nTags

Design and Implementation of a Net-Based and General Tagging System in Django to Improve the Search Quality Compared to Current Systems.

Master Thesis

submitted in partial fulfillment of the requirements for the degree

Master of Science

University of Applied Sciences Ravensburg-Weingarten
Faculty Electrical Engineering and Computer Science
Study program Computer Science

in cooperation with

Instituto Tecnológico y de Estudios Superiores de Monterrey

Olaf Gladis

31st of August 2010

Thesis Advisor: Prof. Dr. Martin Hulin
University of Applied Science Ravensburg-Weingarten
Faculty Electrical Engineering and Computer Science

Second Reader: Dr. Gabriel Valerio Ureña
Instituto Tecnológico de Monterrey

Declaration of independence

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated.

Weingarten, 31st of August 2010 Olaf Gladis

Abstract

In this thesis we will show you a new perspective, how to create a tagging system, that will improve the ways to search objects. Currently there are two ways: you use either ontology or tags. We will explain the differences and the problems of those systems. We will identify them and design a system that will be more resistant to those problems. We implemented this system and tests are showing that this system will be better or at least equally well.

Contents

1 Introduction

Note of thanks

I thank everyone who helped me to achieve my goal, Gabriel Valerio for believing in me from the first minute, Professor Martin Hulin for helping me out when I got stuck, my girlfriend Rachel for giving me the space and time I needed the last month and supported me all the time, my cousin Martin for the support and proofreading and testing, my room mates Felix, Cyril, Sarah, my brother Arne and my friend Krishan for helping me all the hours of tagging and testing.

Why did I choose this topic?

Many people asked me why I chose this topic. At the beginning, I worked at the homepage of my father. It was really old and I wanted to change not only the design, but the whole system. At the time I created a novel design I thought about the new features; How they could be implemented, and what would be realistic to expect. I thought about a nice photo library where you can add new photo, tag them, and they would automatically appear in one or multiple albums.

Because this is a hunting web page, there are many pictures of hunted animals, too. So I wanted to create something like a global filter to hide those dead animals for people who would be offended by them.

To make things easy for me adding new photos and not attaching the same tags repeatedly, I wanted a very simple system where you just tag *springbok* and the system would know that it's an animal. The other part—e. g. the photo library—would just query *animal* and would get the springbok in return. And even later if you add another animal to the system, you simply need to tell the system one time that *zebra* is an animal, too. There is no need to rewrite the source code or even restart the program.

After a while, I started to search for a free[1] tagging system with those features, but I didn't find one. Some time later, I thought more and more about that imaginary system. I focused my ideas on making the handling much easier and what the system should provide. In the end, I accepted that there is no system that will cover my needs, and I decided to implement it by myself.

In the meantime, my thesis started, and I had no clue what to do. I looked over my idea and saw that it was already complex enough to cover the requirements for my thesis. I had lots of ideas what to implement. So I took my time to research this topic and realized that many people already investigated about tagging system/tagging behavior and almost all of them pointed out the same problems that I had with the current systems [HGM06], [TS09], [WYYH09], [MBC+09]. I was surprised that those problems were already known for about five years [HGM06] and nobody created[2] a new system that would deal with those problems. They developed nice algorithms to save the old systems. In fact, I was happy about that I can contribute something useful to the open source community, and I can combine my hobby with my thesis.

1.1 Tim and Simon

We are using two fictional characters Tim and Simon. They are like Alice and Bob in cryptography[3]. Simon is a person like you and me, but most of the time he is *searching*. On the other hand, Tim is someone who creates content, and he is *tagging* it, so that Simon will find it while he is searching. It is a lot easier to follow Tim and Simon instead of the tagger and the searcher, so this is hopefully an easement for you.

1.2 Tagging systems

1.2.1 A tagging system in a nutshell

You have some objects and you want to attach some labels to those objects that you, or other people can find those objects easier. In this case, you will use

1 We meant free like freedom, not like free beer.
2 I checked only free open source programs for web systems. It is possible that there are some closed source or commercial products that are already using such systems, but I am not aware of.
3 http://en.wikipedia.org/wiki/Alice_and_Bob

a system that attaches labels—tags—to your objects. You can also query this system to get all objects with a given tag.

1.2.2 Important Definitions

A tag is a short string that keeps some knowledge and is attached or tagged to an object, in some cases another information like an amount, geographic information, etc. is attached (triple tags).

An object is everything that can be uniquely identified. This could be a photo, a video, a text file, a reference to an object in the real world like a DVD or a car, etc.

A query is a set of instructions, how to select the object. E. g. which tags should be considered and, which are excluded. Some systems will allow only a very simple grammar to build the instruction.

2 Currently used tagging systems

This chapter will give an overview of current tagging systems from two perspectives, first different systems compared to their type of tag organization and secondly by their power of their search queries. We will show the advantages and the disadvantages for those systems.

2.1 By tag organization

2.1.1 Flat systems

General information: These are the most common tagging systems. They are used by Flickr, Picasa, iPhoto, Google Code, stackoverflow, Blogger and many other programs and websites. Usually there is one extra field in the database where tags can be added as a comma separated list. Sometimes it is realized as a many-to-many relationship.

Advantages: The reason why so many people are using this system is, because it is very simple and therefore, easy to implement and not intensive in computing.

Disadvantages: On the other hand, there are a lot of problems with those systems. Let's say Tim and Simon are from different regions or even countries. They will use a totally different vocabulary. The probability for two people to apply the same term to the same object is 12%[GWFD87]. If both are "lazy" and don't apply more tags to the object, Simon won't find the content that was created by Tim. This is unsatisfying for the two of them.

Other problems are homographs—for example, *apple* could refer to the company or the fruit, *jaguar* to the animal or to the car. *Plane* could refer to an aircraft, to a tool or to a geometrical figure.

The last problem is, how can more generic queries be generated without removing a keyword? The system does not know how the tags are in relation among each other.

2.1.2 Taxonomies

General information: Taxonomies are used if the comparison of tags is needed, or the possibility to query more generally or specifically is wanted. The tags are placed in a hierarchy like the biological classification of animals, plants, mushrooms, etc. We can think of the tag organization like the file system structure with folders and files.

It is used by the F-spot photo manager in Gnome and Studio Line[1](another photo manager for Windows) and some others.

Advantages: Due to the fact that the tags are organized in a hierarchy, the system is able to give Simon some hints what tags are more general or more specific. This tagging system is really strong, Simon can approximate slowly towards his object, even if he doesn't know the specific name in the beginning.

For example, he is looking for a picture with a *Chihuahua*, but he can't recall the name (or forgets the *h* in one syllable). In that case, he could "ask" the system: "Give me all pictures with dogs" and asks for a more general group. There are only view groups, it is very likely that he also remembers that a Chihuahua is part of the *toy dog* group. And from there he will find *Chihuahua*.

Another example: Simon is writing an article, and he needs a picture with a small dog. He only knows *Chihuahua* and not the *toy dog* group or other toy dogs. In this case, he could query for Chihuahuas, and if he doesn't find a good picture he can tell the system to make a more general search and will get all the pictures with toy dogs instead.

In addition, Tim has the advantage, that he doesn't need to attach so many tags like with the *flat system*. Because of the hierarchy, he only needs to attach the most specific tags.

Last but not least, it can handle homographs. A tag called *plane* could mean the aircraft, the tool or the geometric figure, so the tag would be subsumed under those taxa(categories) to represent their different meanings.

1 since I don't have Windows on my computer I didn't test that software

Disadvantages: We have the same word problem like the *flat system* with persons from different regions.

For example, if we decide to differentiate between "*apple*" as a "*computer company*" and "*apple*" as fruit, we categorize "*apple*" as a "*computer company*" and as a "*fruit*". Let's presume we want now "*apple*" (the fruit) also within the category "*healthy fruits*".

That is possible. We divide the taxa "*fruits*" in two sub categories of "*normal fruits*" and "*healthy fruits*". If we have another category of "*bread*" and also the sub categories "*normal bread*" and "*healthy bread*", then most of us would want both healthy categories as a sub category of "*healthy food*". We could just copy those categories inside the "*healthy food*" category, but then we have three different apples—the "*computer company*", the "*fruit/healthy fruit*" and the "*healthy food/healthy fruit*". Now we have to decide either we want to handle homographs, or we want multiple hierarchies.

In case we choose us for the multiple hierarchies we have one more disadvantage. When someone updates the "*fruit–healthy fruit*" and adds a new fruit, then the same fruit must be added to "*healthy food–healthy fruit*" to keep the system consistent.

Another problems are the constraints from Clay Shirky that must be met that such a system will work well. He pointed those in his article [Shi05]. A small corpus of objects and stable and restricted entities is needed. The same is valid for expert catalogers (tagger), expert users (searchers) and an authoritative source of judgment.

We need a small corpus and stable and restricted entities so that nobody will lose the overview. The expert users are needed to file the objects and search for them at the correct places. The authoritative source is needed to make the rules because sometimes things are not easy to categorize. For example, some people would put an SUV in the category "*big cars*", but the government defined that SUVs are "*small trucks*". Music, by the way, has no such authority who defines categories for each song/band. That is done by the band itself. There are bands that sound very similar, but are playing in "different" genres.

If the tagger and the searcher are the same person this will cause fewer problems, because the tagger and the searcher have the same expertise and are also the authority and have the power to decide what is correct, but he needs to be consistent.

2.1.3 Ontologies

General information: Ontologies aren't tagging systems, but you can describe objects with the help of them and find them again. Ontologies are a great chapter of the artificial intelligence. For example, it is possible, when you have the following informations: Aristotle is Greek and all Greeks are human. The system could find the conclusion that Aristotle is human[1].

You might know the buzz word "semantic web" that's all about ontology. There is a semantic web search called Swoogle[2], but couldn't find any useful web page[3]).

Advantages: The advantages are, that Tim can describe every object really precisely. Something like "on picture 1 is Alice, and she is wearing a coat, there is also Bob, and he is wearing a pair of jeans and a sweater. They are talking.". The system must know that Alice and Bob are persons. Persons could wear clothes and talk to other persons (This is a very simplified example).

Simon could now ask for photos with more than two persons, or a person wearing a special cloth, or a person doing a certain action.

The big advantage of Ontologies are, that it is possible to reason information that is not said explicitly inserted.

Disadvantages: On the other hand, this will work only for a small amount of objects and rules, because each query will result in many calculations and transformations for each object and each part of information.

The same disadvantages for Taxonomies from Clay Shirky are valid for the ontology, too. Expert users are required because they need to learn a new language to describe the objects and query them. If this system should be used within a public web site and reach a lot people (that have no degree in computer science) a much simpler system is mandatory.

1 http://en.wikipedia.org/wiki/Reasoning
2 http://swoogle.umbc.edu
3 Neither for hunting nor for programming.

2.1.4 Some attempts to avoid problems

Homographs Noriko Tomuro and Andriy Shepitsen from the DePaul University trying to identify Homographs with the help of Wikipedia[TS09].

Hierarchy Paul Newman is trying to generate a hierarchy from a flat data set[HGM06]

2.2 By search capabilities

2.2.1 *AND* or *OR*

General information: This is the simplest search. Many systems just allow one of them. Picasa for example, searches for the tags with an *AND* conjugation, the Playstation blog[1] uses only an *OR* conjugation.

Advantages: The implementation of this search is also very easy: First the different tags are split and then directly translated into a SQL string. E. g. "`SELECT * FROM photo WHERE tag=`*`tag1`* `OR tag=`*`tag2`*". It is only one SQL query, what makes it fast!

