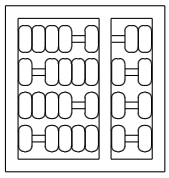

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Retrieving and Storing Data from Folksonomies

Hugo Alves *André Santanchè*

Technical Report - IC-12-15 - Relatório Técnico

May - 2012 - Maio

Retrieving and Storing Data from Folksonomies

Hugo Alves* André Santanchè†

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Pesquisa desenvolvida com suporte financeiro parcial do CNPq e FAPESP

†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

Contents

1 Introduction

The popularization of web-based systems offering services for content storage, indexing and sharing fostered a rapid growth of content available on-line. There are more than 5 billion images hosted on Flickr[1] and more than 180 million URL addresses on Delicious[2]. These systems increasingly rely on tag-based metadata to organize and index all the amount of data. The tags are provided by users connected in social networks, who are free to use any word as tag; there is no central control. The term folksonomy – combining the words "folk" and "taxonomy" [8] – has been used to characterize the product which emerges from this tagging in a social environment.

In order to analyze, index and classify their content, web systems compare tags attached to resources. Instead of considering the semantics of each tag in the comparison, tag-based systems usually rely on string matching approaches. While ontologies are increasingly adopted to enrich tags semantics, one common problem with the proposals to associate tags to formal ontologies concerns their unidirectionality, i.e., ontologies improve tags semantics, or the implicit/potential semantics of folksonomies is extracted to produce ontologies.

Differently from traditional techniques, we proposed a fusion approach, called *folksonomized ontology* (FO), which goes beyond this unidirectional perspective [3]. In one direction, the ontologies are "folksonomized", i.e., the latent semantics from the folksonomic tissue is extracted and fused to ontologies. On the other direction, the knowledge systematically organized and formalized in ontologies gives structure to the folksonomic semantics, enhancing operations involving tags, e.g., content indexation and discovery. The folksonomic data fused to an ontology will tune it up to contextualize inferences over the repository.

In our fusion approach both ontologies and folksonomies are enriched in the process. This symbiosis is explored to:

Tag disambiguation: by finding groups of related tags and mapping them to ontology concepts, the FO can be applied to disambiguate tags and find the ones that are more related, going beyond statistical analyses by using semantic similarity metrics.

Tag suggestion: the current folksonomy systems consider only co-occurrence information to suggest related tags to users; a FO has a richer set of semantic relations among concepts, supporting suggestion of tags that were not used together before – folksonomies cannot do that.

Semantic similarity: a FO can support the computation of semantic similarity between concepts and, by extension, between tags; so, they can expand the usual techniques that focus only at syntactical similarity and co-occurrence of tags, achieving better results in discovery operations.

Ontology evolvement: a FO can be used to find missing relations in ontologies; the high co-occurrence between two groups of tags, and their corresponding concepts, can indicate a necessary relation in the ontology, if it does not exist yet.

[1] `http://blog.flickr.net/en/2010/09/19/5000000000/` - retrieved on November, 2011

[2] `http://blog.delicious.com/blog/2008/11/delicious-is-5.html` - retrieved on November, 2011

In order to build a practical tool to validate our proposal, we have implemented a software module to access and collect data from folksonomy-based web systems. We confronted the model adopted by each system with models proposed in the literature, in order to propose a generic model to represent and store the collected data.

The goal of this technical report is to detail this work. In Section 2 we synthesize related work concerning formal models to represent folksonomies. In Section 3 we discuss implementation aspects of our module, which interacts with these systems to collect and store folksonomic data. In Section 4 we analyse the approach adopted by folksonomy-based systems to represent and store their folksonomies, including their Application Programming Interfaces (APIs). The systems that will be analysed here are Delicious [1] and Flickr [2].

2 Formal Model for Folksonomies

In folksonomy-based systems, users can attach a set of tags to resources. These tags are not tied to any central vocabulary, so the users are free to create and combine tags. Some strengths of folksonomies are their easiness of use and the fact that they reflect the vocabulary of their users [5]. In a first glimpse, tagging can transmit the wrong idea of a poor classification system. However, thanks to its simplicity, users are producing millions of correlated tags. It is a shift from classical approaches – in which a restricted group of people formalize a set of concepts and relations – into a social approach – in which the concepts and their relations emerge from the collective tagging [7]. In order to perform a systematic folksonomy analysis, to subsidize the extraction of its potential semantics, researchers are proposing models to represent its key aspects. Gruber [4] models a folksonomy departing from its basic "tagging" element, defined as the following relation:

$$Tagging(object, tag, tagger, source) \tag{1}$$

In which *object* is the described resource, *tag* is the tag itself – a string containing a word or combined words –, *tagger* is the tag's author, and *source* is the folksonomy system, which allows to record the tag provenience (e.g., Delicious, Flickr etc.).

In order to formalize a folksonomy Mika [6] departs from a tripartite graph with hyperedges. There are three disjoint sets representing the vertices:

$$T = \{t_1, \ldots, t_k\}, U = \{u_1, \ldots, u_l\}, R = \{r_1, \ldots, r_m\} \tag{2}$$

In which the sets *T, U* and *R* correspond to tags, users and resources sets respectively.

A folksonomy system is a set of annotations A relating these three sets:

$$A \subseteq T \times U \times R \tag{3}$$

The folksonomy itself is a tripartite hypergraph:

$$H(T) = \langle V, E \rangle \tag{4}$$

In which $V = T \cup U \cup R$, and $E = \{\{t, u, r\} \mid (t, u, r) \in A\}$

The folksonomy analysis can be simplified and directed by reducing this tripartite hypergraph into three bipartite graphs: TU relating tags to users, UR relating users to resources and TR relating tags to resources [6]. A graph TT is a relevant extension of this model for representing relations between tags. It allows to represent the co-occurrence of tags. The same approach can be applied to the user and resource sets.

3 Implementation

In this section we describe the tool we have implemented to retrieve data from Delicious and Flickr, as well as the database model. The implementation adopted the python language[3]. The data was stored by using the SQLite[4] database manager.

The access of Flickr data required the implementation of the code to handle its protocol and to treat the results of the requests. The module to retrieve the data from Delicious, on the other hand, adopted a third-party library: DeliciousAPI[5].

During the development, we faced an unexpected behavior of the library. The reason was a change in the structure of Delicious' pages. This is still a challenge to be faced in web services research, mainly in public web services: whenever servers change their interfaces, the clients will break if there is not backwards compatibility. In order to fix it, we developed a patch to adjust the access, contributing to the community to fix the error[6].

3.1 Database Model

As mentioned before, our database model results from a comparative analysis of related work and models adopted by folksonomy based systems. The logical modeling is depicted in the Figure 1. There are three main entities – `User`, `Resource` and `Tag` – following the model presented in Section 2. The `Resource` entity was specialized to better characterize its representation in the systems Flickr (`Photo`) and Delicious (`URL`). As the database was designed to simultaneously afford tags of many systems, the `Source` entity keeps track of the origin of the tag. This is an important information, since our algorithms were designed to work with a single folksonomy system each time.

The physical modeling is based on the logical one. However, it includes control tables and flags to indicate: users already processed, resources already visited, and so on. Some auxiliary tables – prefixed by `count_` – record the counts of occurrences or co-occurrences of analyzed items; they will support statistics produced by our system. It is composed of 13 tables, as we further detail:

control: records control data related to the execution of the process, like the timestamps of the last requisition to the systems. `(name TEXT, value TEXT)`

count_resource: records the count of each resource. `(rid INTEGER, sid INTEGER, count INTEGER)`

[3] `http://www.python.org/`
[4] `http://www.sqlite.org/`
[5] `https://github.com/quuxlabs/DeliciousAPI`
[6] `https://github.com/quuxlabs/DeliciousAPI/commit/1cea76941797d6807ac8411b0e8437aa92a35aa5`

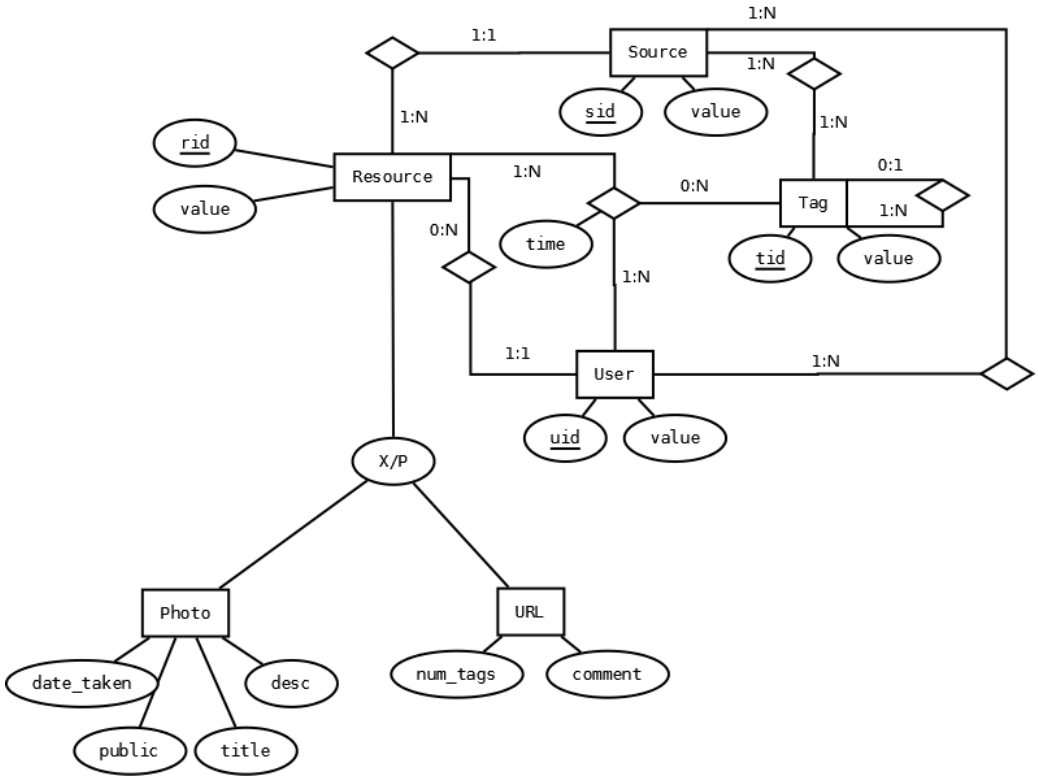

Figure 1: Logical modeling of the database

count_rt: records the count of each pair (resource, tag). `(rid INTEGER, tid INTEGER, sid INTEGER, count INTEGER)`

count_tag: records the count of each tag. `(tid INTEGER, sid INTEGER, count INTEGER)`

count_tt: records the count of each pair of tags. `(t1 INTEGER, t2 INTEGER, sid INTEGER, count INTEGER)`

count_tu: records the count of each pair (tag, user). `(tid INTEGER, uid INTEGER, sid INTEGER, count INTEGER)`

count_ur: records the count of each pair (user, resource). `(uid INTEGER, rid INTEGER, sid INTEGER, count INTEGER)`

count_user: records the count of each user. `(uid INTEGER, sid INTEGER, count INTEGER)`

resource: records each resource, which has an internal id `rid` and a reference to the specific identification in the source system – e.g., a URL for Delicious or a Flickr internal identifier – in the `value` field. `(rid INTEGER PRIMARY KEY, sid INTEGER, value TEXT, done NUMERIC)`

source: records each source, which has an internal id `sid` and the specification of the source (e.g., delicious or flickr) in the `value` field. `(sid INTEGER, value TEXT)`

tag: records each tag, assigning an internal id `tid` to each one. `(tid INTEGER, sid INTEGER, value TEXT)`

tagging: records each tagging triple (resource, tag, user) in a given source. `(sid INTEGER, uid INTEGER, rid INTEGER, tid INTEGER, time TEXT)`

user: records each user, assigning for each value (string of the username) an internal id. `(uid INTEGER, sid INTEGER, value TEXT, done NUMERIC)`

