

## Worcester Polytechnic Institute Digital WPI

---

Major Qualifying Projects (All Years)

Major Qualifying Projects

---

April 2009

# EBAY QUERY LINGUISTIC SERVICE

Antoniya Toneva Statelova  
*Worcester Polytechnic Institute*

Radoslav Valentinov Petranov  
*Worcester Polytechnic Institute*

Follow this and additional works at: <https://digitalcommons.wpi.edu/mqp-all>

---

### Repository Citation

Statelova, A. T., & Petranov, R. V. (2009). *EBAY QUERY LINGUISTIC SERVICE*. Retrieved from <https://digitalcommons.wpi.edu/mqp-all/1073>

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact [digitalwpi@wpi.edu](mailto:digitalwpi@wpi.edu).

Project Number: DXF-EB92

**EBAY QUERY LINGUISTIC SERVICE**

A Major Qualifying Project Report

submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

---

Radoslav Petranov

---

Antoniya Statelova

Date: April 18, 2009

Approved:

---

Professor David Finkel, Major Advisor

# Abstract

---

In this project we designed, tested and implemented a query service for expanding and normalizing titles and searches. With regards to finding synonyms in titles, phrase recognition algorithms were developed as well. An additional service was created to find appropriate categories for a query to assist the search of synonyms. An application was then built on top of these two services to help users expand their searches and build advanced queries with no additional knowledge.

# Table of Contents

---

|   |    |
|---|----|
| Abstract .....  | 2  |
| Table of Figures .....  | 4  |
| Table of Tables .....   | 6  |
| 1. Background .....   | 7  |
| 1.1. eBay Overview .....  | 7  |
| 1.2. Query Expansion .....  | 7  |
| 1.3. Web Service Protocols.....   | 10 |
| 1.3.1. Simple Object Access Protocol (SOAP) .....   | 10 |
| 1.3.2. Representational State Transfer (REST) over Hypertext Transfer Protocol (HTTP) ..... | 11 |
| 1.4. Web Service Technologies.....  | 13 |
| 2. Synonym Look-Up Service (SLS) .....  | 13 |
| 2.1. Information Storage .....  | 13 |
| 2.1.1. Reading from file .....  | 14 |
| 2.1.2. Hash tables and vectors .....  | 15 |
| 2.1.3. MySQL Database .....   | 16 |
| 2.1.4. Comparison and Conclusion .....  | 16 |
| 2.2. JavaServlet Implementation of Service .....  | 17 |
| 2.2.1. Phrase Recognition Algorithms.....   | 18 |
| 2.2.2. Service Implementation.....  | 21 |
| 3. Category ID Suggesting Service (CISS).....   | 22 |
| 4. Advanced Listing Finder (ALF) .....  | 23 |
| 5. System details and Maintenance .....   | 29 |
| 6. Issues Encountered .....   | 32 |
| 6.1. MySQL Connector/J.....   | 32 |
| 6.2. XMLHttpRequest API in JavaScript.....  | 33 |
| References .....  | 34 |

# Table of Figures

---

|  |    |
|--|----|
| Figure 1 - Example of query expansion via the PubMed thesaurus (2) ..... | 8  |
| Figure 2 – The SOAP request structure .....                              | 11 |
| Figure 3 - The SOAP response structure .....                             | 11 |
| Figure 4 - The REST request structure .....                              | 12 |
| Figure 5 - The REST response .....                                       | 12 |
| Figure 6 - Look up for a keyword in the hash table scenario .....        | 15 |
| Figure 7 - State of SLS .....  | 18 |
| Figure 8 - Start with keyword1 .....                                     | 18 |
| Figure 9 - Add keyword2 and try again .....                              | 18 |
| Figure 10 - Start with keyword2 .....                                    | 18 |
| Figure 11 - Add keyword3 to phrase .....                                 | 19 |
| Figure 12 - Start with whole string .....                                | 19 |
| Figure 13 - Remove a word and try again .....                            | 19 |
| Figure 14 - Remove words until left with "keyword1" .....                | 19 |
| Figure 15 - Start with all the remaining words .....                     | 20 |
| Figure 16 - Look at remaining words .....                                | 20 |
| Figure 17 - Look at remaining words .....                                | 20 |
| Figure 18 - Start with first two words .....                             | 20 |
| Figure 19 - "keyword1" remains .....                                     | 20 |
| Figure 20 - Look at the next two words .....                             | 21 |
| Figure 21 - Consider the last window .....                               | 21 |
| Figure 22 - Phase 1 of ALF .....   | 24 |

|   |    |
|---|----|
| Figure 23 - Phase 2: Selecting a category .....             | 25 |
| Figure 24 - Phase 3: Select phrases .....                   | 26 |
| Figure 25 - Modified phrases update the synonym list .....  | 26 |
| Figure 26 - Phase 2 with Additional Options requested ..... | 27 |
| Figure 27 - Show results and refine search .....            | 28 |
| Figure 28 - Application Set Up.....                         | 29 |
| Figure 29 - SLS File Structure .....                        | 30 |
| Figure 30 - "main" table details.....                       | 31 |

# Table of Tables

---

Table 1 – Abstract example for SLS table ..... 16

Table 2 - Real world example for SLS table..... 16

Table 3 - Test results..... 17

# 1. Background

---

Before looking at the analysis and design of the web service for the project, it would be useful to become familiar with the history of eBay to clarify the relevance of this project to the company. Also some terms and protocols will be introduced, since they will be referred to further on in the paper. These include query expansion, Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) over Hypertext Transfer Protocol (HTTP). A brief introduction to the different languages available for developing web services will also be included.

## 1.1. eBay Overview

---

In 1995 Pierre Omidyar created “AuctionWeb”. At the time, “AuctionWeb” was the first of its kind, which is what led to its success; it allowed people to offer and seek items through it, thus creating an online marketplace. It started as a small project at first hosted on Omidyar’s personal computer only to grow incrementally into the company we today know as eBay. Today the company is open to 39 markets, has approximately 276 million users registered worldwide, and accounts for more than \$2039 worth of goods exchanged every second. eBay has grown to contain an online self-regulating economy where the people themselves determined the behavior of the market. [4]

The more people joined this network, the more information the company had to maintain; more entries were being listed, more users were signing up for accounts, and more changes were occurring concurrently on the site. Searching through all this information efficiently and maintaining it consistent prove to be a challenge. Also all the information collected from user activity provides a resource for analysis and improving designs. Analyzing emerging patterns in user behavior and making use of them leads to many possibilities for expansion and improvement of the implementation already in place.

The Research Labs were created at eBay for just those purposes – optimization (in searching, stream processing, computing), machine learning, visualization and cross-platform development. [5] The problem of search is just as important as it has ever been and has become extremely complex in trying to save time and space. Additionally, different searching principles have been developed to offer new kinds of services to people, such as clustering similar items or offering more specific product classification. Furthermore, human error in listings on eBay can create noise within the data on the website, so these errors have to also be managed or at least recorded. This grows into an even greater issue when you add the complexity of the natural language and all attempts at processing it using machines. All these issues are tackled by the eBay research labs who take on the endeavor to create new features and improve the already existing ones.

## 1.2. Query Expansion

---



As the amount of stored data on computer systems grows, so does the demand for its precise management. While storing information in many cases is a short process and takes constant time, retrieving it from the database may not be so easy and efficient.

The way searching works on most systems today is by using keywords. Whenever data needs to be retrieved from a database, the user specifies a number of terms that describe the desired information and then the searching mechanism goes through the database and determines what the suitable matches are. This, however, can be a very burdensome and inaccurate process in the sense that in many occasions the initial keywords may not return what the user needs. This is caused by a number of reasons such as the inability of the user to accurately describe the information that he or she demands, the existence of spelling mistakes, the difference in vocabulary and way of expression of the various database contributors.

