An Artifact-Centric Activity Model for Analyzing Knowledge Intensive Processes

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

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

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

Technical Report
UNSW-CSE-TR-201210
March 2012

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Many processes in organizations involve knowledge workers. Understanding and analyzing knowledge-intensive processes is a challenge for organizations today. As knowledge-intensive processes involve human judgements in the selection of activities that are performed, process execution path can change in a dynamic and ad-hoc manner. Case management is a common approach to support knowledge-intensive processes and almost always involves the collection and presentation of a diverse set of artifacts. In case scenarios, understanding ad-hoc processes entails identifying the interactions among people and artifacts, where artifacts are developed and changed gradually over a long period of time as a case is long running and it changes hands over time. We present a framework, simple abstractions and a language for the explorative querying and understanding of the knowledge-intensive processes. Analyzing the set of activities on artifacts helps in understanding the case process. We introduce two concepts of timed folders to represent evolution of artifacts over time, and activity paths to analyze proposed framework. We have implemented the approach on top of FPSPARQL, a graph query language for analyzing business processes execution. The evaluation shows the viability and efficiency of our approach.

1 Introduction

Many processes in organizations today involve knowledge workers. Understanding and analyzing knowledge-intensive processes is a challenge for organizations today. Many knowledge-intensive processes, e.g. those in domains such as healthcare and governance involve human judgements in the selection of activities that are performed. This lead to dynamic and ad-hoc changes of process execution paths in different process instantiations. Activities of knowledge workers in knowledge intensive processes involve directly working on and manipulating artifacts to the extent that these activities can be considered artifact-centric activities. Case management [35], also known as case handling, is a common approach to support knowledge-intensive processes and almost always involves the collection and presentation of a diverse set of artifacts, and capturing human activities around artifacts.

In case management applications, understanding ad-hoc processes entails identifying the interactions among people and artifacts, where artifacts are developed and changed gradually over a long period of time as a case is long running and changes hands over time. In particular, when case analysts want to find an answer to precise questions, their first stop is usually to understand the evolution of artifacts over periods of time. This, emphasizes the artifact-centric nature of case management process where *time* becomes an important part of the equation. To portray the evolution of artifacts, there is a need to collect meta-data about facts (e.g. artifacts, activities on top of artifacts, and related actors) and relationship among them from various systems/departments over time. This will enable case analysts to apply their knowledge (i.e. extract information about facts and the relationship among them) on knowledge intensive processes.

To understand knowledge-intensive processes, the focus should be on interactions among actors (i.e. people/services) and artifacts over time, where there is no central system to capture such activities at different systems/departments. In case management applications, this is challenging, as artifacts can be accessed/modified by different actors over time, various versions of artifacts can be generated in different sysems/departments, and each artifact version can be derived from various sources. To address these challenges, We present a framework, simple abstractions and a language for the explorative querying and understanding of the knowledge-intensive processes. Analyzing the set of activities on artifacts helps in understanding the case process. The unique contributions of the paper are as follows:

- We propose a temporal graph model for representing the process activities on artifacts, over time, in knowledge intensive processes. This model allows: (i) representing artifacts (and their evolution), actors, and interactions between them through activity relationships; (ii) identifying derivation of artifacts over periods of time; and (iii) discovering timeseries of actors and artifacts in case management applications.
- We introduce two concepts of *timed-folders* to represent evolution of artifacts over time, and *activity-paths* to represent the process which led to artifacts.
- We extend FPSPARQL [5], a graph query language for analyzing processes execution, for explorative querying and understanding of the knowledge-intensive processes. We introduce simple templates for querying evolution, derivation, and timeseries of artifacts.
- We provide a front-end tool for assisting users to create queries in an easy way and visualizing proposed graph model and query results.

The remainder of this paper is organized as follows: We fix some preliminaries in section 2. Section 3 presents an example scenario. In section 4 we present the knowledge intensive process model. In section 5 we propose a query language for querying the proposed model. In section 6

we describe the query engine architecture, implementation, and evaluation experiments. Finally, we discuss related work in section7, before concluding the paper with a prospect on future work in section 8.

2 Preliminaries

Definition 1. [artifact] An artifact can be defined as a digital representation of something, i.e. data object, that exists separately as a single and complete unit and has a unique identity. An artifact can be a *mutable* object, i.e. its attributes (and their values) are able or likely to change over periods of time. An artifact Ar is represented by a set of attributes $\{a_1, a_2, ..., a_k\}$, where k represents the number of attributes.

Definition 2. [artifact version] An artifact may appear in many versions. A version v is an *immutable* deep copy of an entity at a certain point in time. An artifact Ar can be represented by a set of versions $\{v_1, v_2, ..., v_n\}$, where n represents the number of versions. An artifact can capture its *current state* as a version and can restore its state by loading it. Each version v_i is represented as a data object that exists separately and has a unique identity. Each version v_i consists of a snapshot, a list of its parent versions, and meta-data, such as commit message, author, owner, or time of creation. In order to represent the history of an artifact, it is important to create archives containing all previous states of an artifact. The archive allows us to easily answer certain temporal queries such as retrieval of any specific version from the archive and finding the history of an artifact. Archives can be managed using temporal databases [25].

Definition 3. [activity] An activity defined as an action performed on or caused by an artifact version. For example, an action can be used to create, read, updated, or delete an artifact version. We assume that each distinct activity does not have a temporal duration. A timestamp τ can be assigned to an activity.

Definition 4. [process] A process defined as group of related activities performed on or caused by artifacts. A starting timestamp τ and a time interval d can be assigned to a process.

Definition 5. [actor] An actor defined as an entity acting as a catalyst of an activity, e.g. a person or a piece of software that acts for a user or other programs. A process may have more than one actor enabling, facilitating, controlling, affecting its execution.

Definition 6. [artifact evolution] In case management applications, artifacts develop and change gradually over a long period of time as a case is long running and it changes hands over time. Consequently, *artifact evolution* can be defined as the series of related activities on top of an artifact over different periods of time. These activities can take place in different organizations/departments/systems and various actors may act as the catalyst of activities. Documentation of these activities will generate meta-data about actors, artifacts, and activity relationships among them over time.

3 Example Scenario

To understand the problem, we present an example scenario in the domain of case management. This scenario is based on breast cancer treatment cases in Velindre hospital [35]. Figure 3.1-A represents a case instance, in this scenario, where a General Practitioner (GP) suspecting a patient