The complete schema of the database is as follows:

```
1  CREATE TABLE source (sid INTEGER PRIMARY KEY, value TEXT);
2  CREATE TABLE control (name TEXT, value TEXT);
3  CREATE TABLE user (uid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE
       NOCASE, done NUMERIC, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid,
       value));
4  CREATE TABLE tag (tid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE
       NOCASE, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value));
5  CREATE TABLE resource (rid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE
       NOCASE, done NUMERIC, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid,
       value));
6  CREATE TABLE tagging (sid INTEGER, uid INTEGER, rid INTEGER, tid INTEGER, time
       TEXT, FOREIGN KEY(sid) REFERENCES source(sid), FOREIGN KEY(uid) REFERENCES
       user(uid), FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(tid)
       REFERENCES tag(tid), UNIQUE(sid, uid, rid, tid));
7  CREATE TABLE 'count_tu' (tid INTEGER, uid INTEGER, sid INTEGER, count INTEGER,
       FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(uid) REFERENCES user(uid),
       FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(tid, uid, sid));
8  CREATE TABLE 'count_ur' (uid INTEGER, rid INTEGER, sid INTEGER, count INTEGER,
       FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(rid) REFERENCES
       resource(rid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(uid, rid,
       sid));
9  CREATE TABLE 'count_rt' (rid INTEGER, tid INTEGER, sid INTEGER, count INTEGER,
       FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(tid) REFERENCES
       tag(tid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(rid, tid, sid));
10 CREATE TABLE 'count_tt' (t1 INTEGER, t2 INTEGER, sid INTEGER, count INTEGER,
       FOREIGN KEY(t1) REFERENCES tag(tid), FOREIGN KEY(t2) REFERENCES tag(tid),
       FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(t1, t2, sid));
11 CREATE TABLE 'count_tag' (tid INTEGER, sid INTEGER, count INTEGER, FOREIGN
       KEY(tid) REFERENCES tag(tid), FOREIGN KEY(sid) REFERENCES source(sid),
       UNIQUE(tid, sid));
12 CREATE TABLE 'count_user' (uid INTEGER, sid INTEGER, count INTEGER, FOREIGN
       KEY(uid) REFERENCES user(uid), FOREIGN KEY(sid) REFERENCES source(sid),
       UNIQUE(uid, sid));
13 CREATE TABLE 'count_resource' (rid INTEGER, sid INTEGER, count INTEGER, FOREIGN
       KEY(rid) REFERENCES resource(rid), FOREIGN KEY(sid) REFERENCES source(sid),
       UNIQUE(rid, sid));
```

3.2 Tool Model

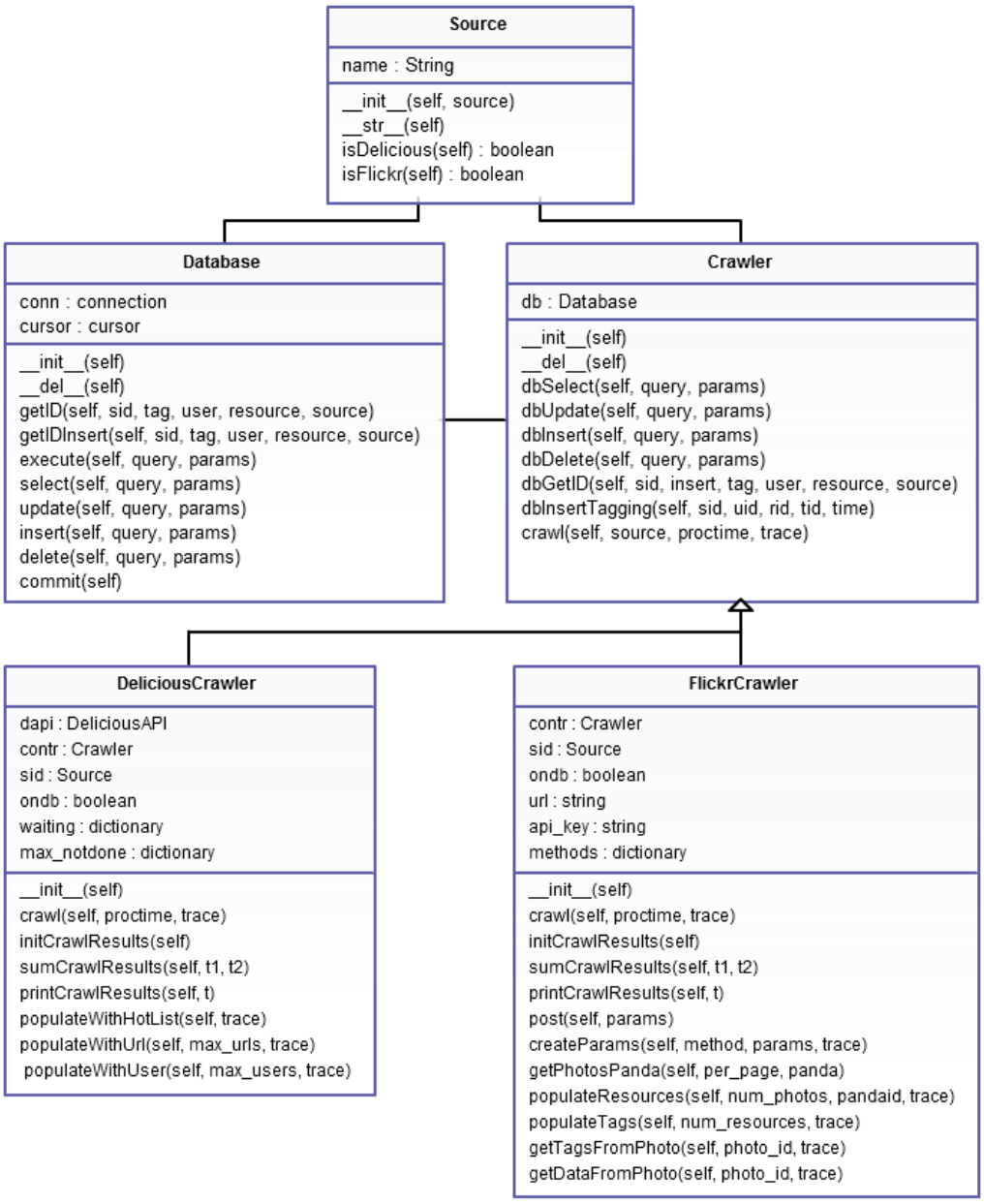

Figure 2: Class diagram

Figure 2 presents a UML diagram of the main classes of our tool. The five classes depicted in the figure are:

Source: represents a specific folksonomy system.

Database: abstracts and centralizes all database operations.

Crawler: abstract class whose instances represent a crawler a its operations.

DeliciousCrawler: implements the crawler operations for the Delicious context.

FlickrCrawler: implements the crawler operations for the Flickr context.

The `Crawler` abstract class makes simpler to extend the system, since it standardize the API to the crawler mechanism. Each new system will require only a new implementation extending `Crawler`.