One of the solutions to this challenging information retrieval (IR) problem is query expansion (QE). Query expansion is a technique that improves retrieval performance by reformulating the original query – either adding new terms or reweighing the original terms. [1] In the context of web search engines, query expansion refers to the evaluation of the user specified keywords and the expansion of the query using words or phrases with a similar meaning. This technique usually uses a thesaurus; a list of words that the query expansion terms are selected from. Other methods involve finding a statistical or linguistic relation between the query and a set of documents. A good overview of Query Expansion can be found in Efthimiadis' *Query Expansion*. [3]

Query expansion, based on the means of obtaining information, can be divided into three major categories: Manual QE, Interactive QE and Automatic QE. [3]

Manual query expansion is based on the ability of the user to give the system more precise information about the data he or she is looking for. In other words, manual QE demands user intervention. This technique assumes that the user has advanced understanding of the system, the indexing mechanism and the domain knowledge, which is rarely the case. [2] Figure 1 best demonstrates the difference between a simple user-defined query and a more advanced system-generated query.

- User query: cancer
- PubMed query: ("neoplasms"[TIAB] NOT Medline[SB]) OR "neoplasms"[MeSH Terms] OR cancer[Text Word]
- User query: skin itch
- PubMed query: ("skin"[MeSH Terms] OR ("integumentary system"[TIAB] NOT Medline[SB]) OR "integumentary system"[MeSH Terms] OR skin[Text Word]) AND (("pruritus"[TIAB] NOT Medline[SB]) OR "pruritus"[MeSH Terms] OR itch[Text Word])

Figure 1 - Example of query expansion via the PubMed thesaurus [2]

The first and third lines show simple user-defined queries. The second and fourth lines depict more complex versions of the same two queries respectively. Only very advanced syntax understanding would allow any user to create such sophisticated queries that would return more relevant information. This

inherent complexity is the main reason that makes manual query expansion an unsuitable technique for most available search engines.

Interactive Query Expansion refers to techniques where the user has some interaction with the system in the query expansion process. This set of techniques includes Relevance Feedback, which is based on collecting sets of users' opinions on the search results with which they have been presented. The results from that feedback then affect the future search results for that exact query; the search results which have been rated to be better go up in the list, the ones rated to be worse go down.

Automatic query expansion (AQE), on the other hand requires no additional input from the user, since all the work is done by the system itself. In this way, AQE provides an expanded search to the users with no additional feedback on their part, which saves them both time and patience. However, AQE is more unreliable than the previous two query expansion techniques, since the system might not be educated enough to "guess" for what the user was searching. Thus it is important to perfect this type of expansion, so it accumulates knowledge about queries and results most accurately and comprehensively. AQE includes techniques based on search results and knowledge-collecting structures.

AQE is additionally used in collaboration with IQE techniques in such a way that with every search the system updates the values of the query and documents it is searching through so that the next time the query is processed, the query value is matched up with more appropriately valued documents. This method also includes the generation of partial queries to enhance the search. This allows a broader spectrum of results, since the queries generated are not as restricting as the original one. On the other hand, this method of ranking documents doesn't relate the queries and documents, but rather holds collections of parts of queries which are related to one another. In this way, when performing the search those related parts can be interchanged or added to make the resulting set of documents more precise in its subject matter.

One method of storing such relations would be an association thesaurus: a particular list of words with weights, which can be used interchangeably within queries. The success of the exchange of words in the query is based on the term weights since they correspond to the relevance of each term to the list in the thesaurus. For example, the association thesaurus includes the word "bag" and the words associated with it are "purse" with 80% relevance and "backpack" with 70% relevance. This thesaurus information means that if a query contains the word "bag", there's an 80% chance we'd want to replace "bag" with "purse" and a 70% chance we'd want to replace it with "backpack".

Another method of storing is term clustering; it stores thematically related words, which when added to the query would make the search results a lot more specific. For the "bag" example, this will involve storing "bag", "purse" and "backpack" in one cluster and picking the most appropriate one for a specific query containing any one of them.

Similarly there exists term co-occurrence which stores parts of queries that are observed to often appear together, so the query could be expanded to include one part, if the other is already in place (3). For example, if you're searching for "bag", a popular phrase has been proven to be "travelling bag" so it will be suggested to you by the query expander.

## 1.3. Web Service Protocols

---

In 1998 Microsoft created a protocol for internet communication to replace all the middleware technologies at the time. This Simple Object Access Protocol (SOAP) would not depend on platform and language thus providing a standard for any client-server web service communication. They came out with the first version (SOAP 1.0) of the protocol in 1999 and two newer versions have been released since then (SOAP 1.1 and SOAP 1.2). Both newer versions of SOAP have become a well known and commonly implemented standard. But in 2000 with his PHD dissertation, Roy Thomas Fielding created a new school of architecture for web services – Representational State Transfer (REST). Fielding's idea proposed a web service based on a standard already ubiquitous in use - Hypertext Transfer Protocol (HTTP).

### 1.3.1. Simple Object Access Protocol (SOAP)

---

SOAP is an XML-based protocol that defines a set of rules for structuring messages allowing information exchange between applications. SOAP is not tied to any particular transport protocol but the widespread usage of HTTP makes it a very popular choice. SOAP allows programmers to create message exchange and client-server communication frameworks that are completely OS and programming language independent.

The best way to describe what a SOAP web service looks like is by giving an example. A typical SOAP client-server communication over the HTTP transport layer can be seen in Figure 2 and is described below.

```
GET /zipTemp HTTP/1.1
```

```
Host: example.com
```

```
Content-Type: application/soap+xml; charset=utf-8
```

```
Content-Length: nnn
```

```
<?xml version="1.0"?>
```

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
```

```
  xmlns:sth="http://www.example.com/weather-service">
```

```
  <env:Body>
```

```
    <sth:GetWeatherAtZIP>
```

```
<sth:zip>01609</sth:zip>
</sth:GetWeatherAtZIP>
</env:Body>
</env:Envelope>
```

Figure 2 – The SOAP request structure

The first four lines from the text in Figure 2 represent the HTTP binding of the SOAP message. Simply put they tell the HTTP protocol to send the XML code to the specified URI (in this case 'http://www.example.com/zipTemp') using the GET method. The XML itself is an example of the typical SOAP structure described in the 'soap-envelope' namespace. The entire message is enclosed within the <Envelope> node which in turn must contain the <Body> node. There are a number of optional nodes that can also be included within <Envelope> and <Body> but we don't need to focus on them at the moment. In this case the application on the server side will know that the requested zip will be stored within the <GetWeatherAtZIP> and <zip> nodes.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn
```

```
<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  xmlns:sth="http://www.example.com/weather-service">
  <env:Body>
    <sth:GetWeatherAtZIPResponse>
      <sth:zipTemp>85.2</sth:zipTemp>
    </sth:GetWeatherAtZIPResponse>
  </env:Body>
</env:Envelope>
```

Figure 3 - The SOAP response structure

Once the request has been received, the server has a number of options. It can either: read it, do something and not send a response (in the case of one-way message communication); it can read it and signal an error; or it can read it and send back an appropriate response. In the example above, the server sends the XML SOAP envelope preceded by the HTTP binding lines that simply notify the client if the request was successfully received on the other side. In the example shown in Figure 3, the server sends a response containing the temperature at the requested zip code.

## 1.3.2. Representational State Transfer (REST) over Hypertext Transfer Protocol (HTTP)

---

REST is an architectural style that can be summed up as four verbs: GET, POST, PUT, and DELETE. The verbs have the following operational equivalents:

- GET – read
- POST – create, update, delete
- PUT – create, update
- DELETE – delete

REST revolves around the idea of a more concise and less complex information exchange with less needless overhead. Instead of sending each request in a separate message, REST would use the URI and the HTTP method that it was sent by, to determine what the necessary action is. A service to get the details of a user called 'jsmith', for instance, would be handled using an HTTP GET to 'http://example.com/users/jsmith'. Deleting the user would use an HTTP DELETE, and creating a new one would be done with a POST.

