A Framework and a Language for Analyzing Cross-Cutting Aspects in Ad-hoc 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-201228
November 2012

THE UNIVERSITY OF
NEW SOUTH WALES

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

Abstract

Ad-hoc processes have flexible underlying process definition. The semi-structured nature of ad-hoc process data requires organizing process entities, people and artifacts, and relationships among them in graphs. The structure of process graphs, describing how the graph is wired, helps in understanding, predicting and optimizing the behavior of dynamic processes. In many cases, however, process artifacts evolve over time, as they pass through the business's operations. Consequently, identifying the interactions among people and artifacts over time becomes challenging and requires analyzing the cross-cutting aspects of process artifacts: process artifacts, like code, has cross-cutting aspects such as versioning and provenance. Analyzing these aspects will expose many hidden interactions among entities in process graphs. We present a framework, simple abstractions and a language for analyzing cross-cutting aspects in ad-hoc processes. 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. The approaches presented in this paper have been implemented on top of FPSPARQL, Folder-Path enabled extension of SPARQL, and experimentally validated on real-world datasets.

1 Introduction

Ad-hoc processes, a special category of processes, have flexible underlying process definition where the control flow between activities cannot be modeled in advance but simply occurs during run time [22, 20, 45]. The semistructured nature of ad-hoc process data requires organizing process entities, people and artifacts, and relationships among them in graphs. The structure of process graphs, describing how the graph is wired, helps in understanding, predicting and optimizing the behavior of dynamic processes. In many cases, however, process artifacts evolve over time, as they pass through the business's operations. Consequently, identifying the interactions among people and artifacts over time becomes challenging and requires analyzing the cross-cutting aspects [31] of process artifacts.

A cross-cutting concern defined as a "universal program behavior, one that is potentially needed in many disparate parts of a program, but is often developed and modeled separately" [23]. In particular, aspect-oriented programming (AOP) [31] paradigm can be applied to the process artifacts in BP execution data, where AOP is used to add support for cross-cutting aspects (e.g., object versioning, event logging, and memory management) to existing code without directly modifying that code [23]. In particular, process artifacts, like code, has cross-cutting aspects such as versioning (what are the various versions of an artifact, during its lifecycle, and how they are related), provenance [16] (what manipulations were performed on the artifact to get it to this point), security (who has access to the artifact over time), and privacy (what actions were performed to protect or release artifact information over time). In this chapter we focus on cross-cutting aspects which are related to the *evolution* of business process artifacts over time, i.e., versioning and provenance. Analyzing these aspects will expose many hidden interactions among entities in process graphs.

As an example, 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. Such processes, almost always involves the collection and presentation of a diverse set of artifacts, where artifacts are developed and changed gradually over a long period of time. Case management [46], also known as case handling, is a common approach to support knowledge-intensive processes.

In order to represent cross-cutting aspects in ad-hoc processes, there is a need to collect meta-data about entities (e.g., artifacts, activities on top of artifacts, and related actors) and relationship among them from various systems/departments over time, where there is no central system to capture such activities at different systems/departments. 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 analyzing cross-cutting aspects in ad-hoc processes. The unique contributions of the paper are as follows:

- We propose a temporal graph model for representing cross-cutting aspects in ad-hoc processes. This model enables supporting timed queries and weaving cross-cutting aspects, e.g., versioning and provenance, around business artifacts to imbues the artifacts with additional semantics that must be observed in constraint and querying ad-hoc processes. In particular, the 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 process graphs.

- 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 [7, 5], a graph query language for analyzing processes execution, for explorative querying and understanding of cross-cutting aspects in ad-hoc processes. We introduce simple templates for querying evolution, derivation, and timeseries of artifacts. We present the evaluation results on the performance and the quality of the results using a number of process event logs.
- 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 case management applications. In Section 4 we introduce a data model for representing cross-cutting aspects in ad-hoc processes. In Section 5 we propose a query language for querying the proposed model. In Section 7 we describe the query engine implementation and evaluation experiments. Finally, we discuss related work in Section 8, before concluding the paper in Section 9.

2 Preliminaries

Definition 1. ['Artifact'] An artifact is 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 is 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 artifact 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 [36].

Definition 3. ['Activity'] An activity is defined as an action performed on or caused by an artifact version. For example, an action can be used to create, read, update, 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 is 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 is 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 ad-hoc processes, artifacts develop and change gradually over time as they pass through the business's operations. 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.

Definition 7. ['Provenance'] Provenance refers to the documented history of an *immutable* object which tracks the steps by which the object was derived [16]. This documentation (often represented as graphs) should include all the information necessary to reproduce a certain piece of data or the process that led to that data [38].

Definition 8. ['Cross-cutting Aspects'] Aspect-Oriented Programming [31] (AOP), has been proposed as a technique in computing for improving separation of concerns in software. In AOP, a concern is a particular set of behaviors needed by a computer program, is modeled as an aspect, and can be as general as database interaction or as specific as performing a calculation. In this context, cross-cutting concerns are aspects of a program which affect other concerns, e.g., object versioning, event logging, and memory management. Process artifacts, like code [23], also has cross-cutting aspects such as versioning, provenance, security (who has access to the artifact over time), and privacy (what actions were performed to protect or release artifact information over time). In this paper we focus on cross-cutting aspects which are related to the *evolution* of business process artifacts over time, i.e., versioning and provenance. Analyzing these aspects will expose many hidden interactions among entities in process graphs.

Definition 9. ['Case Management' and 'Case'] Most important processes for organizations today involve knowledge work. An example of this is the case of government agencies, banks, big legal firms and insurance providers where complex customer and service interactions need to be handled. Case Management, also known as case handling [46], can be defined as a common approach to support knowledge intensive processes. When a customer initiates a request for some services, the set of interactions among people (e.g., customer and other relevant participants) and artifacts from initiation to completion is known as the 'case'. Understanding such ad-hoc processes entails identifying the interactions, among people and artifacts, where artifacts are developed and changed gradually over a long period of time.

3 Example Scenario: Case Management

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 [46]. Figure 3.1-A represents a case instance, in this scenario, where a General Practitioner (GP) suspecting a patient has cancer, updates 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.