3.3 Source Code

In this section we present the source of the tool, according the model presented in the previous section. Each block contains a class and comments explaining its functionality.

```
1  import sys                                # print without \n
2  import urllib                             # http request and manipulation
3  import sqlite3                            # database
4  import time, datetime                     # time processing
5  from xml.etree.ElementTree import parse # parser xml
6  import deliciousapi                       # delicious data access
7  import random
8
9  """ Object Source - encapsulates the current folksonomy system
10 """
11 class Source(object):
12     def __init__(self, source = 'all'):
13         if source in ['all', 'flickr', 'delicious']:
14             self.name = source
15         else:
16             self.name = 'all'
17
18     def __str__(self):
19         return self.name
20
21     """ true if the source is delicious
22     """
23     def isDelicious(self):
24         return self.name == 'delicious'
25
26     """ true if the source is flickr
27     """
28     def isFlickr(self):
29         return self.name == 'flickr'
```

```
30 """ Object Database - encapsulates the database operations
31 """
```

```
32 class DataBase(object):
33
34     """ constructor - try to create the tables if they don't exist
35         dbfile - name of the database file
36     """
37     def __init__(self, dbfile = 'folk.sqlite'):
38         self.conn = sqlite3.connect(dbfile)
39         self.conn.text_factory = str
40         self.cursor = self.conn.cursor()
41         self.cursor.execute('CREATE TABLE IF NOT EXISTS user (uid INTEGER PRIMARY
               KEY, sid INTEGER, value TEXT COLLATE NOCASE, done NUMERIC, FOREIGN
               KEY(sid) REFERENCES source(sid), UNIQUE(sid, value))')
42         self.cursor.execute('CREATE TABLE IF NOT EXISTS tag (tid INTEGER PRIMARY
               KEY, sid INTEGER, value TEXT COLLATE NOCASE, FOREIGN KEY(sid)
               REFERENCES source(sid), UNIQUE(sid, value))')
43         self.cursor.execute('CREATE TABLE IF NOT EXISTS source (sid INTEGER
               PRIMARY KEY, value TEXT COLLATE NOCASE)')
44         self.cursor.execute('CREATE TABLE IF NOT EXISTS resource (rid INTEGER
               PRIMARY KEY, sid INTEGER, value TEXT COLLATE NOCASE, done NUMERIC,
               FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value))')
45         self.cursor.execute('CREATE TABLE IF NOT EXISTS tagging (sid INTEGER, uid
               INTEGER, rid INTEGER, tid INTEGER, time TEXT, FOREIGN KEY(sid)
               REFERENCES source(sid), FOREIGN KEY(uid) REFERENCES user(uid),
               FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(tid)
               REFERENCES tag(tid))')
46         self.cursor.execute('CREATE TABLE IF NOT EXISTS control (name TEXT, value
               TEXT)')
47         self.cursor.execute('CREATE TABLE IF NOT EXISTS count_tag (tid INTEGER,
               sid INTEGER, count INTEGER, FOREIGN KEY(tid) REFERENCES tag(tid),
               FOREIGN KEY(sid) REFERENCES source(sid))')
48         self.cursor.execute('CREATE TABLE IF NOT EXISTS count_user (uid INTEGER,
               sid INTEGER, count INTEGER, FOREIGN KEY(uid) REFERENCES user(uid),
               FOREIGN KEY(sid) REFERENCES source(sid))')
49         self.cursor.execute('CREATE TABLE IF NOT EXISTS count_resource (rid
               INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(rid) REFERENCES
               resource(rid), FOREIGN KEY(sid) REFERENCES source(sid))')
50         self.cursor.execute('CREATE TABLE IF NOT EXISTS count_tu (tid INTEGER,
               uid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(tid) REFERENCES
               tag(tid), FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(sid)
               REFERENCES source(sid))')
51         self.cursor.execute('CREATE TABLE IF NOT EXISTS count_ur (uid INTEGER,
               rid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(uid) REFERENCES
               user(uid), FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN
               KEY(sid) REFERENCES source(sid))')
52         self.cursor.execute('CREATE TABLE IF NOT EXISTS count_rt (rid INTEGER,
               tid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(rid) REFERENCES
               resource(rid), FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(sid)
               REFERENCES source(sid))')
```

```
53          self.cursor.execute('CREATE TABLE IF NOT EXISTS count_tt (t1 INTEGER, t2
                INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(t1) REFERENCES
                tag(tid), FOREIGN KEY(t2) REFERENCES tag(tid), FOREIGN KEY(sid)
                REFERENCES source(sid))')
54          self.conn.commit()
55
56
57      """ destructor - close the resources
58      """
59      def __del__(self):
60          self.cursor.close()
61          self.conn.close()
62
63
64      """ find the id of the given entity
65          if more than one entity is given, the order [tag; user; resource; source]
                is important
66          return the id or 'None' if failed
67      """
68      def getID(self, sid, tag = None, user = None, resource = None, source = None):
69          res = None
70          try:
71              if tag is not None:
72                  self.cursor.execute('select tid from tag where value = ? and sid
                        = ?', (tag.lower(), sid,))
73              elif user is not None:
74                  self.cursor.execute('select uid from user where value = ? and sid
                        = ?', (user.lower(), sid,))
75              elif resource is not None:
76                  self.cursor.execute('select rid from resource where value = ? and
                        sid = ?', (resource.lower(), sid,))
77              elif source is not None:
78                  self.cursor.execute('select sid from source where value = ?',
                        (source.lower(),))
79              res = self.cursor.fetchone()
80              if res is not None: res = res[0]
81          except:
82              pass
83          return res
84
85
86      """ return the id of the given entity, if it isn't in the database, insert
            and return the id
87          if more than one entity is given, the order [tag; user; resource; source]
                is important
88          return a pair <id, already>:
89             id - the id of the given entity - None if failed
90             already - [boolean] True if the entity was already in the db; False
                   otherwise - None if failed
```

```
91      """
92      def getIDInsert(self, sid, tag = None, user = None, resource = None, source =
            None):
93          # try to find the id in database
94          id = self.getID(sid, tag = tag, user = user, resource = resource, source
                = source)
95          if id is not None: return id, True
96
97          # if it isn't, try to insert
98          try:
99              if tag is not None:
100                 self.cursor.execute('insert into tag values(NULL, ?, ?)', (sid,
                        tag.lower(),))
101                 self.conn.commit()
102                 return self.getID(sid, tag = tag), False
103             elif user is not None:
104                 self.cursor.execute('insert into user values(NULL, ?, ?, 0)',
                        (sid, user.lower(),))
105                 self.conn.commit()
106                 return self.getID(sid, user = user), False
107             elif resource is not None:
108                 self.cursor.execute('insert into resource values(NULL, ?, ?, 0)',
                        (sid, resource.lower(),))
109                 self.conn.commit()
110                 return self.getID(sid, resource = resource), False
111             elif source is not None:
112                 self.cursor.execute('insert into source values(NULL, ?)',
                        (source.lower(),))
113                 self.conn.commit()
114                 return self.getID(sid, source = source), False
115         except Exception,e:
116             print '[!]', e
117             return None, None
118
119
120     """ execute the given query and return the results of it
121     """
122     def execute(self, query, params):
123         try:
124             self.cursor.execute(query, params)
125             self.conn.commit()
126             return self.cursor.fetchall()
127         except Exception,e:
128             print '[!]', e
129             return None
130
131
132     """ execute the given select query and return the results of it
133     """
```

```
134     def select(self, query, params):
135         try:
136             self.cursor.execute(query, params)
137             return self.cursor.fetchall()
138         except Exception,e:
139             print '[!]', e
140             return None
141 
142     """ execute the given update query
143         if the parameter 'commit' is False, the commit is delayed
144     """
145     def update(self, query, params, commit = True):
146         try:
147             self.cursor.execute(query, params)
148             if commit: self.conn.commit()
149         except Exception,e:
150             print '[!]', e
151 
152 
153     """ execute the given insert query
154         if the parameter 'commit' is False, the commit is delayed
155     """
156     def insert(self, query, params, commit = True):
157         try:
158             self.cursor.execute(query, params)
159             if commit: self.conn.commit()
160         except Exception,e:
161             print '[!]', e
162 
163 
164     """ execute the given delete query
165         if the parameter 'commit' is False, the commit is delayed
166     """
167     def delete(self, query, params, commit = True):
168         try:
169             self.cursor.execute(query, params)
170             if commit: self.conn.commit()
171         except Exception,e:
172             print '[!]', e
173 
174     """ execute the commit in the database
175     """
176     def commit(self):
177         self.conn.commit()
```

```
178 """ Object Crawler - encapsulates the crawler operations
179 """
180 class Crawler(object):
181 
```

```
182     """ constructor - sets the database
183     """
184     def __init__(self, db = None):
185         if db is None:
186             self.db = DataBase()
187         else:
188             self.db = db
189 
190 
191     """ destructor - deletes the database
192     """
193     def __del__(self):
194         del self.db
195 
196 
197     """ calls the select of the current database
198     """
199     def dbSelect(self, query, params):
200         return self.db.select(query, params)
201 
202 
203     """ calls the update of the current database
204     """
205     def dbUpdate(self, query, params):
206         self.db.update(query, params)
207 
208 
209     """ calls the delete of the current database
210     """
211     def dbDelete(self, query, params):
212         self.db.delete(query, params)
213 
214 
215     """ get the id of the given entity.
216         the optional parameter 'insert' indicates if is necessary to insert the
                entity in the database
217     """
218     def dbGetId(self, sid, insert = False, tag = None, user = None, resource =
            None, source = None):
219         if not insert:
220             return self.db.getID(sid, tag, user, resource, source)
221         else:
222             return self.db.getIDInsert(sid, tag, user, resource, source)
223 
224 
225     """ insert a 'tagging object' (a triple user, resource, tag associated with a
            source) in the database
226     """
227     def dbInsertTagging(self, sid, uid, rid, tid, time = None):
```

```
228
229         # no parameter (except 'time') can be 'None'
230         if sid is None or uid is None or rid is None or tid is None:
231             print 'invalid values', sid, uid, rid, tid
232             return
233
234         # transaction [begin] - delay commit until transaction ends
235         # update the counters -
236         #   if there's no entity in db: counter = 1; else counter += 1.
237         r = self.db.select('select count from count_tag where tid = ? and sid =
                ?', (tid, sid))
238         if r == []: self.db.insert('insert into count_tag values (?, ?, 1)',
                (tid, sid), commit = False)
239         else: self.db.update('update count_tag set count = ? where tid = ? and
                sid = ?', (int(r[0][0]) + 1, tid, sid), commit = False)
240
241         r = self.db.select('select count from count_user where uid = ? and sid =
                ?', (uid, sid))
242         if r == []: self.db.insert('insert into count_user values (?, ?, 1)',
                (uid, sid), commit = False)
243         else: self.db.update('update count_user set count = ? where uid = ? and
                sid = ?', (int(r[0][0]) + 1, uid, sid), commit = False)
244
245         r = self.db.select('select count from count_resource where rid = ? and
                sid = ?', (rid, sid))
246         if r == []: self.db.insert('insert into count_resource values (?, ?, 1)',
                (rid, sid), commit = False)
247         else: self.db.update('update count_resource set count = ? where rid = ?
                and sid = ?', (int(r[0][0]) + 1, rid, sid), commit = False)
248
249         r = self.db.select('select count from count_tu where tid = ? and uid = ?
                and sid = ?', (tid, uid, sid))
250         if r == []: self.db.insert('insert into count_tu values (?, ?, ?, 1)',
                (tid, uid, sid), commit = False)
251         else: self.db.update('update count_tu set count = ? where tid = ? and uid
                = ? and sid = ?', (int(r[0][0]) + 1, tid, uid, sid), commit = False)
252
253         r = self.db.select('select count from count_ur where uid = ? and rid = ?
                and sid = ?', (uid, rid, sid))
254         if r == []: self.db.insert('insert into count_ur values (?, ?, ?, 1)',
                (uid, rid, sid), commit = False)
255         else: self.db.update('update count_ur set count = ? where uid = ? and rid
                = ? and sid = ?', (int(r[0][0]) + 1, uid, rid, sid), commit = False)
256
257         r = self.db.select('select count from count_rt where rid = ? and tid = ?
                and sid = ?', (rid, tid, sid))
258         if r == []: self.db.insert('insert into count_rt values (?, ?, ?, 1)',
                (rid, tid, sid), commit = False)
```

```
259         else: self.db.update('update count_rt set count = ? where rid = ? and tid
                = ? and sid = ?', (int(r[0][0]) + 1, rid, tid, sid), commit = False)
260
261         # insert the 'tagging object'
262         self.db.insert('insert into tagging values(?, ?, ?, ?, ?)', (sid, uid,
                rid, tid, time), commit = False)
263
264         # get all tags in the same post
265         tags = self.db.select('select distinct tid from tagging where rid = ? and
                uid = ? and sid = ?', (rid, uid, sid))
266
267         for _t in tags:
268             t = _t[0]
269             # convention - id t1 is always less than id t2
270             if tid < t:
271                 t1 = tid
272                 t2 = t
273             else:
274                 t1 = t
275                 t2 = tid
276
277             # update the counter of tag - tag relation
278             r = self.db.select('select count from count_tt where t1 = ? and t2 =
                    ? and sid = ?', (t1, t2, sid))
279             if r == []:
280                 self.db.insert('insert into count_tt values (?, ?, ?, 1)', (t1,
                        t2, sid), commit = False)
281             else:
282                 self.db.update('update count_tt set count = ? where t1 = ? and t2
                        = ? and sid = ?', (int(r[0][0]) + 1, t1, t2, sid), commit =
                        False)
283
284         # transaction [end]
285         self.db.commit()
286
287     """ the main method that get the data
288         the parameter 'proctime' is the minimum amount of time processing. The
                actual time may be (and usually is) greater.
289     """
290     def crawl(self, source = Source(), proctime = 10, trace = False):
291
292         # create the crawler object of the given source
293         if source.isDelicious():
294             dc = DeliciousCrawler(self)
295             try: dc.crawl(proctime = proctime, trace = trace)
296             except Exception,e:
297                 print e
298                 pass
299         elif source.isFlickr():
```

```
300             fc = FlickrCrawler(self)
301             try: fc.crawl(proctime = proctime, trace = trace)
302             except Exception,e:
303                 print e
304                 pass
305         # all sources
306         else:
307             start = time.time()
308             dc = DeliciousCrawler(self)
309             fc = FlickrCrawler(self)
310             td = dc.initCrawlResult()
311             tf = fc.initCrawlResult()
312 
313             # run until the timeout
314             while True:
315                 try:
316                     # minimum processing time
317                     t = dc.crawl(proctime = 1, trace = trace)
318                     td = dc.sumCrawlResults(td, t)
319                 except Exception,e:
320                     print e
321                     pass
322                 try:
323                     # minimum processing time
324                     t = fc.crawl(proctime = 1, trace = trace)
325                     tf = fc.sumCrawlResults(tf, t)
326                 except Exception,e:
327                     print e
328                     pass
329 
330                 # verify if the timeout has ben reached
331                 end = time.time()
332                 delta = end - start
333                 if (delta / 60) > proctime: break
334 
335             # print the results
336             if trace:
337                 print '$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$'
338                 print 'Total time:', datetime.timedelta(seconds = delta)
339                 dc.printCrawlResults(td)
340                 fc.printCrawlResults(tf)
```

```
341 """ Object DeliciousCrawler - encapsulates the delicious crawler operations
342 """
343 class DeliciousCrawler(object):
344 
345     """ constructor - initialize the variables
346     """
347     def __init__(self, contr):
```

```
348
349         # api object
350         self.dapi = deliciousapi.DeliciousAPI(user_agent = "folklicious v0.1.5")
351         # controller - the main crawl object
352         self.contr = contr
353         # source id
354         self.sid, ondb = self.contr.dbGetId(None, insert = True, source =
                'delicious')
355         # waiting time for each data category
356         self.waiting = {'hotlist': 1800, 'user': 300, 'url': 30}
357         # max elements in the 'not done' list
358         self.max_notdone = {'resources': 200}
359
360     """ the main method that get the data
361         the parameter 'proctime' is the minimum amount of time processing.
362         the actual time may be (and usually is) greater.
363     """
364     def crawl(self, proctime = 10, trace = False):
365
366         # start the processing
367         start = time.time()
368
369         # verify if more than x seconds have been elapsed since the last
370         # requisition to hotlist - delicious restriction
371         oldtime = self.contr.dbSelect("select value from control where name =
                'del.hotlist.timestamp'", ())
372         old = int(oldtime[0][0])
373         interval = time.time() - old
374         if interval < self.waiting['hotlist']: # 30min
375             if trace: print 'wait more', (self.waiting['hotlist'] - interval),
                    'seconds to call populateWithHotList'
376             size_hot = 0
377         else:
378             # populate the hotlist
379             size_hot = self.populateWithHotList(trace)
380             # set the time of the requisition
381             self.contr.dbUpdate('update control set value = ? where name =
                    "del.hotlist.timestamp"', (int(round(time.time())),))
382
383         # initialize the variables
384         waiting = 0
385         size_url = 0
386         size_usr = 0
387         count_pop_user = 0
388
389         # process until timeout
390         while True:
391
392             # 'progress bar'
```

```
393             for i in range(10, waiting):
394                 sys.stdout.write('.')
395                 time.sleep(1)
396                 if i == waiting - 1: print ''
397             count_pop_user += 1
398 
399             # verify if more  than x seconds have been elapsed since the last
400             # requisition to hotlist - delicious restriction
