WETSUIT: An Efficient Mashup Tool for Searching and Fusing Web Entities

Stefan Endrullis
University of Leipzig
endrullis@informatik.uni-leipzig.de

Andreas Thor
University of Leipzig
thor@informatik.uni-leipzig.de

Erhard Rahm
University of Leipzig
rahm@informatik.uni-leipzig.de

ABSTRACT

We demonstrate a new powerful mashup tool called WETSUIT (Web EnTity Search and fUsIon Tool) to search and integrate web data from diverse sources and domain-specific entity search engines. WETSUIT supports adaptive search strategies to query sets of relevant entities with a minimum of communication overhead. Mashups can be composed using a set of high-level operators based on the Java-compatible language Scala. The operator implementation supports a high degree of parallel processing, in particular a streaming of entities between all data transformation operations facilitating a fast presentation of intermediate results. WETSUIT has already been applied to solve challenging integration tasks from different domains.

1. INTRODUCTION

Mashups follow a light-weight and programmatic approach for on-the-fly data integration which is complementary to common database-oriented approaches, such as data warehouses or query mediators. Mashups have become quite popular to integrate both web and enterprise data and many tools and frameworks have been developed [3]. Typical systems such as Yahoo! Pipes [1], Damia [5], or Semantic Web Pipes [4] offer a powerful and easy-to-use interface allowing even inexperienced users to access and integrate data they need. To this end they provide operators (e.g., filter, merge, or join) to process sets of entities in user-specified workflows.

However, current mashup dataflows are mostly comparatively simple and do not yet exploit the full potential of programmatic data integration, e.g., to analyze larger sets of web data. In particular they are limited to simple query approaches to retrieve relevant entities from hidden data sources or entity search engines. Furthermore, they typically provide only simple approaches to deal with dirty data, e.g., for entity resolution.

To overcome such deficiencies we have developed a new mashup framework called WETSUIT (Web EnTity Search and fUsIon Tool)[2] for the integration of web data, e.g., product offers from e-commerce shops, citation data from bibliographic web databases, etc.. Its design is also based on experiences with our earlier mashup approaches [6, 7]. Distinctive features of WETSUIT include the following:

- Mashups are specified in a domain specific language embedded in the Java-compatible language *Scala*[3]. The language comprises high-level set-oriented operators facilitating compact workflow definitions (scripts). The embedding in a general purpose language (Scala or Java) provides significant advantages over other mashup systems. First, developers can more easily reuse existing (Java) code and can implement specific procedures that are not captured by the script language, e.g., an HTML parser for screenscraping web sites. Second, mashup workflows compile into standard Java bytecode and can thus be used in other Java or Scala programs. Third, mashup developers can benefit from common development tools for Java such as IDEs, debuggers, or unit tests.
- WETSUIT provides powerful operators for entity resolution and advanced search strategies. In particular, adaptive search strategies can be employed to retrieve larger sets of relevant entities from entity search engines with a minimum of communication costs [2]. Queries are determined by source-specific query generators and the most promising queries are executed based on the characteristics of input entities and previous query results.
- WETSUIT supports a high degree of parallel processing, in particular a streaming of entities between all data transformation operations facilitating a fast presentation of intermediate results. This feature supports highly interactive web applications where first results are quickly shown to the user.
- WETSUIT has already been applied to solve challenging integration tasks from different domains. We will demonstrate two such use cases using bibliographic and movie-related web data. The bibliographic mashups determines the top-cited papers (based on Google Scholar citations) for any conference, journal or author listed at DBLP. The movie mashup determines all current movie offers of a city and filters them according to IMDB information such as the average user rating.