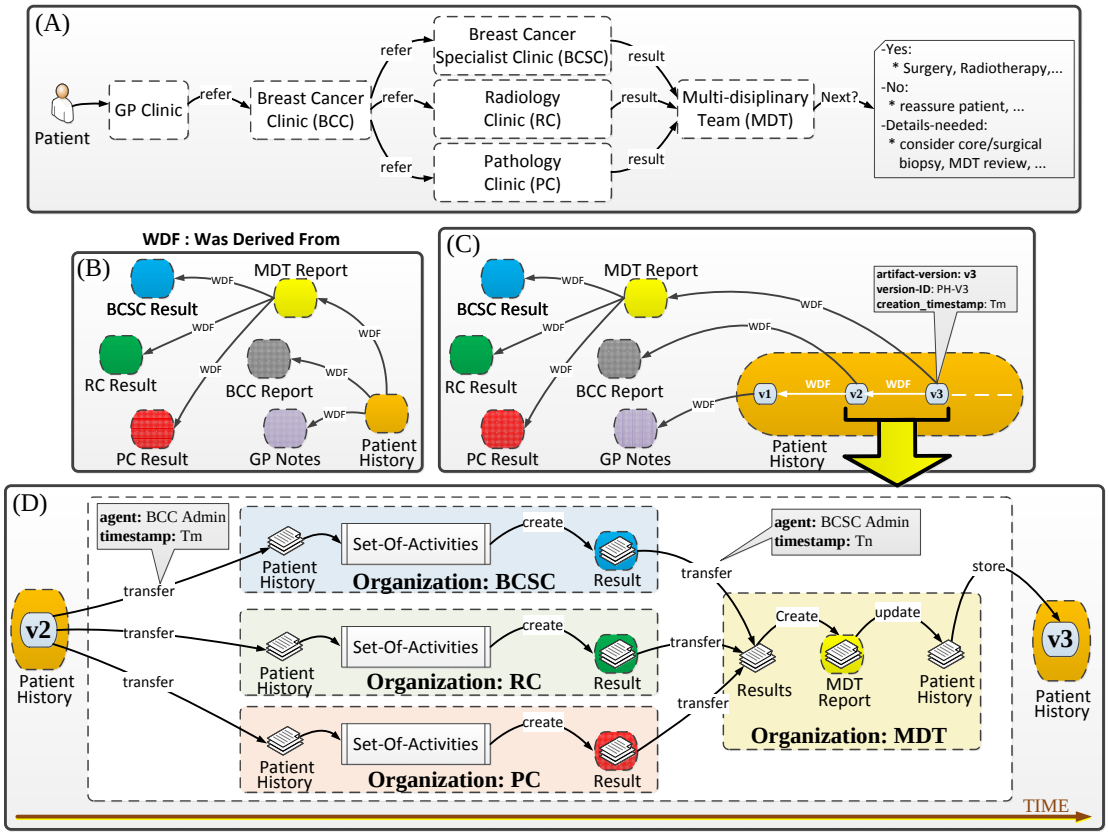

Figure 3.1: Example case scenario for breast cancer treatment including a case instance (A), parent artifacts, i.e. ancestors, for patient history document (B) and its versions (C), and set of activities which shows how version v_2 of patient history document develops and changes gradually over time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

4 Representing Cross-cutting Aspects

4.1 Time and Provenance

Provenance refers to the documented history of an *immutable* object and often represented as graphs. The ability to analyze provenance graphs is important as it offers the means to verify data products, to infer their quality, and to decide whether they can be trusted [37]. In a dynamic world, as data changes, it is important to be able to get a piece of data as it was, and its provenance graph, at a certain point in time. Under this perspective, the provenance queries may provide different results for queries looking at different points in time. Enabling time-aware querying of provenance information is challenging and requires: (i) explicitly representing the time information in the provenance graphs, and (ii) providing timed abstractions and efficient mechanisms for time-aware querying of provenance graphs over an ever increasing volume of data.

The existing provenance models, e.g., the open provenance model (OPM) [34, 38], 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. For example, annotations assigned to an artifact (e.g., a file or Web resource) today may no longer be relevant to the future representation of that entity, as entity attributes are very likely to have different states over time and the temporal annotations may or may not apply to these evolving states.

Due to the implicit treatment of time, abovementioned approaches do not enable explicit representation of the evolution of relevant subgraphs (i.e., group of interrelated objects such as versions of artifacts) and paths (i.e., discovering historical paths through provenance graphs forms the basis of many provenance queries [15, 27, 30]) over time. For example, the shortest path from a business artifact to its origin may change over time [41] as provenance metadata forms a large, dynamic, and time-evolving graph. In particular, versioning and provenance are important cross-cutting aspects of business artifacts and should be considered in modeling the evolution of artifacts over time.

4.2 AEM Data Model and Timed Abstractions

We propose an artifact-centric activity model for ad-hoc processes to represent interaction between actors and artifacts over time. This graph data model (i.e., AEM: Artifact Evolution Model) can be used to represent the cross-cutting aspects in ad-hoc processes and analyze the evolution of artifacts over periods of time. We use and extend the data model proposed in [7] to represent AEM graphs. In particular, AEM data model supports: (i) uniform representation of nodes and edges; (ii) structured and unstructured entities; and (iii) *folder* nodes, which enable grouping related entities, and *path* nodes, which help in analyzing the history of artifacts.

In this paper, we introduce two concepts of *timed* folders and *timed* paths, which help in analyzing AEM graphs. Timed folder and path nodes can show their evolution for the time period that they represent. In AEM, we assume that the interaction among 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.

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 timed 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: 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 whose identity will be the version unique ID and timestamps $\tau_{activity}$.

Timed Folder Node: We proposed the notion of folder node in [7]. A *timed* folder is 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 (see Section 6). New members can be added to timed folders over time. 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.

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 and can be modeled using timed path nodes.

Activity Relationships: An activity is an *explicit* relationship that directly links two entities in the AEM 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 an AEM entity; and (ii) archiving activities, include *actions* such as storage and transfer of an AEM entity.
- *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. Relationship can be codified using regular expressions in which alphabets are the nodes and edges from the graph [7]. 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 and can be modeled using timed path nodes.

A *timed* path node is defined as a timed container for a set of related entities which are connected through *transitive* relationships. We define a timed path node for each change-aware query which results in a set of paths. The change-aware query, documents the evolution of path node by adapting a monitoring code snippet (see Section 6). New paths can be added to timed path nodes over time. Entities and relationships in a timed path node are represented as a subgraph $P_{(\tau_1,\tau_2)} = (V_{(\tau_1,\tau_2)}, E_{(\tau_1,\tau_2)})$, where $V_{(\tau_1,\tau_2)}$ is a set of related nodes representing instances of entities in time which added to the path node P between a time period of τ_1 and τ_2, and $E_{(\tau_1,\tau_2)}$ is a set of directed edges representing *transitive* relationships among these related nodes. It is possible to capture the evolution of the path node $P_{(\tau_1,\tau_2)}$ between a time period of τ_1 and τ_2.

For example, Figure 4.1-A represents a set of activities showing how version v_2 of patient history develops and changes gradually over time and evolves into version v_3. A query which results in a set of paths, can be used to discover all/specific path(s) between v_2 and v_3, and group them under an activity path (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. Figure 4.1-C illustrates the attributes for the constructed activity path and its storage and representation. We use triple tables to store objects (object-store) and relationships among them (link-store) in graphs [7].

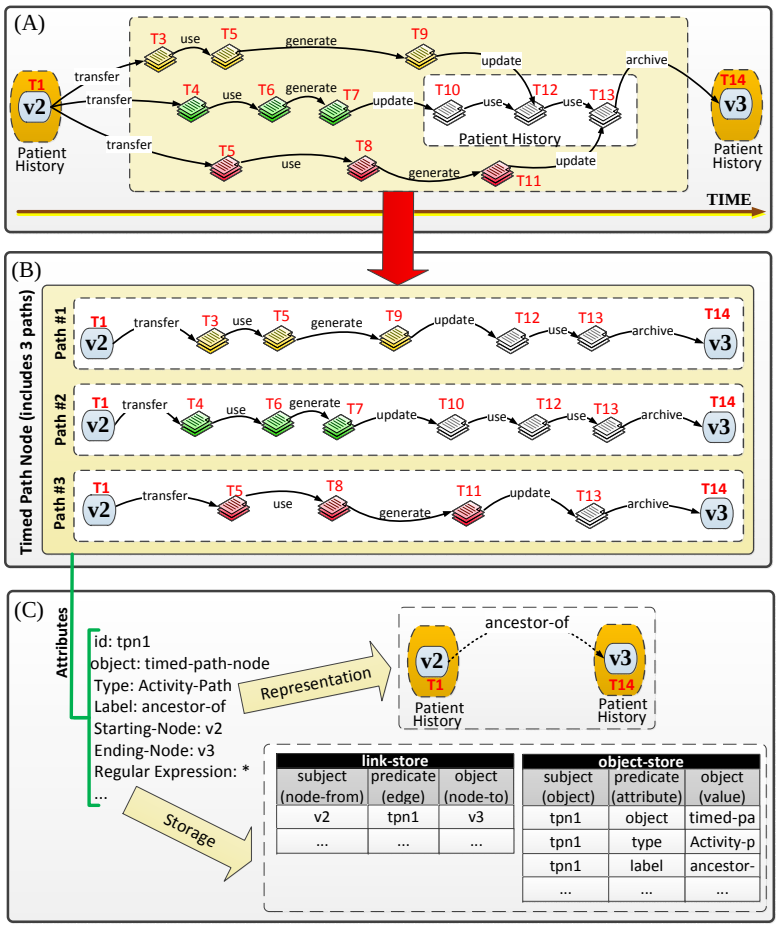

Figure 4.1: Implicit and explicit relationships between versions v_2 and v_3 of patient history document including: (A) activity edges; (B) constructed activity-path stored as a timed (path node) abstraction; and (C) representation and storage of the activity path.