401             oldtime = self.contr.dbSelect("select value from control where name =
                    'del.url.timestamp'", ())
402             old = int(oldtime[0][0])
403             interval = time.time() - old
404             if interval < self.waiting['url']: # 15min
405                 if trace: print 'wait more', (self.waiting['url'] - interval),
                        'seconds to call populateWithUrl'
406                 waiting += 1
407             else:
408                 # populate with url as seed
409                 size_url += self.populateWithUrl(1, trace)
410 
411             # only populate with user if tried to populate with url 20 times
412             if count_pop_user == 20:
413 
414                 # verify if more  than x seconds have been elapsed since the last
415                 # requisition to hotlist - delicious restriction
416                 oldtime = self.contr.dbSelect("select value from control where
                        name = 'del.user.timestamp'", ())
417                 old = int(oldtime[0][0])
418                 interval = time.time() - old
419                 if interval < self.waiting['user']: # 15min
420                     if trace: print 'wait more', (self.waiting['user'] -
                            interval), 'seconds to call populateWithUser'
421                     waiting += 1
422                 else:
423                     # populate with user as seed
424                     size_usr += self.populateWithUser(1, trace)
425 
426                 # reset the counter
427                 count_pop_user = 0
428 
429             # verify if the minimum processing time has been reached
430             end = time.time()
431             delta = end - start
432             if (delta / 60) > proctime:
433                 if trace:
434                     print 'del :: time reached', datetime.timedelta(seconds=delta)
435                     print 'del ::', size_hot, 'new resources from hotlist'
436                     print 'del ::', size_url, 'new tuples'
437                     print 'del ::', size_usr, 'new urls'
```

```
438                 break
439
440         # return the elapsed time, and the amount of elements
441         return (delta, size_hot, size_url, size_usr)
442
443     """ initialize the results
444     """
445     def initCrawlResult(self):
446         return (0, 0, 0, 0)
447
448     """ sum two sets of results
449     """
450     def sumCrawlResults(self, t1, t2):
451         return (t1[0] + t2[0], t1[1] + t2[1], t1[2] + t2[2], t1[3] + t2[3])
452
453     """ print the results
454     """
455     def printCrawlResults(self ,t):
456         print '$del :: total time', datetime.timedelta(seconds=t[0])
457         print '$del ::', t[1], 'new resources from hotlist'
458         print '$del ::', t[2], 'new tuples'
459         items = self.contr.dbSelect('select count(*) from resource where sid = ?
                and done = 0', (self.sid,))
460         print '$del ::', t[3], 'new urls,', items[0][0], 'to be processed'
461
462     """ populate the db using the hotlist
463     """
464     def populateWithHotList(self, trace = False):
465         if trace: print 'del :: populateWithHotList'
466
467         # get the resources not processed yet
468         notdone = self.contr.dbSelect('select count(*) from resource where sid =
                ? and done = 0', (self.sid,))[0][0]
469         if notdone > self.max_notdone['resources']:
470             if trace: print 'There are %d (%d) resources notdone.' % (notdone,
                    self.max_notdone['resources'])
471             return 0
472         # get the URLs in hotlist
473         urls = self.dapi.get_urls()
474         if trace: print len(urls), 'urls retrieved'
475         count = 0
476
477         # for each URL
478         for u in urls:
479             # try to insert in db
480             rid, ondb = self.contr.dbGetId(self.sid, insert = True, resource = u)
481             if not ondb:
482                 count += 1
483                 if trace: print 'inserted', u
```

```
484             elif trace: print 'already inserted', u
485         if trace: print count, 'new urls'
486 
487         # return the number of new objects stored
488         return count
489 
490     """ populate the db using the URL
491     """
492     def populateWithUrl(self, max_urls = 10, trace = False):
493         if trace: print 'del :: populateWithUrl - max_urls:', max_urls
494 
495         # get the resources not processed yet
496         res = self.contr.dbSelect('select value from resource where done = 0 and
                sid = ? limit ?', (self.sid, max_urls,))
497         total = len(res)
498         if total > max_urls: total = max_urls
499         if trace: print total, 'urls'
500         curr = 1
501         count = 0
502 
503         # process each resource ...
504         for _r in res:
505             # ... until reach the maximum
506             if curr > max_urls: break
507 
508             r = _r[0]
509             if trace: print 'resource %03d/%03d' %(curr, total)
510 
511             # get the bookmarks associated with that URL
512             # with 'max_bookmarks=0' all of them are retorned, but it consumes
                    more time
513             meta = self.dapi.get_url(r, max_bookmarks=0)
514             # store the requisition time
515             self.contr.dbUpdate('update control set value = ? where name =
                    "del.url.timestamp"', (int(round(time.time())),))
516 
517             # process the bookmarks
518             total_bookmarks = len(meta.bookmarks)
519             if trace: print 'resource %s - %d bookmarks' %(r, total_bookmarks)
520             if total_bookmarks > 0 :
521                 curr_bookmarks = 0
522                 for b in meta.bookmarks:
523                     curr_bookmarks += 1
524                     # [user, taglist, comment, time]
525                     user = b[0]
526                     taglist = b[1]
527                     timestamp = b[3]
528 
529                     # user id
```

```
530                     uid, ondb = self.contr.dbGetId(self.sid, insert = True, user
                            = user)
531                     # resource id
532                     rid, ondb = self.contr.dbGetId(self.sid, insert = True,
                            resource = r)
533
534                     # failed to insert
535                     if uid is None or rid is None:
536                         break
537
538                     for t in taglist:
539                         count += 1
540                         # tag id
541                         tid, ondb = self.contr.dbGetId(self.sid, insert = True,
                                tag = t)
542                         # insert the tagging object
543                         self.contr.dbInsertTagging(self.sid, uid, rid, tid,
                                str(timestamp))
544                         if trace:
545                             print 'ids:', uid, rid, tid
546                             print 'inserted', (user, r[:60], t, str(timestamp))
547                             print '[%03d/%03d resources][%03d/%03d bookmarks][%4d
                                    inserts]' %(curr, total, curr_bookmarks,
                                    total_bookmarks, count)
548
549                 # set the resource as done
550                 self.contr.dbUpdate('update resource set done = 1 where value = ?
                        and sid = ?', (r, self.sid,))
551             else:
552                 # set the resource as done - no tags
553                 self.contr.dbUpdate('update resource set done = 1 where value = ?
                        and sid = ?', (r, self.sid,))
554                 if trace: print 'no tags -> done', r[:75]
555             curr += 1
556
557         # return the number of new objects stored
558         return count
559
560     """ populate the db using the URL
561     """
562     def populateWithUser(self, max_users = 10, trace = False):
563         if trace: print 'del :: populateWithUser - max_users:', max_users
564
565         # get the resources not processed yet
566         notdone = self.contr.dbSelect('select count(*) from resource where sid =
                ? and done = 0', (self.sid,))[0][0]
567         # if there are more resources than the maximum, don't try to populate
                with user
568         if notdone > self.max_notdone['resources']:
```

```
569             if trace: print 'There are %d (%d) resources not done.' % (notdone,
                    self.max_notdone['resources'])
570             return 0
571
572         # get the users not processed yet
573         users = self.contr.dbSelect('select value from user where done = 0 and
                sid = ? limit ?', (self.sid, max_users,))
574         total = len(users)
575         if total > max_users: total = max_users
576         if trace: print total, 'users'
577         curr = 1
578         count = 0
579
580         # process each user ...
581         for _u in users:
582             # ... until reach the maximum
583             if curr > max_users: break
584
585             u = _u[0]
586             if trace: print 'user %03d/%03d' %(curr, total)
587
588             try:
589                 # get the bookmarks associated with that user
590                 meta = self.dapi.get_user(u, max_bookmarks = 10)
591                 # store the requisition time
592                 self.contr.dbUpdate('update control set value = ? where name =
                        "del.user.timestamp"', (int(round(time.time())),))
593             except:
594                 # if there was an error, mark that resource as already processed
                        and continue to the next one
595                 if trace: print 'Delicious error: user', u, 'marked as done'
596                 self.contr.dbUpdate('update user set done = 1 where value = ? and
                        sid = ?', (u, self.sid,))
597                 continue
598
599             if len(meta.bookmarks) > 0 :
600                 for b in meta.bookmarks:
601                     # [url, taglist, title, comment, time]
602                     r = b[0]
603                     # resource id
604                     rid, ondb = self.contr.dbGetId(self.sid, insert = True,
                            resource = r)
605                     if not ondb:
606                         count += 1
607                         if trace: print 'inserted', r, 'id:', rid
608                     else:
609                         if trace: print 'already inserted', r, 'id:', rid
610                     if trace: print '[%03d/%03d] %d inserts' %(curr, total, count)
611
```

```
612             # set the user as done
613             self.contr.dbUpdate('update user set done = 1 where value = ? and sid
                    = ?', (u, self.sid))
614             curr += 1
615 
616         # return the number of new objects stored
617         return count
```

```
618 """ Object FlickrCrawler - encapsulates the flickr crawler operations
619 """
620 class FlickrCrawler(object):
621 
622     """ constructor - initialize the variables
623     """
624     def __init__(self, contr):
625         # controller - the main crawl object
626         self.contr = contr
627         # source id
628         self.sid, ondb = self.contr.dbGetId(None, insert = True, source =
                'flickr')
629         # base URL
630         self.url = 'http://api.flickr.com/services/rest/'
631         # api key - REPLACE WITH YOUR OWN API KEY
632         self.api_key = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
633         # api methods
634         self.methods = {
635             'echo':     'flickr.test.echo',
636             'pInfo':    'flickr.photos.getInfo',
637             'pPList':   'flickr.people.getPublicPhotos',
638             'uPList':   'flickr.contacts.getPublicList',
639             'uPPhotos': 'flickr.photos.getContactsPublicPhotos',
640             'pRecent':  'flickr.photos.getRecent',
641             'tHot':     'flickr.tags.getHotList',
642             'tPhoto':   'flickr.tags.getListPhoto',
643             'panda':   'flickr.panda.getPhotos',
644         }
645 
646     """ the main method that get the data
647         the parameter 'proctime' is the minimum amount of time processing.
648         the actual time may be (and usually is) greater.
649     """
650     def crawl(self, proctime = 10, trace = False):
651 
652         # start the processing
653         start = time.time()
654 
655         # initialize the variables
656         size_tag = 0
657         size_res = 0
```

```
658         count_pop_res = 0
659
660         # process until timeout
661         while True:
662             count_pop_res += 1
663
664             # populate the db with tags
665             size_tag += self.populateTags(1, trace = trace)
666
667             # only populate with user if tried to populate with tags 20 times
668             if count_pop_res == 20:
669                 size_res += self.populateResources(1, trace = trace)
670                 count_pop_user = 0
671
672             # verify if the minimum processing time has been reached
673             end = time.time()
674             delta = end - start
675             if (delta / 60) > proctime:
676                 if trace:
677                     print 'fli :: time reached', datetime.timedelta(seconds=delta)
678                     print 'fli ::', size_tag, 'new tuples'
679                     print 'fli ::', size_res, 'new resources'
680                 break
681
682         # return the elapsed time, and the amount of elements
683         return (delta, size_tag, size_res)
684
685     """ initialize the results
686     """
687     def initCrawlResult(self):
688         return (0, 0, 0)
689
690     """ sum two sets of results
691     """
692     def sumCrawlResults(self, t1, t2):
693         return (t1[0] + t2[0], t1[1] + t2[1], t1[2] + t2[2])
694
695     """ print the results
696     """
697     def printCrawlResults(self, t):
698         print '$fli :: total time', datetime.timedelta(seconds=t[0])
699         print '$fli ::', t[1], 'new tuples'
700         items = self.contr.dbSelect('select count(*) from resource where sid = ?
                and done = 0', (self.sid,))
701         print '$fli ::', t[2], 'new resources,', items[0][0], 'to be processed'
702
703     """ execute a post request
704     """
705     def post(self, params):
```

```
706         return urllib.urlopen(self.url, params)
707
708     """ create the params object
709     """
710     def createParams(self, method, params, trace = False):
711         params['method'] = method
712         params['api_key'] = self.api_key
713         if trace: print params
714         return urllib.urlencode(params)
715
716     """ get random photos - this service is called panda in flickr
717     """
718     def getPhotosPanda(self, per_page = None, panda = None):
719         # choose the panda - 'ling ling', 'hsing hsing', and 'wang wang'
720         if panda is None: panda = random.randint(1, 3)
721         if panda == 1:
722             lst = {'panda_name': 'ling ling'}
723         elif panda == 2:
724             lst = {'panda_name': 'hsing hsing'}
725         else:
726             lst = {'panda_name': 'wang wang'}
727         if per_page is not None:
728             lst['per_page'] = per_page
729
730         # return the result
731         return self.post(self.createParams(self.methods['panda'], lst))
732
733     """ populate the db using resources
734     """
735     def populateResources(self, num_photos, pandaid = None, trace = False):
736         if trace: print 'fli :: populateResources - num_photos:', num_photos,
              'pandaid:', pandaid
737
738         # the limit is 500 photos
739         if num_photos <= 500:
740
741             # xml with photos from pandas
742             pxml = self.getPhotosPanda(num_photos, pandaid)
743             # get the information
744             root = parse(pxml).getroot().find('photos')
745             interval = root.get('interval')
746             lastupdate = root.get('lastupdate')
747             photos = root.findall('photo')
748             total = len(photos)
749             curr = 1
750             count = 0
751
752             # for each photo of the xml
753             for p in photos:
```

```
754                 # insert in the db
755                 pid = p.get('id')
756                 rid, ondb = self.contr.dbGetId(self.sid, insert = True, resource
                        = pid)
757                 if not ondb:
758                     count += 1
759                 if trace: print '%s: %03d/%03d' % (rid, curr, total)
760                 curr += 1
761 
762             # flickr gives the lastupdate time and the waiting interval
763             wait = time.time() - (int(lastupdate) + int(interval))
764             # so, verify if need to wait
765             if wait > 0:
766                 if trace: print 'waiting', wait, 'before next requisition'
767                 time.sleep(wait)
768 
769         # more than 500 photos - multiple requests
770         else:
771             n = 0
772             r = 0
773             while n < num_photos:
774                 r += self.populateResources(500)
775                 n += 500
776             return r
777 
778         # return the number of new objects stored
779         return count
780 
781     """ populate the db using tags
782     """
783     def populateTags(self, num_resources = 10, trace = False):
784         if trace: print 'fli :: populateTags - num_resources:', num_resources
785 
786         # get the resources not processed yet
787         res = self.contr.dbSelect('select rid from resource where done = 0 and
                    sid = ? limit ?', (self.sid, num_resources,))
788         curr = 1
789         if len(res) > num_resources:
790             total = num_resources
791         else:
792             total = len(res)
793         count = 0
794 
795         # process each resource ...
796         for _r in res:
797             rid = _r[0]
798 
799             # ... until reach the maximum
800             if curr > total:
```

```
801                 break
802
803             if trace: print 'resources: %03d/%03d' % (curr, total)
804
805             # get the data from photo
806             data = self.getDataFromPhoto(rid, trace = trace)
807             notags = True
808
809             for d in data:
810                 # user id
811                 uid, ondb = self.contr.dbGetId(self.sid, insert = True, user =
                        d['user'])
812                 # tag id
813                 tid, ondb = self.contr.dbGetId(self.sid, insert = True, tag =
                        d['tag'])
814
815                 # insert the tagging object
816                 self.contr.dbInsertTagging(self.sid, uid, rid, tid)
817                 count += 1
818                 if trace: print '%d new tuple: %s' % (count, (self.sid, uid, rid,
                        tid,))
819                 notags = False
820
821             # set the resource as done
822             self.contr.dbUpdate('update resource set done = 1 where rid = ? and
                    sid = ?', (rid, self.sid,))
823             if trace: print 'rid:', rid, 'set done - notags:', notags
824             curr += 1
825
826         # return the number of new objects stored
827         return count
828
829     """ return the xml file with the tags of a given photo
830     """
831     def getTagsFromPhoto(self, photo_id):
832         return self.post(self.createParams(self.methods['tPhoto'], {'photo_id':
                photo_id}))
833
834     """ return the data from a given photo
835     """
836     def getDataFromPhoto(self, photo_id, trace = False):
837
838         # get the tags
839         tags = self.getTagsFromPhoto(photo_id)
840         root = parse(tags).getroot()
841         stat = root.get('stat')
842         if stat == 'fail' and trace: print '[!] Flickr error:',
                root.find('err').get('msg'), 'photo_id', photo_id
843
```

```
844         # prepare the result with all pairs {user; tag}
845         result = []
846         for t in root.findall('photo/tags/tag'):
847             result.append({'user': t.get('author'), 'tag': t.text})
848 
849         # return the result data
850         return result
```

```
851 # the main execution
852 if __name__ == "__main__":
853     # create the crawler object
854     c = Crawler()
855     # execute a crawl operation in all sources and with minimum 60 minutes
856     c.crawl(Source(), proctime=60, trace=True)
857 
858     # example of crawl in delicious with minimum 45 minutes
859     # c.crawl(Source('delicious'), proctime = 45, trace = True)
860     # example of crawl in flickr with minimum 15 minutes
861     # c.crawl(Source('flickr'), proctime = 15, trace = True)
862     # delete the object and release all resources
863     del c
```