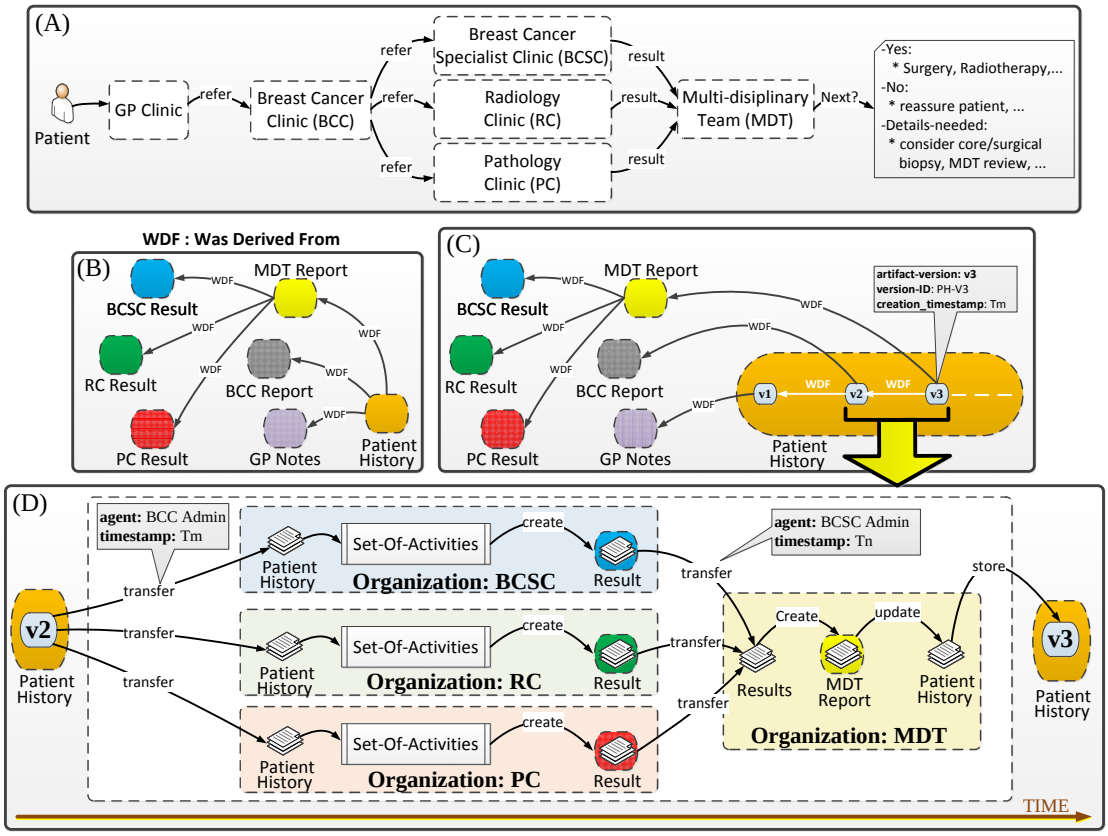

Figure 3.1: Example case scenario for breast cancer treatment including a case instance (A), parent artifacts, i.e. ancestors, for patient history document (B) and its versions (C), and set of activities which shows how version v_2 of patient history document develops and changes gradually over time and evolves into version v_3 (D).

has cancer, update patient history, and referring the patient to a Breast Cancer Clinic (BCC). BCC checks the patients history and requests assessments such as an examination, imaging, fine needle aspiration, and core biopsy. Therefore, BCC administrator refers patient to Breast Cancer Specialist Clinic (BCSC), Radiology Clinic (RC), and Pathology Clinic (PC), where these departments apply medical examinations and send the results to Multi-Disciplinary Team (MDT). The results are gathered by the MDT coordinator and discussed at the MDT team meeting involving a surgeon oncologist, radiologist, pathologist, clinical and medical oncologist, and a nurse. Analyzing the results and the patient history, MDT will decide for next steps, e.g., in case of positive findings, non-surgical (Radiotherapy, Chemotherapy, Endocrine therapy, Biological therapy, or Bisphosphonates) and/or surgical options will be considered. During interaction among different systems, organizations and care team professionals, a set of artifacts will be generated. Figure 3.1-B represents parent artifacts, i.e. ancestors, for patient history document, and Figure 3.1-C represents parent artifacts for its versions. Figure 3.1-D represents a set of activities which shows how version v_2 of patient history document develops and changes gradually over time and evolves into version v_3.

4 Representing Knowledge Intensive Processes

We propose an artifact-centric activity model for knowledge intensive processes to represent interaction between actors and artifacts. This graph data model (i.e. AEM: Artifact Evolution Model) can be used to represent the evolution of artifacts over periods of time. In AEM, We assume that interaction between actors and artifacts is represented by a directed acyclic graph $G_{(\tau_1,\tau_2)} = (V_{(\tau_1,\tau_2)}, E_{(\tau_1,\tau_2)})$, where $V_{(\tau_1,\tau_2)}$ is a set of nodes representing instances of artifacts in time, and $E_{(\tau_1,\tau_2)}$ is a set of directed edges representing activity relationships among artifacts. It is possible to capture the evolution of AEM graphs $G_{(\tau_1,\tau_2)}$ between timestamps τ_1 and τ_2.

4.1 AEM Entities

An entity is an object that exists independently and has a unique identity. AEM consists of two types of entities: artifact versions and folder nodes. Folder nodes represent evolution of artifacts over time.

Artifact Version: Artifacts are represented by a set of instances each for a given point in time. For example, artifact Ar is represented by the set of instances $\{Ar_{t_1}, Ar_{t_2}, Ar_{t_3}, ...\}$ where $\{t_1, t_2, t_3, ...\}$ indicates the activity timestamps at distinct points in time. Artifact instances considered as data objects that exist separately and have a unique identity. An artifact instance can be stored as a new version, as different instances of an entity for different points in time/departments/systems, may have different attribute values. An artifact version can be used over time, annotated by activity timestamps $\tau_{activity}$, and considered as a graph node, i.e., its identity will be the version unique ID and timestamps $\tau_{activity}$.

Timed Folder Node: We proposed the notion of folder node in [5]. Timed folders defined as a timed container for a set of related entities, e.g., to represent artifacts evolution (Definition 6). Timed folders, document the evolution of folder node by adapting a monitoring code snippet. Entities and relationships in a timed folder node are represented as a subgraph $F_{(\tau_1,\tau_2)} = (V_{(\tau_1,\tau_2)}, E_{(\tau_1,\tau_2)})$, where $V_{(\tau_1,\tau_2)}$ is a set of related nodes representing instances of entities in time added to the folder F between timestamps τ_1 and τ_2, and $E_{(\tau_1,\tau_2)}$ is a set of directed edges representing relationships among these related nodes. It is possible to capture the evolution of the folder $F_{(\tau_1,\tau_2)}$ between timestamps τ_1 and τ_2.

4.2 AEM Relationships

A relationship is a directed link between a pair of entities, which is associated with a predicate defined on the attributes of entities that characterizes the relationship. AEM consists of two types of relationships: activity and activity-path. Activity-paths can be used for efficient graph analysis.

Activity Relationships: An activity is an *explicit* relationship that directly links two entities in the graph, and is defined as an action performed on or caused by an artifact version. Activity relationships can be described by a set of attributes:

- *What* (i.e. type) and *How* (i.e. action), two types of activity relationships can be considered in AEM: (i) lifecycle activities, include actions such as creation, transformation, use, or deletion of a AEM entity; and (ii) archiving activities, include actions such as storage and transfer of a AEM entity.

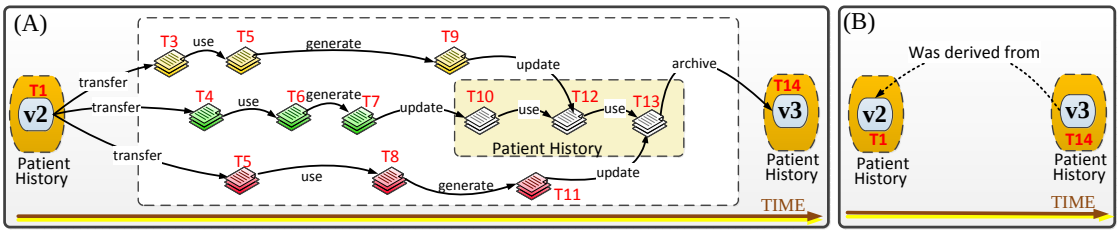

Figure 4.1: Implicit and explicit relationships between versions v_2 and v_3 of patient history document including: (A) activity edges, and (B) constructed activity-path.

- *When*, to indicate the timestamp in which the activity has occurred.
- *Who*, to indicate an actor that enables, facilitates, controls, or affects the activity execution.
- *Where*, to indicated the organization/department where the activity happened.
- *Which*, to indicate the system which hosts the activity.
- *Why*, to indicate the goal behind the activity, e.g. fulfilment of a specific phase or experiment.

These attributes, e.g. actors and organizations, can be stored as individual objects and used for annotating activity edges in the graph.