Notice that the AEM graph model supports the uniform representation of nodes and edges: path nodes have unique identities and considered as first class abstractions. In this paper we use path nodes to represent activity paths. As illustrated in Figure 4.1-C, activity paths have set of mandatory attributes, e.g., id, type, starting node, and ending node, and also descriptive attributes.

5 Querying Cross-cutting Aspects

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 [7, 5], a Folder-, Path-enabled extension of SPARQL, 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 graphs. In this paper we extend FPSPARQL to support querying cross-cutting aspects in ad-hoc processes. Moreover, we propose simple query templates for discovering derivation, evolution, and timeseries of artifacts over periods of time

5.1 Formalizing AEM Queries

Artifact-centric 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 like data representation. Our representation, supports uniform representation of nodes and edges: edges are treated as first class citizens where any edge can be described by an arbitrary set of attributes. However, that is not the case in the RDF data model where it is not supported that edges can be described by any attribute information. In our model, a *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 URI references, and β represents set of blanks. Subjects, predicates and objects, 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. In particular, an *RDF graph* is a finite set of triples [14].

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, an activity path, in AEM graph, is a sequence of triples, where the object of each triple in the sequence coincides with the subject of its successor triple in the sequence, and the predicate is an activity relationship having the following mandatory attributes: 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 and apply further operations to discovered paths, we use $pconstruct$ and $apply$ commands proposed in [7]. 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 with *discover* statement which enables process analysts to extract information about facts and the relationship among them 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.
 }
```

This statement can be used for discovering evolution of artifacts (using *evolutionOf* construct), derivation of artifacts (using *derivationOf* construct), and timeseries of artifacts/actors (using *timeseriesOf* con-

struct). The *filter* statement restrict the result to those activities for which the filter expression evaluates to true. Variables such as artifact (e.g., artifact version), type (e.g., lifecycle or archiving), action (e.g., creation, use, or storage), actor, and location (e.g., organization) will be defined in *where* statement.

In order to support temporal aspects of the queries, we adapted the time semantics proposed in [50]. We introduce the special construct, 'timesemantic(fact, [t1, t2, t3, t4])' in FPSPARQL, which is used to represent the $fact$ to be in a specific time interval $[t1, t2, t3, t4]$. A fact may have no temporal duration (e.g., a distinct activity) or may have temporal duration (e.g., series of activities such as process instances). Table 5.1 represents the time-semantics that we support in FPSPARQL queries. The *when* construct, i.e., $when(t1, t2, t3, t4)$ proposed in the above extension, will be automatically translated to *timesemantic* construct in FPSPARQL. Following we will introduce derivation, evolution, and timeseries queries.

5.3 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 process 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 FPSPARQL query for this example.

```
1 discover.evolutionOf(?artifact1,?artifact2);
2 where{
3  ?artifact1 @id v2. ?artifact2 @id v3.
4  #Path node attributes
5  ?pathAbstraction @id tpn1.
6  ?pathAbstraction @label 'ancestor-of'.
7  ?pathAbstraction @description 'version evolution'.
8 }
```

In this example, the *evolutionOf* statement is used to represent the evolution of version v_3 (i.e., variable '?artifact2') from version v_2 (i.e., variable '?artifact1'). The variable '?pathAbstraction' is reserved to identify the attributes for the path node to be constructed. Notice that, by specifying the 'label' attribute (line 6), the implicit relationship, with ID 'tpn1', between versions v_2 and v_3 will be added to the graph (see the query translation). Also, if Adam would be interested to see the whole evolution of version v_3, he does not need to specify the first parameter, e.g., in "evolutionOf(,?artifact2)". In the above example, attributes of variables '?artifact1' and '?artifact2' can be defined in the *where* clause. Considering Figure 4.1-B, 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. This query will automatically be translated to the following path construction query proposed in [7]:

Table 5.1: FPSPQARL 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]

```
1  pconstruct tpn1 as ?evolution (?startNode,?endNode, *)
2  where {
3   ?startNode @id v2.
4   ?endNode @id v3.
5   #defining the path node attributes
6   ?evolution @timed true. #timed path node
7   ?evolution @type Activity-Path.
8   ?evolution @Starting-Node v2.
9   ?evolution @Ending-Node v3.
10  ?evolution @type Activity-Path.
11  ?evolution @label 'ancestor-of'.
12  ?evolution @description 'version evolution'.
13  #add the activity-path to the graph
14  ?evolution @addToLinkStore 'triple(v2,tpn1,v3)'.
15 }
```

In [7], we introduced the $PCONSTRUCT$ command to construct a path node. This command is used to discover transitive relationships between two entities and store it under a path node name. In this paper we extend this command with two attributes: (i) '@timed': setting the value of attribute '@timed' to $true$ for the path node, will assign a monitoring code snippet to this path node (line 6). The code snippet is responsible for updating the path node content over periods of time: new paths can be added to timed path nodes over time; and (ii) '@addToLinkStore': this attribute is used to add the activity-path to the original graph, using a simple triple format (line 14). In this example the value 'triple(v2,tpn1,v3)' for this attribute, will generate the link between versions v_2 and v_3 represented in Figure 4.1-C. Attribute 'label' (line 11) shows the label of this implicit edge in the graph. The value '*' for the regular expression (line 1) will discover all the paths between the starting node, v_2, and the ending node, v_3. Variable '?evolution' represents the activity-path to be constructed, i.e., 'tpn1' (lines 6 to 12). As mentioned earlier, activity paths have set of mandatory attributes, e.g., id, type, starting node, and ending node, and also descriptive attributes, e.g., description (line 12).

Query Filters. 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 the patient history evolution, for the patient having the id 'X14', during November 2012 in radiology clinic. Following is the sample FPSPARQL query for this example.

```
1  discover.evolutionOf( ,?artifact);
2  filter( where(?location),
3          when(?t1,?,?,?t2));
4  where{
5   ?artifact @patient-ID 'X14'.
6   ?location @name 'radiology'.
7   ?t1 @timestamp '11/1/2011 @ 0:0:0'.
8   ?t2 @timestamp '12/1/2011 @ 0:0:0'.
9   #timestamp: M/D/Y @ h:m:s
10  #Path node descriptive attributes
11  ?pathAbstraction @id 'tpn2'.
```

```
12 }
```

In this example, *filter* statement (lines 2 and 3) is used to restrict the result to those activities, happened during November 2011 (lines 7 and 8) in radiology clinic (line 6). As Adam is interested to see the whole evolution of patient history document, he didn't specify the first parameter in the *evolutionOf* construct, i.e., "evolutionOf(,?artifact2)" (line 1). This query will automatically be translated to the following path construction query:

```
1  pconstruct tpn2 as ?patientHisroty
2    (?startNode,?endNode, ?edge (?node ?edge)* )
3  where {
4   #regular expression
5    ?startNode @isA entityNode.
6    ?startNode @id ?stID. ?startNode @patient-ID X14.
7    ?endNode @isA entityNode.
8    ?endNode @id ?stID. ?endNode @patient-ID X14.
9    ?node @isA entityNode. ?node @patient-ID X14.
10   ?edge @isA edge.
11   ?edge @where 'radiology'.
12   ?edge @timestampe ?ts.
13   filter(timesemantic(?ts,[t1,?,?,t2])).
14   ?t1 @timestamp '11/1/2011 @ 0:0:0'.
15   ?t2 @timestamp '12/1/2011 @ 0:0:0'.
16  #defining the path node attributes
17   ?patientHisroty @timed true.
18   ?patientHisroty @type Activity-Path.
19   ?patientHisroty @Starting-Node ?stID.
20   ?patientHisroty @Ending-Node 'X14-med-doc'.
21 }
```

In this example variables '?startNode' and '?endNode' denotes any artifact related to the patient having the ID 'X14' which being used between timestamps '?t1' and '?t2' (lines 14 and 15). Respectively, variables '?edge' and '?node' denotes any edges and nodes in the transitive relationship between starting and ending nodes in the regular expression (lines 9 to 12). To discover the activities applied to artifacts in radiology clinic, the attribute 'where' for the activity relationship '?edge' is set to the value 'radiology' (line 11). The variable '?patientHisroty' is used to define path node attributes (lines 17 to 20). The *when* statement (i.e. $when(t1, ?, ?, t2)$) in the evolution query have been translated to the special construct $timesemantic(?ts, [t1, ?, ?, t2])$ in FPSPARQL (line 13) which is used to represent the activity timestamps '?ts', to be in a specific time interval $[t1, ?, ?, t2]$. Recall from previous example, that this implicit relationship will not be added to the original graph as the attribute 'label' have not been specified in the evolution query.

5.4 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 a 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 the patient history (see Figure 3.1-C). Following is the sample FPSPARQL query for this example.