Disadvantages: A more complex requirement which pictures are needed, like: "All pictures from Berlin or Munich with animals". This must result in two different queries that will be combined by hand. E. g. *`Munich`* `AND` *`animal`* and *`Berlin`* `AND` *`animal`*. That is not user friendly and will cost a lot of extra time.

Improvements: Some forums e. g. giving the possibility to search for specific words and also to exclude some. They use it for a full text search not for tags. Nevertheless, it would be easy to transfer this feature into a tagging system. We can define two queries, one for the selection and a second one to exclude some objects.

1 http://blog.us.playstation.com

2.2.2 Disjunctive normal form

General information: The disjunctive normal form[1] (DNF) is a logical formula which is a disjunction of conjunctive clauses and would look like this:
`( Munich AND animal ) OR ( Berlin AND animal )`[2]

Advantages: Because the grammar is so simple it is quite easy to build a tree from the query string. The string just has to be split at the *OR*s and each of those clauses has to be split again at the *AND*s. There is no need for loops or recursion. There is one more big advantage nearly every query can be transfered into a DNF[3].

Disadvantages: Even so, it is in some cases also a disadvantage, that somebody has to transform those queries into the DNF. Those transformations are not every time as easy as in the example above. Not everyone who wants to search can transform his query.

Even if Simon could transform his query, he will lose a lot of time for the transformation and again to enter the long string. However, this is good enough for systems that just send their own, predefined queries, so this transformation must only be done once.

Improvements: It is possible to add the feature to transform every boolean algebra query into a DNF.

2.2.3 A free combination of `and`, `or` and braces `(,)`

General information: This form has the same power as the DNF. The queries could be built like a mathematical equation. A more complex example would be:
`sunset and ( ( cloudless and mountain ) or ( tree and cloudy ) )`

When we add the negation operator, e. g. `!`, we could exclude tags without the need of two different queries. A query could look like:
`sunset and ( cloudless or nice clouds ) and ( !dead animals or (springbok and !rifle and !(hunter or huntress ) )`

1 http://en.wikipedia.org/wiki/Disjunctive_normal_form
2 The brackets are only for a better readability and could be ignored.
3 Compared to the possible queries of pure boolean algebra.

Advantages: There are no limitations to the depth of brackets[1]. The queries could be formed more similar to the spoken requirements.

This queries in the formal language theory would be part of the Chomsky hierarchy type 2[2] and not part of the Chomsky hierarchy type 3[3]. So Simon has a lot more freedom how to write his queries compared to the previously discussed query types.

Disadvantages: The effort to implement such a parser is much more complex than a simpler system like DNF. It consumes more execution time for long queries.

Improvements: The query language could be extended to add the possibility of querying tag relations also. E. g. “person wears sunglasses”

2.2.4 A context specific query

General information: As an example for such a query language, we will show you SPARQL[4]. SPARQL is a recursive acronym and stands for SPARQL Protocol and RDF Query Language. It is a query language for RDF data on the Semantic Web with formally defined meaning. And its syntax is very similar to SQL. If you want to know how many persons have the name Alice you could send this query:

```
SELECT COUNT(?person) AS ?alices
WHERE {
?person :name "Alice" .
}
```

Advantages: You can do much more than just query for the tags that are attached to that photo, you can query even for relations between the tags—e. g. “Show me the photos where Alice wears a necklace”.

1 except limitations of the computer (memory) or with the implementation (max recursion depth).
2 context free languages
3 regular languages
4 http://www.w3.org/TR/2009/WD-sparql-features-20090702/

Disadvantages: You need to be an expert user to search, since you need to learn the query language first. The implementation is quite complex and the execution time is even worse than the previous.

2.3 Conclusion

You read the pros and cons for the different tagging systems. You have millions and millions of documents that should be tagged that the searching time would be a problem. In this case, you should stick with a flat system and take the disadvantages with the search capabilities.

If you need to describe your objects as detailed as they are in the nature, including their relationships between them, you need an Ontology. As a trade-off you need to learn a new language for the description and the queries, and you can't tag much.

If the searching part is human you may choose the Taxonomy and a power full query language. If a computer will query always the same queries you might stick with a DNF query. Even so, the taxonomy needs expert users and an authoritative source of judgment since one tag can only be in one category.

However, in my case every combination had its disadvantages: My father doesn't have the time to learn a new description language to describe his articles and photos on the web page, and the visitors won't use a complex query language to search for photos. The taxonomy isn't the right choice, either since we want many people to use the system.

I didn't want to stick with a simple system because we would need to tag a lot more to give the end users a more or less good result. None of those systems had a good possibility to be multilingual. They didn't offer the possibility to define synonyms and there was no possibility to have tags in two different taxa[1]. These were the reasons why I decided to develop my own tagging system.

1 http://en.wikipedia.org/wiki/Taxonomy

3 nTags

3.1 Requirements

First of all, we want for our tags arranged in a *hierarchy*. We need to make multiple categories possible (like in Wikipedia) so the easiest attempt is to order the tags in a **net-based** structure. It should be possible that this system can be used as a collaborative tagging system. Therefore, **homographs** and **synonyms** are very important. It shouldn't be limited to photos—there should be **general relations** between tags and objects. The system should be able to handle *complex search queries* but should still be **easy to use**. Another nice feature would be **triple tags**. Last but not least, it needs to create **tag clouds**.

A short description of the different requirements

Hierarchy/net-based: The tags are in a parent-child relationship among each other. However, to be able to handle homonyms, a simple hierarchy isn't sufficient. The tags need the possibility to have multiple parents. This will result in a net-based structure.

Synonyms: Since different people are using different words for the same objects, it is useful to allow them to use their own words for the description and the search. The tagging system has to link between them.

General relation: The system should have the option to be used for many different objects at once.

Complex search queries: We want the possibilities of boolean algebra—as it was described in Section 2.2.3 on page 10—to build our queries. This should be trivial to handle for new users. Simple queries should be with the simple and more common systems[1].

Triple tags: triple tags are tags with attached information, like a *people*- or *persons*-tag with a people count value.

Tag cloud: A cloud for the most popular tags.

3.2 Design decisions

3.2.1 Redefining the tag

The first definition of a tag was very simple: "A tag is a string". Now we need to extend this definition to be able to fulfill the requirements.

One of the demands are to handle homographs. That means we need to distinguish between two different tags with the same text. The common way is to add a unique id to each element.

The next need is the support for synonyms. This could be represented as a list of ids.

For a hierarchical/net-based structure, we need two more elements: a list of children and a list of parents. It is important, that there is a list of parents, not just one parent. In this case, we can guarantee the capability to be present in more than one category.

Finally, we need a way to represent the language of the current tag. This could be realized with a short string.

The new formal description of a tag would be as 6-tuple of *id*, *text*, *language*, *synonym list*, *parent list* and *children list*. See the following formal description and the representation of the tag *giraffe*:

```
(<id>, <text> , <lang>, <syn_list>, <parent_list>, <child_list>)
(43 , 'giraffe', 'en' , [226,] , [213,] , [159,])
```

1 Just AND or OR without the possibility to exclude.

3.2.2 How to handle synonyms

Each tag could have synonyms. This relation is a symmetric relationship, this means that if the tag "*cloudless*" has the synonym "*no clouds*", "*no clouds*" will automatically need the synonym "*cloudless*".

However, let's say "*cloud*" has two synonyms "*cloudy*" and "*smoke*". "*Smoke*" has an additional synonym "*pollution*". Should "*cloudy*" be also a synonym of "*pollution*"?

Well, we could define that we want only the direct synonyms. It would save a lot of trouble while searching through the tree but that has also disadvantages. The user needs consistency. That means if someone searched for a "*no clouds*" picture at one day and found a great photo. He should find the same picture the other day just searching for "*cloudless*". Otherwise he will be confused. On the other hand, the organization of the tags would be very complicated. Tim has to consider which of the direct synonyms which parent/child relationship has, because this would influence the search results.

A simple and neat solution is when you can define a set of tags that are equally valid and could be replaced with each other. The persons who would organize the tags need to do much fewer steps this way. Let's assume there are two sets of synonym-tags and the admin notices that they should be one set. He just needs to add one link to combine them. The first solution would require to connect every single tag from one set to each tag of the other set or even worse: check each tag if it is a real synonym with each tag for the other set.

If you just need to check one synonym group for a valid translation into another language, and not for each individual tag, new translations can be done much faster.

I decided to choose the second approach to have a proper solution because this could save a lot of time for maintenance and better internationalization support for the future.

3.2.3 How to store the tags

There are two different ways how the system could achieve that.

Type 1: When a new tag is attached to an object, the tagging system will automatically add also all synonyms and all ancestor[1] (including uncles[2]) of that tag. The advantage is, that the system doesn't have to fold every tag every time a query is sent. So the same query code could be used for the flat system.

Type 2: The tag is attached immediately. When someone sends a search query the system will replace every tag with `( original_tag or synonyms or ancestors )`. The advantage is that it is very easy to attach new tags and the system doesn't need to look all the hierarchy up when he attaches the tag.

The disadvantage of the second type is that you have to do the replacement every single time a new query is sent. On the other hand you save a lot of connections.

One disadvantage of the first system is, for example, our demonstration photo site, has 1275 unique tags, 17713 attached tags for 3136 objects. The first solution would need 76848 tag connections for almost the same results. These are more than four times the tag connections. This information is redundant because you can look the "missing" tags from the tag relations up.

Another disadvantage for the first system is that a lot of photos need to be updated when the relationships between the tags are changing. When just new relationships are added everything should work fine, but when relations are deleted it could get complicated. Let's say the tag "*green*" was accidentally added as a parent to "*animal*", and it is removed again. Every object that tagged "*animal*" need to be checked. Afterwards the objects need to be verified that they haven't other tags that have "*green*" as their parent—like "*green grass*"—and delete the tag "*green*". Even now it is possible that the tag "*green*" was added manually.

The third disadvantage of the first system is that the query possibilities are limited. Assume Simon is searching for photos with one or more persons, but he doesn't want any unknown person. With Type 2 he just queries all pictures that have the tag person or one of its synonyms or descendants. In the second step, every photo with an unspecific person[3] will be excluded. The first system would exclude every photo again, because the unspecific tags are also directly attached. Only if the system knows what tag was attached originally he could manage that correctly. This, however, would need even more space in the database.

1 all parents and their parents and so on
2 the parents of the synonyms
3 These are pictures with one of the tags *person*, *male*, *female*, *child*, . . . but without considering the descendants

Our decision was to use Type 2 since this would result in a smaller and clearer structure.

3.3 The tag implementation

3.3.1 The simple Tag

The tag itself: For the implementation of the tag structure, we defined at section 3.2.1 on page 14, we used the class that is shown in listing 3.1. It contains the minimalistic class definition of a tag that is able to provide all the feature requests in Django[1].

A big advantage of Django is, that it comes with a powerful object-relational mapper, URL mapper and nice template mechanism. There is no need to think how to implement each table in the database or how to write correct and save SQL queries. It is possible to focus on the logic behind the web interface. The template engine helps a lot to divide strictly the logic from the visualization. This is helpful if a) the layout got changed or b) the program will be ported to a different language.

Listing 3.1: Tag class for net-based structure synonym capabilities and multilingual support

```
class Tag(models.Model):
    text = models.CharField(max_length=200)
    children = models.ManyToManyField('self', blank=True, null=True,
                                      symmetrical=False,
                                      related_name='parents')
    synonyms = models.ManyToManyField('self', blank=True, null=True)
    i18n = models.CharField(max_length=12)
```

The general relationship: This part describes the implementation of the general relationship with Django. We will describe the way how Django deals with it, so that the concept could be transferred to other programming languages as well. First we will show you a use case with just one set objects that could be tagged and the problems that will occur when that scheme should be updated to a second object.

