IMPLEMENTATION OF FULL TEXT SEARCH FOR OPENGEOCODING.ORG

Shailesh Shrestha [a *] and Franz-Josef Behr [b]

[a,b] Department of Geomatics, Computer Science and Mathematics, University of Applied Sciences Stuttgart
Schellingstraße 24, D-70174 Stuttgart (Germany),
[a] shreshai@yahoo.com, [b] franz-josef.behr@hft-stuttgart.de

KEY WORDS: Full Text Search, phonetics matching, intelligent data retrieval, PostgreSQL

ABSTRACT:
This paper describes the stepwise procedure for integration of full text search functionality in Opengeocoding.org which is a free and participatory community oriented project for assembling geocoded address. The full text search functionality provides an intelligent retrieval and suggestion of the addresses already stored in the database as a user types in a field to find a specific address. For the development of such tool, PostgreSQL has been used as Database Management System (DBMS), PHP has been used for server side scripting, and JavaScript and jQuery libraries have been used for client side scripting.

1 Introduction

1.1 Background

The addition of a spatial component to other types of information exponentially increases ability of decision makers in diverse fields to take well informed decisions. Therefore, people always have been interested to know 'where is something' and wanted to have an ability to visualize it in an interactive map. This has been greatly simplified with the emergence of freely available web mapping applications like Google Maps, Yahoo Maps or Virtual Earth. However these commercial online mapping tools have great detail coverage of street data and address data in most of the developed countries but address information is limited in 'information poor' developing countries which cover most of the part of southern part of the Earth. Therefore to build the gap between 'information rich' North and 'information poor' South, a free and participatory community oriented geocoding service project named "Opengeocoding.org" was initiated in 2007. The project aims to collect geocoded address data, e. g. postal addresses and coordinates, in order to make them available and usable by web based services on a worldwide level with a focus on developing countries (Behr, Rimayanti and Li, 2008; Behr, 2010). The project was founded with the belief that a person living in a particular area best knows local surrounding and many more locations around the vicinity. The idea was to attract people to voluntarily contribute and assemble geographic data and store the information in a database which could be later provided as a service to geocode a place in different format such as XML, KML or JSON. Thus the project aims to be one of the major hubs of freely available data collected based on principle of Volunteered Geographic Information (VGI).

Since its inception, only about 700 users have come together to collect geographic data from different parts of the globe. The database has been continuously growing but not in a pace that was expected before. Therefore, the database as well as the web interfaces of the project is in the process of major overhaul to attract many more users providing with many additional features which enhance the functionality aspect as well as performance. The database has been moved from MySQL to the more robust open source database management system (DBMS) PostgreSQL.

This paper describes the integration of PostgreSQL's full text search (FTS) functionality to provide an intelligent retrieval and suggestion of the addresses already stored in the database as a user types in a field to find a specific address.

* Corresponding author

2 Overview of FTS in Opengeocoding.org

The database of opengeocoding.org contains vast source of address information, either collected from its users or acquired from publicly available datasets on the internet. Therefore, it is felt that there is an immediate necessity to develop a tool by which a user can filter and find desired address information from a gigantic source of information pool. The tool should execute the search in a quick time and should provide an intelligent mechanism that automatically searches for matching entries and displays a list of results to choose from. The suggested results should be narrowed to find better matches as user types more characters of an address he/she is interested in. The tool should also accommodate room for mistyped words to provide more flexibility to the users.

For the development of such a tool, PostgreSQL was used as Database Management System (DBMS), PHP for server side scripting, and JavaScript and jQuery libraries were used for client side scripting. jQuery is a fast and concise JavaScript library that simplifies many task on client-side, i. e. document traversing, event handling, animating, and AJAX interactions for rapid web development (The jQuery Project, 2010). In addition, the autocomplete plugin found in the official website of jQuery is used.