Activity-Path: Defined as an *implicit* relationship that is a container for a set of related activities which are connected through a path, where a path is a transitive relationship between two entities showing the sequence of edges from the starting entity to the end. This relationship can be codified using regular expressions [5] in which alphabets are the nodes and edges from the graph. We define an activity-path for each query which results in a set of paths between two nodes. Activity-paths can be used for efficient graph analysis. For example, Figure 4.1-A represents a set of activities which shows how version v_2 of patient history develops and changes gradually over time and evolves into version v_3. A path query (see section 5) can be used to discover all/specific path(s) between v_2 and v_3, and group them under an activity path labeled 'was derived from' (Figure 4.1-B). Merging activities using activity-paths will not lose information, as activities that are important to the user will be visible after the merger.

`Discussion.` Activity-paths are different from path-nodes presented in [5], as activity-paths are graph edges which can be used to merge all paths between two entities in AEM graph. Path-nodes are containers for the set of paths, not necessarily between two entities, and considered as graph nodes. We use path-nodes to discover activity paths in AEM graphs to answer queries about derivation, evolution, and timeseries of artifacts.

5 Querying AEM Graphs

Querying AEM graphs needs a graph query language that not only supports primitive graph queries but also is capable of: (i) constructing timed folders and group related activities (paths). In general, the output of every query can be stored as folder/path and used for further querying; (ii) applying further queries to constructed folders/paths, e.g. to analyze their evolution or

understand the merged activities over time; and (iii) applying external tools and algorithms (e.g. to discover shortest path and frequent patterns) to AEM graphs for further analysis.

FPSPARQL [5] (a Folder-Path enabled extension of SPARQL [29]) is a graph query processing engine which supports primitive graph queries, constructing folders/paths, applying further queries to constructed folder/path nodes, and applying external tools and algorithms to graph. There are two levels of queries in FPSPARQL: (a) Graph-level Queries: at this level SPARQL is used to query graphs; and (b) Node-level Queries: at this level FPSPARQL extends SPARQL to construct and query folder nodes and path nodes. In this paper we extend FPSPARQL (TFP-SPQARL: timed FPSPARQL) to support time-aware querying of AEM graphs.

5.1 Formalizing AEM Queries

Many knowledge intensive process queries require traversal of AEM graphs. In order to represent AEM graphs and formalize path queries, we model our prototype based on an RDF data representation. In RDF model, an *RDF triple* (**S**ubject, **P**redicate, **O**bject) can be defined as an element of $(\upsilon \cup \beta) \times \upsilon \times \tau$, where τ represents RDF terminology, υ represents set of of URI references, and β represents set of blanks. An *RDF graph* is a finite set of RDF triples.

Considering ℓ as set of literals, $\upsilon \cup \ell$ will represent the vocabulary ν. Let ν be the set of names appearing in AEM graph and $\nu_{edge} \subseteq \nu$ be a set of names on the arcs in the graph. The label on each $e \in \nu_{edge}$ defines a relationship between the entities in the graph and also allows us to navigate across the different nodes by a single hop. Consequently, a *path* in an RDF graph is a sequence of RDF triples, where the object of each triple in the sequence coincides with the subject of its successor triple in the sequence [9]. In AEM, an activity-path is a path defined over the AEM vocabulary ν using regular expressions [5] in which alphabets are the nodes and edges from the graph, and activity edges have the following mandatory attributes (see Section 4.2): what, how, when, who, where, and which.

5.2 Simplifying Path Queries

Discovering (activity) paths through AEM graphs forms the basis of many AEM queries. In order to discover paths through AEM graphs and apply further operations on the discovered path(s) we use *pconstruct* and *apply* commands proposed in FPSPARQL [5]. In FPSPARQL, writing path queries and generating regular expression can be complex and requires being familiar with FPSPARQL/SPARQL syntax. In this paper, we extend FPSPARQL (TFP-SPQARL: timed FPSPARQL) with *discover* statement which enables case analysts to apply their knowledge (i.e. extract information about facts and the relationship among them) on the AEM graphs in an easy way. This statement has the following syntax:

```
discover.[evolutionOf(artifact1,artifact2) |
          derivationOf(artifact) |
          timeseriesOf(artifact|actor)];
filter( what(type),
        how(action),
        who(actor),
        where(location),
        which(system),
        when(t1,t2,t3,t4));
where{
 #define variables such as artifact, actor, and location.
}
```

Table 5.1: TFP-SPQARL time semantics.

Time Semantic	Time Range
in, on, at, during	[t,t,t,t]
since	[t,t,?,?]
after	[t,?,?,?]
before	[?,?,?,t]
till, until, by	[?,?,t,t]
between	[t,?,?,t]

This statement can be used for discovering evolution of an artifact (using *evolutionOf* construct), derivation of an artifact (using *derivationOf* construct), and timeseries of artifacts/actors (using *timeseriesOf* construct). The *filter* statement restrict the result to those activities for which the filter expression evaluates to true. Variables such as artifact (e.g., version/artifact), type (e.g., lifecycle or archiving), action (e.g., creation, use, or storage), actor, location (e.g., organization), and system will be defined in *where* statement.

In order to support temporal aspects of the queries, we introduce the special construct, $when(t1, t2, t3, t4)$, which is used to represent the fact (e.g. activity) to be in a specific time interval $[t1, t2, t3, t4]$. Table 5.1 represents the time-semantics that we support in TFP-SPARQL queries. A fact may have no temporal duration, e.g. an activity, or may have temporal duration, e.g., an activity-path. Details on time semantics and sample queries can be found in [6]. Following we will introduce derivation, evolution, and timeseries queries.

Evolution Queries. In order to query the evolution of an artifact, case analysts should be able to discover activity paths among entities in AEM graphs. In particular, for querying the evolution of an AEM entity En, all activity-paths on top of En ancestors should be discovered. For example, considering the motivating scenario, Adam (a case analyst) is interested to see how version v_3 of patient history evolved from version v_2 (see Figure 3.1-D). Following is the sample TFP-SPQARL query for this example.

```
discover.evolutionOf(?artifact1,?artifact2);
where{
 ?artifact1 @id v2.
 ?artifact2 @id v3.
}
```

In this example, *evolutionOf* statement is used to represent the evolution of version v_3 (i.e., variable *?artifact2*) from version v_2 (i.e., variable *?artifact1*). Note that, if Adam would be interested to see the whole evolution of version v_3, he didn't need to specify the first parameter, e.g. "evolutionOf(,?artifact2)". In the above example, attributes of variables $?artifact1$ and $?artifact2$ can be defined in the *where* clause. Considering Figure 4.1-A, the result of this query will be a set of paths between versions v_2 and v_3, and can be stored in an activity-path (Figure 4.1-B). This query will automatically be translated to the following FPSPARQL query:

```
pEconstruct v3-v2-evolution-edge
(?startNode, ?endNode, RE:'?edge1 (?artifact ?edge)+ ?edge2') ?evolution
where {
 ?evolution @direction EtoS.
 ?evolution @type activity-edge.
```

```
 ?evolution @label v3-v2-evolution.
 ?evolution @id 'v3v2evl'.
 ?evolution @description 'version evolution'.
 ?startNode @isA entityNode.
 ?startNode @id v2.
 ?endNode @isA entityNode.
 ?endNode @id v3.
 ?artifact @isA entityNode.
 ?edge @isA activityEdge.
 ?edge1 @isA activityEdge.
 ?edge2 @isA activityEdge.
}
```

To construct a path-edge, we introduce the *pEconstruct* command (see Section 5.3). This command is used to discover transitive relationships between two entities and store it under a path-edge name. Variable ?*evolution* represents the path-edge to be constructed, i.e. 'v3-v2-evolution-edge'. Attribute 'direction' is used to define the direction of the activity-path to be constructed, where: i) *StoE*, will construct a directed edge from starting node to the ending node; and ii) *EtoS*, will construct a directed edge from ending node to the starting node. Attribute 'label' shows the label of this implicit edge in the graph. Attributes 'type' and 'id' are used to define the type and ID of the constructed edge respectively. Variables ?*startNode* and ?*endNode* defined to show the starting node and ending node. The regular expression '?edge1 (?artifact ?edge)+ ?edge2' defined to find all activity paths between v_2 and v_3.