1 Django is a powerful web framework, with an object-relational mapper, a built in admin interface, a template language and a powerful URL to function mapping. See http://djangoproject.com for more information

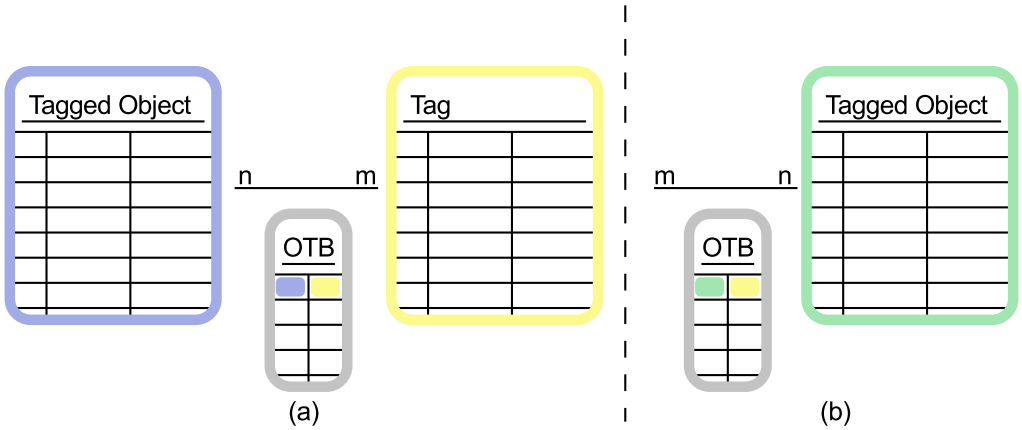

Figure 3.1: (a) The many-to-many relationship between the tagged object and the tag needs a third table—OTB, object-tag-bridge—in our database.
(b) if a second tagged object is added an additional OTB table is needed.

Figure 3.1 (a) is showing us the simple approach to add tags to one object. To connect the tags with objects with a many-to-many relationship, there is a need for a third object—let's call it object tag bridge (OTB). All those objects will be created as a table inside the database automatically and the OTB doesn't need to be define as a Django class. The ORM will independently look inside the OTB when someone asks for the list of attached tags. The second picture (b) shows how the database gets changed when an additional object is added. You can see that a second OTB is created, so for each new tagged object there will be two additional tables in the database. It would be great, if it is possible to add only one table.

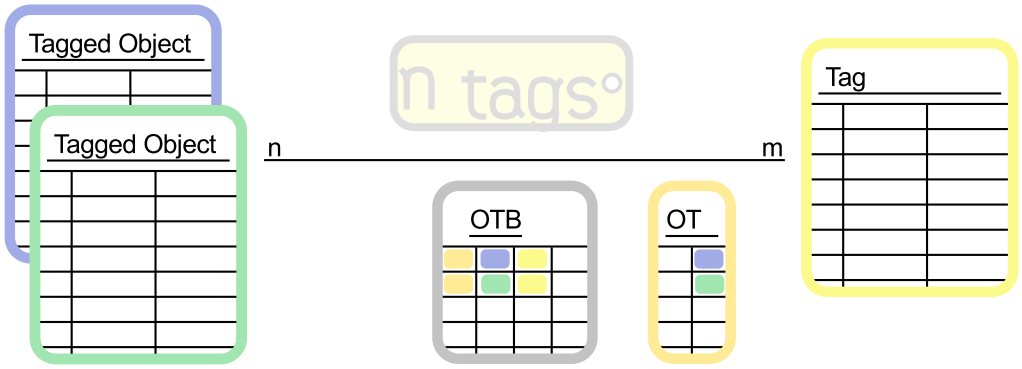

Figure 3.2: A general relationship requires a more complex OTB and one additional table for the object types (OT).

Figure 3.2 on the preceding page shows how Django will solve this problem with the help of a general relationship. In this case, the OTB gets a third column that defines, which tagged object type is meant. Now we have to define the OTB class by our self. With the tuple of the object id and the object type is each individual object uniquely identified. Our goal was, to make it as easy as possible. So we added a useful API to add, remove and clear tags from our objects in a way so the user won't know that there is an additional table involved in the background.

The self defined OTB class is mandatory to add the triple tag feature to our system.

3.3.2 Triple Tags

Triple tags are one step further to the ontology. They allow to add additional information to a tag. This could be the position of a person in a picture. A triple tag itself is a new Object that is attached to one specific tag-object relation—in our case to the *ObjectTagBridge*. Figure A.1 on page 50 shows the UML diagram for the relations between all related objects.

3.3.3 Tag cloud

There are two parts involved: a) counting/calculating each tag and b) displaying all/the most popular tags.

Gather the tag count

In a simple system, something like in listing 3.2 is done. We count for every tag the occurrences, if they are used, they will be put in a list with their name, the tag itself (if someone needs the tag to get a translation, etc.) and the number of appearances. In the end, the list will be sorted and returned.

Listing 3.2: outline of get_tag_list()

```
1 def get_tag_list():
      tag_counter = {}
3     ret_list = []

5     for tag in Tag.objects.all():
          tag_counter[tag] = ObjectTagBridge.objects.filter(tag=tag).
              count()
7
      for tag, amount in tag_counter.iteritems():
```

```
9           if amount > 0:
                ret_list.append((tag.text, tag, amount))
11      return sorted(l, key=itemgetter(2), reverse=True)
```

Unfortunately our job is a bit more complicated. Simon and every other person who will use the search doesn't need or even want the information how often each tag was used, they want the information how many objects will return for each tag. So we need to take all the ancestors and synonyms into account and add them to the actual value.

Because every tag could be replaced with a synonym and wouldn't change anything it's better to combine them and just print one of them in a tag cloud. In case the top 20 of the most popular tags are requested the tags "*human*", "*Mensch*", "*person*", "*Person*" and "*Typ*" will be returned with the exact same object count. This is a waste because if someone is reading one of them, he will know what the tags are standing for. In a worst case scenario, all the returned tags are synonyms. It would be better to group the synonyms and return 20 different tag groups.

Get representative tag

Since the group needs a specific name, we need to choose which of the tags will be representing the group. Therefore, we implemented the function with the name `get_representative_tag()`. It takes the preferred language—e. g. "en-us"—as one argument. It searches all synonyms for the correct language. Wether there is just one it will be returned. If it finds more of them, it will look for those with the flag "*is_representative*"[1] and return it . If there are still no tags it checks the current tag if it is from the same language and return itself. Otherwise the first tag with the exact language match will be returned.

If there are no tags with the exact language it will do the same again with a more generic language. In our case, the new language would be "*en*". Every tag where the language starts with "*en*" is considered. Now it is possible that the tag "*colour*" will be returned, even if US-English was requested in the first place. If even this pass will return no result the default language is used, and the same steps will be proceeded again. If all else fails, the current tag is returned.

This function can also be used to display the tags in the correct language, even if the tagger added them as a different one. And also to filter "bad" words for those

1 We implemented this flag only for this function. Since this flag is not important for other functions we decided just to show just a minimal version of a tag to help focusing.

who would be offended, without affecting the freedom of speech and block those totally.

Calculate the font size

This should be a very easy task, you take a set of tags count how often they occur and calculate a proper font size to display them. There are two different functions you could use to produce quite good results:

$$fontsize = \frac{max_{in} - min_{in}}{max_{font} - min_{font}} \times count_{tag} + min_{font}$$

and the logarithmic function[1].

$$fontsize = \frac{log(count_{tag}) - log(min_{in})}{log(max_{in}) - log(min_{in})} \times (max_{font} - min_{font}) + min_{font}$$

The difference of both formulas isn't always very visible, only if the difference between two following[2] tag amounts is too big. Figure 3.3 on the following page is showing a case where the linear font size algorithm will produce an unfavorable result. There is just one big tag—John—and all the other tags are very small. Since the tags are distributed logarithmic, you won't see small changes when the numbers are small. Figure A.4 on page 53 is showing the tag distribution of our test tagging system.

The second formula will compensate this fact. You won't see a big difference between a tag with a count of 1000 and another with a count of 1500, but you will see a difference between two tags with the count of 1 and 50. Figure 3.4 on the following page is showing the same tags with the same values of occurrences but with the logarithmic font size algorithm. You can now clearly identify the second most popular tag—John. This formula produces on average a larger font size. The penalty is, that we need more space to display the tag cloud.

1 I found the logarithmic function at google http://blogs.dekoh.com/dev/2007/10/29/choosing-a-good-font-size-variation-algorithm-for-your-tag-cloud/.

2 When you are sorting the tags by the amount of their occurrences.

Figure 3.3: A tag cloud with the linear font size algorithm with an extreme difference in the amount of the tags.

Figure 3.4: A tag cloud with the logarithmic font size algorithm with an extreme difference in the amount of the tags.

3.3.4 Tagging

This section will describe one possible integration of our net-based tagging system. The integration itself will strongly influence how someone has to tag. If the integration is poorly integrated, the system won't behave as good as it could. Our tagging system itself provides this solution only as an example. Since such a behavior is implemented mostly on the presentation part of a project and our tagging system is part of program logic everybody has a chance to implement the behavior the way he likes it. The design of Django pushes you that you strictly separate program logic and visualization.

When Tim is tagging his photos, he just enters the name of a tag. Meanwhile a list of tags will be queried that are similar to the name Tim entered. A small auto-complete-pop-up appears and shows him a list of possible tags. This is very useful when there are homographs—two words with the same spelling but a different meaning. In this case, there would be additional information displayed, so Tim could decide which is the right one. After the correct tag was selected the id of the tag and the id of the object are transmitted to the tagging system. See A.3 on page 52 for a possible sequence of AJAX calls while Tim is adding one tag.

The tagging system will create a new *ObjectTagBridge* and link it to the object and the tag. In the case, it should connect an unknown tag to the object, it will try first to find a tag with the exact spelling if this wasn't successful, it will ignore the case and search again. When more than one tag is retrieved an error is retrieved. After all when one tag was returned this one is used and if there was no tag a new one is created. See Figure A.2 on page 51 for a detailed flow chart.

Possible features

There are also some more nice features that could be implemented, e. g. look what tags are already attached to this object and check what tags are used often in that combination and suggest them—similar to [ZC08]. However, this would cost a lot of calculation time that might be more interesting as a desktop application. Lei Wu is taking a step forward, and wants also to take visual correlations into account[WYYH09].

3.4 The Search

The most important part of a tagging system is the search. People attaching tags to objects, so that they or others can find them easily later. Nobody would use a tagging system, just to tag objects and view the tags like comments.

In simple cases, you can just retrieve a list of objects that have one or more given tags attached. More complex system can also handle more complex queries with *AND*, *OR* and *NOT* conjugations.

For the next part, you should know Tim and Simon. Look at Section 1.1 on page 2 for more information.

3.4.1 A simple query

First of all, we want to check simple queries before we explain how complex ones are handled. A simple tagging system would transform the query—"`tree`"—from Simon to the SQL query: "`GET Objects WHERE tag='tree'`". Let's assume that we have a tag structure as shown in Figure 3.5 on the following page. In such a case, we should query for "`GET Objects WHERE tag='tree' OR tag='Baum' OR tag='arbol' OR tag='Eiche' OR tag='Kameldornbaum' OR`

`tag='camelthorn' tree OR tag='pino'` in order to retrieve all objects tagged with "`tree`", its synonyms or a more specific tree name.