4 Folksonomy Systems

4.1 Flickr

Flickr is an online community and an image and video hosting website. It has a large user community and huge amount of resources (mainly images). Whenever authors uploads an image they insert tags to describe it. The main search mechanism of Flickr is based on tags.

Flickr offers a web service API to access its data by non-commercial applications. One important aspect of the API is that all data should be codified by using the UTF-8 standard. If the API receives a sequence in any codification but UTF-8, it assumes that the codification is ISO-8859-1 and then transforms it to UTF-8. Any other codification could result in incorrect data. In the following subsections we detail aspects of the Flickr API.

4.1.1 Authentication

The services can be accessed in non-authenticated or authenticated modes. There are API methods that are only available to the authenticated users. The authentication process is described in the Flickr OAuthAPI webpage[7]. In our study we focused on public data. Therefore, we did not use the authenticate methods.

[7] `http://www.flickr.com/services/api/auth.oauth.html`

4.1.2 Definitions

Services' protocols and data objects exchanged by Flickr and clients follow a set of basic standard definitions further detailed.

Dates[8] There are two types of dates: *taken* – the date when the photo was taken; *posted* – the date when the photo was posted on the system. *Taken* dates must follow the MySQL 'datetime' format (e.g., 2004-11-29 16:01:26) and they have 4 levels of accuracy:
0 Y-m-d H:i:s
4 Y-m
6 Y
8 circa...