To better illustrate the difference between SOAP and REST, we will re-write the simple client-server communication from the previous section, but this time following the RESTful style for writing web services.

```
GET /zipTemp/01609 HTTP/1.1
Host: example.com
Accept: text/xml
Accept-Charset: utf-8
```

**Figure 4 - The REST request structure**

The only information needed for a RESTful web service is the URI and the sending method used. The HTTP request in Figure 4 tells the server that it needs to return the temperature associated with zip-code 01609.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<sth:tempResults xmlns:sth="http://example.com/weather-service">
  <sth:zip>01609</sth:zip>
  <sth:temp>80.7</sth:temp>
</sth:tempResults>
```

**Figure 5 - The REST response**

The response (Figure 5) consists of an HTTP binding shell that describes that status and type of the request, and a short XML that contains that zip and the requested temperature. Note that the zip is only included for validation purposes and is not really crucial for the service in general.

## 1.4. Web Service Technologies

---

Since the discussed protocols SOAP and REST over HTTP are language independent, implementation of a web service can be done using one of several languages possible - Java, PHP, Ruby, Perl, or Python. Some of these (such as Java, Python, and Ruby) are more powerful than the others (Perl and PHP). Depending on the application of them, each has its pros and cons so let's discuss each further.

Java is the most ubiquitous object-oriented language among software developers today. The main reasons for its popularity include versatility, efficiency, platform portability and security. Additionally, Java has many developed libraries, which provide more than enough functionality for building and testing web services. The downside to Java though is that since it's platform-independent, its performance suffers. [6]

Similarly, Python and Ruby offer alternative general-purpose object-oriented languages, but they're not as widespread as Java. They each have their strengths; Ruby was created to be highly intuitive for the user, and in it everything is an object. Python, on the other hand, is a language which is open source and relies on the community-based development model. [7, 8] Their performance is significantly better than Java's when it comes to web services, but we still have room for improvement.

Both Perl and PHP are both scripting languages used for generating dynamic html pages, and although they aren't as rich as the previously mentioned languages, their performance is unmatched. They possess all the capabilities for developing a web service, however since Perl and PHP were explicitly created for explicitly for web services, they also perform with a much greater speed.[9,10]

## 2. Synonym Look-Up Service (SLS)

---

The Synonym Look-Up Service is a JavaServlet which runs on a Tomcat server, using a MySQL server as for a backend database. The purpose of the service is to read in a query that the user has entered, attempt to split it into phrases, and return those phrases with their known synonyms in JSON format. Phrases are recognized based on the data we have from synonym files given to us, since no other phrases really matter to us. Following are details on the decisions we made on how to store the data for our service and the algorithms to be used for phrase recognition.

### 2.1. Information Storage

---

EBay has a number of server applications that all use the Java-based Tomcat Servlet technology so we were advised to proceed and develop our web service also in the form of a Java Servlet to be hosted on a Tomcat server. Choosing a way to store all the synonym information, however, wasn't so straightforward. We had 3 possible options:

- Store all the information in the text files that we were provided with and just read from them whenever a request is made to the web service
- Store all the information in the server's random access memory and load it all whenever the application is started or updated
- Store all the information in a database such as MySQL and access it whenever a request is made or whenever the data needs to be updated

To make the right decision we developed and tested all three options described above. The files we started with were flat text files where every new row contains a group of synonyms. A row itself has the following structure:

```
Category MainKeyword <statistics for MainKeyword> Keyword2 <statistics for  
Keyword2> Keyword3 <statistics for Keyword3> etc.
```

Where all the keywords in a row are synonyms in the specified category; if we're trying to look up synonyms for any one of these keywords in the specified category, we will get the whole list of keywords in the row. Additionally, the MainKeyword in the list is considered to be the most often used word from this synonym group. As a rule, every keyword in the file is encountered only once, because in this way any keyword will only be associated with one synonym group from the file instead of having multiple synonym groups and becoming ambiguous.

## 2.1.1. Reading from file

---

The first and conceptually simplest method possible for storing this data is just keeping it in the original file we are given and every time we need to find a word's synonyms we parse through the file. This has a couple of obvious flaws – it will become slower as the size of the file grows, and problems with synchronization may occur, if the file needs to be updated or replaced. The big-O performance of this method will depend on the length of the file; this means that if the file has  $n$  lines of synonyms, the average runtime of the algorithms will be  $n/2$ , so the runtime will be limited by  $O(n)$ . On the plus side, this method of storing will use little runtime memory and won't require much, if any, population and updating time. This search when implemented just looked for any occurrence of a search word or phrase in the line of keywords, rather than looking for the occurrence of the exact phrase, which gave a greater possibility of finding synonyms quicker; this implementation gave us a rough set of results to compare its look up time to the rest of the methods' performances.

## 2.1.2. Hash tables and vectors

The next method for storing data was the complete opposite – the basic idea was to load the whole file into memory, thus entirely optimize the look up time. The initial concept was to store all words in a hash table, where each word was a key and it would be paired up with a value which was an integer. There would then be an additional hash table with keys which consisted of those same integers and values which were the list of words. Every word list look up would take 2 hash table calls. Looking at Figure 6, an example of the look up for Keyword3, we can see that we need a call to get the number of the group which Keyword3 belongs to, and then we need a second call to look up the string value for that group, which is the final piece of information we need.

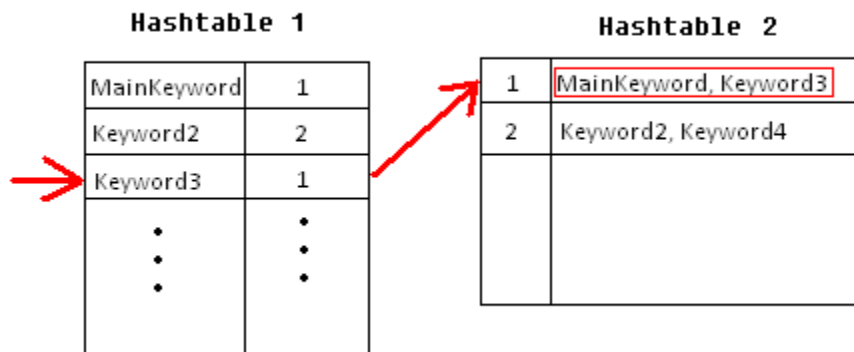


Figure 6 - Look up for a keyword in the hash table scenario

As the data stored grows, the performance of look up will not be affected, since every look up in a hash table takes  $O(1)$ , no matter how large the hash table. This means that the look up time for both hash table look ups will still remain a constant, i.e.  $O(1)$ .

This method of storing although very fast was estimated to use a great amount of runtime memory. The rough estimation was made on 30 thousand categories with approximately 16 thousand synonym group entries for each category, about 6 words per synonym group and 7 letters per word; considering a character is 2 bytes and an integer is 4 bytes, this evaluated to 48.2 gigabytes necessary in memory for storing these tables.

Since that was a huge amount of memory, we thought about still storing all the information in memory, but doing it without as much regard for time optimization. The information was stored in a vector instead of in hash tables; in this vector every row contained just the set of synonyms which were meant to be grouped together (entries similar to the second column of the second hash table in Figure 6). This slowed down the search significantly, but it still performed better than searching through the flat file. The memory-used estimate however was still up to a pretty large number – 37 gigabytes of memory to just load all this file information.



## 2.1.3. MySQL Database

---

The database is simple. We store all the information in a single table where each entry has 3 required components: category, phrase, and synonyms. 'Category' is an integer describing the item category in the eBay database. 'Phrase' can be a single word or two and more words merged together into a phrase. So 'Keyword1 Keyword2' and 'Keyword5' are both examples of valid phrase values in the sample database below. The last component is 'synonyms' which is just a long string that is made up of all the phrase synonyms separated by a delimiter, which in our case is a comma. To make things a little more clear, Table 1 shows an example of a couple valid database entries.

| Category | Phrase            | Synonyms                   |
|----------|-------------------|----------------------------|
| 12345    | Keyword1 Keyword2 | Keyword1 Keyword2,Keyword3 |
| 32424    | Keyword5          | Keyword4, Keyword5         |

Table 1 – Abstract example for SLS table

A real world example can be seen in Table 2.

| Category | Phrase    | Synonyms                 |
|----------|-----------|--------------------------|
| 34968    | brand new | brand new, brand new     |
| 23863    | Handbag   | purse, hand bag, handbag |

Table 2 - Real world example for SLS table

The performance of every MySQL database, similarly to the flat file and contrary to the hash table, depends on the amount of information stored in the tables. If the tables are relatively small, they are kept in cache memory, so no reloading is necessary for a look up to be performed and it will take a constant amount of time, i.e.  $O(1)$ . However the anticipated size of our table will most likely slow down the look up performance, resulting in a performance whose upper limit will be  $O(\log(n))$ .

## 2.1.4. Comparison and Conclusion

---

We created a test that calculates the average time to retrieve a record from the database on a sample file which we were given. We randomly selected 20 different phrases and found the related synonyms using each of the options that we described in the previous 3 sections: text files, hash tables, a vector and MySQL database. The population times are based on the only data set we were provided with, which reflects about a quarter of the phrases and synonyms in one single category. The results from our tests are summarized in Table 3.

|                   | Population Time  | Average Retrieval | Ram Usage | Big-O Notation |
|-------------------|------------------|-------------------|-----------|----------------|
| <b>Text File</b>  | 0 seconds        | 9.6 milliseconds  | 0 KB      | $O(n)$         |
| <b>Hash Table</b> | 149 milliseconds | 0.12 milliseconds | ~460 KB   | $O(1)$         |
| <b>Vector</b>     | 68 milliseconds  | 1.49 milliseconds | ~229 KB   | $O(n)$         |
| <b>MySQL</b>      | 140 seconds      | 6.75 milliseconds | 0 KB      | $O(\log(n))$   |

Table 3 - Test results

Several things were also taken into consideration when making the final choice:

- eBay has more than 30,000 categories and we only had data from one of them. Some categories are likely to have more information in them, making the file size even bigger. We, therefore, expect that the average retrieval time for looking up synonyms from a file is likely to go up for a significant number of the eBay categories.
- The excess of 30,000 categories also means that the RAM required to store all the information in vectors will be approximately 37 GB
- Although populating the MySQL table took 140 seconds on Antoniya's computer, it only took 83 seconds on Rado's machine. In other words, we expect that this time will be significantly decreased when the population script is run on a server.

In conclusion, we decided that using a MySQL database is the best tradeoff between retrieval speed and realistic RAM usage.

## 2.2. JavaServlet Implementation of Service

The Java servlet developed for accessing the synonym database was developed initially in two versions – one developed by Radoslav and the other by Antoniya. Both services were created to recognize phrases in the search string entered by the request, find synonyms for those phrases, and return the set of results to the application requesting the information. The main differences in how these services function consist in the algorithms used to split the request search string accepted and specifics on how phrases are recognized as such (whether they are exact or only partial matches).

The three main algorithms we came up with for phrase recognition were the following:

- You start with a blank phrase and add words to it, while it is still recognized as the beginning of a phrase; after a phrase is found, remove it from the search and repeat. This is most efficient when the known phrase contains few words and don't have similar words in them.
- You start with the whole search string and cut off words one by one, until you find a phrase in the search string; repeat the process with the remaining words which are not associated with a

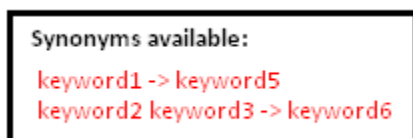
phrase yet. This is most efficient, if the phrases found in search string contain a large number of words.

- You have a window of  $n$  words and the algorithm checks if they constitute a single phrase; while the words in the window are not recognized as a phrase, keep cutting off the last one until they are. This is most efficient if we know the most words that a phrase can contain and it is the most common number of words (example: 2 or 3)

## 2.2.1. Phrase Recognition Algorithms

---

Let's look at a simple example to see the different behavior of these algorithms. Imagine the state of the service was as presented in Figure 7.

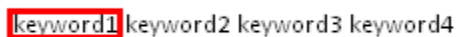