Do you think that Simon would like to enter such a query in order to find every object tagged as a "*tree*"? I doubt it[1]. Since our system knows about the relationships and synonyms it could act a little more intelligent and transform the first query to the second one by itself.

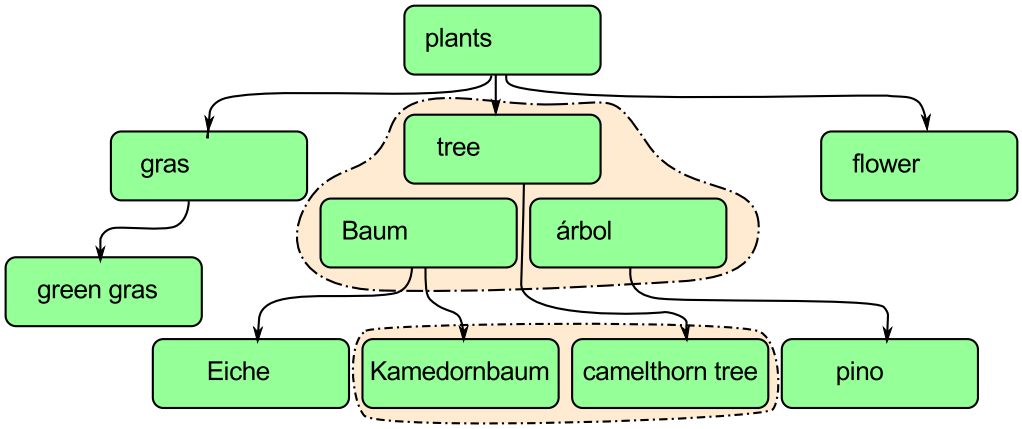

Figure 3.5: These are some tags with children and synonyms. The pointers are pointing from the parents to the children. The dashed fringes are showing the synonym sets.

The search - the simple way

The transformation of the search query is easy when you keep the image of Figure 3.5 in mind. If Simon searches for one tag—in our case "`tree`"—he wants to find all objects that are tagged with:

1. Load tree.
2. Load all synonyms of the previous step.
3. Load all children of the previous step.
4. If there were still new tags, goto back to step 2.

On the other hand, Tim is interested in search tags, which find his object. In this case, we are doing the same as when we are searching in the other direction. We are looking for the parents instead.

1 The test results from 4.2.5 on page 37 are showing that most of the searchers don't have the attitude to do it.

The search - a detailed view

To unfold the query for the tag "`tree`" we need to retrieve all synonyms for every tag[1].

We have a database and symmetrical relationship. So our system ensures that when someone adds the synonym "*arbol*" to "*tree*" it will automatically add the synonym "*tree*" to "*arbol*". It just adds the synonyms between those tags. The previously attached synonym of tree—"`Baum`"—doesn't get informed or changed in any way. So we would always get different result for the synonyms, depending which tag we are asking.

Since there is no explicit limit how deep the synonyms could be nested, we need to loop until we found all synonyms. Now we must be very careful how to design the loop so that we won't get stuck there infinitely.

The first implementation I used only the children links all ancestors. I optimized the code that it has to look just once at each tag to get all of them. Later I developed an even better implementation:

Get all synonyms: Let's take a look at Listing 3.3: We have two lists, the first one—*synonym_list*—contains every tag that is a synonym and that is already checked. We don't want to look at the same tags twice and maybe create an infinity loop. The *tmp_list* contains all the tags, that are identified as synonyms, but we haven't checked them yet if they have other synonyms by themselves.

Now we will loop until the *tmp_list* is empty. Each round we will pop one tag from the temporary list and retrieve its synonyms. The tag itself will be added to the *synonym_list*. The synonyms are added to the *tmp_list* but only if they are neither in the *tmp_list* nor in the *synonym_list*.

Listing 3.3: get all synonyms

```
1     def get_all_synonyms(self):
          synonym_list = []
3         tmp_list = list(self.synonyms.all())
          while len(tmp_list) > 0:
5             synonym = tmp_list.pop()
              synonym_list.append(synonym)
7             for syn in synonym.synonyms.all():
                  if syn not in tmp_list and syn not in synonym_list:
9                     tmp_list.append(syn)
          return synonym_list
```

1 The current tag, its synonyms and all children and other descendants

get all ancestors/descendants: This function is very similar to the previous one that catches all synonyms. Instead of looking for new tags from the synonym link, we are looking for them under the children/parents link. For each tag that is new, we also check for all synonyms and add them to the temporary list. We ensure that there will be no infinity loop in the same way as before.

Both functions, *get_all_ancestors()* and *get_all_descendants()* are using exact the same lines of code except for the pointer to the next tags. That is the reason why we decided to implement both functions just as a wrapper function which calls the *get_whole_tree()* function. It requires one additional argument—the function that will return either the children or the parents of the current tag.

3.4.2 The complex search

AND, OR and NOT queries

Most of the time Simon wants to retrieve objects that will be caught by more than one condition; for example, pictures showing the sky and an animal. Most of tagging systems let you search like this. Some of them let you choose if you want the results that match with all keywords or just with one of them.

If we want this in our system, we have to know that every tag is converted to all of its synonyms/children/ancestors and OR'ed together.

When Simon is querying all photos with the tag `tree` the system will return a list of photos where there is either the tag `tree` or `arbol` or `Baum` or one tag of the more specific tree names is attached. If he is looking for photos with *`tree` and `house`*, the system will check for the photos with a tree and the descendants, etc. and then checking those for a `house` or `building`.

If there is a NOT tag, the system has to exclude objects that have this tag.

Nested queries

We can get much more out of a tagging system when powerful search queries are allowed. That's the reason why our system allows nested queries such as "`sky and ( oryx or (springbok and !rifle) ) and dog`". Hereby the exclamation mark (`!`) is used to negate the tag `rifle`.

We will show a recursive approach to filter the objects correctly. First of all, we are trying to check the syntax of the string. Since there is no limitation for the

tag name[1], we can't check much. We can check the brackets. Are there as many open brackets as closed brackets and are they in the correct order?

The next step is to identify the top-level-braces. For example, "`A or ( B and ( C or D ) )`" would identify the first and the last bracket as one top-level-brace pair. We need this because we want to split our string at *OR*. If there are no *OR*s we would split at *AND*. However, we don't want to split at the *OR* between `C` and `D`—because it is inside our top level cluster.

For each new part, we call the function recursively and combine the result with a binary operation. If in the end one function call gets a query without a bracket and no reserved keyword[2], it will take the name as a string, or the id of the tag and return that tag, all the synonyms and all their descendants.

Nested queries and negation: Lets examine the negation of a nested element, e. g. "*animal* `and !(`*rifle* `or` *hunter*`)`". At the beginning, they will look for the top level cluster, after that they search for free `or`'s. Since the `or` is between the braces it won't be found. But there is an `and` where the string will be split and recursively passed to the same function again and the results conjugated with the boolean and.

The first part is handled like before but the negated part must be transformed to "`!`*rifle* `and !`*hunter*". This is a valid transformation from the boolean algebra, and now we are able to use the same function like before—filter and exclude, with the capabilities to conjugate these query sets with *OR* and *AND*.

3.5 How to use nTags

3.5.1 Our demonstration page

One example use case is a picture database. We have 3136 photos, 1275 different tags and about 17728 tag connections. Some tags were not attached to any photo, but they are useful for the search because they are parents of other tags (that are used). When you are using nTags to improve your own application you need to know that a chain is only as strong as its weakest link. In our case if you have a great application but your tagging system is weak, your whole application might look bad[3]. On the other hand, if you have a good tagging system, but

1 We allowed also spaces and special chars
2 A reserved keyword is either AND, OR, !, or with lower case.
3 It depends of course how important is it to search through your objects.

you implement it in a way that is not user friendly, then nobody is able to find anything either[1].

Autocompletion

This feature is not new. Google is using it for a long time now, but it is very handy. When someone starts typing in the text field to add a new tag, he will get a list of all tags with the same letters. In this case just two or three letters must be typed and then one tag from the suggestion list can be chosen. That saves a lot of time!

AJAX

We made massive use of AJAX calls. The auto complete is made with the help of AJAX, but it could be made with a static list too. There is also an AJAX call when a tag was selected. It is used to attach that tag to current object. Therefore, a HTTP POST message is sent to the web server containing the id of the tag, or if unknown the name, and the id of this object. If the tag doesn't exist, a new tag with this name is created. In the same way, it is possible to delete a tag again. After the link is clicked—that starts the AJAX call—the color of the text box changes and indicates the current status.

Keep it simple

Try to keep the design simple, that not much data/images has to be loaded each time a new site is visited. In this case, your visitors don't have to look for the search field and can find their stuff quite easy.

3.5.2 Other useful applications

Here is a short list of some applications that can be improved much with such a tagging system:

digital cocktail database

1 With the first prototype of the Photo Library, I tagged about 4 tags per minute, after a lot improvements, I and my roommate who never tagged before could attach 12 tags per minute.

digital cookbook A cookbook and a cocktail database are very similar. With the help of nTags it is possible to attach very specific ingredients. With a fuzzy search, it would be possible for a home chef to retrieve also the recipe for "Chili con Carne" even if the author tagged *Merlot* and the user informed the system that he has "red wine" available.

Another improvement would be an implementation of a user-configured filter/warning system for special ingredients in case someone has allergies or is a vegetarian. It would be possible to create simple rules to add automatically additional tags like "*vegan food*" or "*food with nuts*".

file database Many companies have a lot of files, where a lot of people need access to. Nowadays, most of them create a samba share an everybody can access them via the network. However, often a hierarchical structure is not good enough. They could use nTags to index all the files and folders from the file system and tag them. When a folder is tagged, all sub folders and files could also be referenced by this tag. In this case, they could tag fast and efficiently all their files and help their employees to retrieve the important project files.

media database It would be similar to the file database but specialized for media files. E. g. an electronic video rental shop or a library. Books are already tagged by librarians. With a net-based system it would be easier to search for those tags because you don't need to enter the exact same words.

file indexer E. g. the first step would define the important words of document. Clustering algorithms could be used for that, those important words would be tagged to the object. The next step would be to define synonyms, children, parents, etc. This part must be done by a human, but it would improve the search results a lot.

4 Tests

4.1 First tests

4.1.1 General overview

For our first test, we tagged nearly 3000 photos. The tester got a spoken order to search for photos that match specific criteria. He had to perform the same search on two different systems, one system was our system with the net-based structure and synonyms, the other one was a flat system. Both systems had exactly the same pictures and the same tags attached. The web interface was also identical. We build the flat system by adding an additional parameter to our search function. When this parameter was set, all relations like parent-child or synonyms would be ignored.

4.1.2 Test description

Our plan was to keep the number of the images that should be found clearly. Even so, the search query that should be entered also has to be compact. Our order was: “Find a woman who is at the beach and wearing sunglasses, but I don’t want to see the sea.”. The tester should now enter the search query and tell me which of the photos fits the given order. This test should be done twice, once for each system. We measured the time to be able to compare the correctness of a) the returned tags, b) the answer from the tester and c) the time that he needed for each system.

4.1.3 Expected results

I was sure, that the test persons will be faster when they are using our system, even if they need a little more time to enter the more specialized query. However, to go through all photos and check them manually kill a lot of time. The flat system would provide less accurate results which would lead to more manual

checking and more search attempts. I also expected that our system will return more in quantity and more accurate results.