On the other hand, *posted* dates are always in the unix timestamp format, i.e., an unsigned integer specifying the number of seconds since Jan 1st 1970 GMT.

Buddyicons[9] Buddyicon is a 48x48 pixel image that represent the user – an "avatar". It is necessary to inform the user's NSID (user id), icon server and icon farm to access the buddyicon of a user.

If the icon server parameter is a number greater than zero. A URL to request the icon takes the format:

```
http://farm{icon-farm}.static.flickr.com/{icon-server}/
buddyicons/{nsid}.jpg
```

There is also a URL to request the default buddyicon:

```
http://www.flickr.com/images/buddyicon.jpg
```

URLs[10] The photo URL is built by using the following parameters: photo ID, server ID, farm ID, and secret. The URL takes the format:

```
http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}.jpg
or
http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}_[mstzb].jpg
or
http://farm{farm-id}.static.flickr.com/{server-id}/
{id}_{o-secret}_o.(jpg|gif|png)
```

The second URL affords one of the following size suffixes (indicated in the URL by brackets):
s – small square 75x75
t – thumbnail, 100 on longest side

[8] http://www.flickr.com/services/api/misc.dates.html
[9] http://www.flickr.com/services/api/misc.buddyicons.html
[10] http://www.flickr.com/services/api/misc.urls.html

m – small, 240 on longest side
- – medium, 500 on longest side
z – medium 640, 640 on longest side
b – large, 1024 on longest side
The last URL is specific for images in the original size:
o – original image, either a jpg, gif or png, depending on source format

Tags[11] When a photo has tags, the format of the tag field in XML is as follows:

```
<tag id="1234" author="12037949754@N01" raw="woo yay">wooyay</tag>
```

The parameters are:
id – The photo id.
author – The NSID of the user who added the tag.
raw – The "raw" version of the tag - as entered by the user. This version can contain spaces and punctuation.
tag-body – The "clean" version of the tag – as processed by Flickr.

4.1.3 Access

API keys are necessary to access and use the flickr data. The process to obtain those keys is the following:

1. Register to Flickr (as a user).
2. Go to the API request page (`http://www.flickr.com/services/apps/create/apply/`).
3. Choose the appropriate option (in this work we used "Non-Commercial").
4. Fill in the form.

In the end of this process the API keys are generated.

4.1.4 Request Protocols

There are three request protocols: REST[12], XML-RPC[13], and SOAP[14]. We adopted REST since it is the simplest option and it meets our needs.

The following pair requisition/response illustrates a REST requisition on Flickr. In this case we are using a fictitious API key.

```
http://www.flickr.com/services/rest/?method=flickr.test.echo&format=rest
&foo=bar&api_key=cc4094c55264c02ec2a83001b95a0837

<rsp stat="ok">
```

[11] `http://www.flickr.com/services/api/misc.tags.html`
[12] `http://www.flickr.com/services/api/request.rest.html`
[13] `http://www.flickr.com/services/api/request.xmlrpc.html`
[14] `http://www.flickr.com/services/api/request.soap.html`

```
  <method>flickr.test.echo</method>
    <format>rest</format>
    <foo>bar</foo>
  <api_key>cc4094c55264c02ec2a83001b95a0837</api_key>
</rsp>
```

4.1.5 Response Formats

There are five response formats: REST[15], XML-RPC[16], SOAP[17], JSON[18], and PHP[19]. Again, we chose the REST format.

4.1.6 API Methods

In this section we briefly describe the API methods that are of most relevant to this work. The complete list of methods can be viewed in the API page (`http://www.flickr.com/services/api/`).

auth: methods to authenticate the app.

- flickr.auth.checkToken[20]: returns the credentials attached to a token
- flickr.auth.getFrob[21]: returns the frob to be used in authentication
- flickr.auth.getFullToken[22]: returns the full token from a mini-token
- flickr.auth.getToken[23]: returns the token from a frob

contacts

- flickr.contacts.getList[24]: returns the contact list from a user
- flickr.contacts.getPublicList[25]: returns the public contact list from a user (doesn't need authentication)

galleries

- flickr.galleries.getInfo[26]: returns the information of a gallery
- flickr.galleries.getList[27]: returns the list of galleries

[15] `http://www.flickr.com/services/api/response.rest.html`
[16] `http://www.flickr.com/services/api/response.xmlrpc.html`
[17] `http://www.flickr.com/services/api/response.soap.html`
[18] `http://www.flickr.com/services/api/response.json.html`
[19] `http://www.flickr.com/services/api/response.php.html`
[20] `http://www.flickr.com/services/api/flickr.auth.checkToken.html`
[21] `http://www.flickr.com/services/api/flickr.auth.getFrob.html`
[22] `http://www.flickr.com/services/api/flickr.auth.getFullToken.html`
[23] `http://www.flickr.com/services/api/flickr.auth.getToken.html`
[24] `http://www.flickr.com/services/api/flickr.contacts.getList.html`
[25] `http://www.flickr.com/services/api/flickr.contacts.getPublicList.html`
[26] `http://www.flickr.com/services/api/flickr.galleries.getInfo.html`
[27] `http://www.flickr.com/services/api/flickr.galleries.getList.html`

- flickr.galleries.getListForPhoto[28]: returns the list of galleries that contains a given photo
- flickr.galleries.getPhotos[29]: returns the photos of a given gallery

interestingness

- flickr.interestingness.getList[30]: returns the most interesting photos of a specific date

machinetags

- flickr.machinetags.getNamespaces[31]: returns a list of unique namespaces
- flickr.machinetags.getPairs[32]: returns a list of unique namespace and predicate pairs
- flickr.machinetags.getPredicates[33]: returns a list of unique predicates
- flickr.machinetags.getRecentValues[34]: returns the most recent machinetags
- flickr.machinetags.getValues[35]: returns a list of unique values for a namespace and predicate

panda

- flickr.panda.getList[36]: returns a list of pandas (photo services)
- flickr.panda.getPhotos[37]: returns a list of photos of the given panda

people

- flickr.people.findByEmail[38]: returns a user's NSID, given his/her email address
- flickr.people.findByUsername[39]: returns a user's NSID, given his/her username
- flickr.people.getInfo[40]: gets information about a user
- flickr.people.getPhotos[41]: returns photos from the given user's photostream
- flickr.people.getPhotosOf[42]: returns a list of photos containing a particular Flickr member

[28] `http://www.flickr.com/services/api/flickr.galleries.getListForPhoto.html`
[29] `http://www.flickr.com/services/api/flickr.auth.getToken.html`
[30] `http://www.flickr.com/services/api/flickr.interestingness.getList.html`
[31] `http://www.flickr.com/services/api/flickr.machinetags.getNamespaces.html`
[32] `http://www.flickr.com/services/api/flickr.machinetags.getPairs.html`
[33] `http://www.flickr.com/services/api/flickr.machinetags.getPredicates.html`
[34] `http://www.flickr.com/services/api/flickr.machinetags.getRecentValues.html`
[35] `http://www.flickr.com/services/api/flickr.machinetags.getValues.html`
[36] `http://www.flickr.com/services/api/flickr.panda.getList.html`
[37] `http://www.flickr.com/services/api/flickr.panda.getPhotos.html`
[38] `http://www.flickr.com/services/api/flickr.people.findByEmail.html`
[39] `http://www.flickr.com/services/api/flickr.people.findByUsername.html`
[40] `http://www.flickr.com/services/api/flickr.people.getInfo.html`
[41] `http://www.flickr.com/services/api/flickr.people.getPhotos.html`
[42] `http://www.flickr.com/services/api/flickr.people.getPhotosOf.html`

- flickr.people.getPublicPhotos[43]: gets a list of public photos for the given user

photos

- flickr.photos.getAllContexts[44]: returns all visible sets and pools the photo belongs to
- flickr.photos.getContactsPhotos[45]: fetches a list of recent photos from the calling users' contacts
- flickr.photos.getContactsPublicPhotos[46]: fetches a list of recent public photos from a users' contacts
- flickr.photos.getContext[47]: returns next and previous photos for a photo in a photostream
- flickr.photos.getCounts[48]: gets a list of photo counts for the given date ranges for the calling user
- flickr.photos.getInfo[49]: gets information about a photo. The calling user must have permission to view the photo
- flickr.photos.getPerms[50]: gets permissions for a photo
- flickr.photos.getRecent[51]: returns a list of the latest public photos uploaded to flickr
- flickr.photos.getSizes[52]: returns the available sizes for a photo. The calling user must have permission to view the photo
- flickr.photos.getUntagged[53]: returns a list of your photos with no tags
- flickr.photos.getWithGeoData[54]: returns a list of your geo-tagged photos
- flickr.photos.getWithoutGeoData[55]: returns a list of your photos which haven't been geo-tagged
- flickr.photos.recentlyUpdated[56]: returns a list of your photos that have been recently created or which have been recently modified
- flickr.photos.search[57]: returns a list of photos matching some criteria

[43] `http://www.flickr.com/services/api/flickr.people.getPublicPhotos.html`
[44] `http://www.flickr.com/services/api/flickr.photos.getAllContexts.html`
[45] `http://www.flickr.com/services/api/flickr.photos.getContactsPhotos.html`
[46] `http://www.flickr.com/services/api/flickr.photos.getContactsPublicPhotos.html`
[47] `http://www.flickr.com/services/api/flickr.photos.getContext.html`
[48] `http://www.flickr.com/services/api/flickr.photos.getCounts.html`
[49] `http://www.flickr.com/services/api/flickr.photos.getInfo.html`
[50] `http://www.flickr.com/services/api/flickr.photos.getPerms.html`
[51] `http://www.flickr.com/services/api/flickr.photos.getRecent.html`
[52] `http://www.flickr.com/services/api/flickr.photos.getSizes.html`
[53] `http://www.flickr.com/services/api/flickr.photos.getUntagged.html`
[54] `http://www.flickr.com/services/api/flickr.photos.getWithGeoData.html`
[55] `http://www.flickr.com/services/api/flickr.photos.getWithoutGeoData.html`
[56] `http://www.flickr.com/services/api/flickr.photos.recentlyUpdated.html`
[57] `http://www.flickr.com/services/api/flickr.photos.search.html`

photos.licenses

- flickr.photos.licenses.getInfo[58]: fetches a list of available photo licenses for Flickr

photosets

- flickr.photosets.getContext[59]: returns next and previous photos for a photo in a set
- flickr.photosets.getInfo[60]: gets information about a photoset
- flickr.photosets.getList[61]: returns the photosets belonging to the specified user
- flickr.photosets.getPhotos[62]: gets the list of photos in a set

reflection

- flickr.reflection.getMethodInfo[63]: returns information for a given flickr API method
- flickr.reflection.getMethods[64]: returns a list of available flickr API methods

tags

- flickr.tags.getClusterPhotos[65]: returns the first 24 photos for a given tag cluster
- flickr.tags.getClusters[66]: returns a list of tag clusters for the given tag
- flickr.tags.getHotList[67]: returns a list of hot tags for the given period
- flickr.tags.getListPhoto[68]: gets the tag list for a given photo
- flickr.tags.getListUser[69]: gets the tag list for a given user (or for the user currently logged)
- flickr.tags.getListUserPopular[70]: gets the popular tags for a given user (or for the user currently logged)
- flickr.tags.getListUserRaw[71]: gets the raw versions of a given tag (or all tags) for the user currently logged
- flickr.tags.getRelated[72]: returns a list of tags 'related' to a given tag, based on clustered usage analysis