[1] `http://pipes.yahoo.com`

[2] `http://dbs.uni-leipzig.de/wetsuit`

[3] `http://www.scala-lang.org/`

Proceedings of the VLDB Endowment, Vol. 5, No. 12

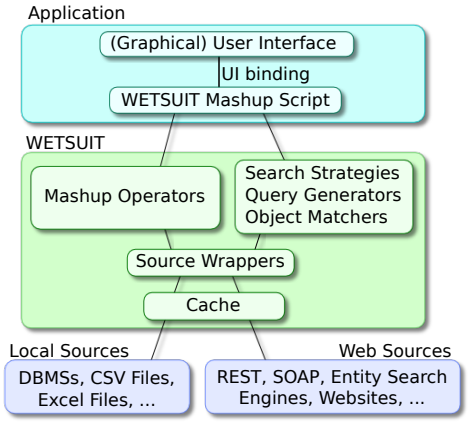

Figure 1: System architecture

Next, we give a brief overview of WETSUIT's architecture and mashup language (Section 2). Section 3 explains how we support different search strategies. Finally, we present our demonstration scenarios (Section 4).

2. WETSUIT

Figure 1 illustrates the three-layered architecture of mashup applications built with WETSUIT. A mashup application itself basically consists of a mashup workflow and a (graphical) user interface. WETSUIT supports the automatic generation of simple GUIs, but developers can also manually design appropriate user interfaces. WETSUIT provides a library of wrappers to access diverse web sources as well as local data sources such as databases or spreadsheets.

In our framework mashup workflows are specified in a domain specific language (DSL) embedded in *Scala*. In contrast to classical programming languages, Scala provides a series of powerful and flexible language features (e.g., implicit casts, operator overloading, infix operators, implicit type inference) enabling the design of concise, well readable, script-like languages as exploited for our workflow language.

In WETSUIT mashup workflows are described through sequences of mashup operators which may together form complex data flow graphs. Table 1 shows a list of the most important operators supported in our framework. We can roughly divide them into user interaction operators and data transformation and integration operators. The former kind is responsible for presenting results to the user and for acquiring user input, whereas the latter kind performs the actual data handling.

All operators are set-oriented, i.e., they consume and/or produce sets of entities. Entities are either numeric values or instances of Scala (or Java) classes which allow for an intuitive and object-oriented modeling of entity types. Figure 2 shows a simple type definition for entity type `Publication` consisting of three attributes: `title`, `authors`, and `year`.

A major feature of WETSUIT is the capability of quickly presenting intermediate results. To this end, our mashup framework supports inter-operator and intra-operator parallelism in a completely transparent fashion to the mashup developer. The former one is implemented by executing all operators of a mashup workflow in a pipelined manner, i.e., results of each operator immediately stream to subsequent operators so that all operators can work in parallel as long as they have entities to process. Another speedup can be achieved through data parallelism within operators, i.e., several mashup operators process multiple input entities at the same time by exploiting different cores of modern CPUs.

Table 1: Selected operators of WETSUIT

Operator	Definition
`inputOne`(label) / `inputMany`(label)	asks user for one / many input value(s); *label* denotes component caption
`selectOne`(label) / `selectMany`(label)	displays intermediate results and asks user to select one / many of them
`outputOne`(label) / `outputMany`(label)	presents results
`map`(f)	maps each input entity to a new entity using the mapping function f
`flatMap`(f)	maps each input entity to a set of new entities using the mapping function f
`filter`($cond$)	filters input entities based on the filter condition $cond$
`groupBy` ($groupAttr$)	groups the input entities by $groupAttr$; `groupBy` has to be followed by `filterTop` or `aggregateValue`
`filterTop` (n) `by` ($orderAttr$)	for each group / whole input set: filters the top n elements on $orderdAttr$
`aggregateValue` ($value$) `via` (agg, [rev])	for each group / whole input set: aggregates $value$ using aggregation function agg and its reverse operation rev (rev needed if input entities can be revoked)
`findAt` (ese)	searches the input entities at ESE ese using an `implicitly` defined search strategy
`matchWith` set_2 `using` ($matcher$)	matches the input entities against entities of set_2 by using match strategy $matcher$
`union`, `intersect`, `diff`	classical set operators
`join` set_2 `on` ($theta$)	θ-join of two entity sets using the binary function $theta$ as join condition