I also wanted to show the limit where our system isn't strong enough to give 100% exact results. Since it cannot describe relations between tags like "*Person A is wearing sunglasses*". It is possible to get false positives.

4.1.4 Experimental drain

We started with the flat system and gave the order. The tester entered: "`woman and sunglasses and beach and !(sea)`", there was no photo returned. He deleted everything and typed it again, he thought maybe there was a spelling error. Still, no photos returned. He discarded the "`and !(sea)`", again no photos. He discarded "`woman`„ and 12 photos appeared. After an examination of all photos, the tester gave me the ids of the three correct photos.

Right after that the tester should find the same pictures with our system. Right after the first try he got five pictures returned and selected the three correct ones.

4.1.5 The results

I canceled the test because I realized, that this test wasn't scientific enough. I hoped that my precautions were sufficient.

What went wrong?

I looked at all the returned pictures from our system and checked the tags. From the two wrong pictures that were returned, the first one was missing the `sea` tag[1] and the second one had a woman and a man who was wearing the sunglasses. Nevertheless, none of them had the explicit tag `woman`. Since I tagged those pictures[2], and I knew those persons, I just tagged their name and not woman, because this was redundant information in my eyes. I also wanted to finish the tagging process as fast as possible.

1 It was a very small part of the background. Most of the time only important objects are tagged.
2 We were about 5 persons who tagged the 3000 photos but these 5 pictures were tagged by me coincidentally.

The tendency of the result was correct, but it was much to strong. I realized that if someone is tagging content specially for a flat system, he is automatically attaching multiple tags with the same meaning. My next thought was to add additional tags to that system, that we could compare them better. The big question now was: “How?” For retagging all photos again for the flat system was not enough time left.

I could write a little script that imports X% of the calculated synonyms. However how can I choose the X? If I choose 100%, I have the same results like our system, if X is set to 0%, there are the problems like before. In this case, I could choose how “good” my system would be just by picking that number.

Even if I find a number and can explain why this number is correct, I have the problem to determine which synonyms or parents should be preferred. Just choosing it randomly wouldn’t represent the environment neither.

Another big problem was that the order was represented in a language. E. g. when I asked them to find pictures with roads in it, most of them typed in road or roads. Nobody tried street or streets. Even if I gave the orders in a different language, then they translated it and took the translation with the most similar spelling.

4.2 The second tests

4.2.1 Solve those problems

The fairest way to choose what tags should be used is to do the tagging again. I decided to tag 200 new random pictures. 200 because we didn’t have the time to tag more and random pictures because I didn’t want that the tagger chose exactly the same tags for every picture in a series. There should also not be a series of pictures at all, because if there are 200 pictures of few guys boarding down a dune, there wouldn’t be much freedom to search for.

The next big issue was the influence on the tester by the choice of words. Scientists all over the world have the same problem: How can a biologist observe animals and document their natural behavior, when his presence (could) will change the actions of the animals when they are aware of him. Or in quantum physics are the influences much deeper, when you observe (measure) such a quantum you will destroy it, because in that second when you are “looking” at it, it will lose, for example, his superposition and collapses into a single identity.

We need to reduce the influence as much as possible, but we need to push our tester in a direction to let them search for something that we can compare. When a set of people did the same (similar) tasks, we can do that.

We came up with the idea to show a little picture that contains the objects that should be in the photos we are looking for. Should these pictures be drawn or simple similar photos from the database as a reference? I chose the drawn picture, because here we have the most freedom to add or reduce the level of detail. We can add more details or remove them. It is very hard to find the correct photo with the minimum of details that shows everything. We could mark or point to the most important information but there are two problems with that. A) the tester will get influenced by the unimportant objects and b) it is more difficult to show that in my written thesis. The drawn pictures are self-explanatory and I don't need to worry about the printing quality, since they are black and white.

4.2.2 Tagging

Two people tagged 200 pictures in a random order for two times. The first time with the knowledge that they are using a net-based tagging system. The second time they are tagging the same pictures in a different order but for a flat system. This time it was impossible to look up the old tags that were attached the last time.

Tagger A (net-based system):

- 100 photos
- 40 min
- 406 raw tags (10.15 $\frac{rawtags}{min}$)
- $\hat{=}$1691 tags (42.28 $\frac{tags}{min}$)

Tagger B (net-based system):

- 100 photos
- 60 min
- 613 raw tags (10.30 $\frac{rawtags}{min}$)
- $\hat{=}$3012 tags (50.2 $\frac{tags}{min}$)

Tagger A (flat system):

- 100 photos
- 75 min
- 949 tags (12.63 $\frac{tags}{min}$)

Tagger B (flat system):

- 100 photos
- 77 min
- 934 tags (12.13 $\frac{tags}{min}$)

Analysis: As we can see, both Tagger A and Tagger B had around 10 tags per minute with the net-based system and about 12 with the flat system. There could be two reasons why this happens. The first, when they are tagging with the flat system, they knew the photos already and had the vocabulary prepared. The second reason could be that you need more time to think about the more specific tags.

In an interview after the tagging, one person told that the thinking time was notable. For the second tagger felt both idle times more or less equal. The change in the tagging rate changed from one tag every six seconds to one tag every five seconds. So in average both spend one second per tag longer with the net-based system.

The second time I told them that they have to tag for a flat system, and they should give their best to help other people to find those photos. This is the main reason why Tagger A tagged more than twice as much for the flat system. Tagger B, on the other hand, put much more effort for the tagging from the beginning.

Even if the tags per minute value for flat-system is higher than the raw tags per minute for the net-based, the calculated/folded tags per minute value of the net-based system is much higher than that. Even though Tagger A tagged twice as much for the flat system the amount of calculated tags compared to the tags of the flat system much higher.

Both taggers spent more time for the flat system and didn't get as many tags as the net-based got for it.

We also analyzed the specific tags and check if the tags that were attached to the flat-system was also attached to the net-based and vice versa. We counted the three different possibilities: The tags with the label *forgot* are those, that were tagged with the net-based system (or could be deduced from it) but not attached to the flat system. The tags with the label *both* appeared in both versions and those with the label *new* were used only in the flat system.

Tagger A:	Tagger B:
forgot 1118	*forgot* 1932
both 403	*both* 671
new 538	*new* 261

We have a correlation that largest set is in both cases the forgotten tags. This was expected not like relative high values of new items. We could explain it with the work of [GWFD87], he proved that the probability is about 12% when two persons should name a common object that they will choose the same name. I believe even when you are looking through your photos many times you will find

often new tags you forgot. That is one of the reasons why Wikipedia's articles aren't written in one piece.

Tagger A had a special hight value of *new* tags because in the first pass. When there were more people, he just tagged `persons`, the second pass he tagged `persons`, `man`, `woman`, `men`, and so on.

It might be interesting what kind of tags were in the *forgot* set. Most of the top entries where the German translation of the tags. When you are tagging some objects it is not easy to switch from one language to another rapidly. The next type of tags where the top level tags that only describe a set but wouldn't apply to a specific object. E. g. *actions* or *countries*. It is not common to search for `country` when someone is searching for photos from Namibia or Mexico. Tagger A used the net-based system also for the first time and didn't know its mechanisms completely. So he couldn't use the system (100%) efficiently.

4.2.3 Searching

I got four testers that performed the search test. None of them are tagging objects by themselves on a regularly basis. One of them tagged about 250 photos earlier. And two others had an education in IT. None of the testers saw the photos before the test.

Breadboard construction

We had seven sketches that were showing different objects. Each tester had to look at them first, describe in his own words what he is seeing. Then he should choose the tags he was looking for. He had the possibility to see a list of all tags that were attached to the subset of photos. He wasn't aware, which tagging system he is using. (Since nobody of them uses such a system regularly, it wouldn't change the results to a lot.)

My part was more the observer or coordinator. I didn't answer questions about what was meant by the sketches. I asked those questions: "What are you seeing in that picture?" "What tags are you looking for when you want to find similar pictures?" "Why are you using the specific query?" Sometimes when they were forgetting one object, I asked: "What do you also see in that picture?"

They had the possibility to look up synonyms or translations.

At the beginning they performed the search with our net-based system. After the first pass, I used exactly the same queries for the flat system. However, this

time from the second tagging pass. When there were significantly fewer results I asked them again. “What would you try next?”

For a better comparison I took those queries again and used them with our net-based system but the same (second) tag set.

4.2.4 The sketches

The sketches can be found in Section B.1 on page 55 until page 58. Since I don’t want to influence you too much, I describe the sketches in the Appendix on page 59. Now you have the choice to look a the sketches without reading the description and simulate how the testers tried to search, or read the explanation and compare it with your thoughts.

4.2.5 The results

I made a table for each person. Each row stands for one sketch. The three bigger columns represent the three systems—the net-based system with the first tag set, the flat system with the second tag set and again the net-based system but this time with the second tag set. The three smaller columns for each of those systems have the labels *ff* (found first), *t* (tries) and *fl* (found last). The found first represents the number of photos that were returned after the first try. A try counts also as a lookup if this specific tag doesn’t exist. And the found last represent the number of photos that were returned in the end. *t* and *fl* could be empty when all important photos were found with the first try. When *ff* is empty and *t* shows a - (minus sign), the tester gave up because he didn’t find the correct tag (combination).

Table 4.1: Test results: person 1

Bild	**Set 1 net**			**Set 2 flat**			**Set 2 net**		
	ff	**t**	**fl**	**ff**	**t**	**fl**	**ff**	**t**	**fl**
1	0	3	1	0	-	0	0	3	1
2	5	2	6	16	2	20	16	2	21
3	8			5			9		
4	-23	4	6	-14	4	3	-36	4	18
5	4			3			3		
6	-5	2	4	-4	2	2	-5	2	3
7	1	6	6	0	15	5	1	6	5

Table 4.2: Test results: person 2

Bild	Set 1 net			Set 2 flat			Set 2 net		
	ff	t	fl	ff	t	fl	ff	t	fl
1	1			0	-	0	1		
2	5	2	6	16	2	20	16	2	21
3	5			3			6		
4	0	6	5	0	-		0	6	18
5	4			-4	2	3	-4	2	3
6	-10	6	-	-10	-		-10	-	
7	1	8	-	1	-		1	-	

Table 4.3: Test results: person 3

Bild	Set 1 net			Set 2 flat			Set 2 net		
	ff	t	fl	ff	t	fl	ff	t	fl
1	-3			-3			-3		
2	5			16			16		
3	8			5			9		
4	-6	2	3	0	5	3	4	3	3
5	-5			-4			-4		
6	-5	4	-	-4	-		-5	-	
7	1	3	6	1	-		1	3	6

Table 4.4: Test results: person 4

Bild	Set 1 net			Set 2 flat			Set 2 net		
	ff	t	fl	ff	t	fl	ff	t	fl
1	0	3	1	0	-		0	3	1
2	1			6			6		
3	2	3	1	2	4	1	4	3	3
4	5			0	2	14	19		
5	0	2	4	0	2	3	0	2	4
6	4			2			3		
7	1	2	4	0	8	-	1	2	2

4.2.6 Analysis

When you are comparing the net-based systems with the flat system you will see, that there is not even one case where the net-based system needed more attempts than the flat system. They also have always more results, except number 2. This sketch showed *water* and *sand* (but most of them saw *sea* and *beach*). The reason is, that sea or beach are in some pictures only a little part of the background so Tagger 1 didn't tag those in the first tagging iteration.