`Discussion. [when/where/who/which queries]` Adam can use the *filter* statement to answer to specific evolution questions: (i) *when queries*: what happens to the artifact during the first three weeks that they are received?; (ii) *where queries*: what happens to the artifact in radiology clinic?; (iii) *who queries*: who (which roles) work on the artifact?; and (iv) *which queries*: what happens to the artifact in the Wiki system? For example, Adam is interested to see who work on patient history document during November 2012 in radiology clinic. Following is the sample TFP-SPQARL query for this example.

```
select ?actor
discover.evolutionOf( ,?artifact);
filter( who(?actor),
        where(?location),
        when("11/1/2011 @ 0:0:0",?,?,"12/1/2011 @ 0:0:0"));
where{
 ?artifact @id 'X14-med-doc'.
 ?location @name 'radiology'.
 #timestamp: M/D/Y @ h:m:s
}
```

In this example, *filter* statement is used to restrict the result to those activities, happened November 2011 in radiology clinic. The *select* statement is used to specify the actor(s) who work on patient history document.

Derivation Queries. In AEM graphs, derivation of an entity En can be defined as all entities which En found to have been derived from them. In particular, if entity En_b is reachable from entity En_a in the graph, we say that En_a is an ancestor of En_b. The result of derivation

query for an AEM entity will be a set of AEM entities, i.e., its ancestors. For example, considering the motivating scenario, Adam is interested to query the derivation of version v_3 of patient history (see Figure 3.1-C). Following is the sample TFP-SPQARL query for this example.

```
discover.derivationOf(?artifact);
where{
 ?artifact @id v3.
}
```

In this example, *derivationOf* statement is used to represent the derivations of version v_3 of patient history. Attributes of variable $?artifact$ can be defined in the *where* clause. Considering Figure 3.1-C, the result for this query will be the set "{MDT-report, BCSC-result, RC-result, PC-result}". This query will automatically translated to the following FPSPARQL query:

```
select ?startNode
pconstruct derivation_v3 (?startNode,?endNode,RE:'?edge (?node ?edge)*')
where { ?startNode @isA entityNode.
 ?endNode @isA entityNode.
 ?endNode @type artifactVersion.
 ?endNode @id v3.
 ?node @isA entityNode.
 ?edge @isA edge.
}
```

In [5], we introduced the *pconstruct* statement to discover paths: i) between two nodes; ii) starting from a specific node and ending to a set of nodes; and iii) starting from a set of nodes and ending to a specific node. In this example, we used *pconstruct* statement to discover paths between set of starting nodes (ancestors) to a specific ending node (version v_2 of patient history). The result of this query will be set of artifacts/versions (variable *?startNode* in *select* statement) reachable from version v_2 of patient history document. Details about *pconstruct* statement and how to specify regular expressions (e.g. "RE:?startNode,?endNode,?edge (?node ?edge)*") can be find in [5].

`Discussion.` Adam can use the *filter* statement to answer specific derivation questions. For example, he can find specific artifacts which v_2 was derived from them: (i) in radiology clinic (using *where* statement); (ii) between the time periods τ_1 and τ_2 (using *when(τ_1,?,?,τ_2)* statement); or (iii) in a specific system (using *which* statement). For example, Adam is interested to find all ancestors of version v_3 of patient history (see Figure 3.1-C) generated in radiology clinic between February and March 2011. Following is the sample TFP-SPQARL query for this example.

```
discover.derivationOf(?artifact);
filter( where(?location),
        when("2/1/2011 @ 0:0:0",?,?,"3/1/2011 @ 0:0:0"));
where{
 ?artifact @id v3.
 ?location @name 'radiology'.
 #timestamp: M/D/Y @ h:m:s
}
```

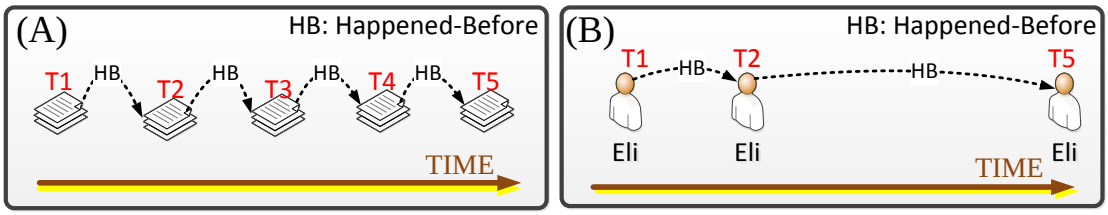

Figure 5.1: Sample timeseries for: (A) patient history document between τ_1 and τ_5; and (B) an actor, i.e. Eli, acting on patient history between τ_1 and τ_5.

In this example, *filter* statement is used to restrict the result to those activities, happened between February and March 2011 in radiology clinic.

Timeseries Queries. In analyzing AEM graphs, it is important to understand the timeseries, i.e. a sequence of data points spaced at uniform time intervals, of artifacts and actors over periods of time. To achieve this, we introduce *timeseriesOf* statement. The result of artifact/actor timeseries queries will be a set of artifact/actor over time, where each artifact/actor connected through a happened-before edge. For example, Adam is interested in Alex's activities on the patient history document between timestamps τ_1 and τ_5. Following is the sample TFP-SPQARL query for this example.

```
discover.timeseriesOf(?actor);
filter( when("T1",?,?,"T5") );
where{
 ?actor @id Eli-id.
}
```

In this example, *timeseriesOf* statement is used to represent the timeseries of Eli (i.e., variable *?actor*). Attributes of variable ?*actor* can be defined in the *where* clause. Figure 5.1-B represents the timeseries of Eli for activities she did on top of patient history document. Figure 5.1-A represents time series of patient history document between τ_1 and τ_5. Similar to evolution and derivation queries, *timeseriesOf* statement can be used with/without *filter* statement, where *filter* statement can be used to answer specific timeseries questions.

5.3 Constructing Timed Folders and Activity-Paths

Section 5.2 represented how to query AEM graphs in an easy way without having knowledge about FPSPARQL syntax. In this section, we introduce FPSPARQL queries for constructing timed-folders and activity-paths. To construct a *timed folder* node, we use FPSPARQL's *fconstruct* statement. We extend this statement with "?folder @timed true" pattern. Setting the value of attribute *timed* to *true* for the folder, will assign a monitoring code snippet to this folder. The code snippet is responsible for updating the folder content over periods of time. To construct a path-edge, we introduce the *pEconstruct* command. This command is used to discover transitive relationships between two entities and store it under a activity-path name.

Constructing Timed Folder Nodes. To construct a *timed* folder node, we extend FPSPARQL's *fconstruct* statement. This command is used to group a set of related entities or folders. The syntax for a basic construction query of a timed folder node is given as follows:

```
fconstruct <Folder_Node Name> as ?folder
[select ?var1 ?var2 ... | (Folder1, Folder2,...)]
where {
 ?folder @timed true.
 #(other patterns)
}
```

A query can be used to define a new timed 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 folder nodes. A set of user-defined attributes for this folder can be defined in the *where* statement. Setting the value of attribute *timed* to *true* for the folder, will assign an intelligent agent to this folder. The intelligent agent is responsible for updating the folder content over periods of time. For example, considering Figure 3.1-C, a timed folder can be constructed to represent patient history artifact. New versions (and activities on top of it) can be added to this folder, automatically, over periods of time. Following is a sample FPSPARQL query for this example.

```
fconstruct X14-patient-history as ?med-doc
select ?version
where {
 ?med-doc @timed true.
 ?med-doc @type artifact.
 ?med-doc @id 'X14-Artifact'.
 ?med-doc @description 'history for patient #X14'.
 ?version @isA entityNode.
 ?version @patient-ID X14.
}
```

In this example, variable $?med-doc$ represents the folder node to be constructed, i.e. 'X14-patient-history'. Setting the attribute *timed* to *true* for the folder, will assign an intelligent agent to this folder. This folder is of type 'artifact'. The attribute 'description' used to describe the folder. Variable $?version$ in a AEM entity and represents the patient history versions to be collected. Attribute 'patient-ID' indicated that the version is related to the patient history of patient number X14.