A stumbling block for a fast presentation of first results are blocking operators such as `diff`, `aggregateValue`, and `filterTop`. These operators usually require the availability of the entire input set in order to perform their operation and to generate their results thereby breaking pipelining parallelism. To overcome this limitation our framework implements these operators differently and allows them to produce preliminary results which can be later updated or revoked by these operators when new input entities arrive. This feature allows for instance a citation application to continuously update the top-cited papers as new citation data arrives or an eCommerce application to update the lowest-priced offers for a product of interest. We can thus provide users quickly with first results while additional search queries are being processed in the background. Updates and revokes are automatically propagated to the subsequent operators as well.

Figure 3 shows a simple WETSUIT mashup for online citation analysis. The workflow starts with asking the user to enter an author name. The authors matching the name are looked up in DBLP[4] and presented to the user. When the user selects one of the authors all his/her DBLP publications are determined and handed over to the search strategy, initiated by operator `findAt(Scholar)`. This complex operator tries to find all corresponding publications at Google Scholar[5] by generating suitable queries, sending them to Scholar and matching the results against the input publications. The result of the entity search is a set of correspondences consisting of a `domain` entity (DBLP publication),

[4] `http://www.informatik.uni-trier.de/~ley/db/`
[5] `http://scholar.google.com/`

```
case class Publication(title: String, authors: String,
    year: Int)
```

Figure 2: Sample type definition in Scala

```
inputOne ("Author name: ", "")
flatMap (name => Dblp.Author.where("name like ?",
    "%"+name+"%"))
selectOne ("Select one author")
flatMap (_.publications)
findAt (Scholar)
groupBy (_.range) filterTop 1 by (_.sim)
groupBy (_.domain) aggregateValue (_.range.citations)
    via (_+_, _-_)
outputMany ("DBLP pubs with citations")
```

Figure 3: Mashup for an online citation service

```
object Scholar extends Ese {
  type ResultType = Publication

  // declare search predicates
  val keywords = predicate("as_q")
  val authors  = predicate("as_sauthors")

  // how queries are mapped to URLs
  val queryMapper = UrlParamQueryMapper(
    urlPrefix = "http://scholar.google.de/scholar?",
    and       = " ",    // separator for conjunctions
    or        = " OR ") // separator for disjunctions

  // parser for search engine results
  val resultParser = ScholarParser
}
```

Figure 4: Definition of an entity search engine (simplified)

a `range` entity (Scholar publication), and a similarity value `sim`. In order to avoid that one Scholar publication may match to multiple DBLP publications the `filterTop` operation filters out all correspondences but the best match for each range entity (Scholar publication). Finally, the second `groupBy` line aggregates the Scholar citation counts for each domain entity (DBLP publication) which are afterwards presented to the user. A screenshot of this mashup application is shown in Figure 7.

3. SEARCH STRATEGIES

A huge amount of information on the web is hidden behind entity search engines (ESE) or hidden databases such as Google Scholar or IMDB[6]. Such sources provide access only to specific types of entities (e.g., publications or movies) and typically offer multiple search predicates to specify search conditions (e.g., on the publication title or authors).

Figure 4 shows an example for the definition of a Google Scholar-like entity search engine in WETSUIT. The search engine provides access to entities of type `Publication` and supports the search predicates `keywords` and `authors`. The `queryMapper` describes how queries are mapped from a logical expression (our internal query format) to URLs of the search engine. In the example we use a query mapper which composes the query URL in the same way a web browser would compose the request from a classical web form after the search button was pressed. Some ESEs such as Scholar support logical operators (e.g., `AND` and `OR`) within input fields. In WETSUIT query generators may exploit such operators, especially the disjunction of terms is promising in order to search for several entities within one query. The actual format for these operators can be specified as parameters `and` and `or` of the query mapper, e.g., to specify that spaces between search terms denote a conjunction.