```
Synonyms available:  
keyword1 -> keyword5  
keyword2 keyword3 -> keyword6
```

Figure 7 - State of SLS

where the list of synonyms for “keyword1” includes “keyword5” and the list of synonyms for “keyword2 keyword3” as a phrase includes “keyword6”. Assume this service receives the search string “keyword1 keyword2 keyword3 keyword4”.

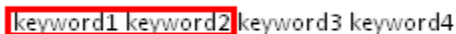
If the first type of phrase recognition algorithm were run on this string, these are the steps that would be taken for finding the phrases:



```
keyword1 keyword2 keyword3 keyword4
```

Figure 8 - Start with keyword1

- Consider keyword1 and check if it is the beginning of any phrase, and it is, since keyword1 -> keyword5.



```
keyword1 keyword2 keyword3 keyword4
```

Figure 9 - Add keyword2 and try again

- Since keyword1 was found as a beginning of a phrase, see if you can add keyword2 to that phrase and get synonyms for it. But “keyword1 keyword2” is not in our synonym table, so we remove “keyword2” and note that we’ve found a phrase “keyword1” and its synonyms “keyword5”



```
keyword1 keyword2 keyword3 keyword4
```

Figure 10 - Start with keyword2

- Start a new phrase, considering “keyword2”; we continue to add words, since it’s the beginning of a phrase – “keyword2 keyword3”

keyword1 **keyword2 keyword3** keyword4

Figure 11 - Add keyword3 to phrase

- “keyword3” is added to the phrase, so now we’re considering “keyword2 keyword3”

Similarly to Figure 9, “keyword4” gets added but the phrase is no longer found in the look-up table so we remove it and add “keyword2 keyword3” as a phrase with its synonyms “keyword6”. At last “keyword4” is considered and since it has no synonyms, it is added as a phrase by itself. All these steps result in the correct phrase separation of the query into “keyword1”, “keyword2 keyword3”, “keyword4”.

The concept of the second algorithms is the exact opposite – you take the whole search string and then start removing the words until you find a phrase.

**keyword1 keyword2 keyword3 keyword4**

Figure 12 - Start with whole string

- Check if the whole search string is a recognized phrase, but we do not have “keyword1 keyword2 keyword3 keyword4” in our look up table, so we cut off a word

**keyword1 keyword2 keyword3** keyword4

Figure 13 - Remove a word and try again

- Check if the remaining piece is recognized as a phrase; it also is not, since “keyword1 keyword2 keyword3” is not related to a synonym list;

**keyword1** keyword2 keyword3 keyword4

Figure 14 - Remove words until left with "keyword1"

- We keep following previous steps until we’re left only with “keyword1” (neither of the phrases “keyword1 keyword2” or “keyword1 keyword2 keyword3” are recognized as phrases); since “keyword1” by itself is recognized as a phrase (and there are no more words to remove), it gets stored

keyword1 **keyword2 keyword3 keyword4**

Figure 15 - Start with all the remaining words

- Reset the phrase to all the remaining words, and check if they're recognized as a phrase; since they're not, remove the last word

keyword1 **keyword2 keyword3** keyword4

Figure 16 - Look at remaining words

- Look at the remaining phrase "keyword2 keyword3"; it's found in our look-up table so it's added as a phrase to our list of phrases found

keyword1 keyword2 keyword3 **keyword4**

Figure 17 - Look at remaining words

- The remaining words consist only of "keyword4" so even though it's not recognized as a phrase, it's a single word so it gets added to the list of phrases

This results in completing the execution of the second method for phrase recognition. It again results in the query being split into the phrases "keyword1", "keyword2 keyword3", "keyword4".

The third and last algorithm uses a window of words, so for the purpose of our example we will make that window to be two words long. This means that similarly to the second algorithm explained we will take words and remove from them until phrases are found, however, we will not be taking all the remaining words for consideration but only the next two.

**keyword1 keyword2** keyword3 keyword4

Figure 18 - Start with first two words

- Consider the first two words, and since they don't form a phrase, remove "keyword 2"

**keyword1** keyword2 keyword3 keyword4

Figure 19 - "keyword1" remains

- The single word remains, so it gets added to the list of phrases; then consider the next two words

keyword1 keyword2 keyword3 keyword4

Figure 20 - Look at the next two words

- The next two words considered are “keyword2 keyword3” and they’re recognized as a phrase so they get added to the list of synonyms

keyword1 keyword2 keyword3 keyword4

Figure 21 - Consider the last window

- Since only one word remains, it gets considered by itself; a single word even if not a recognized phrase gets added to the list of phrases so “keyword4” gets stored

This achieves the same result for the list of phrases as the previous two algorithms. After all the phrases are known, their synonyms can simply be populated by direct look-ups in the table and returned by the service.

## 2.2.2. Service Implementation

---

Radoslav’s service implements the third out of these three methods. This service looks for the longest possible exact matching phrase which has synonyms. It starts with a string of length  $n$  and if synonyms are found, it records the phrase and continues the phrase search from the first word that is not part of the newly found phrase. Because of this, the algorithm doesn’t need to look for partial matches; the word which follows a phrase has either already been considered to be part of the phrase, or it makes the phrase contains too many words, and needs not be considered. The HTTP request for the service follows the syntax shown below:

```
http://host:8080/FFWS/rService/typeSearch/excludeReps/n/category/searchString
```

In this request, the category and search string are obviously the category in which to look for phrase synonyms and the search string, which to split into phrases. The typeSearch value is not so obvious – it can be set to one of 3 values – “single”, “synonyms”, or “generalize”. If “single” is chosen, then the whole search string is considered as a single string and it takes only one look up to get its synonyms. This setting requires no phrase recognition and it is the same for both services. The “synonyms” setting returns a JSON format of the phrases found and their synonyms, which is the general use of the service. The last setting possible is “generalize”, which can be looked at as a form of normalization – it returns the phrases found in the search string and the most common phrases that can replace them. For example, if you have the set of synonyms “Dolce and Gabbana” to be “D&G”, “Dolce&Gabbana”, etc., then if any of these phrases is recognized in a search string the value of their generalization will be “Dolce and Gabanna”, since it is the most commonly used phrase for expressing the idea.

The last unmentioned parameter in the search is the `excludeReps` parameter, which allows the user to ask for all synonyms which do not contain the current phrase within them. For example, if you're searching for "dolce", the results will include "dolce and gabbana", "d&g", "dolce & gabanna", etc. If `excludeReps` is set (vs. `notExcludeReps`) "dolce and gabbana", "dolce & gabanna" and all similar phrases containing "dolce" as a separate word will be excluded from the results. The reason for this is that if you search for `OR("dolce","dolce and gabbana")`, it will give the exact same results as just searching for "dolce", since the addition phrase performs a more restricted search instead of enriching the result list.

Antoniya's service on the other hand implements the first out of these methods. The HTTP request for the service follows a similar syntax to Radoslav's but does not require a value for  $n$ :

<http://host:8080/FFWS/aService/typeSearch/isExactMatch/excludeReps/category/searchString>

Similarly to Radoslav's service, here we see that the type of search, excluding repeating words, category and search string have to be specified, however as previously mentioned, the size of the window of words needs not be specified. Because of the way the service functions, it also allows the possibility of finding partially matching phrase, i.e. finding a set of words in the search string which represent the beginning of a phrase, but do not necessarily complete it. The method for phrase recognition the algorithm uses needs to use partial recognition, however after finding a partially matching phrase it can go back and cut off words until an exact phrase is recovered. But this takes a lot more time than Radoslav's algorithm, since first we build up a phrase and then reverse that break it back down, so it's a lot more inefficient at exact phrase matching. On the plus side, it will find a phrase of any length no matter what the request specifications are.

Both of these services are available on our server, and since they use a set of common structures can easily be maintained simultaneously. There is also the availability to extend the service to implement the third possible algorithm if at any point it proves to be a feasible design decision.

### 3. Category ID Suggesting Service (CISS)

---

The main purpose of Category ID Suggesting Service (CISS) is to take a query and then be able to suggest an appropriate set of categories to the user. Although created and refined in the last minute, this is probably the most essential part of the entire project as without the appropriate category we are simply unable to suggest any synonyms for the user's query.

We were given 3 text files that contain all the queries made on eBay in the past year. Each entry in this file contains a query followed by a list of all categories that users browsed when entering the specific query. The category ID's relevance is in turn evaluated by a number that represents the number of times this specific category ID was viewed each time a person entered the given query. To summarize that, an entry in this file contains of a query and a list of category ID's with appropriate confidence level. We were advised to just load the information in a huge hash table where each entry is a query (i.e. string) pointing to the most popular category ID associated with this query (i.e. another string). As one can imagine, however, these files are far from exhaustive and one can easily find out that a simple addition of a size unit or some additional detail can easily result in a query that does not exist in the file. This shortcoming would leave us with no category to suggest to the user and that makes our entire project practically useless because phrases have very different synonyms in each category. Without an exact category ID SLS doesn't know how to split the initial query and recognize the different phrases.

We, therefore, decided to take a little different approach to creating our CISS. First of all, we need to give the user a choice of category. So instead of just storing one category ID per query, we store the top 4 categories that the user used when searching for results. We then first match the entire query to our hash-table-based database and see if any matches exist. If they do, we return the entire trees of the top 4 categories and ask the user to choose one (category ID). If an exact match is not present we simplify and analyze the query by removing the last word and matching it again. This is repeated until a match is found. This technique assumes that each query is composed by adding more descriptive words only after the main objective of the query is established. In other words we assume that most people will write "shoes dolce and gabbana size 8" instead of "size 8 dolce & gabbana shoes". To be on the safe side, we also decided to also handle the case where the descriptive and not so essential words are added before those holding the main purpose of the query. If no match is found even after all words have been eliminated, CISS will remove the first word and repeat the process of eliminating words from end to front once again, until a match is found.

The drawbacks of this method are several. First of all, this technique can be quite time consuming for longer queries. On the bright side, however, it will only need to go through all possible combinations of adjacent words only if the user begins his query with less essential words and ends with the crux of the query (i.e. "black size 8 shoes" instead of "shoes black size 8".) Secondly, CISS will run till first match is found. A more reliable (and also more time consuming) approach would be to go through the entire query and keep track of the different existing matches. This would allow CISS to suggest a category with a higher confidence level. Another issue is the lack of unit support. In other words, instead of treating "4 GHz" as one word and removing it all together, the service will take an extra cycle to remove the phrase from the query. Unit support, therefore, can potentially make the service a little more efficient and reliable by both decreasing execution time and the chance of giving false results.

Another possible expansion, tied to CISS, that could enhance the entire ALF application with a better user experience would be an advanced category selection tool that would allow the user to still select a category if he is not happy with what CISS suggested (which is unlikely but possible nonetheless).

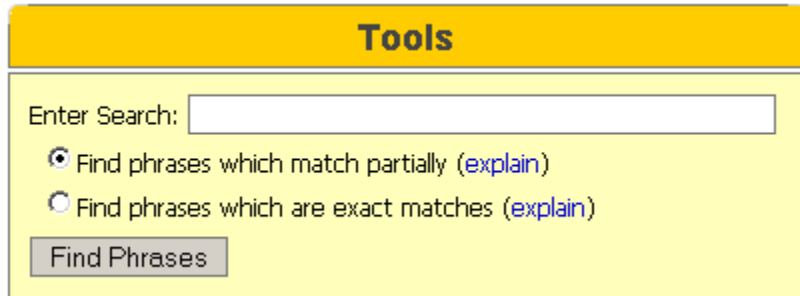
## 4. Advanced Listing Finder (ALF)

---