[58] `http://www.flickr.com/services/api/flickr.photos.licenses.getInfo.html`
[59] `http://www.flickr.com/services/api/flickr.photosets.getContext.html`
[60] `http://www.flickr.com/services/api/flickr.photosets.getInfo.html`
[61] `http://www.flickr.com/services/api/flickr.photosets.getList.html`
[62] `http://www.flickr.com/services/api/flickr.photosets.getPhotos.html`
[63] `http://www.flickr.com/services/api/flickr.reflection.getMethodInfo.html`
[64] `http://www.flickr.com/services/api/flickr.reflection.getMethods.html`
[65] `http://www.flickr.com/services/api/flickr.tags.getClusterPhotos.html`
[66] `http://www.flickr.com/services/api/flickr.tags.getClusters.html`
[67] `http://www.flickr.com/services/api/flickr.tags.getHotList.html`
[68] `http://www.flickr.com/services/api/flickr.tags.getListPhoto.html`
[69] `http://www.flickr.com/services/api/flickr.tags.getListUser.html`
[70] `http://www.flickr.com/services/api/flickr.tags.getListUserPopular.html`
[71] `http://www.flickr.com/services/api/flickr.tags.getListUserRaw.html`
[72] `http://www.flickr.com/services/api/flickr.tags.getRelated.html`

test

- flickr.test.echo[73]: a testing method which echo's all parameters back in the response
- flickr.test.login[74]: a testing method which checks if the caller is logged and then returns his/her username
- flickr.test.null[75]: null test

urls

- flickr.urls.getGroup[76]: returns a url pointing to a group's page
- flickr.urls.getUserPhotos[77]: returns a url pointing to a user's photos
- flickr.urls.getUserProfile[78]: returns a url pointing to a user's profile
- flickr.urls.lookupGallery[79]: returns a gallery info
- flickr.urls.lookupGroup[80]: returns a group NSID, given the url pointing to a group's page or photo pool
- flickr.urls.lookupUser[81]: returns a user NSID, given the url pointing to a user's photos or profile

4.1.7 API Examples

In this section we show some examples of the API. In each example there are two blocks, one containing the request and other with the response.

i Getting Photo Information

```
http://api.flickr.com/services/rest/?method=flickr.photos.getInfo
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&photo_id=120292580

/**
  api_key (required): api key
  photo_id (required): photo id
  secret (optional): if correct the permission check is not performed
**/
```

[73] http://www.flickr.com/services/api/flickr.test.echo.html
[74] http://www.flickr.com/services/api/flickr.test.login.html
[75] http://www.flickr.com/services/api/flickr.test.null.html
[76] http://www.flickr.com/services/api/flickr.urls.getGroup.html
[77] http://www.flickr.com/services/api/flickr.urls.getUserPhotos.html
[78] http://www.flickr.com/services/api/flickr.urls.getUserProfile.html
[79] http://www.flickr.com/services/api/flickr.urls.lookupGallery.html
[80] http://www.flickr.com/services/api/flickr.urls.lookupGroup.html
[81] http://www.flickr.com/services/api/flickr.urls.lookupUser.html

```
<rsp stat="ok">
<photo id="120292580" secret="fca8637ab6" server="47" farm="1"
    dateuploaded="1143731314" isfavorite="0" license="5" rotation="0"
    originalsecret="fca8637ab6" originalformat="png" views="10163" media="photo">
    <owner nsid="67526850@N00" username="Alex Osterwalder" realname="Alexander
        Osterwalder" location="Genvea, Switzerland" />
    <title>Web2.0 Business Model Characteristics</title>
    <description>The outcome of a short late-night brainstorming session on the
        characteristics of a Web2.0 business model. The reflections are based on
        what I write at my (...)</description> <visibility ispublic="1"
        isfriend="0" isfamily="0" />
    <dates posted="1143731314" taken="2006-03-30 22:08:34" takengranularity="0"
        lastupdate="1202967678" />
    <editability cancomment="0" canaddmeta="0" />
    <usage candownload="1" canblog="0" canprint="0" canshare="0" />
    <comments>2</comments>
    <notes />
    <tags>
        <tag id="2017715-120292580-380852" author="67526850@N00" raw="business
            model" machine_tag="0">businessmodel</tag>
        <tag id="2017715-120292580-11227" author="67526850@N00" raw="web2.0"
            machine_tag="0">web20</tag>
        <tag id="2017715-120292580-2956157" author="67526850@N00" raw="business
            model innovation" machine_tag="0">businessmodelinnovation</tag>
        <tag id="2017715-120292580-2956158" author="67526850@N00" raw="business
            model ontology" machine_tag="0">businessmodelontology</tag>
        <tag id="2017715-120292580-2109580" author="67526850@N00"
            raw="osterwalder" machine_tag="0">osterwalder</tag>
    </tags>
    <urls>
        <url type="photopage">
          http://www.flickr.com/photos/osterwalder/120292580/
        </url>
    </urls>
</photo>
</rsp>
```

ii Photo Galleries Information

```
http://api.flickr.com/services/rest/?method=flickr.galleries.getListForPhoto
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&photo_id=2080242123&per_page=5

/**
  api_key (required): api key
  photo_id (required): photo id
  per_page (optional): number of galleries in result. default 100 - max 500
  page (optional): the page of the result. default 1
**/
```

```
<rsp stat="ok">
<galleries total="19" page="1" pages="4" per_page="5" photo_id="2080242123">
    <gallery id="25845796-72157624218638653"
        url="http://www.flickr.com/photos/25891118@N06/galleries/72157624218638653"
        owner="25891118@N06" primary_photo_id="2080242123"
        date_create="1277335620" date_update="1278257567" count_photos="4"
        count_videos="0" primary_photo_server="2209" primary_photo_farm="3"
        primary_photo_secret="55c93c007d"> <title>Nature</title>
        <description />
    </gallery>
    <gallery id="51165959-72157624161029199"
        url="http://www.flickr.com/photos/fer10/galleries/72157624161029199"
        owner="51198098@N04" primary_photo_id="4691319257"
        date_create="1276662136" date_update="1276906396" count_photos="14"
        count_videos="0" primary_photo_server="4007" primary_photo_farm="5"
        primary_photo_secret="f411f6ba4e">
        <title>Bellezas</title>
        <description>Expectacular</description>
    </gallery>
    <gallery id="1344252-72157623919289749"
        url="http://www.flickr.com/photos/68196577@N00/galleries/72157623919289749"
        owner="68196577@N00" primary_photo_id="4536144000"
        date_create="1273628141" date_update="1278099389" count_photos="15"
        count_videos="0" primary_photo_server="4032" primary_photo_farm="5"
        primary_photo_secret="49a59c20ff">
        <title>WHAT?!</title>
        <description />
    </gallery>
    <gallery id="15624814-72157623792903801"
        url="http://www.flickr.com/photos/15646144@N07/galleries/72157623792903801"
        owner="15646144@N07" primary_photo_id="2080242123"
        date_create="1272049888" date_update="1272052268" count_photos="1"
        count_videos="0" primary_photo_server="2209" primary_photo_farm="3"
        primary_photo_secret="55c93c007d">
        <title>For Mobile</title>
        <description />
    </gallery>
    <gallery id="20945644-72157623529610741"
        url="http://www.flickr.com/photos/20966974@N07/galleries/72157623529610741"
        owner="20966974@N07" primary_photo_id="2080242123"
        date_create="1269051584" date_update="1269051616" count_photos="1"
        count_videos="0" primary_photo_server="2209" primary_photo_farm="3"
        primary_photo_secret="55c93c007d">
        <title>Fall</title>
        <description />
    </gallery>
</galleries>
</rsp>
```

iii Public List of User Contacts

```
http://api.flickr.com/services/rest/?method=flickr.contacts.getPublicList
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&user_id=67526850@N00

/**
  api_key (required): api key
  user_id (required): photo id
  per_page (optional): number of result items. default 1000 - max 1000
  page (optional): the page of the result. default 1
**/
```

```
<rsp stat="ok">
<contacts page="1" pages="1" per_page="1000" perpage="1000" total="19">
    <contact nsid="28404674@N00" username="(^_^) wellwin" iconserver="41"
        iconfarm="1" ignored="0" />
    <contact nsid="38075047@N00" username="dgray_xplane" iconserver="32"
        iconfarm="1" ignored="0" />
    <contact nsid="27009262@N00" username="Dion Hinchcliffe" iconserver="7"
        iconfarm="1" ignored="0" />
    <contact nsid="80095026@N00" username="fanstone" iconserver="53" iconfarm="1"
        ignored="0" />
    <contact nsid="80739942@N00" username="keystone1111" iconserver="3128"
        iconfarm="4" ignored="0" />
    <contact nsid="46752978@N00" username="kisco" iconserver="34" iconfarm="1"
        ignored="0" />
    <contact nsid="92455005@N00" username="laurenthaug" iconserver="4"
        iconfarm="1" ignored="0" />
    <contact nsid="89529267@N00" username="LynetteRadio" iconserver="40"
        iconfarm="1" ignored="0" />
    <contact nsid="92518516@N00" username="modahome" iconserver="120"
        iconfarm="1" ignored="0" />
    <contact nsid="73314839@N00" username="Naaaif" iconserver="110" iconfarm="1"
        ignored="0" />
    <contact nsid="20056291@N00" username="nicolasnova" iconserver="4059"
        iconfarm="5" ignored="0" />
    <contact nsid="21296916@N03" username="Paul Hughes: Design Thinking"
        iconserver="2186" iconfarm="3" ignored="0" />
    <contact nsid="54412022@N00" username="publicmind" iconserver="17"
        iconfarm="1" ignored="0" />
    <contact nsid="46557603@N00" username="Ralf Beuker" iconserver="2386"
        iconfarm="3" ignored="0" />
    <contact nsid="49147885@N00" username="squidish" iconserver="21" iconfarm="1"
        ignored="0" />
    <contact nsid="36112663@N00" username="tangyg" iconserver="0" iconfarm="0"
        ignored="0" />
    <contact nsid="34862120@N08" username="think.smith" iconserver="3126"
        iconfarm="4" ignored="0" />
</contacts>
</rsp>
```