In order to enable efficient and effective entity search our framework goes far beyond the simple search approaches of common mashup tools. WETSUIT supports sophisticated entity *search strategies* [2] exploiting different *query generators* [1] to find sets of entities with a minimum of queries. Search strategies are specific for a certain type of input entities and a certain entity search engine. For example, to quickly retrieve the citations of all publications of an author WETSUIT can start with a single author query rather than querying every publication.

Figure 5 shows a definition of a search strategy that can be used to search for `Publication`s at the publication search engine `Scholar` from Figure 4. The first half of the code defines two query generators. A query generator represents an algorithm for generating queries for a set of input entities, in our case `Publication`s. The first one, named `fallback`, is a naive query generator, i.e., it creates one query per input entity. Specifically, it maps the first author and all title keywords of each input publication to the search predicates `authors` and `keywords`, respectively. The query generator `fv` represents a so-called frequent value query generator which determines frequently occurring authors in the input set to formulate queries with them. The property `minEntities = 2` ensures that each generated query of `fv` covers at least two input entities to ensure a minimal efficiency.

Below the two query generators the `titleSim` similarity measure is defined that can be used for matching publications (entity resolution). It calculates the trigram similarity between two publication titles. Additional similarity measures and their combination can be defined analogously.

Search strategies can be build upon multiple query generators in order to find more relevant results and/or make the entity search more efficient. Figure 5 (lower part) shows an example for the definition of an implicit search strategy that is automatically invoked by calling operator `findAt` for `Scholar`. The search strategy applies the two query generators sequentially. It starts searching for the input entities using query generator `fv` and continues the search using `fallback`. The algorithm stops searching for entities that have been found at least once (`maxResults = 1`) or searched already two times (`maxTrials = 2`). Matching pairs between input entities and search results are identified by using the similarity measure `titleSim` and a threshold of 0.8. Similar matching rules can be applied within the `matchWith` operator. A more detailed description and evaluation of search strategies can be found in [2]. In particular, we outline a fully adaptive strategy of WETSUIT that analyzes the input entities and search results to continuously determine the most promising search queries to execute next.

[6] `http://imdb.com/`

```
object PublicationSearchStrategy extends
    SearchStrategyFor[Publication, Scholar.type] {

  // query generators
  val fallback = naiveQg(
    ese.authors  <= {_.authors.take(1)},
    ese.keywords <= {_.title.keywords}
  )
  val fv = frequentValueQg(
    ese.authors <= {_.authors}
  ) using (minEntities = 2)

  // similarity measures
  val titleSim = sim { (pub1, pub2) =>
      triGramSim(pub1.title, pub2.title) }

  // implicit search strategy
  implicit val seq = sequentialStrategy(fv, fallback).
    using (maxTrials = 2, maxResults = 1,
           matcher = (titleSim >= 0.8))
}
```