```
1  discover.derivationOf(?artifact);
2  where{
3   ?artifact @id v3.
4  }
```

In this example, *derivationOf* statement is used to represent the derivation(s) 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 be translated to the following path construction query:

```
1  select ?startNode from
2  pconstruct derivation_v3
3    (?startNode, ?endNode, '?edge (?node ?edge)*')
4  where {
5   ?startNode @isA entityNode.
6   ?endNode @isA entityNode.
7   ?endNode @type artifactVersion.
8   ?endNode @id v3.
9   ?node @isA entityNode.
10  ?edge @isA edge.
11 }
```

In [7], we used *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 use *pconstruct* statement to discover paths between set of starting nodes (ancestors) to a specific ending node (version v_3 of patient history). The result of this query will be set of artifacts/versions reachable from version v_3 of patient history document. For the sake of simplicity we enabled applying further operations to the constructed path node using select statement (line 1), e.g., the variable '?startNode' in *select* statement will return the ancestors of version v_3 of patient history. It is possible to use *apply* statement for applying further operations to the constructed path nodes (details can be found in [7]). Moreover, the query in this example can be timed, i.e., using '@timed' attribute.

Query Filters. 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 the '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 before March 2011. Following is the sample FPSPARQL query for this example.

```
1 discover.derivationOf(?artifact);
2 filter( where(?location),
3         when(?,?,?,?t1) );
4 where{
5  ?artifact @id v3.
6  ?location @name 'radiology'.
```

```
7   ?t1 @timestamp '3/1/2011 @ 0:0:0'.
8 }
```

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

5.5 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 Eli's activities on the patient history document between timestamps τ_1 and τ_5. Following is the sample FPSPARQL query for this example.

```
1 discover.timeseriesOf(?actor);
2 filter( when("T1",?,?,"T5") );
3 where{
4  ?actor @id Eli-id.
5 }
```

In this example, *timeseriesOf* statement (line 1) is used to represent the timeseries of Eli, i.e., the variable '?actor'. Attributes of variable $?actor$ can be defined in the *where* clause (line 4). 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 timed and may contain *filter* statement, where *filter* statement can be used to answer specific timeseries questions.

5.6 Constructing Timed Folders

In Section 5.3 we discussed how we extend path construction queries to support time aware querying. In this section, we discuss how to construct timed folder nodes. In particular, to construct a *timed folder* node, we use FPSPARQL's $fconstruct$ statement proposed in [7]. We extend this statement with '@timed' attribute. 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: new members can be added to timed folders over time. The syntax for a basic construction query of a timed folder node is given as follows:

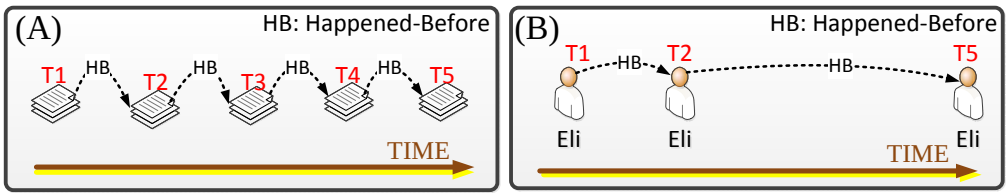

Figure 5.1: Sample timeseries for: (A) patient history document between τ_1 and τ_5; and (B) Eli, an actor, acting on patient history between τ_1 and τ_5.

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

For example, considering Figure 3.1-C, a timed folder can be constructed to represent a patient history document which may contain various versions of this artifact. New versions (and activities on top of it) can be added to this folder over time. Following is a sample FPSPARQL query for this example.

```
1  fconstruct X14-patient-history as ?med-doc
2  select ?version
3  where {
4   #defining the path node attributes
5    ?med-doc @timed true.
6    ?med-doc @type artifact.
7    ?med-doc @description 'history for patient #X14'.
8   #specifying nodes to be added to the folder
9    ?version @isA entityNode.
10   ?version @patient-ID X14.
11 }
```

In this example, variable '?med-doc' represents the folder node to be constructed, i.e., 'X14-patient-history' (line 1). This folder is of type 'artifact' (line 6). Setting the attribute $timed$ to $true$ (line 5) will force new artifacts having the patient ID 'X14' (line 10) to be added to this folder over time. The attribute 'description' used to describe the folder (line 7). The variable '?version' is an AEM entity and represents the patient history versions to be collected (line 9). Attribute 'patient-ID' (line 10) indicate that the version is related to the patient history of the patient having the id 'X14'.

As another example, it is possible to construct a timed folder to monitor the artifacts touched (created, transformed, used, deleted, stored, or transferred) by Eli. As the result of this query, all the artifacts touched by Eli will be added to the constructed timed folder. Moreover, new artifacts can be added to this folder over time. Following is a sample FPSPARQL query for this example.

```
1  fconstruct Eli_artifacts as ?Eli_art
2  select ?artifact1
3  where {
4   ?Eli_art @timed true.
5   ?Eli_art @type artifact.
6   ?Eli_art @description 'artifacts generated by Eli'.
7   #
8   ?artifact1 ?activity ?artifact2.
9   ?artifact1 @isA entityNode.
10  ?artifact2 @isA entityNode.
11  ?activity @isA edge.
12  ?activity @who Eli_id.
13 }
```

In this example, variable $?Eli\_art$ represents the folder node to be constructed, i.e., 'Eli_artifacts' (line 1). This folder is of type 'artifact' (line 5). Setting the attribute $timed$ to $true$ (line 4) will force new

artifacts touched by Eli to be added to this folder over time. The pattern '?artifact1 ?activity ?artifact2' (line 8) illustrates an activity relationship between two artifacts, '?artifact1' and '?artifact2'. The activity relationship '?activity' (line 11) has a set of mandatory attributes discussed in Section 4.2. To discover the artifacts touched by Eli, i.e., '?artifact1', the attribute 'who' for the activity relationship '?activity' is set to Eli's identification (line 12). More activity relationship attributes can be used, e.g., to discover the artifacts generated, deleted, or stored by an actor.

Querying Timed Folder Nodes. In [7], we introduced the *apply* statement to apply further operations to constructed folder nodes. For example, consider a user who is interested to retrieve information about patient history folder, e.g., folder 'X14-patient-history' constructed in previous examples, between timetamps τ_2 and τ_7. Following is the FPSPARQL query for this example.