The next phase of our MQP consists of developing a web application that implements SLS and CISS, and helps eBay users to find more of the listings they need and less of those that are of no use to them. Our Advanced Listing Finder (ALF) works in four major phases:

*Phase 1:*



**Tools**

Enter Search:

Find phrases which match partially ([explain](#))

Find phrases which are exact matches ([explain](#))

Figure 22 - Phase 1 of ALF

We first ask the user to enter a search query in the provided input field. Once the “Search” button is clicked the user goes to Phase 2. The partial and exact matching of phrases corresponds to the settings for SLS, explained in Section 2.2.2.

Phase 2:

### Tools

Enter Search:

Find phrases which match partially ([explain](#))  
 Find phrases which are exact matches ([explain](#))

---

### Select Category

No category selected.

---

Clothing, Shoes & Accessories  
 Women's Accessories & Handbags  
 Handbags & Bags

---

Clothing, Shoes & Accessories  
 Men's Clothing  
 Shirts  
 T-Shirts, Tank Tops

---

Clothing, Shoes & Accessories  
 Women's Clothing  
 T-Shirts & Tank Tops

---

Clothing, Shoes & Accessories  
 Women's Clothing  
 Dresses

Figure 23 - Phase 2: Selecting a category

ALF sends the query to CISS which in turn analyzes it and returns the 4 most commonly viewed categories corresponding to this particular search. Instead of simply listing the categories we decided to give the user a little more freedom and show him the entire category trees. This improves the chances that he can select a category that fits his needs. Clicking on the "Select Category" button will lead the user to the next step.

### Phase 3:

The interface shows a search bar containing 'dolce and gabbana'. Below it is a table with two columns: 'Phrase' and 'We also suggest'. The 'Phrase' column contains 'dolce and gabbana'. The 'We also suggest' column contains a list of suggestions with checkboxes: 'dolce and gabbana' (checked), 'd g' (unchecked), and 'dolceandgabbana' (unchecked). Below the table are two buttons: 'SELECT ALL' and 'Deselect ALL'. At the top right of the table area is a 'Show Options' checkbox. At the bottom of the interface is a 'Show Results' button.

| Phrase            | We also suggest   |
|-------------------|---|
| dolce and gabbana | <input checked="" type="checkbox"/> dolce and gabbana<br><input type="checkbox"/> d g<br><input type="checkbox"/> dolceandgabbana |

Figure 24 - Phase 3: Select phrases

ALF then sends the query to SLS which in turn analyzes it, splits it into phrases and returns all phrases and all available synonyms for each phrase. We then provide the user with a very simple interface that informs him of the phrases and synonyms that SLS has found and also allows him to modify the list of phrases (Figure 24). At this point the user can select all phrases and synonyms that he would like to include in his search by marking the checkbox associated with them. If a modification is made to the search, ALF's interface changes immediately, providing the user with the new list of phrases and their synonyms. When a modification is made to the phrases themselves, ALF changes reflect a change in only the synonyms lists (Figure 25).

The interface shows the search bar with 'dolce and gabbana'. The table now has two rows. The first row has 'dolce and' in the 'Phrase' column and suggestions: 'dolce and' (checked), 'd g' (unchecked), and 'dolceandgabbana' (unchecked). The second row has 'gabbana' in the 'Phrase' column and suggestions: 'gabbana' (checked) and 'gabana' (unchecked). Each row has its own 'SELECT ALL' and 'Deselect ALL' buttons. The 'Show Options' checkbox is still at the top right, and the 'Show Results' button is at the bottom.

| Phrase    | We also suggest   |
|-----------|---|
| dolce and | <input checked="" type="checkbox"/> dolce and<br><input type="checkbox"/> d g<br><input type="checkbox"/> dolceandgabbana |
| gabbana   | <input checked="" type="checkbox"/> gabbana<br><input type="checkbox"/> gabana  |

Figure 25 - Modified phrases update the synonym list

We also give the user an option to incorporate advanced query syntax to his search. The advanced query syntax is completely optional and will show up only if the user clicks on the appropriate checkbox (Figure 26).

The screenshot shows a search interface with a search bar containing the text "dolce and gabbana something else for sake of example". Below the search bar is a table with two rows, each representing a phrase and its synonyms. The first row is for "dolce and gabbana" and the second row is for "something else for sake of example". Each row has a "We also suggest" column with checkboxes for synonyms and "SELECT ALL" / "DESELECT ALL" buttons. The "Search for:" column has a dropdown menu for each phrase, with the second dropdown menu open, showing options: "Leave as is (default)", "Apply OR operator", and "Apply quotations". A "Show Results" button is located at the bottom left of the interface.

| Phrase                             | We also suggest   | Search for: (explain)  |
|------------------------------------|---|--|
| dolce and gabbana                  | <input checked="" type="checkbox"/> dolce and gabbana<br><input type="checkbox"/> d g<br><input type="checkbox"/> dolceandgabbana<br>SELECT ALL<br>DESELECT ALL | <input type="checkbox"/> Hide Options<br>Leave as is (default) ▾<br>Leave as is (default) ▾<br>Leave as is (default) ▾ |
| something else for sake of example | <input checked="" type="checkbox"/> something else for sake of example<br>SELECT ALL<br>DESELECT ALL  | Leave as is (default) ▾<br>Leave as is (default) ▾<br>Apply OR operator<br>Apply quotations                            |

Show Results

Figure 26 - Phase 2 with Additional Options requested

This is achieved by attaching a drop-down menu to each phrase synonym. In this way the user is given the option to specify an advanced syntax type that he wants to apply to each different synonym. The four available options are as follows:

- **Leave as is (default):** This is the option selected by default. It treats each word in the query normally with no advanced syntax applied. It would return all listings that match all words in the phrase in any order. Example: used book
- **Apply OR operator:** This option uses the parentheses syntax which tells the API that any listings containing any of the words in this phrase is a good match. Example: (used, book)
- **Apply quotations:** This is the quotations mark syntax. It is especially useful for titles, names, or anything else where the order of words is important. It will return all results that contain all words in the phrase in the specified order. Example: "used book"
- **Apply asterisk:** This implements the asterisk syntax. It returns all listings that contain any word that begins with the selected phrase. Example: book\* would return listings containing any of the following: book, books, bookshelf, etc

Once the user has selected all the phrases and synonyms of his interest, he can proceed and click on the "Show Results" button. This will lead him to the final phase.

### Phase 4:

We send the new query to the eBay API, parse the results and display them on the right side of the same page, giving the impression that the user has entered the advanced query on the main web site. As this is taking place, ALF analyzes the first 50 listings and records each word in the listings and the number of times it is found in them. Since the results are sorted by “Best Match”, this ensures that ALF analyzes the best results for the user. The interface then presents its findings to the user in the form of the top 40 most frequently encountered words. He is then given an option to mark those that he would like to avoid finding in his search. Once the selection is made, the user clicks on the “Refine Search” button and the process is repeated but this time with newer listings and the user is given another option to refine his search even further.