Querying Timed Folder Nodes. Using the *apply* statement in FPSPARQL, it is possible to apply queries to constructed timed folder nodes. For example, consider a user who is interested to retrieve information about $X14-patient-history$ folder evolution between timetamps τ_2 and τ_7. Following is the FPSPARQL query for this example.

```
(X14-patient-history)
apply (
 select ?a
 where {
   ?a @isA entityNode.
   ?a @timestamp ?ts.
   filter( Timesemantic(?ts,[t2,?,?,t7]) ).
 }
)
```

In this example the query applied to the constructed timed folder node 'X14-patient-history'. Variable $?a$ represents all members (i.e. artifact versions) of the folder node whose (creation) timestamp $?ts$ falls between time τ_2 and τ_7. The *when* statement (i.e. $when(t1, t2, t3, t4)$) in TFP-SPARQL will be translated to $timesemantic(fact, [t1, t2, t3, t4])$ in FPSPARQL which is used to represent the $fact$ to be in a specific time interval $[t1, t2, t3, t4]$. In this example, *timesemantic* statement defines version creation timestamps, i.e. variable $?ts$, to be between timestamps τ_2 and τ_7. Details about FPSPARQL time semantics can be found in [6].

Constructing Path-Edges. Discovering paths through AEM graphs form the basis of many evolution, derivation, and timeseries queries. To construct a path-edge, we introduce the *pEconstruct* command. This command is used to discover transitive relationships between two entities and store it under a path-edge name. The syntax for a basic construction query of a path-edge is given as follows:

```
pEconstruct <Path_Edge Name>
(StartNode,EndNode,RegularExpression) as ?pathNode
where {
 ?pathNode @direction StoE/EtoS.
 #(other patterns)
}
```

A regular expression can be used to define a transitive relationship between two entities [5]. Attributes of starting node, ending node, and regular expression's alphabets (i.e. graph nodes and edges) can be defined in the *where* statement. If the regular expression would not be considered in *pEconstruct* command, all the paths between starting and ending node will be discovered. Moreover, setting the value of attribute *direction* to: i) $StoE$, will construct an edge from starting node $StartNode$ to the ending node $EndNode$; and ii) $EtoS$, will construct an edge from ending node $EndNode$ to the starting node $StartNode$. For example, considering Figure 3.1-D, Adam can be interested in discovering all the activities happening between versions v_2 and v_3 of patient history artifact, and replace them as a derivation link, e.g. 'was-derived-from' edge, from v_3 and v_2 (see Figure 3.1-C). Following is a sample FPSPARQL query for this example.

```
pEconstruct v3-v2-ancestry-edge (?startNode, ?endNode,
            RE:'?edge1 (?artifact ?edge)+ ?edge2') ?derivation
where {
 ?derivation @direction EtoS.
 ?derivation @type ancestry.
 ?derivation @label wasDerivedFrom.
 ?derivation @id 'v3v2drv'.
 ?derivation @description 'version derivation'.
  #specifying starting and ending node.
 ?startNode @isA entityNode.
 ?startNode @id X14-v2.
 ?endNode @isA entityNode.
 ?endNode @id X14-v3.
  #defining RegExp variables
 ?artifact @isA entityNode.
 ?edge @isA activityEdge.
 ?edge1 @isA activityEdge.
```

```
 ?edge1 @type archiving.
 ?edge2 @isA activityEdge.
 ?edge2 @type archiving.
 ?edge1 @action ?action.
 ?edge2 @action ?action.
 FILTER (?action='transfer' || ?action='storage').
}
```

In this example, variable ?*derivation* represents the path-edge to be constructed, i.e. 'v3-v2-ancestry'. Attribute 'direction' used to define the direction of the edge. Attribute 'label' shows the label of this implicit edge in the graph. Attributes 'type' and 'id' used to define the type and ID of the constructed edge respectively. Variable ?*RegExp* used to define the regular expression. Variables ?*startNode* and ?*endNode* defined to show the starting node and ending node. The regular expression '?edge1 (?artifact ?edge)+ ?edge2' defined to find all paths between v_2 and v_3 starting and ending with an activity edge typed as archiving activity and acting to transfer/storage an artifact version.

`Discussion.` In order to discover paths through AEM graph and apply further operations on the discovered path(s) we use *pconstruct* and *apply* commands proposed in FPSPARQL [5]. Note that *pconstruct* command is different from *pEconstruct* command proposed in this paper. In particular *pEconstruct* discover paths between two nodes, group them, and add them as an implicit edge between starting and ending node in AEM graphs. But, *pconstruct* command used to: i) discover paths between two nodes; ii) discover paths (all paths or path having a specific pattern defined by a regular expression) starting from a specific node, but not ending to a specific node; iii) discover paths (all paths or path having a specific pattern defined by a regular expression) ending to a specific node, but not starting from a specific node; and (iv) discover frequent patterns, i.e. paths having a specific pattern defined by a regular expression and not having a specific starting/ending node. The result for *pconstruct* command will be a set of paths which can be stored in a path node (i.e. different from a path-edge). Details about path nodes can be find in [5]. In this paper, we extend path nodes to *timed* path nodes, i.e., defined as a timed container for a set of related entities which are connected through *transitive* relationships. For the sake of simplicity and to prevent confusion with the concept of path-edge, we didn't introduce timed path nodes in this paper. Details about timed path nodes can be find in [6].

6 Architecture, Implementation and Experiments

6.1 Architecture

Figure 6.1 illustrates TFP-SPARQL graph processing architecture which consists of following components:

1. *Graph Loader*: Input graph can be in the form of RDF, N3 (or Notation3, is a W3C standard and shorthand non-XML serialization of RDF models), or XML. We developed a workload-independent physical design by developing a *loader* algorithm. This algorithm is responsible for: (i) validating the input graph; (ii) generating the relational representation of triple store, for manipulating and querying entities, folders, and paths; and (iii) generating powerful indexing mechanisms.

2. *Data Mapping Layer*: is responsible for creating data element mappings between semantic web technology (i.e. Resource Description Framework) and relational database schema.

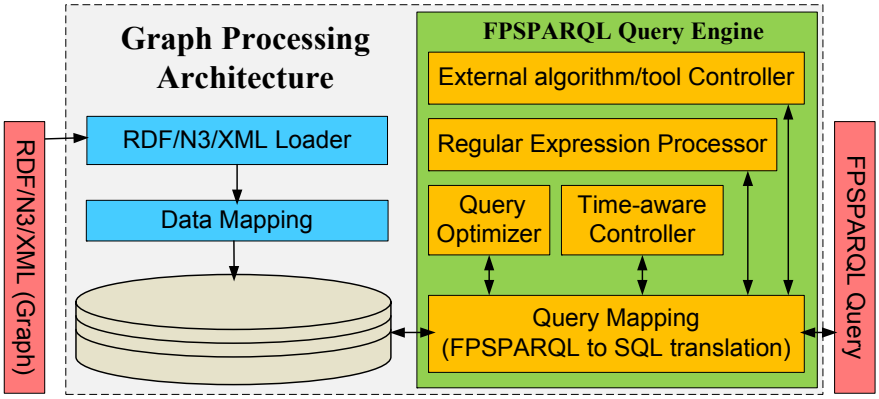

Figure 6.1: FPSPARQL graph processing architecture.

3. *Query Mapping Layer*: is consist of a FPSPARQL parser (for parsing FPSPARQL queries based upon the syntax of FPSPARQL) and a schema-independent FPSPARQL-to-SQL translation algorithm. This algorithm consists of:

   - *SPARQL-to-SQL Translation Algorithm.* 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 [11]. This algorithm supports *Aggregate* queries and *Keyword Search* queries.
   - *Folder Node Construction and Querying.* We use the relational representation of triple RDF store, to store, manipulate, and query folder nodes.
   - *Path Node Construction and Querying.* To describe constraints on the path nodes, we reused expressions proposed in CSPARQL [3].