```
1 (X14-patient-history)
2 apply (
3  select ?a
4  where {
5    ?a @isA entityNode.
6    ?a @timestamp ?ts.
7    filter( timesemantic(?ts,[t2,?,?,t7]) ).
8  }
9 )
```

In this example the query applied to the constructed timed folder node 'X14-patient-history'. Variable '?a' represents the members (i.e., artifact versions) of the folder node whose (creation) timestamp '?ts' (line 6) falls between time τ_2 and τ_7 (line 7). The 'timesemantic' construct is used to filter the requested timestamps.

6 Architecture and Implementation: Temporal Extension

6.1 Architecture

In order to analyze process graphs, we proposed a graph processing engine, i.e., FPSPARQL [7, 5, 4] (a Folder-, Path-enabled extension of the SPARQL), to manipulate and query entities, and folder and path nodes. The query engine is implemented in Java and supports two types of storage back-end:

- Relational Database System: 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. While simple, this schema quickly hits scalability limitations [42]. To avoid this we developed a relational RDF store including its three classification approaches [42]: vertical (triple), property (n-ary), and horizontal (binary). The query engine supports various relational database management systems (e.g., IBM DB2, PostgreSQL, and Microsoft SQL Server) to generate physical storage layer.
- Hadoop File System: We use Hadoop [49], an open source software framework that supports data-intensive distributed applications, data processing platforms to store and retrieve process graphs in Hadoop file system and to support cost-effective and Web-scale processing of process graphs. We use Apache-Pig[1], a high-level procedural language on top of Hadoop for querying large process graphs.

[1] http://pig.apache.org/

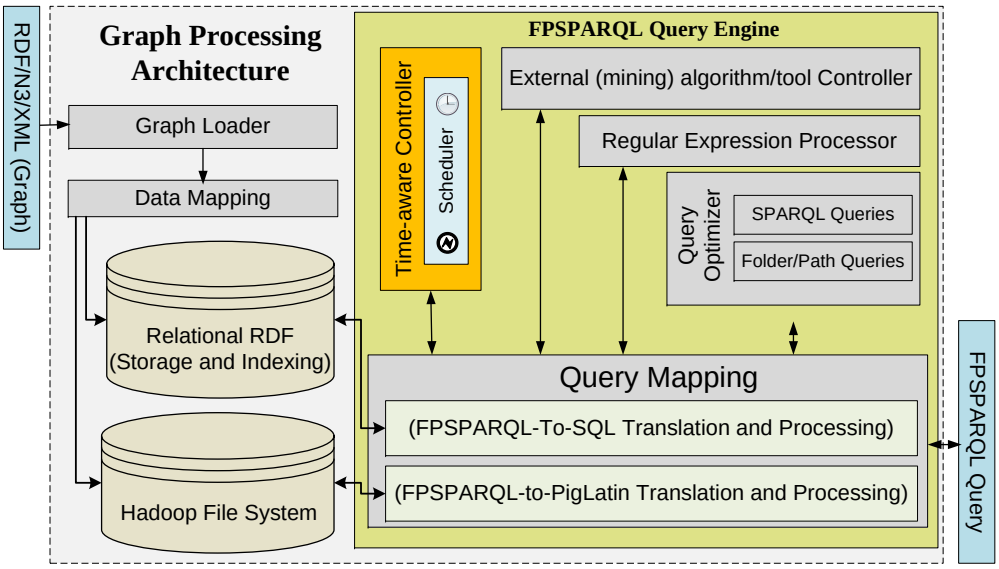

Figure 6.1: FPSPARQL graph processing architecture: analytics extension.

As discussed in [6, 9, 10, 8, 4], FPSPARQL architecture consists of the following components: Graph Loader, Data Mapping Layer, Time-aware Controller, Query Mapping Layer, Regular Expression Processor, External Algorithm/Tool Controller, and Query Optimizer. In this paper, we instrument the Time-aware Controller to support the execution of FPSPARQL temporal queries. In particular, Time-aware Controller will be 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.

Time-aware Controller enables users to set an AEM query as a: (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. The interval attribute can be set manually in the query engine; 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.

Moreover, as discussed in [7], Time-aware Controller is responsible for data changes and incremental graph loading: RDF databases are not static and changes may apply to graph entities (i.e. nodes, edges, and folder/path nodes) over time. Updates are mostly insertions of new triples into the object store and link store. At the current version of FPSPARQL query engine, updates cannot be performed concurrently with queries: there may be the need for full-fledged ACID transactions. Figure 6.1 illustrates FPSPARQL graph processing architecture.

Figure 6.2 illustrates an overview of Time-aware Controller architecture. The controller uses numerical priorities, priorities take values from the set of real numbers $\mathbb{R}$, to express precedence constraints over the set of executing code snippets. The priority of code snippet c_i at time τ is given by $p_i(\tau)$. For two code snippets c_i and c_j and a time point τ, c_i will execute in preference to c_j if and only if $p_i(\tau) > p_i(\tau)$. The Time-aware Controller is composed of the following components:

- Priority Functions: are functions of priority with respect to relative time. They detail how a code snippets priority varies relative to a deadline.
- Time-aware Structure: is a data structure that describes how relative priorities for all agents vary as

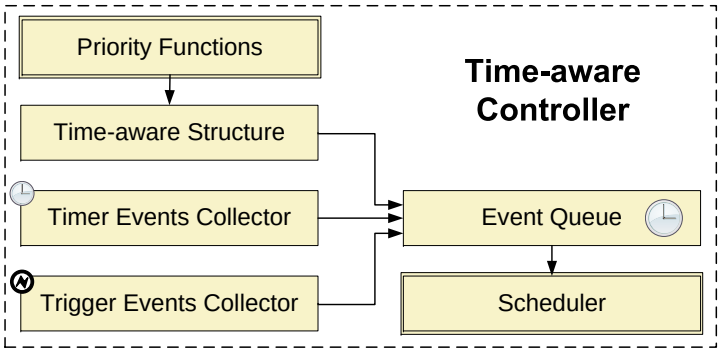

Figure 6.2: Overview of Time-aware Controller architecture.

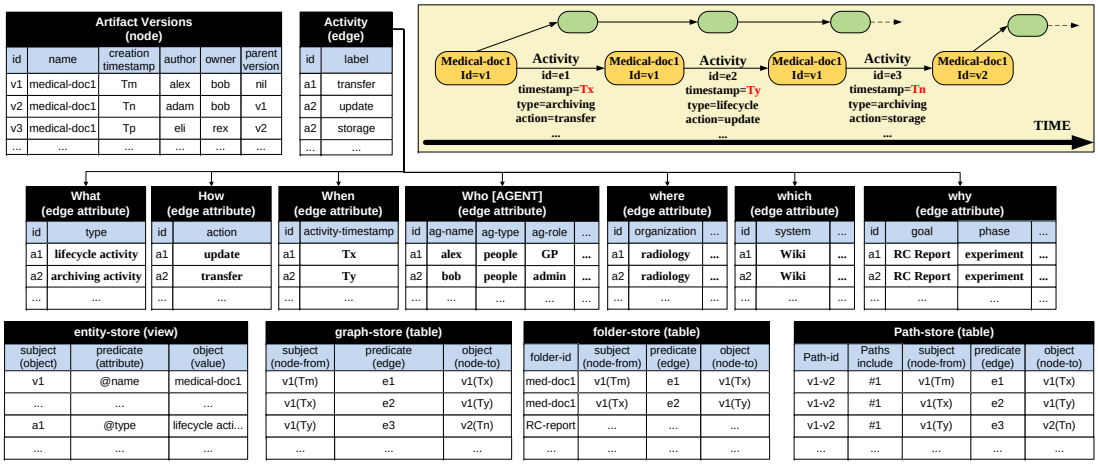

Figure 6.3: Physical layer for storing a sample AEM graph and tables to store AEM entities and relationships.

a function of time. It gives the relative priority ordering of all code snippets possibly executing at time τ.

- Timer Events Collector: is responsible to monitor and manage timer events, i.e., pull queries.
- Trigger Events Collector: is responsible to monitor and manage trigger events, i.e., push queries.
- Event Queue: is a mechanism of dealing with asynchronous events in a synchronous manner. In particular, every time there is a change in the relative precedence of executing code snippets, a timer/trigger event is generated by Time-aware Structure and placed into the Event Queue.
- Scheduler: is responsible for executing timer/trigger events queued in the Event Queue and in the scheduled time.

6.2 Implementation

Details about FPSPARQL query engine implementation is presented in [7]. In this paper, we instrument the query engine with Time-aware Controller. We model graphs based on a RDF like data representation. Figure 6.3 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 among 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 triplestore, 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 triplestore. The 'folder-id' column is added to this triplestore 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. Moreover, we have implemented a front-end tool to assist process 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 operations (i.e., evolution, derivation, or timeseries) on top of selected entity. The proposed templates (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 6.4-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. For the AEM graph exploration, we provide users with a timeline like graph visualization with facilities such as zooming in and out. Figure 6.4-B illustrates a screenshot of this tool while generating an evolution query.

7 Experiments

7.1 Datasets