The screenshot displays a web interface divided into two main sections: "Tools" and "Results".

**Tools Section:**

- Buttons: "Start New Search" and "Edit Previous Search".
- Section: "Select all the phrases you would like to EXCLUDE from your search:"
- Grid of checkboxes for exclusion:
 

|   |  |
|---|--|
| <input type="checkbox"/> new                | <input checked="" type="checkbox"/> sunglasses |
| <input type="checkbox"/> d&g                | <input type="checkbox"/> black                 |
| <input checked="" type="checkbox"/> t-shirt | <input checked="" type="checkbox"/> shades     |
| <input type="checkbox"/> d.                 | <input type="checkbox"/> free                  |
| <input type="checkbox"/> 2009               | <input type="checkbox"/> rock                  |
| <input type="checkbox"/> shiroi             | <input type="checkbox"/> neko                  |
| <input type="checkbox"/> ladies             | <input type="checkbox"/> size                  |
| <input type="checkbox"/> s                  | <input type="checkbox"/> men                   |
| <input type="checkbox"/> music              | <input type="checkbox"/> case                  |
| <input type="checkbox"/> sleeve             | <input type="checkbox"/> women                 |
| <input type="checkbox"/> white              | <input type="checkbox"/> dg                    |
| <input type="checkbox"/> 501                | <input type="checkbox"/> m                     |
| <input type="checkbox"/> shirt              | <input type="checkbox"/> short                 |
| <input type="checkbox"/> sexy               | <input type="checkbox"/> puck                  |
| <input type="checkbox"/> long               | <input type="checkbox"/> 218                   |
| <input checked="" type="checkbox"/> zipper  | <input type="checkbox"/> shipping              |
| <input type="checkbox"/> brand              | <input type="checkbox"/> dress                 |
| <input type="checkbox"/> sz                 | <input type="checkbox"/> l                     |
| <input type="checkbox"/> art                | <input type="checkbox"/> vintage               |
| <input checked="" type="checkbox"/> glasses | <input type="checkbox"/> unisex                |
- Button: "Refine Results"

**Results Section:**

- Text: "7208 items found for: (dolce and gabbana,d g,dolceandgabbana) Original query returned 6077 items"
- Button: "Next Page"
- Table of search results:
 









| Item Title  | Bids | Price*   |                     |
|---|------|----------|---------------------|
|  SALE Black Short Sleeve D. G Ladies Women Clothing New    | 0    | \$ 14.99 | Free 41s            |
|  NEW D G WHITE AVIATOR SUNGLASSES SHADES FREE SHIPPING!!   | 0    | \$ 12.99 | Free 44m 13s        |
|  NEW LADIES D G SUNGLASSES TORTOISE SHADES FREE SHIPPING   | 0    | \$ 9.99  | Free 2h 4m 18s      |
|  NEW 2009 LADIES D G SUNGLASSES SHADES FREE SHIPPING!!!   | 0    | \$ 9.99  | Free 2h 5m 36s      |
|  brand new D & G women high heel shoes size 39 EU        | 5    | \$ 3.79  | Free 3h 28m 36s     |
|  DOLCE GABBANA D&G DG 6010 BLACK 01/87 SUNGLASSES        | 13   | \$ 98.89 | Free 25d 7h 45m 29s |
|  NEW D&G BLACK MESH DRESS SKIRT 40 2 DOLCE & GABBANA     | 0    | \$ 195   | Free 3h 59m 1s      |
|  NEW D&G DOLCE & GABBANA SUNGLASSES MOD. D&G B024 501/87 | 0    | \$ 79.99 | Free 4h 1m 48s      |

Figure 27 - Show results and refine search

The screenshot above gives an idea of what the web page looks at this moment. ALF displays 50 results per page and allows the user to go back and forth through pages using the “Next Page” and “Previous Page” buttons. The text field on top of the entire result’s section displays some information about the number of results that were returned by the original query and the number of results that were returned by ALF’s expanded and refined query. All previous menus and options are still accessible through the “Edit Previous Search” button in the “Tools” section.

# 5. System details and Maintenance

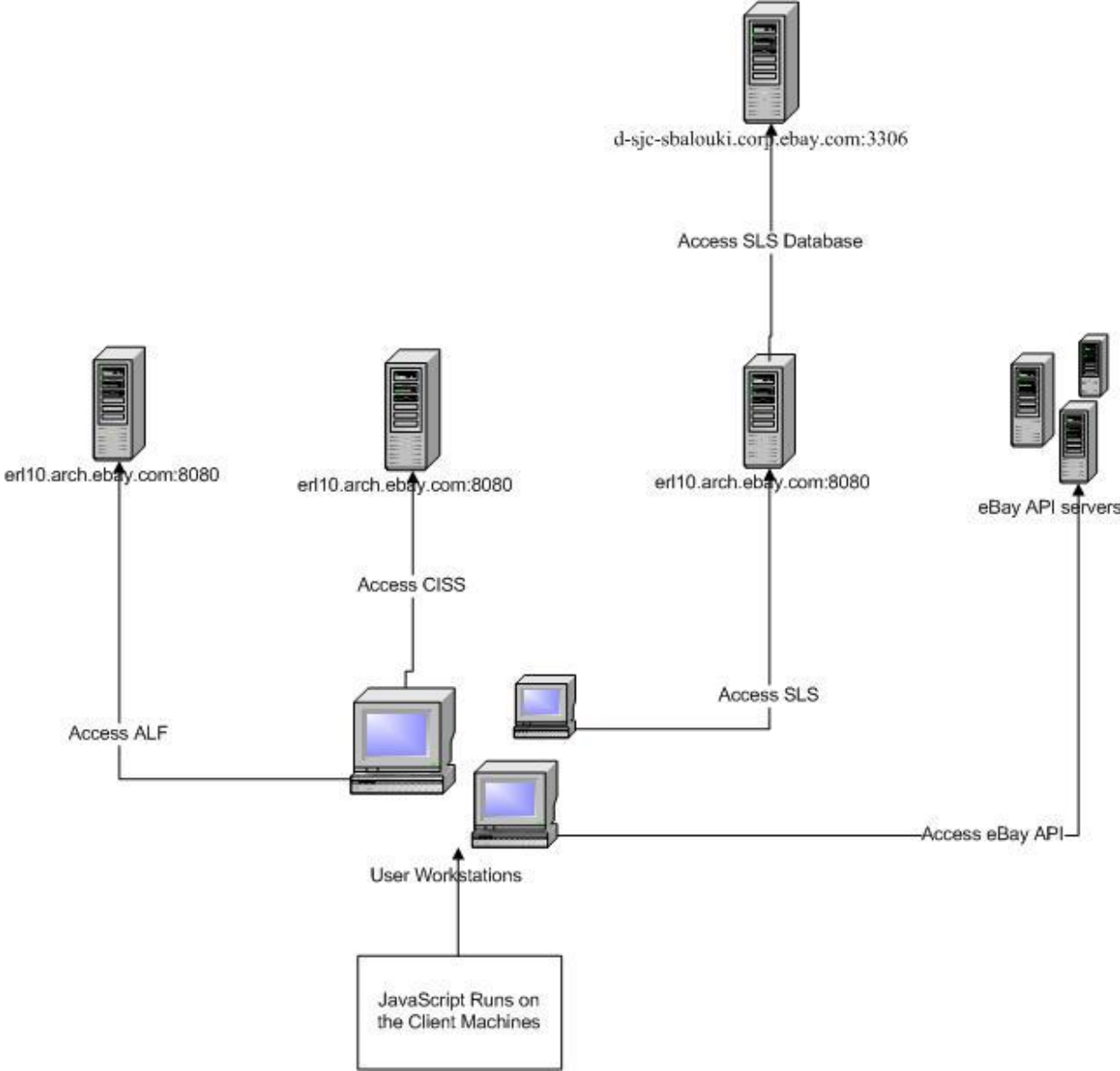


Figure 28 - Application Set Up

The final implementation of our application incorporated several services and tried to create an interactive and educating user experience. Figure 28 shows the specific services used and their physical locations. The application itself (ALF) is located on the erl10.arch.ebay.com server, and it's accessible via port 8080 (the Tomcat server) running on the machine. On the same server (erl10.arch.ebay.com) we also host our SLS and CISS, but they can be relocated on separate machines, if necessary. The SLS also uses

the “wsdb” database on the d-sjc-sbalouki.corp.ebay.com via port 3306 (MySQL server) to access the synonym data, stored there off the flat files. The last service our application uses is the eBay developer’s API, whose details and documentation can be found at <http://developer.ebay.com>.

ALF by itself is a JavaScript driven website, which uses AJAX to create a more pleasurable user experience. All the steps described in the ALF section are created as separate DOM elements and shown or hidden in their assigned <div> tags as necessary depending on the current step the user is on. There are a couple of main functions in the script which correspond to loading the page and the functionality of the three main buttons – find phrases, show results, and refine results. There are many helper show/hide functions, as well as information processing functions and global variables to keep track of details necessary to maintain the state of the site. All of these can be found properly documented in the script itself.

SLS is one of our two JavaServlet services, which allows us to look up synonyms. It follows the file structure shown in Figure 29.

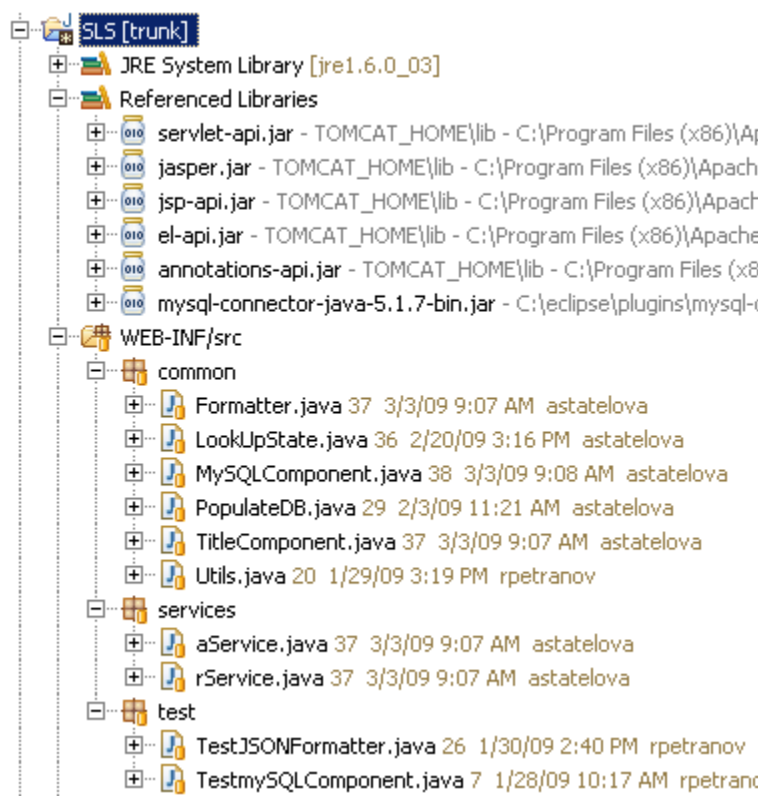


Figure 29 - SLS File Structure

The external library MySQL Connector/J is imported in the project and is necessary for the proper connectivity to the MySQL database. The common package contains all the functions necessary for the functioning of the services, which are located in the services package. The test package contains tests we wrote for some of our classes, which were not meant to be thorough but just test functionality without running the project on a server.

The services package contains the two services SLS offers – Radoslav’s (rservice.java) and Antoniya’s (aservice.java) algorithm implementations, both described in (Section X – will refer to the location in the final write up). The common package contains the classes which create the backbone of both services. The TitleComponent, MySQLComponent and Utils classes contain only static functions, which are grouped by their purpose. The TitleComponent class contains all the functionality necessary for splitting a query into phrases and getting the synonyms. The MySQLComponent class maintains the connection to the MySQL server and makes all the appropriate calls for populating and requesting information from the database. The Utils class contains helper functions for manipulating strings and arrays. The PopulatedDB class is a standalone class which uses MySQLComponent to allow population of the database. The LookUpState and Formatter classes are object-oriented classes, which are used in the TitleComponent on finding the synonym phrases and building the response of the service. The LookUpState class keeps track of the current data being considered as a phrase and details regarding it. The Formatter class keeps track of the found phrases and their synonym lists, which simplifies formatting the final result. Although the current formatting is JSON, adding a simple function to the Formatter class can easily create a different form of output from the service.

As mentioned, the MySQLComponent connects to a database called “wsdb” which is located on a different server - d-sjc-sbalouki.corp.ebay.com. The database contains a single table, which can be seen in Figure 30.