4. *Regular Expression Processor*: is responsible for parsing the described patterns through the nodes and edges in the graph. We developed a regular expression processor which supports optional elements (?), loops (+,*), alternation (—), and grouping ((...)).

5. *External Algorithm/Tool Controller*: is responsible for supporting applying external graph reachability algorithm or mining tools to the graph.

6. *Time-aware Controller*: is responsible for creating a monitoring code snippet and allocate it to a timed folder/path node in order to monitor its evolution and update its content. We enable users to set an AEM query as: (i) pull query, where a time-tracker will be assigned to this query. Time-tracker will trigger the start of the querying process at specific user-defined intervals; or (ii) push query, where a database trigger will be assigned to the entities in the query result. Future changes applied to these entities and their relationships will result in re-executing the query. Users can initialize an intelligent agent in order to allocate it to a timed folder/path node and set its time interval or assign it to a database trigger.

7. *Query Optimizer*: To optimize the performance of queries, we developed four optimization techniques proposed in [10, 32, 11]: (i) selection of queries with specified varying degrees of structure and spanning keyword queries; (ii) selection of the smallest table to query

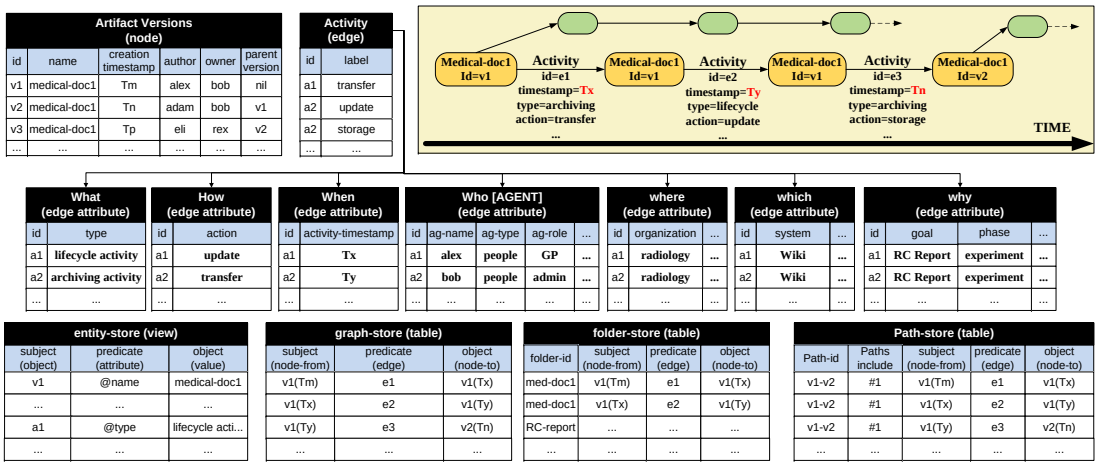

Figure 6.2: Physical layer for storing a sample graph including a sample AEM graph, and tables to store AEM entities and relationships.

based on the type information of an instance; (iii) elimination of redundancies in basic graph pattern based on the semantics of the patterns and database schema; and (iv) create separate tables (property tables) for subjects that tend to have common properties to reduce the self-join problem.

6.2 Implementation

We have implemented a graph processing engine, i.e. FPSPARQL, and the full details of our data model and query engine are presented in [5, 7]. Of the many data models in the literature, we model graphs based on a RDF data representation. The RDF data model is similar to classic conceptual modeling approaches, e.g., class diagrams. RDF is based upon the idea of making statements about resources in the form of subject-predicate-object expressions, i.e. triples. The subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. The simplest way to store a set of RDF statements is to use a relational database with a single table that includes columns for subject, property and object, i.e. triplestore. While simple, this schema quickly hits scalability limitations. To avoid this we developed a relational RDF store including its three classification approaches [32]: vertical (triple), property (n-ary), and horizontal (binary).

Figure 6.2 represents a sample AEM graph and tables to store the graph including: (a) artifact versions, to store AEM entities; (b) activity, to store the relationships between entities. Relationship's attributes can be stored in what, how, when, who, where, which, and why tables; (c) entity store, which is a view on top of graph entities and relationships. This triple store stores the node/edge ID in the *subject* column, node/edge attribute in the *predicate* column, and node/edge value in the *object* column; (d) graph store, which contains directed links between graph entities. This triple store stores the starting node ID in the *subject* column, edge ID in the *predicate* column, and ending node ID in the *object* column; (e) timed folder store, which stores related entities and relationships among them in a triple store. The 'folder-id' column added to this triple store for identifying folders; and (f) timed path store, which stores activity edges between two entities in the graph. The 'path-include' column identifies each path, and the 'path-id' column identifies set of paths considered as an activity-path.

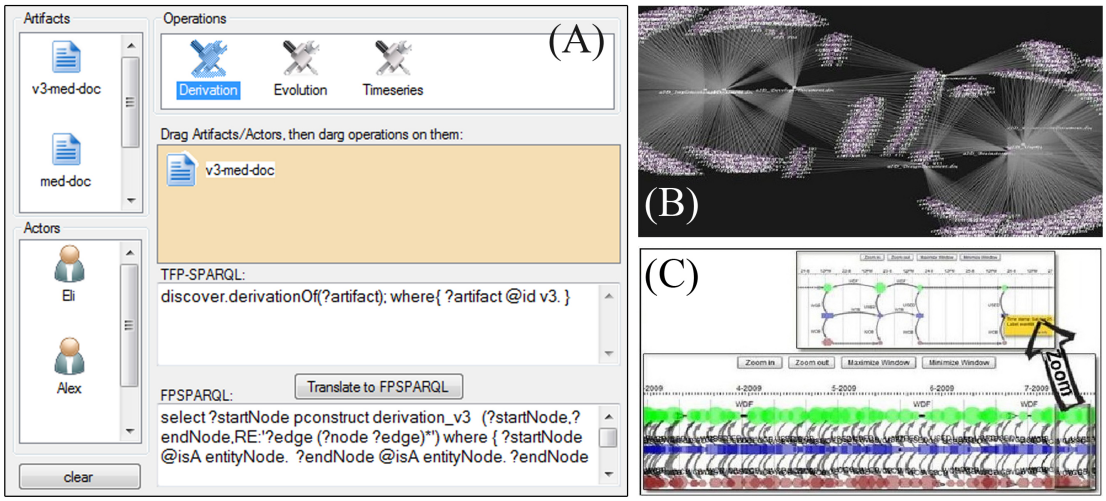

Figure 6.3: Screenshots of front end tool: (a) Query assistant tool; (b) graph visualization tool: to visualize static (non-temporal) graphs such OPM (see Section 6.4); and (c) temporal graph visualization tool: to visualize AEM graphs;

We have implemented a front-end tool to assist case analysts in two steps:

Step1: [Query Assistant] We provided users with a query assistant tool to generate AEM queries in an easy way. Users can easily drag entities (i.e. artifacts and actors) in the activity panel. Then they can drag the operation (i.e. evolution, derivation, or timeseries) on top of selected entity. The TFP-SPARQL template (e.g., for evolution, derivation, and timeseries queries) will be automatically generated. Moreover it is possible to generate the FPSPARQL query by clicking on "Translate to FPSPARQL" button. Also, users can use the tool to generate the regular expressions and other path queries they are interested in. Figure 6.3-A illustrates a screenshot of this tool while generating the derivation query in Section 5.2.

Step2: [Visualizing] We provided users with a graph visualization tool for the exploration of graphs and query results (see Figures 6.3-B and 6.3-C). For the AEM graph exploration, we provide users with a timeline like interface (see Figure 6.3-C) with facilities such as zooming in and out.