We carried out the experiments on three time-sensitive datasets: (i) The real life log of a Dutch academic hospital[1], originally intended for use in the first Business Process Intelligence Contest (BPIC 2011); (ii) e-Enterprise Course, this scenario is built on our experience on managing an online project-based course[2]; and (iii) Supply Chain Management log.

Dutch Academic Hospital

The real-life event log, taken from a Dutch Academic Hospital, contains events related to a heterogeneous mix of patients diagnosed with cancer (at different stages of malignancy) pertaining to the cervix, vulva, uterus and ovary. The event log contains 1143 cases and 150291 events referring to 624 distinct activities. Details about this event log can be found in [13]. Given the heterogeneous nature of these cases, we first applied a preprocessing phase to adapt this dataset to artifact-centric case scenarios. For example, we created more homogeneous subsets of cases, e.g., patients having a particular type of cancer. Then we

[1] http://data.3tu.nl/repository/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54
[2] http://www.cse.unsw.edu.au/∼cs9323

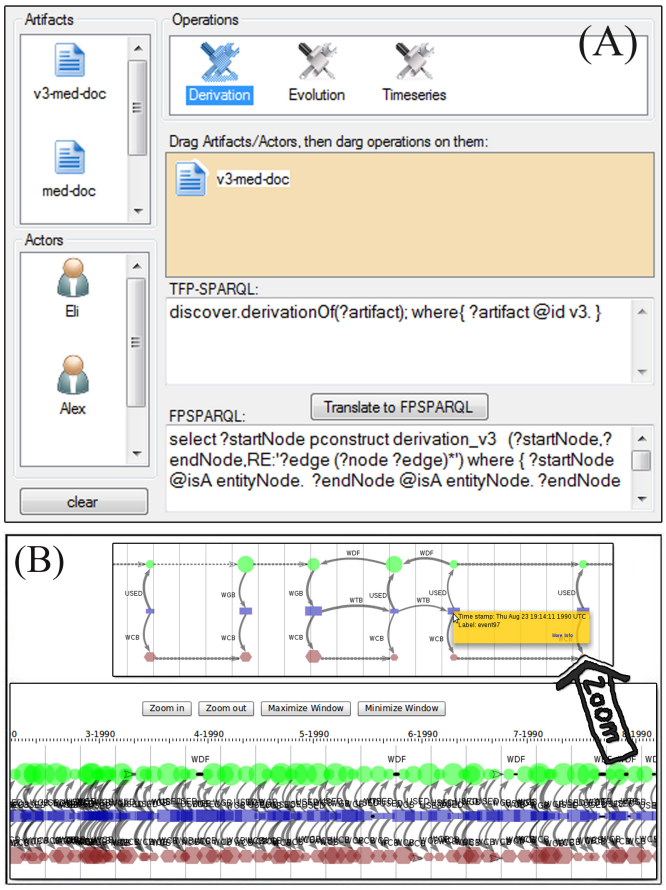

Figure 6.4: Screenshots of front end tool: (A) Query assistant tool; and (B) graph visualization tool: to visualize AEM graphs.

assigned each case in the event log to a patient document history. Afterward, we generated versions for document histories according to event timestamps: the event log contains rich information stored as attributes both at the event level and at the case level. Finally we generated AEM graph model out of preprocesses log, where the generated graph includes artifacts and activity relationships among them.

e-Enterprise Course

This scenario, is built on our experience on managing an online project-based course "e-Enterprise Projects". 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. 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.

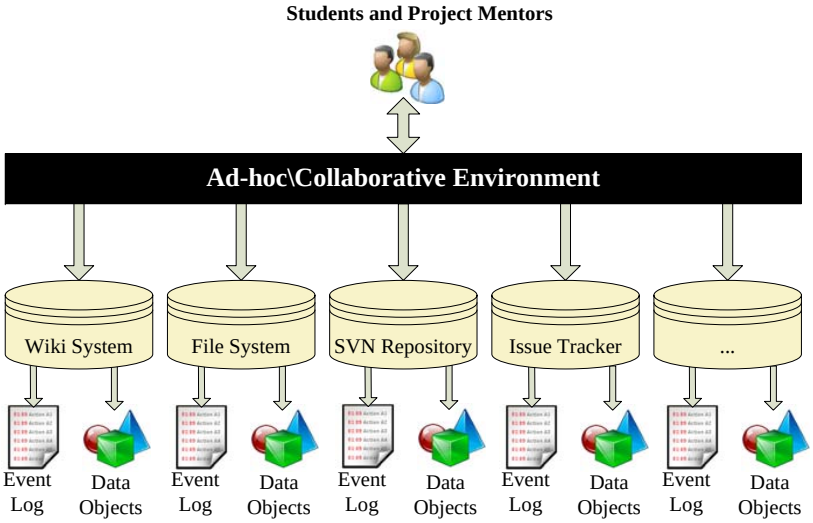

Figure 7.1: e-Enterprise course scenario.

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. Figure 7.1 depicts an illustration of this scenario.

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 this dataset to a case scenario. Details about this dataset can be found in [7].

7.2 Evaluation

We have compared our approach with that of querying open provenance model (OPM) [34, 38]. OPM, a proposal for a standard graph data model and vocabulary for provenance, presents graph nodes as data artifacts, processes, and agents. Five causal relationships are defined in OPM: a process 'used' an artifact, an artifact 'was-Generated-By' a process, a process 'was-Triggered-By' a process, an artifact 'was-Derived-From' an artifact, and a process 'was-Controlled-By' an agent.

We generated two types of graph models, i.e., AEM and OPM, from proposed datasets. The AEM graphs generated based on the proposed model in Section 4.2. The OPM graphs generated based on open provenance model specification [34, 38]. Figure 7.2, represents a sample AEM graph (Figure 7.2-A) for the hospital log, a sample OPM graph generated from a part of AEM graph (Figure 7.2-B), and open provenance model entities and relationships (Figure 7.2-C). Both AEM and OPM graphs for each datasets

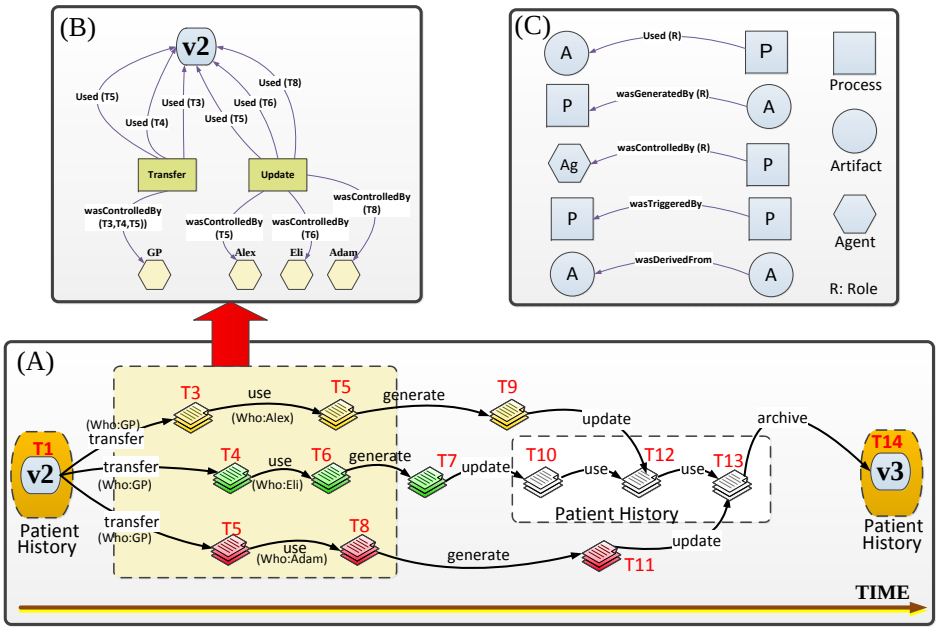

Figure 7.2: A sample AEM graph for the hospital log (A), a sample OPM graph generated from a part of AEM graph (B), and open provenance model entities and relationships (C).

loaded into FPSPARQL query engine. We evaluated the performance and the query results quality using the proposed graphs.

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 7.3 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 7.3 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. We sampled different sizes of the graphs very carefully and based on related cases (patients in the log hospital, projects in the e-Enterprise project, and products in the SCM log) to guarantee the attributes of generated 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 processes 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, 125 paths were discovered and examined by domain experts, and 122 paths (precision=97.6%) considered relevant. And as a result of applying queries to OPM graphs generated from all the datasets, 297 paths discovered, examined by domain experts, and 108 paths (precision=36.4%) considered relevant.

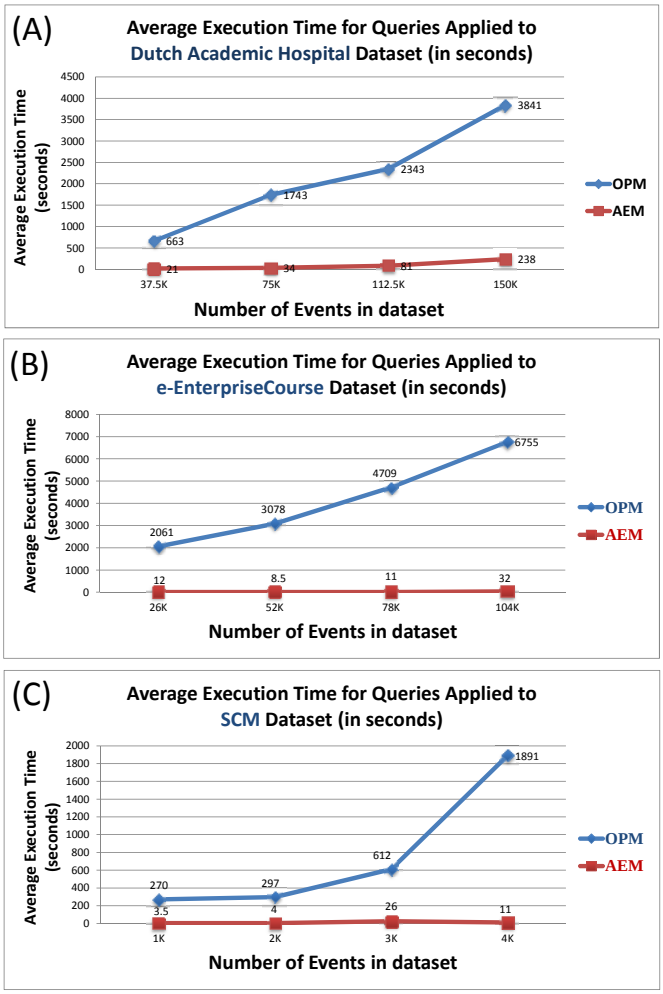

Figure 7.3: The query performance evaluation results, illustrating the average execution time for applying evolution, derivation, and timeseries queries on AEM and OPM graphs generated from: (A) Dutch academic hospital dataset; (B) e-Enterprise course dataset; and (C) SCM dataset.

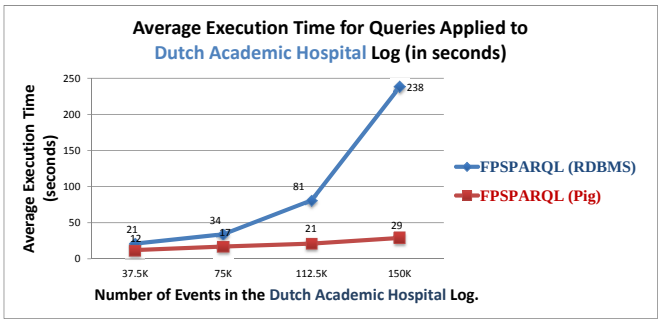

Figure 7.4: The evaluation results, illustrating the performance analysis between RDBMS and Hadoop applied to Dutch academic hospital dataset.

Performance Comparison Between RDBMS and Hadoop Execution Plans. As mentioned earlier, FPSPARQL queries can be run on two types of storage back-end: RDBMS and Hadoop. In this part we compare the performance of query plans on relational triplestores and Hadoop file system. All experiments in this part were conducted on a virtual machine, having 32 cores and 192GB RAM. Figure 7.4 illustrates the performance analysis between RDBMS and Hadoop for queries (average execution time) in Figure 7.3-A applied to Dutch academic hospital dataset. Figure 7.4 shows an almost linear scalability between the response time of FPSPARQL queries applied to Hadoop file system and the number of events in the log.

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. As discussed in [7], there are two types of graph reachability algorithms: algorithms traversing from starting vertex to ending vertex using breadth-first or depth-first search over the graph, and algorithms checking whether the connection between two nodes exists in the edge transitive closure of the graph. 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.

8 Related Work