Just person 3 found a solution for the first picture and the flat system for Sketch 7. He made a complete different search attempt. First he looked for the different objects, then evaluates them and just searches for the most important one. Since *scorpion* is more important than *wood* or *bark* he only searched for scorpion. There were only three pictures with scorpions in it, so it was fast to tell, which was the one with the *bark*.

The first tester took 15(!) different tries to find the correct pictures. This was just because I was sitting next to him, and he didn't want to ruin the test. I asked him afterwards if he would have such a stamina if he has been searching for himself, and he declined. He would have thought that this website doesn't have the right photos for him and give up after three of four tries. Even the other testers gave up sometimes.

It is very hard to merge the four tables because everyone did (slightly) different things. For example, the last one was very fast, but he tagged two weeks earlier also some similar pictures, so he knew the kind of tags that would be attached. He also found fewer photos. However, only because he identified more keywords for himself to look for. For example, sketch 3. Everybody else just queried for *bird*, but he saw *bird*, *flying* and *sky*. I didn't correct him because that would chang his behavior and maybe even influenced the upcoming search attempts.

The test persons should try to get an image in their mind and try to look for similar photos in our system. Since we want to know how the image looks like, we have only that way to pushing them slightly in that direction. The correct result can differ from person to person since everybody has a different sense and a different interpretation of the given pictures. My job was to know what he was looking for, and I knew the photos. When I knew that there were more interesting photos from the searcher's point of view, I didn't switch to the next sketch. This is one reason why the testers didn't give up as early as they would when they are searching just for themselves.

4.3 Conclusion

As you can see in those test results, my system is in both parts superior as the flat system. You can tag more efficient and search faster with better results. You will find your answer faster, when your synonyms are tagged and linked. However, you won't be worse in any condition.

5 Conclusion and future work

We showed you different current tagging systems, pointed their problems and showed some way how to solve or weaken them. Then we identified the source of the problems and showed a theoretical solution to avoid them. We described a basic implementation and proved with tests that our solution is better that the current most used web tagging system—the flat system.

The system is just in the beginning, there is a lot of more work to do, such as plural integration, defining a simple query language for triple tags. Some functions must be rewritten since I experienced other ways how achieve my goal in a nicer way. The cloud tag function for the net-based system is too inefficient for a productive use in the WWW.

The next big step would be to develop a possibility to exchange tags between multiple tagging system instances. So that everybody could use a set of global tags and add his own personal tags that would be meaningless for others. When somebody is tagging his photos with Tim, nearly nobody would know Tim and won't search for him. Or he doesn't want the others to say, hey that's Tim. A decentralized solution would be preferable because if it is just one super server that holds all the tag information, that would be the single point of failure. On the other hand, it would be possible that the organization would stop this server one day and all persons that are relying on that information have bad luck.

Tagging is very useful and also collaborative tagging or folksonomy is very powerful and there are a lot more possibilities in that so it should be kept clearly in mind.

Bibliography

[GWFD87] L. M. Gomez G. W. Furnas, T. K. Landauer and S. T. Dumais.
The vocabulary problem in human-system communication.
Commun. ACM, 30(11):964–971, November 1987.
(cited on pages 5 and 35)

[HGM06] Paul Heyman and Hector Garcia-Molina.
Collaborative creation of communal hierarchical taxonomies in social tagging systems.
Technical report, Computer Science Department, Stanford University, Stanford, CA 94305, April 2006.
(cited on pages 2 and 9)

[MBC+09] Benjamin Markines, Dominik Benz, Ciro Cattuto, Andreas Hotho, Filippo Menczer, and Gerd Stumme.
Evaluating similarity measures for emergent semantics of social tagging.
In *WWW 2009 MADRID! Track: Semantic/Data Web / Session: Mining for Semantics*, 2009.
(cited on page 2)

[Shi05] Clay Shirky.
Ontology is overrated: Categories, links, and tags.
http://shirky.com/writings/ontology_overrated.html, 2005.
(cited on page 7)

[TS09] Noriko Tomuro and Andriy Shepitsen, editors.
Construction of Disambiguated Folksonomy Ontologies Using Wikipedia, 243 S. Wabash, Chicago, IL USA, 2009. DePaul University, College of Digital Media.
(cited on pages 2 and 9)

[WYYH09] Lei Wu, Linjun Yang, Nenghai Yu, and Xian-Sheng Hua.
Learning to tag.
In *WWW 2009 MADRID! Track: Rich Media / Session: Tagging and Clustering*, 2009.
(cited on pages 2 and 23)

[ZC08] Valentina Zanardi and Licia Capra.
Social ranking: Uncovering relevant content using tag-based recommender systems.

Technical report, University College London, 2008.
(cited on page 23)

Glossary

A homograph is one out of a group of words with the same spelling but a different meaning[1]. E. g. bear (v)—to support or carry, bear (n)—the animal..

An object is in our case a uniquely identifiable object, this could be a file on a computer or even a car on the street. But to attach tags to it, you need to identify the object..

An ontology is a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to describe the domain[2].

A query is a list of tags and a description how they describe an object. It is used to find/retrieve objects from a tagging system..

Simon is a person who searches for content with the help of a tagging systems..

A tagging system is a system that handles the relationship between tags and objects. It is capable of returning a list of objects to a given query..

Tim is a person who creates content and attaches tags to them so that other people like Simon could find them..

1 http://en.wikipedia.org/wiki/Homograph

2 Definition from wikipedia, read more at http://en.wikipedia.org/wiki/Ontology_(information_science).

Appendix

A Diagrams

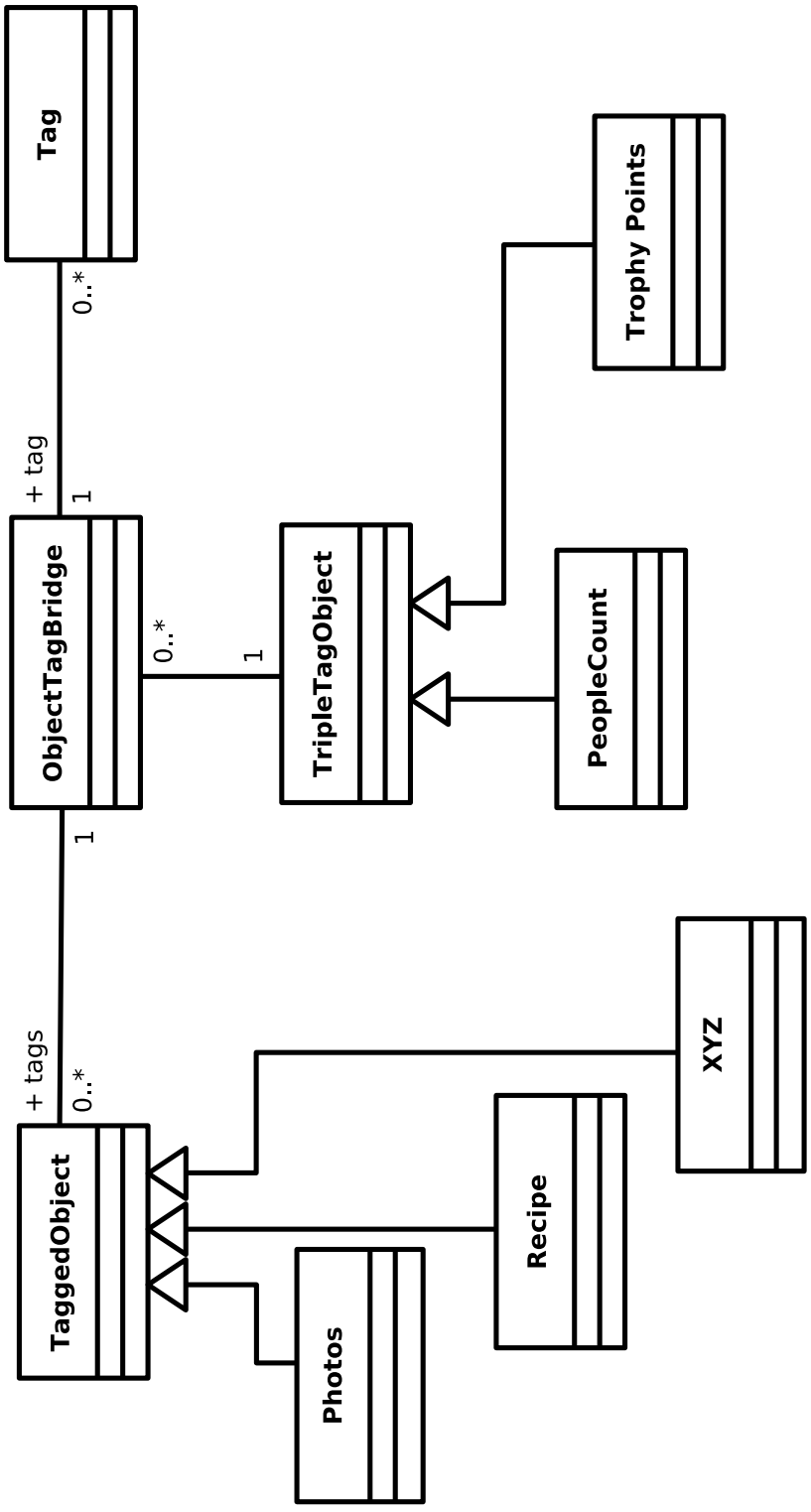

Figure A.1: UML relation for tags, tagged objects and triple tags.

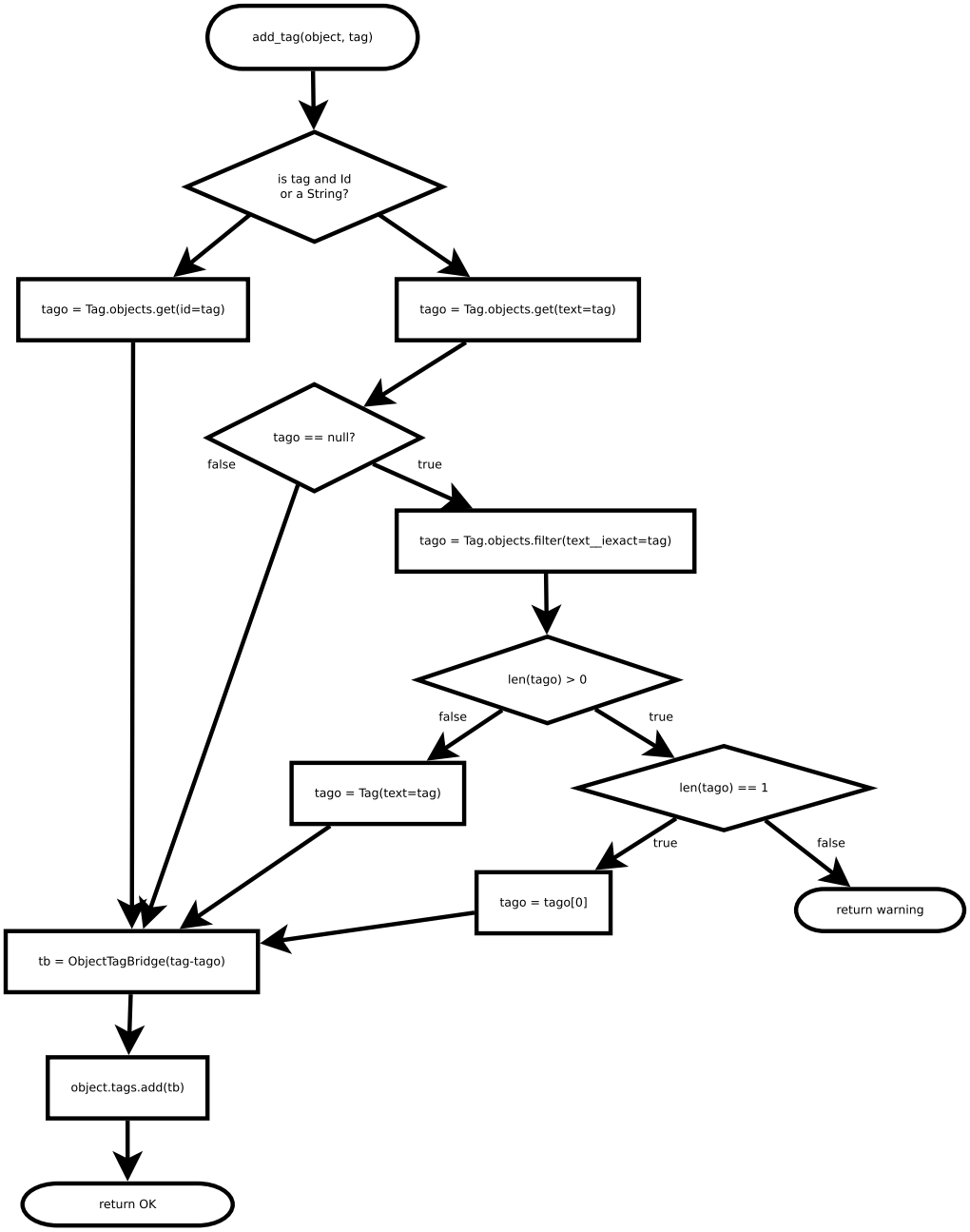

Figure A.2: Flowchart for add_tag().