6.3 Datasets

We carried out the experiments on two time-sensitive datasets:

e-Enterprise Course. This scenario, is built on our experience on managing an online project-based course "e-Enterprise Projects" [1]. In this scenario, each project can be considered as a case process, where various case workers (e.g. students, mentors and lecturers) are involved. As an example, in the 2^{nd} semester of 2009 we had 66 people (60 students + 5 project mentors + 1 lecturer) involved in course activities. During this semester, fifteen projects (i.e. case instances) were defined, where each case handled by group of four students and one mentor.

[1] http://www.cse.unsw.edu.au/∼cs9323

Each mentor supervised 3 projects. The development process of each project went through a sequence of pre-defined phases: brainstorming, requirements analysis, design phase, prototype implementation, testing, and final product delivery. For each phase various artifacts can be created, e.g. brainstorming documents and records, and each artifact version can be derived from various sources, e.g. IEEE or other templates, and can be accessed/modified by different case workers over periods of time.

In order to document the evolution of artifacts, the activities of each project have been documented through a web-based project management system which was equipped with many back-end modules such as: (a) Message board: to exchange message and open discussion topics between the project members; (b) Wiki system: which is used to collaboratively edit documents related to the activities of projects; (c) Blogging system: where each user has their own blog to edit their own posts; (d) File sharing system: where project members can share access to different files and documents; and (e) SVN repository: to synchronize the editing of the projects source codes. This dataset contains 104,050 events.

SCM (Supply Chain Management). This dataset is the interaction log of a supply chain service, developed based on the supply chain management scenario provided by WS-I (the Web Service Interoperability organization). SCM dataset contains 4,050 events. We applied a preprocessing phase to adapt these dataset to a case scenario. Details about this dataset can be found in [27].

6.4 Evaluation

We have compared our approach with that of querying Open provenance model (OPM), see Section 7.2, and evaluated the performance and the query results quality using the proposed datasets. Moreover, the performance of FPSPARQL query engine has been evaluated in [5].

Performance. We evaluated the performance of evolution, derivation, and timeseries queries using *execution time* metric. To evaluate the performance of queries, we provided 10 evolution queries, 10 derivation queries, and 10 timeseries queries. These queries were generated by domain experts who were familiar with the proposed datasets. For each query, we generated an equivalent query to be applied to the AEM graphs as well as the OPM graphs for each dataset. As a result, a set of historical paths for each query were discovered. Figure 6.4 shows the average execution time for applying these queries to the AEM graph and the equivalent OPM graph generated from each dataset. As illustrated in Figure 6.4 we divided each dataset into regular number of events, then generated AEM and OPM graph for different sizes of datasets, and finally ran the experiment for different sizes of AEM and OPM graphs. The evaluation shows the viability and efficiency of our approach.

Quality. The quality of results is assessed using classical *precision* metric which is defined as the percentage of discovered results that are actually interesting. For evaluating the interestingness of the result, we asked domain experts who had the most accurate knowledge about the datasets and the related process to analyze discovered paths and identify what they considered relevant and interesting. We evaluated the number of discovered paths for all the queries (in performance evaluation) and the number of relevant paths chosen by domain experts. As a result of applying queries to AEM graphs generated from all the datasets, 79 paths were discovered and examined by domain experts, and 78 paths (precision=98.7%) considered relevant. And as a result of applying queries to OPM graphs generated from all the datasets, 243 paths discovered, examined by domain experts, and 91 paths (precision=37.4%) considered relevant.

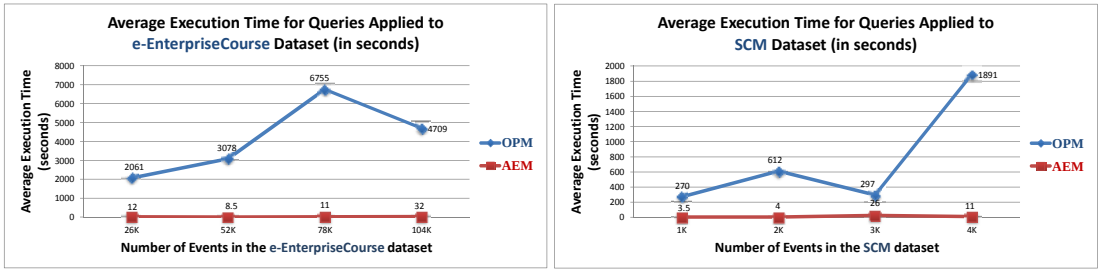

Figure 6.4: The query performance evaluation results, illustrating the average execution time for applying evolution, derivation, and timeseries queries on AEM and OPM graphs generated from (a) e-Enterprise course dataset; and (b) SCM dataset.

Discussion. Evaluation shows that path queries applied to OPM graphs resulted in many irrelevant paths. Moreover, we discovered many cycles in the results of path queries applied to OPM graphs. To eliminate these cycles, we applied the cycle elimination techniques proposed in [2]. To increase the performance of queries, we implemented an interface to support various graph reachability algorithms [2] such as all-pairs shortest path, transitive closure, GRIPP, tree cover, chain cover, path-tree cover, and Sketch. In general, there are two types of graph reachability algorithms [2]: (i) algorithms traversing from starting vertex to ending vertex using breadth-first or depth-first search over the graph, and (ii) 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 imposes a time complexity of $O(n + m)$ and the second approach imposes a space complexity of $O(n^2)$. In both cases, path queries applied to OPM graphs maximized the consumption of memory and processor and resulted in many irrelevant paths and cycles in the query result.

7 Related Work

We study the related work into three main areas: knowledge intensive processes, provenance, and modeling/querying temporal graphs:

7.1 knowledge-intensive processes

Over the past decade, modeling and querying techniques for knowledge-intensive tasks received high interest in the research community. Some of existing approaches [1, 30] for modeling ad-hoc processes focused on supporting ad-hoc workflows through user guidance. Some other approaches [14, 15, 33] focus on intelligent user assistance to guide end users during ad-hoc process execution by giving recommendations on possible next steps. All these approaches focused on user activities and guide users based on analyzing past process executions.

Modeling and querying document-driven processes [23, 37, 16] represent another flavor of knowledge-intensive processes. In [23, 37], a document-driven framework, proposed to model business process management system through monitoring the lifecycle of a document. Dorn et.al. [16], presented a self-learning mechanism for determining document types in people-driven ad-hoc processes through combining process information and document alignment. In these approaches, the document structure is predefined or they presume that the execution of the

business processes is achieved through a business process management system (e.g BPEL) or a workflow process. Another related line of work is artifact-centric workflows [8] where the process model is defined in terms of the lifecycle of the documents. In our model, actors, activities and artifacts are first class citizens, and the evolution of the activities on artifacts over time is the main focus. In our approach, understanding the knowledge-intensive processes and their execution through exploration and querying artifact evolution logs is a major goal: analyzing the set of activities on artifacts helps in understanding the process. Proposed query language, i.e. TFP-SPARQL, provides an explorative approach for querying and understanding of artifact evolutions over time.

7.2 Provenance

provenance refers to the documented history of an object (e.g. documents, data, and resources) or the documentation of processes in an object's lifecycle [12]. Many provenance models [12, 17, 26, 34] have been presented in a number of domains (e.g. databases, scientific workflows and the Semantic Web), motivated by notions such as influence, dependence, and causality. The existing provenance models, e.g., the open provenance model (OPM) [26], treat time as a second class citizen (i.e. as an optional annotation of the data) which will result in loosing semantics of time and makes querying and analyzing provenance data for a particular point in time inefficient and sometimes inaccessible.

