Mashup Feeds: Continuous Queries over Web Services

Junichi Tatemura[1] Arsany Sawires[2] Oliver Po[1]
Songting Chen[1] K. Selcuk Candan[1] Divyakant Agrawal[1] Maria Goveas[3]
[1]NEC Laboratories America, 10080 North Wolfe Road, Suite SW3-350, Cupertino, CA 95014
[2]Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93016
[3]Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287
{tatemura,oliver,songting,candan,agrawal}@sv.nec-labs.com,
arsany@cs.ucsb.edu, mgoveas@asu.edu

1. INTRODUCTION

Mashups are web applications that combine information from several sources, typically provided through simple Web APIs (i.e. web services). Although the concept of mashups has become popular, current mashup applications are limited to very primitive information integration. We claim that this is partly due to the lack of a powerful description language for mashups. Mashups are usually written in procedural programming languages such as JavaScript, and the code rarely separates the user interface layer (dynamic HTML) and the data integration layer well.

Web Service Composition [1] has been studied and developed, including industry efforts on standard composition languages such as BPEL. However, web service composition techniques are not suitable for mashups because they focus on describing processes (or workflows) based on messaging and synchronization primitives; whereas mashups mostly involve data integration rather than process integration. Thus, there is a need for service composition languages with powerful data processing primitives.

Given that the web is becoming more and more dynamic, it is also essential that the data processing primitives support continuous query semantics. Since the web data sources are continuously changing (e.g. news feeds and blogs), it is natural that the information composed of such sources reflect the dynamic nature of the sources. Furthermore, by monitoring the evolution of the sources over time, it is possible to extract extra information (e.g. temporal summaries and tracking information), which is not directly provided by the base data sources. WSMS [2] provides a simple query engine over multiple web services, but it does not support continuous queries.

Mashup Feeds is a system that enables mashup developers to describe new integrated web service feeds as continuous queries over existing feeds and web services. In this demonstration, we present a prototype of the system, which consists of a visual mashup composer, an execution engine, and interfaces for the end users to subscribe to the developed mashup feeds. The demonstration highlights the following main features:

- *Collection-based streams:* In order to enable the extraction of history-based (stateful) summaries and tracking information, the system provides a new collection-based stream processing operator. Unlike the classical stream aggregation operators which deal with streams of tuples (or objects), the collection-based stream operator deals with streams of collections of tuples (or objects). Such new semantics are needed since a single request to a source web service generally results in a collection (a list) of objects rather than a single object. Thus, monitoring a source web service over time requires handling a stream of responses, each of which is a collection of objects.
- *User-defined plugins:* The system works as a workflow engine that can be easily extended with user-defined plugins in order to enable general content manipulation tasks. The demonstration shows two use cases: one demo scenario handles simple tuples, while the other one handles semi-structured objects.

SIGMOD'07, June 11–14, 2007, Beijing, China.
ACM 978-1-59593-686-8/07/0006.

2. MOTIVATING EXAMPLES

We focus on continuous queries that track information over time. We use the following examples:

EXAMPLE 1. *Assume we have two web services: one that returns the current stock price given the stock name, and another one that returns a list of stocks ranked by the exchange volume. We want to identify the stocks with increasing price within highly exchanged stocks.*

EXAMPLE 2. *Assume we have an RSS feed that contains today's news articles on a specific category ranked by popularity on a certain news web site. We want to identify a "thread" of news on a specific topic that has been popular for a week.*

Note that although Example 2 has only one information source, we can still apply the concept of mashups by combining information from the different versions of the content at different times, we call this *self-mashups* or *temporal mashups*.

3. MODEL

A mashup query is visually specified as a workflow graph of service calls and data processing operators.

3.1 Web Service Model

A data-centric web service can be seen as a function that maps an object (request parameters) to a collection of objects (response): $\mathcal{O} \rightarrow \mathcal{C}(\mathcal{O})$, where $\mathcal{C}(\alpha)$ denotes the data

type of a collection of items of type α. In general, a web service can return arbitrarily complex data structures. However, we focus on collections of objects, which is typical in real-world applications. More complex structures can be abstracted as internal structures of a complex object $\mathcal{O}$.