iv Public Photos of User Contacts

```
http://api.flickr.com/services/rest/?method=flickr.photos.getContactsPublicPhotos
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&user_id=67526850@N00

/**
  api_key (required): api key
  user_id (required): user id
  count (optional): number of photos. default 10 - max 50. only used if without
      parameter 'single_photo'
  just_friends (optional): if 1 returns only photos of family and friends
  single_photo (optional): only returns the last photo of each contact
  include_self (optional): if 1 includes photos of the user (specified in
      'user_id')
  extras (optional): extra information (license, date_upload, date_taken,
      owner_name, icon_server, original_format, last_update)
**/
```

```
<rsp stat="ok">
<photo id="120292580" secret="fca8637ab6" server="47" farm="1"
    dateuploaded="1143731314" isfavorite="0" license="5" rotation="0"
    originalsecret="fca8637ab6" originalformat="png" views="10163" media="photo">
    <owner nsid="67526850@N00" username="Alex Osterwalder" realname="Alexander
        Osterwalder" location="Genvea, Switzerland" />
    <title>Web2.0 Business Model Characteristics</title>
    <description>The outcome of a short late-night brainstorming session on the
        characteristics of a Web2.0 business model. The reflections are based on
        what I write at my &lt;a
        href=&quot;http://business-model-design.blogspot.com&quot;&gt;business
        model design blog&lt;/a&gt;</description>
    <visibility ispublic="1" isfriend="0" isfamily="0" />
    <dates posted="1143731314" taken="2006-03-30 22:08:34" takengranularity="0"
        lastupdate="1202967678" />
    <editability cancomment="0" canaddmeta="0" />
    <usage candownload="1" canblog="0" canprint="0" canshare="0" />
    <comments>2</comments>
    <notes />
    <tags>
        <tag id="2017715-120292580-380852" author="67526850@N00" raw="business
            model" machine_tag="0">businessmodel</tag>
        <tag id="2017715-120292580-11227" author="67526850@N00" raw="web2.0"
            machine_tag="0">web20</tag>
        <tag id="2017715-120292580-2956157" author="67526850@N00" raw="business
            model innovation" machine_tag="0">businessmodelinnovation</tag>
        <tag id="2017715-120292580-2956158" author="67526850@N00" raw="business
            model ontology" machine_tag="0">businessmodelontology</tag>
```

```
        <tag id="2017715-120292580-2109580" author="67526850@N00"
            raw="osterwalder" machine_tag="0">osterwalder</tag>
    </tags>
    <urls>
        <url type="photopage">
          http://www.flickr.com/photos/osterwalder/120292580/
        </url>
    </urls>
</photo>
</rsp>
```

v Latest Public Photos

```
http://api.flickr.com/services/rest/?method=flickr.photos.getRecent
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&per_page=10

/**
  api_key (required): api key
  extras (optional): extra information (description, license, date_upload,
      date_taken, owner_name, icon_server, original_format, las_update, geo,
      tags, machine_tags, o_dims, views, media, path_alias, url_sq, url_t, url_s,
      url_m, url_o)
  per_page (optional): number of result items. default 100 - max 500
  page (optional): page of the result. default 1
**/
```

```
<rsp stat="ok">
<photos page="1" pages="100" perpage="10" total="1000">
    <photo id="4771876711" owner="30428372@N05" secret="94ef60dcfa" server="4098"
        farm="5" title="Picture0136" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4771876775" owner="10047346@N00" secret="5cd4161426" server="4122"
        farm="5" title="Aberdeenshire" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4771876795" owner="29221546@N07" secret="4b06f6eb86" server="4080"
        farm="5" title="P1030381" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4771876809" owner="89235411@N00" secret="30b600dcd9" server="4123"
        farm="5" title="P1000277" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4771876827" owner="51617540@N07" secret="085439dc86" server="4095"
        farm="5" title="01winery1" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4772515510" owner="58562067@N00" secret="d1e84f605b" server="4134"
        farm="5" title="IMG_5164" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4772515590" owner="50585245@N06" secret="db914e1f92" server="4079"
        farm="5" title="DSC00558" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4772515614" owner="10887912@N03" secret="cc143872a3" server="4116"
        farm="5" title="IMG_2268" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4772515634" owner="32128624@N05" secret="f171de864e" server="4093"
        farm="5" title="DSC00216" ispublic="1" isfriend="0" isfamily="0" />
    <photo id="4772515640" owner="73657575@N00" secret="71e526d11e" server="4118"
        farm="5" title="Therion @ GMM 2010" ispublic="1" isfriend="0"
        isfamily="0" />
```

```
</photos>
</rsp>
```

vi Most Popular Tags

```
http://api.flickr.com/services/rest/?method=flickr.tags.getHotList
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&period=week

/**
  api_key (required): api key
  period (optional): time period of result. 'day' (default) or 'week'
  count (optional): number of result items. default 20 - max 200
**/
```

```
<rsp stat="ok">
<hottags period="week" count="20">
    <tag score="100">me2mobileme2photo</tag>
    <tag score="100">canadaday2010</tag>
    <tag score="100">tdf10</tag>
    <tag score="100">japanexpo</tag>
    <tag score="100">happybirthdayamerica</tag>
    <tag score="100">animeexpo2010</tag>
    <tag score="100">zurifascht</tag>
    <tag score="100">cosfest</tag>
    <tag score="100">glasto2010</tag>
    <tag score="100">huracanalex</tag>
    <tag score="100">macysfireworks</tag>
    <tag score="100">canadadayfireworks</tag>
    <tag score="100">happy4th</tag>
    <tag score="100">peachtreeroadrace</tag>
    <tag score="100">jul10</tag>
    <tag score="100">redwhiteandboom</tag>
    <tag score="100">goodwoodfestivalofspeed2010</tag>
    <tag score="100">rondevanfrankrijk</tag>
    <tag score="100">marincountyfair</tag>
    <tag score="100">stpaulscarnival</tag>
</hottags>
</rsp>
```

4.2 Delicious

Delicious is a social bookmarking service on the web launched in 2003, i.e., a service where its users can save and share bookmarks (URL adresses). It is important to stress that our study was done before the acquisition of delicious by AVOS Systems. Thus, the following content might be outdated.

4.2.1 Authentication

It is possible to access public data from Delicious in an anonymously way, by using the web feeds (a data format to publish content) service of the system. On the other hand, in order to access private data, the requests must be authenticated by using the OAuth – an open protocol to enable an application to access end user information from a Web service. The process of authentication is described in OAuth Authorization Flow web page[82].

i OAth Python Library There is a python library that supports Oauth authentication: oauth2[83].

4.2.2 Feeds

To access public data from Delicious, there are read-only data feeds[84], which adopted in our study, since our work is focused on public data. The response of feed requests comes in two possible formats – RSS[85] and JSON[86].

i Update Rate
Due to practical reasons, the Delicious system does accept update requests of the feeds very often. RSS feeds, for instance, may not be updated more than twice an hour. Requesting data more often than allowed by the system may result in HTTP 503 errors, indicating either that the requests were blocked or throttled by the servers.

ii Feeds Available
All feeds follow this base URL prefix:

```
http://feeds.delicious.com/v2/{format}/
```

Where the placeholder {`format`} is the feed format: `rss` or `json`.

The following parameters are accepted:

?count = {1..1000} limit the results – default (15).

?plain or ?fancy disable or enable HTML content.

?callback=js call allows the inclusion of a wrapper call. Only JSON data.

Additional placeholders used in URLs further described are:

{format} rss or json.

{username} user's login name on delicious

[82] `http://developer.yahoo.com/oauth/guide/oauth-auth-flow.html`
[83] `http://github.com/simplegeo/python-oauth2`
[84] `http://delicious.com/help/feeds`
[85] `http://en.wikipedia.org/wiki/RSS_(protocol)`
[86] `http://json.org/`

{tag+[tag+...+tag]} tag or intersection of tags.

{url md5} MD5 hash of a URL.

{key} security key that allows view private data.

iii URL Patterns for Feeds

- Recent bookmarks:

```
http://feeds.delicious.com/v2/{format}/recent
```

- Recent bookmarks by tag:

```
http://feeds.delicious.com/v2/{format}/tag/{tag[+tag+...+tag]}
```

- Bookmarks for a specific user:

```
http://feeds.delicious.com/v2/{format}/{username}
```

- Private bookmarks for a specific user:

```
http://feeds.delicious.com/v2/{format}/{username}?private={key}
```

- Bookmarks for a specific user by tag(s):

```
http://feeds.delicious.com/v2/{format}/{username}/{tag[+tag+...+tag]}
```

- Private bookmarks for a specific user by tag(s):

```
http://feeds.delicious.com/v2/{format}/{username}/{tag[+tag+...+tag]}
?private={key}
```

- Public summary information about a user:

```
http://feeds.delicious.com/v2/{format}/userinfo/{username}
```

- A list of all public tags for a user:

```
http://feeds.delicious.com/v2/{format}/tags/{username}
```

- A list of related public tags for a user/tag combination:

```
http://feeds.delicious.com/v2/{format}/tags/{username}/{tag[+tag+...+tag]}
```

- Bookmarks from subscriptions of a given user:

```
http://feeds.delicious.com/v2/{format}/subscriptions/{username}
```

- Private feed for of third-party suggested bookmarks for a given user:

```
http://feeds.delicious.com/v2/{format}/inbox/{username}?private={key}
```

- Bookmarks from network members of a given user:

```
http://feeds.delicious.com/v2/{format}/network/{username}
```

- Bookmarks from network members of a given user by tag:

```
http://feeds.delicious.com/v2/{format}/network/{username}/{tag[+tag+...+tag]}
```

- A list of network members of a given user:

```
http://feeds.delicious.com/v2/{format}/networkmembers/{username}
```

- Recent bookmarks for a URL:

```
http://feeds.delicious.com/v2/{format}/url/{url md5}
```

- Summary information about a URL:

```
http://feeds.delicious.com/v2/json/urlinfo/{url md5}
```

5 Conclusion

Folksonomies maintained by web systems are an important source of information. In order to access and manage them these web systems usually provide web APIs. As a partial result of a research we are conducting concerning folksonomies, we have developed the tool presented here, which can access, retrieve and store data from folksonomies.

In this technical report we showed the tool we developed, detailing the strategy to access folksonomy based systems. We also showed our work of a unified tag database, derived from the comparison of related work and models adopted by web systems.

References

[1] Delicious. `http://www.delicious.com`. Retrieved on November, 2011.

[2] Flickr. `http://www.flickr.com`. Retrieved on November, 2011.

[3] H. Alves and A. Santanchè. Folksonomized ontologies - from social to formal. *XVII Brazilian Symposium on Multimedia and the Web*, 2011.

[4] T. Gruber. Ontology of folksonomy: A mash-up of apples and oranges. *International Journal on Semantic Web & Information Systems*, 3(2):1–11, 2007.

[5] A. Mathes. Folksonomies - cooperative classification and communication through shared metadata. *Computer Mediated Communication*, 2004.

[6] P. Mika. Ontologies are us: A unified model of social networks and semantics. *Journal of Web Semantics*, 5(1):5–15, 2007.

[7] C. Shirky. Ontology is Overrated: Categories, Links, and Tags. `http://www.shirky.com/writings/ontology_overrated.html`, 2005. Retrieved on May, 2011.

[8] T. Vander Wal. Folksonomy. `http://vanderwal.net/folksonomy.html`, 2007. Retrieved on April, 2011.