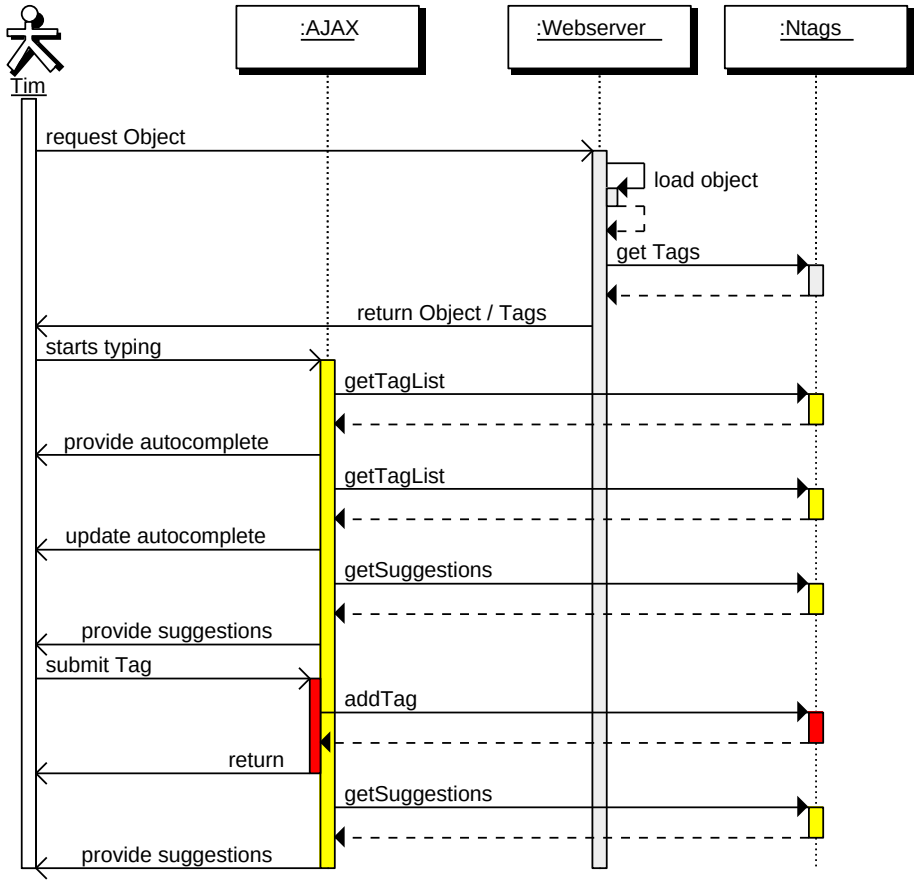

Figure A.3: This Figure is showing the calls that are being made while Tim is adding tags. First, The object is retrieved from the webserver. It will query the tagging system to retrieve the tags and send those with the object back to Tim. When he starts to type, a AJAX call is made directly to the tagging system (In fact, it is made to the webserver, but it is passed through directly.). That call transmits the characters that Tim wrote, the answer contains the names and the ids from similar tags. Every time the tag name changed too much a new AJAX query is sent. As soon as Tim decided for a correct tag it will be transmitted to second part of the tagging system, and added to the object. The last call could be a query to ask for suggestions.

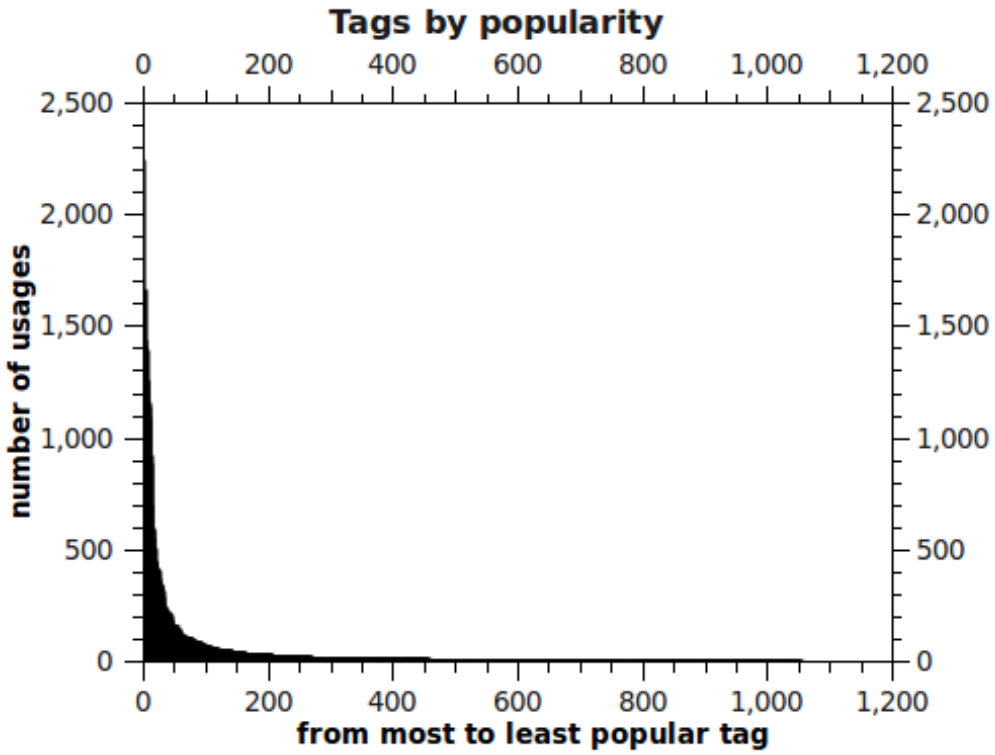

Figure A.4: Tag distribution by popularity.

B Test Sketches

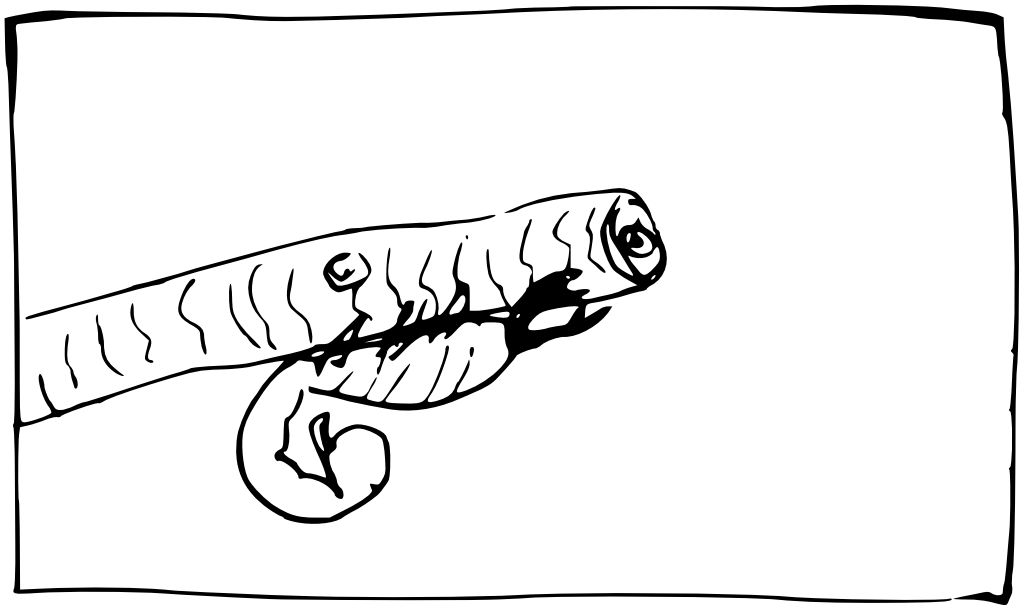

Figure B.1: Sketch 1.

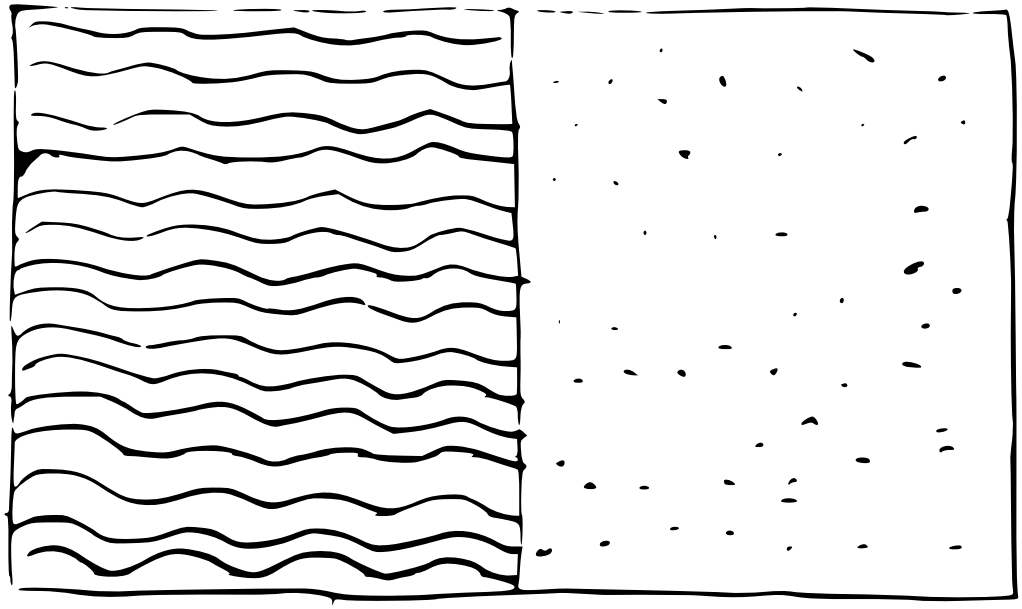

Figure B.2: Sketch 2.