WSMS [2] models a web service as a virtual relation and applies the concept of binding patterns: A request specifies values for the *bound* attributes, and the response returns values for the *free* attributes. We use the same model: A web service $WS : \mathcal{O}_{in} \rightarrow \mathcal{C}(\mathcal{O}_{out})$ is modeled as a virtual collection of composite objects: $\mathcal{C}_{WS} : \mathcal{C}(< \mathcal{O}_{in}, \mathcal{O}_{out} >))$, where $< \mathcal{O}_{in}, \mathcal{O}_{out} >$ denotes the type of a pair of complex objects $\mathcal{O}_{in}$ and $\mathcal{O}_{out}$. A web service call is a selection $\sigma_{\mathcal{O}_{in}}$ over this collection. A feed of a web service WS is the stream of responses retrieved from WS using the same $\mathcal{O}_{in}$ at different times. Each item in the stream is of type $\mathcal{C}(\mathcal{O}_{out})$.

3.2 Mashup Query Model

A new web service mashup $\mathcal{O}_{in} \rightarrow \mathcal{C}(\mathcal{O}_{out})$ is defined as a view $\mathcal{C}(< \mathcal{O}_{in}, \mathcal{O}_{out} >)$ given as $C(O_{in}) \bowtie_P Q(P)$ where $C(O_{in})$ is a collection of input data and $Q(P)$ is a query over other web services, where P is a set of parameters for Q. A join operator $\bowtie_P$ specifies assignment of P with values taken from input object O_{in}. A call to the web service mashup executes $Q(P)$ with variables in P assigned. Maintaining feeds of this mashup is to maintain a materialized view of $C(O_{in}) \bowtie_P Q(P)$ where $C(O_{in})$ contains all the subscribed parameters.

To compose $Q(P)$, we support various query operators closed in $\mathcal{C}(\mathcal{O})$. They are categorized into stateless operators and stateful operators.

3.2.1 Stateless Operators

Stateless operators depend only on the current data $\mathcal{C}(\mathcal{O})$ to generate output $\mathcal{C}(\mathcal{O})$. The current system supports basic operators such as *Select*, *Map* (projection is a special case of map), *Join* (which takes two input collections and outputs one collection: $\mathcal{C}(\mathcal{O}) \times \mathcal{C}(\mathcal{O}) \rightarrow \mathcal{C}(\mathcal{O})$), *Aggregate*, and *Sort*. The system enables mashup developers to use expressions similar to those in SQL by providing a tool for mapping between objects and tuples, and a set of arithmetic and boolean operators. It is also possible to extend the system by user-defined plugins for custom functions, such as content similarity and summarization.

3.2.2 Stateful Operators

We define and implement *collection-based stream processing* semantics to enable information extraction by monitoring source evolution over time. The following subsection describes the operator we use for this task.

3.3 Collection-based Stream Operator

A stream of collections is modeled as a *collection-based stream* $S_C = \{C_0, C_1, C_2, \cdots\}$, where C_0 is the current collection and $\{C_1, C_2, \cdots\}$ is the history of the collections. In the context of a mashup query, a stream S_C is generated and processed by a collection-based stream operator, which results in a single collection $\mathcal{C}(\mathcal{O})$. This operator is logically composed of the following sequence of sub-operators (where $\mathcal{S}(\alpha)$ denotes the data type of a stream of items of type α):

- *Subscribe:* $\mathcal{C}(\mathcal{O}) \times T \rightarrow \mathcal{S}(\mathcal{C}(\mathcal{O}))$
- *Join:* $\mathcal{S}(\mathcal{C}(\mathcal{O})) \rightarrow \mathcal{C}(\mathcal{S}(\mathcal{O}))$
- *Select:* $\mathcal{C}(\mathcal{S}(\mathcal{O})) \rightarrow \mathcal{C}(\mathcal{S}(\mathcal{O}))$
- *Map:* $\mathcal{C}(\mathcal{S}(\mathcal{O})) \rightarrow \mathcal{C}(\mathcal{O})$