We study the related work into three main areas: artifact-centric processes, provenance, and modeling/querying temporal graphs:

8.1 Artifact-centric Processes

Knowledge-intensive processes almost always involves the collection and presentation of a diverse set of artifacts, and capturing human activities around artifacts. This, emphasizes the artifact-centric nature of such processes where *time* becomes an important part of the equation. Many approaches [29, 25, 11, 17, 12] used business artifacts, that combine data and process in a holistic manner, as the basic building block. Some of these works [29, 25, 17] used a variant of finite state machines to specify lifecycles. Some theoretical works [12, 11] explored declarative approaches to specifying the artifact lifecycles following an event oriented style.

Another line of work in this category, focused on modeling and querying artifact-centric processes [33, 48, 21]. In [33, 48], a document-driven framework, proposed to model business process management systems through monitoring the lifecycle of a document. Dorn et.al. [21], 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 [11] where the process model is defined in terms of the lifecycle of the documents. Some other works [1, 40, 18, 19, 43], focused on modeling and querying techniques for knowledge-intensive tasks. Some of existing approaches [1, 40] for modeling ad-hoc processes focused on supporting ad-hoc workflows through user guidance. Some other approaches [18, 19, 43] focused 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.

In our model, actors, activities, artifacts, and artifact versions are first class citizens, and the evolution of the activities on artifacts over time is the main focus. The AEM model supports timed queries and enables weaving cross-cutting aspects, e.g., versioning and provenance, around business artifacts to imbues the artifacts with additional semantics that must be observed in constraint and querying ad-hoc processes.

8.2 Provenance

Many provenance models [16, 24, 38, 44] 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) [38], 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 [30, 27, 51, 15, 35]. In ProQL [30], 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 [27] proposed to use a semi-structured data model for handling provenance and extended the Lorel query language for traversing and querying provenance graph. NetTrails [51] proposed a declarative platform for interactively querying network provenance in a distributed system in which query execution performs a traversal of the provenance graph. RDFProv [15] is an optimized framework for scientific workflow provenance querying and management. Missie et al. [35] presented a provenance model and query language for collection-oriented workflow systems. They emphasize on querying the provenance of collection of activities. The proposed framework, do not support modeling, querying and analyzing the evolution of group of related entities over time.

8.3 Modeling/Querying Temporal Graphs

In recent years, a plethora of work [28, 32, 41] 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. [41] proposed a historical graph-structure to maintain analytical processing on such evolving graphs. Moreover, authors in [32, 41] proposed 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 for querying graphs (e.g., [3, 26, 39, 47]) provide temporal extensions of existing graph models and languages. Tappolet et al. [47] provided temporal semantics for RDF graphs. They proposed τ-SPARQL for querying temporal graphs. Grandi [26] presented another temporal extension for SPARQL, i.e. T-SPARQL, aimed at embedding several features of TSQL2 [36] (temporal extension of SQL). SPARQL-ST [39] and EP-SPARQL [3] 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 time-sensitive queries, propose timed abstractions to store the result of such queries, and enable analyzing the evolution of such timed abstractions over time. Moreover, we extended FPSPARQL [7], our previous work, to support temporal queries and monitor the result of such queries over time.

9 Conclusion and Future Work

In this paper, we have presented an artifact-centric activity model (AEM: Artifact Evolution Model) for ad-hoc processes. This model supports timed queries and enables weaving cross-cutting aspects, e.g., versioning and provenance, around business artifacts to imbues the artifacts with additional semantics that must be observed in constraint and querying ad-hoc processes. Two concepts of timed folders and activity-paths have been introduced, which help in analyzing AEM graphs. Folders enabled grouping related entities and paths helped 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 FPSPARQL, 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 real-world datasets. The results of evaluation showed 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] M. Adams, A.H.M.T. Hofstede, D. Edmond, and W.M.P.V.D. Aalst. Facilitating flexibility and dynamic exception handling in workflows through worklets. In *CAiSE*, 2005.

[2] C.C. Aggarwal and H. Wang. *Managing and Mining Graph Data*. Springer Publishing Company, Incorporated, 2010.

[3] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, 2011.

[4] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, and Hamid R. Motahari-Nezhad. An artifact-centric activity model for analyzing knowledge intensive processes. Technical Report unsw-cse-tr-201210, University of New South Wales, 2012.

[5] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Hamid R. Motahari-Nezhad, and Mohammad Allahbakhsh. A framework and a language for on-line analytical processing on graphs. In *WISE*, pages 213–227, 2012.

[6] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Hamid R. Motahari-Nezhad, and Mohammad Allahbakhsh. Online Analytical Processing on Graphs (GOLAP): Model and Query Language. Technical Report unsw-cse-tr-201214, University of New South Wales, 2012.

