = (V, E)$ as directed graph that has n nodes and m edges, the first approach incurs high cost as $O(n + m)$ time which requires too much time in querying. The second approach results in high storage consumption in $O(n^2)$ which requires too much space. In this experiment, we used the GRIPP [32] algorithm which has the querying time complexity of $O(m - n)$, index construction time complexity of $O(n + m)$, and index size complexity of $O(n + m)$.