The first sub-operator, *Subscribe*, generates the collection-based stream by sampling the source for each interval T. Then *Join* joins the collections (as relations) into one collection, in which each row is logically a stream of objects, thus the resulting type is $\mathcal{C}(\mathcal{S}(\mathcal{O}))$. Then some of the rows are selected by the sub-operator *Select*. Finally, each selected row (stream) is collapsed by the *Map* sub-operator into one object, and thus the final type is $\mathcal{C}(\mathcal{O})$. Note that this sequence of operations is only logical; on the physical implementation level, the final output collection $\mathcal{C}(\mathcal{O})$ may be incrementally maintained as new collections are received. The following subsections explain *Join*, *Select*, and *Map* in more detail.

3.3.1 Join

This sub-operator logically joins the collections of the stream into one collection, in which each row is a stream of objects that describe the temporal evolution of a single real-world entity. In Example 1, each entity is a stock symbol; in Example 2, each entity is a news topic which evolves over time in related news articles.

To determine which objects should be joined together, the *Join* sub-operator takes an input parameter p which is a binary predicate that holds between two objects iff they belong to the same real-world entity. In Example 1, $p(o_i, o_j)$ holds iff both o_i, o_j have the same stock id. In Example 2, $p(o_i, o_j)$ holds iff both o_i, o_j are *similar* news articles according to some custom content similarity metric (which is added as a user-defined plugin). Note that a single news article (object) in a collection $\mathcal{C}_i$ may be split into more than one related news *threads* in the following collection $\mathcal{C}_{i+1}$; in other words, p is not necessarily a one-one relation between successive collections.

By applying p in a chained fashion through the successive collections, all the *related* objects that describe some entity get joined together to form one stream. Left (current) outer join is applied to current and previous collections so that the result consists of chains of objects starting from the current ones. Here, in real-world applications, it may not be sufficient to check this chained relation p only between objects in adjacent (time-wise) collections, it may be desirable to track entities across collections in the stream. For example, it is common that a web service returns the top-k results of the request query from its database; in this case, an entity (object) may disappear from the collection stream and then reappear again in a later collection due to ranking, even if the object does not disappear in the source database. To do that we need some notion of *allowance* in the join operation, similar in spirit to the outer join operation which can show objects even if they do not join with other ones. We introduce an allowance parameter δ as an input to the *Join* sub-operator. Two objects can be joined together by p as long as they appear at most δ *time ticks* apart from each other. This implies that if the information related to some entity (e.g. news topic) disappears from the collection stream for more than δ *ticks*, this entity is considered inactive and is dropped out of the tracking procedure. If it appears again later, it is dealt with as a new entity.

Formally, the *Join* sub-operator is denoted as:
$\bowtie_{p,\delta} (C_0, C_1, \cdots) \equiv$

$\{(o_0, o_1, \cdots) : p(o_i, o_{i+1}), o_i \in C_j, o_{i+1} \in C_{j+k}, 0 < k < \delta\}$

3.3.2 Select

This sub-operator selects some streams (rows) from the collection of streams output by the *Join* sub-operator. The selection condition is given as a function: $\mathcal{S}(\mathcal{O}) \rightarrow Boolean$. Traditional stream processing operations (e.g. aggregation) could be used within this function. In Example 2, we can select popular news threads by aggregating popularity score (ranking) over one week window.

3.3.3 Map

This sub-operator maps each selected stream (row) to a single object. To do that, it takes an input function $\mathcal{S}(\mathcal{O}) \rightarrow \mathcal{O}$. In Example 1, we can extract the most recent price as well as the average price over a specific time window. The Map operator extracts/aggregates data from each stream independently of the other streams. Aggregation across multiple streams can be done by a stateless operator following the collection-based stream operator.