[7] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Hamid R. Motahari-Nezhad, and Sherif Sakr. A query language for analyzing business processes execution. In *Business Process Management (BPM), 9th International Conference, Clermont-Ferrand, France*, pages 281–297, 2011.

[8] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Hamid R. Motahari-Nezhad, and Sherif Sakr. FPSPARQL: A language for querying semi-structured business process execution data. Technical Report unsw-cse-tr-1103, University of New South Wales, 2012.

[9] Seyed-Mehdi-Reza Beheshti, Hamid R. Motahari-Nezhad, and Boualem Benatallah. Temporal Provenance Model (TPM): Model and query language. Technical Report UNSW-CSE-TR-1116, University of New South Wales, 2011.

[10] Seyed-Mehdi-Reza Beheshti, Sherif Sakr, Boualem Benatallah, and Hamid R. Motahari-Nezhad. Extending SPARQL to support entity grouping and path queries. Technical Report UNSW-CSE-TR-1019, University of New South Wales, 2010.

[11] 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.

[12] K. Bhattacharya, R. Hull, and J. Su. A data-centric design methodology for business processes. In *Handbook of Research on Business Process Modeling, chapter 23*, pages 503–531, 2009.

[13] R.P. Jagadeesh Chandra Bose and W.M.P.V.D. Aalst. Analysis of Patient Treatment Procedures: The BPI Challenge Case Study. Technical Report BPM-11-18, BPMCenter.org, 2011.

[14] T. Cadenhead, V. Khadilkar, M. Kantarcioglu, and B.M. Thuraisingham. A language for provenance access control. In *CODASPY*, pages 133–144, 2011.

[15] A. Chebotko, S. Lu, X. Fei, and F. Fotouhi. RDFProv: A relational RDF store for querying and managing scientific workflow provenance. *Data Knowl. Eng.*, 69(8):836–865, 2010.

[16] J. Cheney, L. Chiticariu, and W.C. Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1:379–474, April 2009.

[17] D. Cohn and R. Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, 32(3):3–9, 2009.

[18] C. Dorn, T. Burkhart, D. Werth, and S. Dustdar. Self-adjusting recommendations for people-driven ad-hoc processes. In *BPM*, pages 327–342, 2010.

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

[20] C. Dorn, C.A. Marín, N. Mehandjiev, and S. Dustdar. Self-learning predictor aggregation for the evolution of people-driven ad-hoc processes. In *BPM*, pages 215–230, 2011.

[21] C. Dorn, C.A. Marn, N. Mehandjiev, and S. Dustdar. Self-learning predictor aggregation for the evolution of people-driven ad-hoc processes. In *BPM*, pages 215–230, 2011.

[22] S. Dustdar, T. Hoffmann, and W.M.P.V.D. Aalst. Mining of ad-hoc business processes with teamlog. *Data Knowl. Eng.*, 55(2):129–158, 2005.

[23] C.E. Dyreson. Aspect-oriented relational algebra. In *EDBT*, pages 377–388, 2011.

[24] J. Freire, D. Koop, E. Santos, and C.T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engg.*, 10:11–21, May 2008.

[25] C.E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *ICSOC*, pages 181–192, 2007.

[26] F. Grandi. T-SPARQL: a TSQL2-like temporal query language for RDF. In *International Workshop on Querying Graph Structured Data*, pages 21–30, 2010.

[27] D.A. Holland, U. Braun, D. Maclean, K.K. Muniswamy-Reddy, , and M. Seltzer. Choosing a data model and query language for provenance. In *IPAW*, 2008.

[28] P. Holme and J. Saramki:. Temporal networks. *CoRR*, abs/1108.1780, 2011.

[29] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM Conferences (2)*, pages 1152–1163, 2008.

[30] G. Karvounarakis, Z.G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*. ACM, 2010.

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

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

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

[34] N. Kwasnikowska, L. Moreau, and J. Van Den Bussche. A formal account of the open provenance model. *submitted*, pages 1–49, 2010.

[35] P. Missier, N.W. Paton, and K. Belhajjame. Fine-grained and efficient lineage querying of collection-based workflow provenance. In *EDBT*, pages 299–310, 2010.

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

[37] L. Moreau. Provenance-based reproducibility in the semantic Web. *J. Web Sem.*, 9(2):202–221, 2011.

[38] L. Moreau, J. Freire, J. Futrelle, R.E. Mcgrath, J. Myers, and P. Paulson. The open provenance model: An overview. IPAW'08, pages 323–326, Berlin, Heidelberg, 2008. Springer.

[39] M. Perry, P. Jain, and A.P. Sheth. SPARQL-ST: Extending SPARQL to support spatiotemporal queries. In *Geospatial Semantics and the Semantic Web*, pages 61–86, 2011.

[40] H.A. Reijers, J.H.M. Rigter, and W.M.P.V.D. Aalst. The case handling case. *Int. J. Cooperative Inf. Syst.*, 12(3):365–391, 2003.

[41] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *VLDB*, 4(11):727–737, 2011.

[42] S. Sakr and G. Al-Naymat. Relational processing of RDF queries: a survey. *SIGMOD Rec.*, 38(4):23–28, 2009.

[43] H. Schonenberg, B. Weber, B.F.V. Dongen, and W.M.P.V.D. Aalst. Supporting flexible processes through recommendations based on history. In *BPM*, pages 51–66, 2008.

[44] J. Shen, E. Fitzhenry, and T.G. Dietterich. Discovering frequent work procedures from resource connections. In *IUI*, pages 277–286, 2009.

[45] T. Stoitsev, S. Scheidl, and M. Spahn. A framework for light-weight composition and management of ad-hoc business processes. In *TAMODIA*, pages 213–226, 2007.

[46] K.D. Swenson, N. Palmer, and B. Silver. *Taming the Unpredictable Real World Adaptive Case Management: Case Studies and Practical Guidance*. Future Strategies Inc, 2011.

[47] J. Tappolet and A. Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *ESWC*, pages 308–322, 2009.

[48] J. Wang and A. Kumar. A framework for document-driven workflow systems. In *BPM*, pages 285–301, 2005.

[49] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, original edition, June 2009.

[50] Q. Zhang, F.M. Suchanek, L. Yue, and G. Weikum. TOB: Timely ontologies for business relations. In *WebDB*, 2008.

[51] W. Zhou, Q. Fei, S. Sun, T. Tao, A. Haeberlen, Z.G. Ives, B.T. Loo, and M. Sherr. NetTrails: a declarative platform for maintaining and querying provenance in distributed systems. In *SIGMOD Conference*, pages 1323–1326, 2011.