Discovering historical paths through provenance graphs forms the basis of many provenance query languages. In ProQL [21] a query takes a provenance graph as an input, matches parts of the input graph according to path expression and returns a set of paths as the result of the query. PQL [19] uses a semi-structured data model for handling provenance and extends Lorel query language for traversing and querying provenance graph. NetTrails [38] proposes a declarative platform for interactively querying network provenance in a distributed system in which query execution performs a traversal of the provenance graph. RDFProv [10] is an optimized framework for scientific workflow provenance querying and management. Missie et. al. [24] present a provenance model and query language for collection-oriented workflow systems. They emphasize on querying the provenance of collection of activities. These related activities are not considered as first class objects in the proposed graph. Moreover, they do not support modeling, querying and analyzing the evolution of group of related entities over time. These approaches lead to an increased query complexity in analyzing the the evolution of artifacts over time.

7.3 Modeling/Querying Temporal Graphs

In recent years, a plethora of work [20, 22, 31] has focused on temporal graphs to model evolving, time-varying, and dynamic networks of data. They capture a snapshot for various states of the graph over time. For example, Ren et. al. [31] propose a historical graph-structure to maintain analytical processing on such evolving graphs. Moreover, authors in [22, 31] propose approaches to transform an existing graph into a similar temporal graph to discover and describe the relationship between the internal object states. In our approach, we propose a temporal artifact evolution model to capture the evolution of time-sensitive data where this data can be modeled as temporal graph. We also provide abstractions and efficient mechanisms for time-aware querying of AEM graphs.

Approaches to querying graphs (e.g. [4, 18, 28, 36]) provide temporal extensions of existing graph models and languages. Tappolet et. al. [36] provide temporal semantics for RDF graphs. For querying temporal graphs, they propose τ-SPARQL. Grandi [18] presents another temporal

extension for SPARQL, i.e. T-SPARQL, aimed at embedding several features of TSQL2 [25] (temporal extension of SQL). SPARQL-ST [28] and EP-SPARQL [4] are extensions of SPARQL supporting real time detection of temporal complex patterns in stream reasoning. Our work differs from these approaches as we enable registering a time-sensitive query once, propose timed abstractions (i.e. folders and paths) to store the result of such queries, and enable analyzing the evolution of such timed abstractions over time. Moreover, we extend FPSPARQL [5], our previous work, to support temporal queries and monitor the result of such queries over time.

8 Conclusion and Future Work

In this paper, we have presented an artifact-centric activity model (i.e. AEM: Artifact Evolution Model) for knowledge intensive processes. Two concepts of timed folders and activity-paths have been introduced, which help in analyzing AEM graphs. Folders enable grouping related entities and paths help in analyzing the history of entities in time. Timed folders and activity-paths show their evolution for the time period they represent. We have extended our previous work, FPSPARQL [5], which is a query language for analyzing business processes execution, to query and analyze AEM graphs. To evaluate the viability and efficiency of the proposed framework, we have compared our approach with that of querying OPM models where time is considered as annotation. We have conducted experiments over realworld datasets. The results of evaluation show the viability and efficiency of our approach. A front-end tool has been provided to facilitate the exploration and visualization of AEM graphs and assisting users with generating evolution, derivation, and timeseries queries. As future work, we plan to design a visual query interface to support users in expressing their queries over the conceptual representation of the AEM graph in an easy way. Discovering the AEM model from existing unstructured artifact data in the enterprise is another interesting line of future work.

Bibliography

[1] Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Facilitating flexibility and dynamic exception handling in workflows through worklets. In *CAiSE Short Paper Proceedings*, 2005.

[2] Charu C. Aggarwal and Haixun Wang, editors. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer, 2010.

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

[4] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: a unified language for event processing and stream reasoning. In *WWW*, pages 635–644, 2011.

[5] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Hamid R. Motahari Nezhad, and Sherif Sakr. A query language for analyzing business processes execution. In *BPM*, pages 281–297, 2011.

[6] Seyed-Mehdi-Reza Beheshti, Hamid Reza Motahari-Nezhad, and Boualem Benatallah. Temporal Provenance Model (TPM): Model and query language. Unsw-cse-tr-1116, University of New South Wales, 2010.

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

[8] K. Bhattacharya, C. Evren Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *BPM*, pages 288–304, 2007.

[9] Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani M. Thuraisingham. A language for provenance access control. In *CODASPY*, pages 133–144, 2011.

[10] 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.

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

[12] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[13] Richard Cyganiak. A relational algebra for SPARQL. *HP-Labs Technical Report, HPL-2005-170*, 2005.

[14] Christoph Dorn, Thomas Burkhart, Dirk Werth, and Schahram Dustdar. Self-adjusting recommendations for people-driven ad-hoc processes. In *BPM*, pages 327–342, 2010.

[15] Christoph Dorn and Schahram Dustdar. Supporting dynamic, people-driven processes through self-learning of message flows. In *CAiSE*, pages 657–671, 2011.

[16] Christoph Dorn, César A. Marín, Nikolay Mehandjiev, and Schahram Dustdar. Self-learning predictor aggregation for the evolution of people-driven ad-hoc processes. In *BPM*, pages 215–230, 2011.

[17] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.

[18] F. Grandi. T-sparql: a tsql2-like temporal query language for rdf. In *International Workshop on Querying Graph Structured Data*, pages 21–30, 2010.

[19] David A. Holland, Uri Braun, Diana Maclean, Kiran-Kumar Muniswamy-Reddy, and Margo Seltzer. Choosing a Data Model and Query Language for Provenance. In *IPAW*, 2008.

[20] Petter Holme and Jari Saramäki. Temporal networks. *CoRR*, abs/1108.1780, 2011.

[21] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *SIGMOD Conference*, pages 951–962, 2010.

[22] Vassilis Kostakos. Temporal graph. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.

[23] Jong-Yih Kuo. A document-driven agent-based approach for business processes management. *Information & Software Technology*, 46(6):373–382, 2004.

[24] Paolo Missier, Norman W. Paton, and Khalid Belhajjame. Fine-grained and efficient lineage querying of collection-based workflow provenance. In *EDBT*, pages 299–310, 2010.

[25] Theophano Mitsa. *Temporal Data Mining*. Chapman & Hall/CRC, 1st edition, 2010.

[26] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul T. Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric G. Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Comp. Syst.*, 27(6):743–756, 2011.

[27] H.R. Motahari-Nezhad, R. Saint-Paul, F. Casati, and B. Benatallah. Event correlation for process discovery from web service interaction logs. *VLDB J.*, 20(3):417–444, 2011.

[28] Matthew Perry, Prateek Jain, and Amit P. Sheth. SPARQL-ST: Extending sparql to support spatiotemporal queries. In *Geospatial Semantics and the Semantic Web*, pages 61–86, 2011.

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

[30] Hajo A. Reijers, J. H. M. Rigter, and Wil M. P. van der Aalst. The case handling case. *Int. J. Cooperative Inf. Syst.*, 12(3):365–391, 2003.

[31] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 4(11):726–737, 2011.

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

[33] Helen Schonenberg, Barbara Weber, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Supporting flexible processes through recommendations based on history. In *BPM*, pages 51–66, 2008.

[34] Jianqiang Shen, Erin Fitzhenry, and Thomas G. Dietterich. Discovering frequent work procedures from resource connections. In *IUI*, pages 277–286, 2009.

[35] Keith D. Swenson, Nathaniel Palmer, Bruce Silver, Layna Fischer, and Thomas Koulopoulos. *Taming the Unpredictable Real World Adaptive Case Management: Case Studies and Practical Guidance*. Future Strategies Inc, 2011.

[36] Jonas Tappolet and Abraham Bernstein. Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In *ESWC*, pages 308–322, 2009.

[37] Jianrui Wang and Akhil Kumar. A framework for document-driven workflow systems. In *Business Process Management*, pages 285–301, 2005.

[38] Wenchao Zhou, Qiong Fei, Shengzhi Sun, Tao Tao, Andreas Haeberlen, Zachary G. Ives, Boon Thau Loo, and Micah Sherr. Nettrails: a declarative platform for maintaining and querying provenance in distributed systems. In *SIGMOD Conference*, pages 1323–1326, 2011.