4. SYSTEM ARCHITECTURE

Figure 1 illustrates the system architecture. It works as an intermediary server between web services and end users. A developer can create and publish a new web service by composing a mashup query over existing web services. An end user can subscribe to a web service mashup by registering input parameters. The system consists of three major components: the Developer Module, the Subscription Module, and the Execution Module.

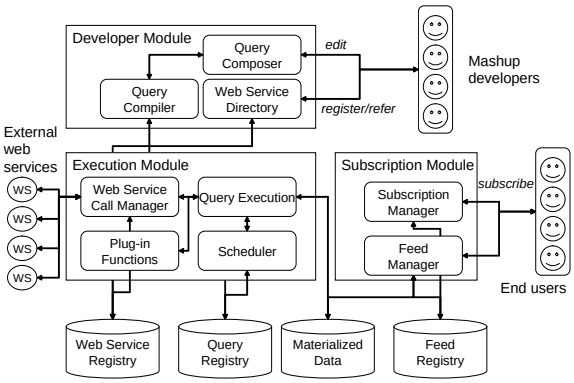

Figure 1: Mashup Feeds Architecture

The Developer Module provides a web interface for the service directory through which developers can register to mashup web services developed in the system as well as to external web services. Registered information include end points (URI), message protocol (REST,SOAP), output format (RSS, ATOM, etc), and default mapping between objects and tuples. Using a visual query composer, a developer can compose a query as a graph of operators. The query gets associated with a new web service mashup, which is registered in the web service registry.

The Subscription Module allows end users to subscribe to web service mashups. Given input parameters (i.e., O_{in}), it generates a new URL for the new feed (i.e., $C(O_{out})$). The Execution Module maintains materialized data that is used to generate the RSS feed.

5. DEMO SCENARIO

In Example 1, we first combine two web services (stock price and stock amount) and feed it into a collection-based stream operator. The parameters of operators are specified interactively through graphical forms. In the paper, however, let us describe them in a textual way to save the space. The collection-based stream operator can be specified as follows:

```
FOR EVERY '1 hour'
JOIN new.symbol = prev.symbol
      and new.price > prev.price
WINDOW 30
WHERE count(*) >= 4
ORDER BY current.price LIMIT 10
RETURN
    (current.symbol as symbol,
    current.price as price,
    count(*) as count)
```

The `FOR EVERY` clause specifies the interval of subscription. The `JOIN` clause specifies relationship between two objects (referred to as `new` and `prev`) in consecutive times. The selection part consists of a `WHERE` clause and a ranking (`ORDER BY`) clause and filters join results (a set of sequences). A function `count(*)` aggregates a sequence to count its length. The `RETURN` clause corresponds to a map sub-operator that aggregates each stream into an object (or tuple), where `current` refers to the most recent object in the stream.

In Example 2, the operator can be specified as follows:

```
FOR EVERY '1 day'
JOIN f:sim(now.text,prev.text) > 0.6
GAP 1 WINDOW 7
ORDER BY sum(f:sim(text,$query)) LIMIT 10
RETURN
    (current.title as title,
    current.text as text,
    a:summary(text) as summary,
    count(*) as count)
```

In this example, `JOIN` tracks threads of news articles where adjacent articles are similar to each other. These threads are ranked by aggregated similarity scores to a text given by the user, represented as an immutable variable `$query`. The `GAP` specifies δ value allowing 1-day gap in a news thread. This example uses user-defined plug-in functions: `f:sim` and `a:summary` (which is an aggregation function). Note that this query tracks many-to-many relationships. Our implementation incrementally maintains such join results, which are internally represented as a graph structure, and enumerates top-k result for each time step.

We will simulate external web services to demonstrate how our mashup feeds can track their changes.

6. REFERENCES

[1] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, Nov/Dec 2004.

[2] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proc. of VLDB 2006*, pages 355–366, September 2006.