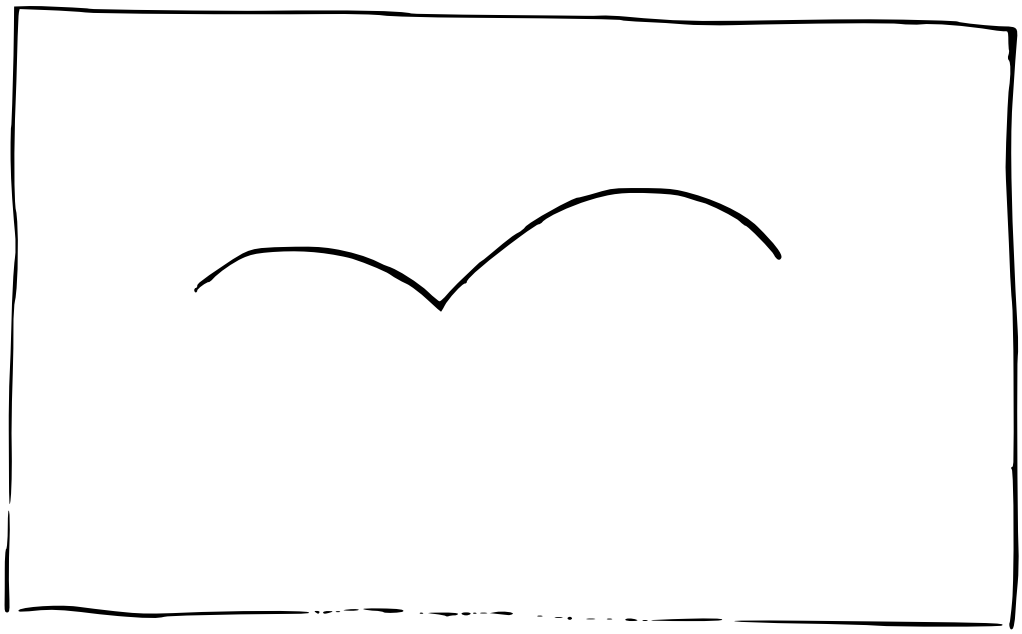

Figure B.3: Sketch 3.

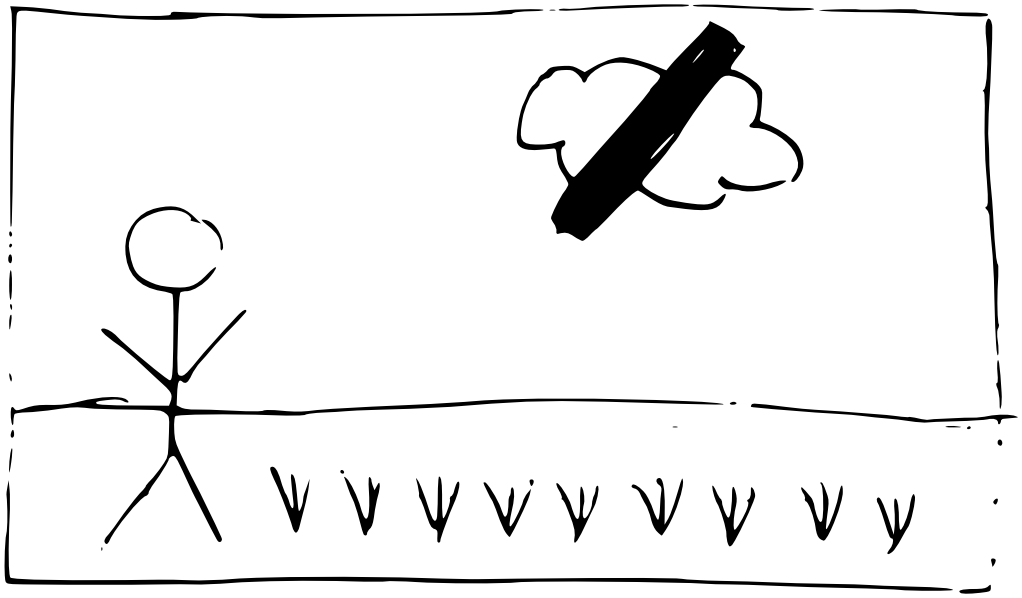

Figure B.4: Sketch 4.

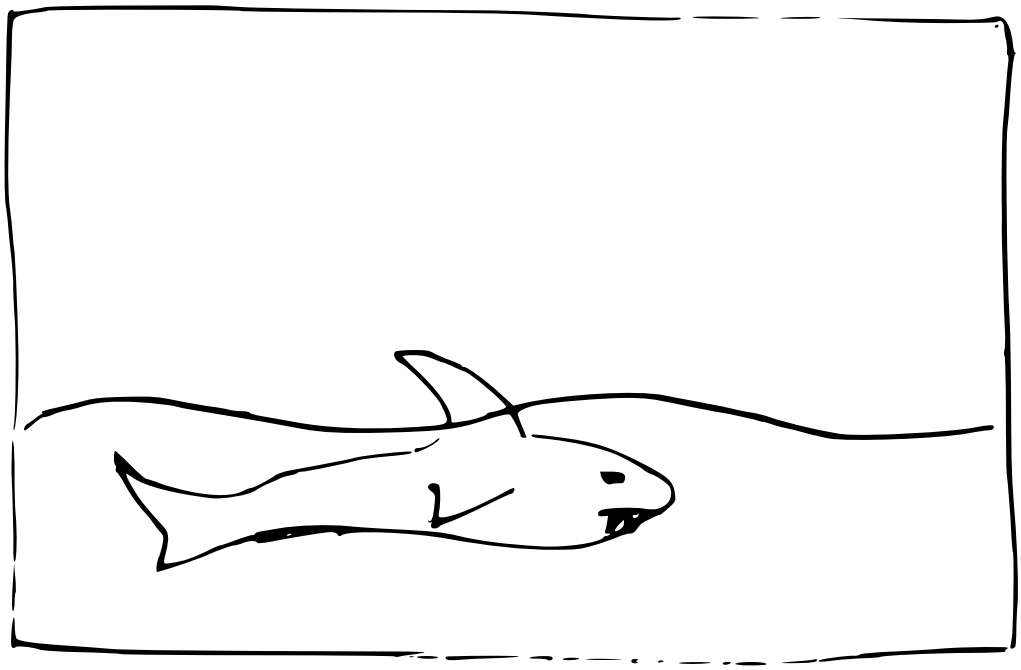

Figure B.5: Sketch 5.

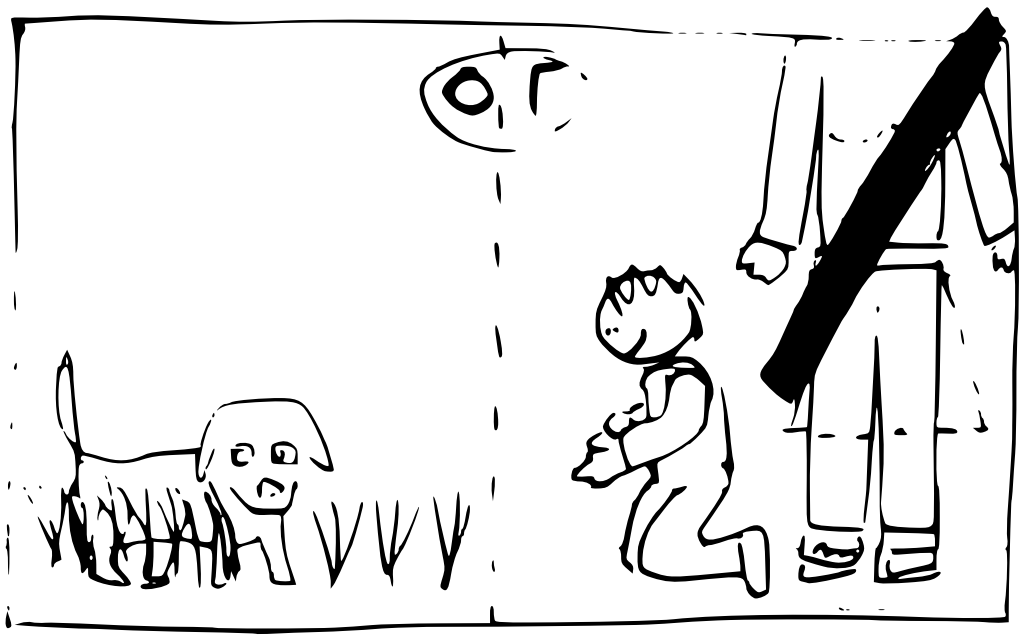

Figure B.6: Sketch 6.

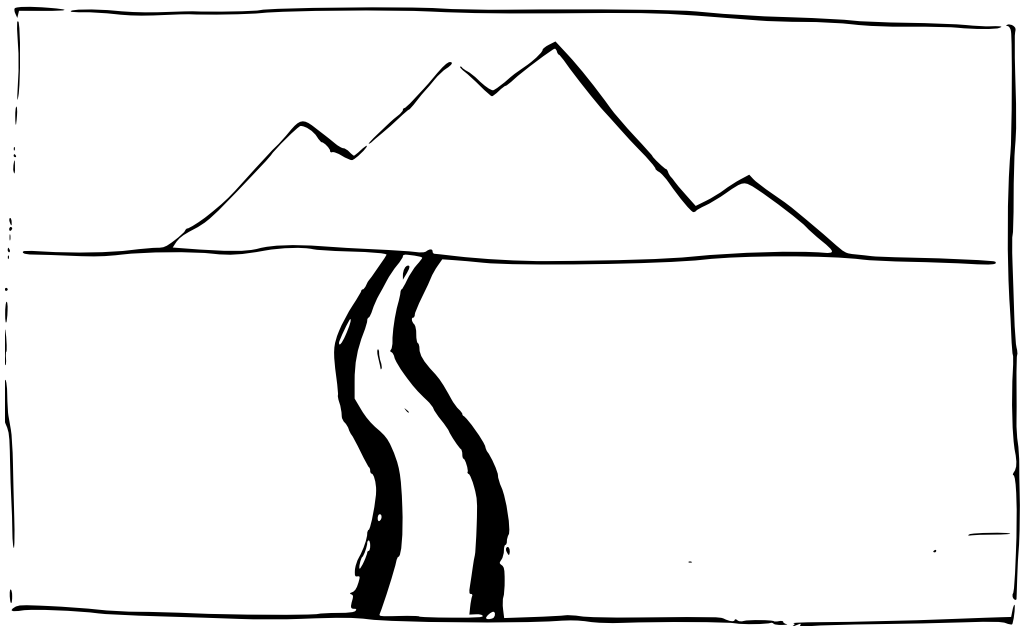

Figure B.7: Sketch 7.

B.0.1 Sketch description

Sketch 1

Most people saw a scorpion on a branch. Unfortunately, that picture was tagged with "*scorpion*" and "*bark*". The net-based system would return that if the user searched for "*scorpion*" and "*wood*".

Sketch 2

It should be "*water*" and "*sand*" but most of them saw just "*sea*" and "*beach*".

Sketch 3

Sketch 3 is showing a "*bird*". One tester associated also "*sky*" and "*flying*" with it.

Sketch 4

It should be "*person* and *grass* and *cloudless*" or "...and !(*clouds*)".

Sketch 5

This Sketch is showing: "*sea*" and "*shark*". Some saw also "*water*" instead of sea.

Sketch 6

This was the most complex sketch. The perfect result was when they searched for: "(*dog* and *grass*) or (*child* and !(*man* or *woman*))". Many testers tried !(*parents*)

Sketch 7

This sketch was more difficult than I expected. The testers should search for "*track*", "*mountain*" and "*sky*". All of them search for "*road*" and "*street*" or even "*river*" instead. Only a few tried tracks.