```
mysql> explain main;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| cat   | varchar(80) | NO   | PRI |          |       |
| word  | varchar(80) | NO   | PRI |          |       |
| syns  | varchar(400) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

Figure 30 - "main" table details

The category and word attributes are defined to be the primary key of the table, so it is required for the combination to be unique and neither of them can be null, but no additional restrictions exist. The list of synonyms is the third attribute in the table, which will be looked up based on its category/word primary key. This is the reason a word search has to contain a specific category related to it, otherwise multiple results can be found and since the service only returns the first result from the list, it can offer no guarantee whether the one returned will be correct.



## 6. Issues Encountered

---

### 6.1. MySQL Connector/J

---

MySQL Connector/J is the driver we used to make MySQL calls from SLS. The driver allows a connection to be established with the MySQL server, specific parameters to be set for this connection (full list can be seen at <http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-configuration-properties.html>), and then query and get results from the database specified to be in use. Since the connection takes time to be established, and it is used by SLS as a read only connection, we decided to create a static connection at the start-up of the service and keep it in use for all requests made to the service.

This decision worked properly, until the connection became idle for a long time. Inactivity over the connection would make it drop and not be restored until the whole service was reset (since that is when it would be initialized). Since this behavior was not feasible for our service, we had to come up with a way to keep the connection from dropping.

A complex and obvious way to correct this issue was to keep the connection active artificially by pinging it every time we fear the connection will timeout from inactivity. This potential solution consists in creating a separate thread to run alongside the main service and have the job of pinging through the connection every 7.5 hours (since the connection times out after 8 hours of inactivity). However, thread management and synchronization is not the first choice for solving the problem, since it's more likely to overcomplicated things and create more problems along the way, so we looked for a simpler solution to our problem.

Using the parameter possibilities that the JDBC (Java Database Connectivity) allowed and an additional validation query to the database, we came up with an alternative solution. Setting the parameter "autoReconnect" to true, forces the connection to be reestablished, if it had been dropped due to inactivity. However, if a call is made to a connection which has been disconnected due to inactivity, as a result, the call reactivates the connection, instead of actually getting processed itself. So the "Select 1;" validation query is sent before every actual query to make sure the connection is still active, or if it's not – that it gets activated for the actual query call. This solution also allows the connection to be inactive whenever the service is being idle, so maintenance can be done on the database itself, if necessary.

## 6.2. XMLHttpRequest API in JavaScript

---

The XMLHttpRequest API in JavaScript allows http calls to be made through DOM to a server without having to reload the current page or navigate the user away from it to get new data. The XMLHttpRequest is used as an AJAX technique which allows the JavaScript to send a request to a different URL than the one the user is currently browsing, get the results back, and incorporate them in the current webpage immediately in any way necessary. For example, we used this technique for getting results from SLS in ALF after the user had entered a search query. The XMLHttpRequest allowed us to send a request to the server hosting SLS immediately once the user pressed the search button, the service then returned the response, which in its turn was displayed on the website via JavaScript DOM.

The XMLHttpRequest object however has two modes – synchronous and asynchronous. By definition, synchronous means that the HTTP request and received response have to be completed before the script goes on with executing. Asynchronous would then be expected to consist of the request and received response happening in parallel with the execution of the rest of the script. However, the asynchronous mode allows for the script to keep running, until the response from the HTTP call is received and interrupts it to load the data received. But we want to use the data received as a response from the HTTP call immediately after the call itself, and in the case of asynchronous mode, this response doesn't get processed quickly enough. Although asynchronous was a recommended mode of execution for the request, we decided against it because of these timing issues, which didn't allow us to get our response data in time. On the other hand, if the SLS becomes too slow and takes time to receive and process data, the page might end up freezing due to this synchronous request we're making.

Another issue which we encountered when working with XMLHttpRequest was a security issue it has on requesting websites from another server. If a request is sent to a different machine, no matter what the network specifics on it are, as long as it's not the same server the JavaScript was downloaded from, an error occurs because of a security breach. The workaround for that was a local JavaServer Page, which the XMLHttpRequest sends a request to with the URL it wants the source of, and on its behalf the JSP fetches the page from the specified URL and hands it back to the JavaScript.

# References

---

1. Vechtomova, Olga and Wang, Ying. "A study of the effect of term proximity on query expansion", *Journal of Information Science*, 2006, p. 11.
2. Cui, Hang; Wen, Ji-Rong; Nie, Jian-Yun; Ma, Wei-Ying. "Query Expansion by Mining User Logs", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, July-August 2003, Pages 829-839
3. Efthimiadis, Efthimis. "Query Expansion", *Annual Review of Information Systems and Technology (ARIST)*, 1996, Pages 121-187
4. Internal Communications (DL-eBay-IC-iWeb-News@ebay.com). "eBay At-A-Glance". eBay Internal Web. Jan.31, 2008. <<http://iweb3.corp.ebay.com/Company/Pages/eBayAt-A-Glance.aspx>>
5. eBay Inc. (2007). "Research Focus Areas". Retrieved on March 3, 2008 from <http://www.ebayresearchlabs.com/erlresearchfocus.html>
6. Sun Microsystems. "Learn about Java Technologies". Retrieved March 3, 2008 from <http://java.com/en/about/>
7. Python Software Foundation. "About Python". Retrieved March 3, 2008 from <http://www.python.org/about/>
8. Stewart, Bruce. "An Interview with the Creator of Ruby". 29 Nov. 2001, retrieved March 3, 2009 from <http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html>
9. The Perl Foundation (2002-2009). "The Perl Directory: About Perl". Retrieved March 3, 2009 from <http://www.perl.org/about.html>
10. The PHP Group (2001-2009). "What is PHP?". Retrieved March 3, 2009 from <http://www.php.net/>