Figure 5: Example for a search strategy

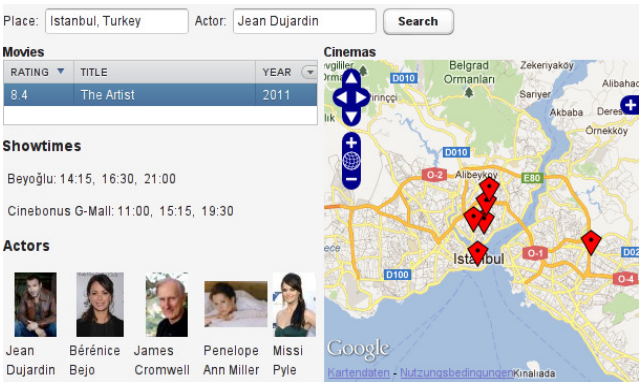

Figure 6: Screenshot of the Cinema Mashup

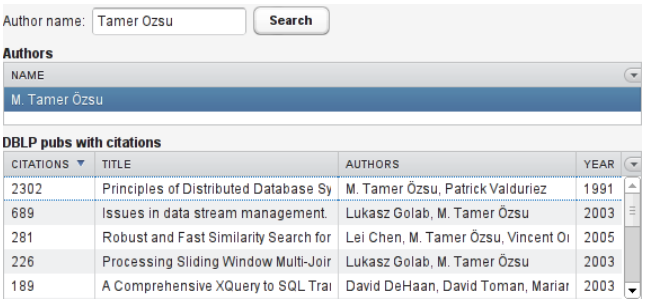

Figure 7: Screenshot of the Online Citation Service

4. DEMONSTRATION DESCRIPTION

During the demonstration we will illustrate how WETSUIT can be employed for mashup development. To this end, we provide two scenarios (Cinema and Publications) that showcase WETSUIT's core functionality.

Scenario "Cinema": We will demonstrate an example cinema mashup built with WETSUIT. The audience can search for an actor and a city and the mashup will show movie offers in the city starring the specified actor. For each movie, screening times, locations, and actor information will be displayed (see Figure 6). Movies are ranked by their IMDB rating. To this end, the cinema mashup retrieves all movie offers of the city using Google's cinema search. It then searches for the corresponding movies in the IMDB. After the audience has been convinced of the correct and reliable operation of the script, the audience can change the script's functionality. First, an additional filter can be incorporated that shows movies with an IMDB rating above a given threshold only. The threshold can either be hard-coded in the mashup or realized as an additional parameter. In the latter case the audience will experience WETSUIT's automatic GUI adjustments, i.e., an additional input field will be generated based on the mashup script. Second, the movie ranking can be modified. We will change the script so that it additionally retrieves the number of won Oscars for all actors and producers for each movie. Movies can then be ranked based on their number of Oscars.

Scenario "Publications": The second scenario illustrates the effect of search strategies with the help of our online citation mashup. For all DBLP publications of a specified author or venue, the mashup retrieves matching Google Scholar entries and summarizes their citation counts (see Figure 7). On the one hand, a simple search strategy that only sends one query with the author's or venue's name is sufficient for a quick overview to identify the top-cited papers. On the other hand, extensive evaluation of all papers for subsequent data analysis requires an exhaustive search strategy because small changes in citation counts may already have significant impact on derived statistics, e.g., h-index. It is therefore crucial to retrieve all matching publications in Google Scholar. The audience will experience the runtime and data quality differences between both strategies. Finally, we will demonstrate a sophisticated strategy that executes the most-promising queries based on the performance (i.e., effectiveness and efficiency) of previously executed queries. We thereby showcase WETSUIT's approach of cost-effective search strategies, i.e., achieving the (almost) complete result with a minimal number of queries.

5. REFERENCES

[1] S. Endrullis, A. Thor, and E. Rahm. Evaluation of Query Generators for Entity Search Engines. In *Int. Workshop USETIM*, 2009.

[2] S. Endrullis, A. Thor, and E. Rahm. Entity Search Strategies for Mashup Applications. In *ICDE*, pages 66–77, 2012.

[3] G. Lorenzo et al. Data Integration in Mashups. *SIGMOD Record*, 38(1):59–66, 2009.

[4] D. L. Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni. Rapid Prototyping of Semantic Mash-ups through Semantic Web Pipes. In *WWW*, pages 581–590, 2009.

[5] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh. Damia: Data Mashups for Intranet Applications. In *SIGMOD*, pages 1171–1182, 2008.

[6] A. Thor, D. Aumueller, and E. Rahm. Data integration support for mashups. In *Int. Workshop on Information Integration on the Web*, 2007.

[7] A. Thor and E. Rahm. CloudFuice: A Flexible Cloud-Based Data Integration System. In *ICWE*, pages 304–318, 2011.