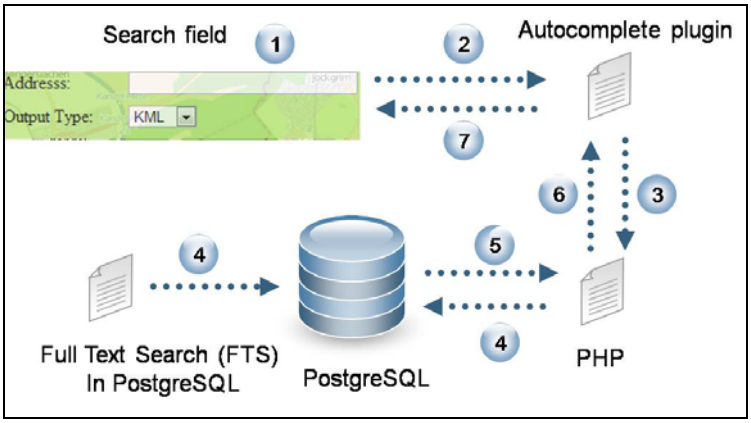

Figure 1: Executing steps in implementation of FTS with PostgreSQL backend

2.1 Autocomplete plugin jQuery

The autocomplete plugin is one of the most popular plugins in jQuery. When linked with an input field, it enables users to quickly find and select values from the matching list as they type, leveraging searching and filtering. Autocomplete can be customized to work with various data sources such as array with local data, a string specifying a URL or a Callback (The jQuery Project, 2010). For database driven application, a callback is most appropriate where a PHP file can be attached to select data from a database. Integration of autocomplete in a web page can be achieved in a very simplistic and convenient way without much hassle and time requirement. To get started, one needs a copy of jQuery as well autocomplete plugin which can be easily downloaded from the official jQuery website. Else, one can also link to the resources where files are stored saving bandwidth, if bandwidth is an issue, preventing the multiple loading of the files if a user's website cache has already stored ht jQuery library. Once the required jQuery files are properly linked, attaching autocomplete feature to any input field is a piece of cake as shown in Table 1.

```
$("#input_field").autocomplete("XYZ.php",{options})
```

Table 1: Using an autocomplete feature with a particular div in a html

Once a user starts typing in a test input field identified as ‘input_field’, typed words are sent to a ‘XYZ.php’ as a HTTP GET request with a parameter ‘q’. Now, ‘XYZ.php’ communicates with the database and uses the FTS feature of PostgreSQL to retrieve the most appropriate results and to send them back to the client. The data fed into the autocomplete feature can be customized for display as required. In our case, the address data extracted from the database using FTS also have ‘row id’ and ‘tablename’ which are necessary for displaying the extracted data on the map are hidden when displaying results by the autocomplete. Also, many other customizations were done such as to control numbers of character needed to type before the autocomplete request is to invoke and the time frame between each request

2.2 Full Text Search in PostgreSQL

The full text search (FTS) feature in PostgreSQL was developed by Oleg Bartunov and Teodor Siegev (2007) from University of Moscow who are actively engaged in the development of PostgreSQL. According to the developers, FTS is a search for documents which satisfy a query string 'query' and returns documents in some order according to their 'similarity' to the 'query'. In earlier versions of PostgreSQL, FTS was available as a contribution module, first as 'tsearch' and then as 'tsearch2' which has to be downloaded and installed afterwards but since PostgreSQL 8.3 version, it is installed by default.

FTS mainly involves three major steps (Bartunov and Sigaev, 2007).

- Parsing documents to lexemes: The initial step is to parse the documents into so called 'lexems' or 'tokens'. There are various classes of lexemes in PostgreSQL for example E-mail addresses, numbers, words and many more.
- The next step is to normalize the 'lexems' into so called *stem* or *infinite* from using a linguistic rule. This means all the similar words such as giving, given, gave or gives are reduced to the common stem word give. PostgreSQL provides linguistic support for all the major languages such as English, French, German, Spanish etc. Altogether 15 major linguistic configurations are available in PostgreSQL.
- The last step is to store the normalized stem words in an optimized way to allow faster searching operation. Therefore, it is always necessary to perform indexing in a stem words. PostgreSQL provides four different types of indexing namely B-tree, Hash, GIST and GIN indexing (The PostgreSQL Global Development Group, 2010).

2.3 Functional aspects of opengeocoding.org

Before describing the implementation of the FTS for the opengeocoding.org, it is worthwhile to have a short description about the functional aspects of the site. As mentioned previously, the site is voluntary driven by many users who have a common motto to assemble geographic data throughout the world with high emphasis on developing countries. The site provides a geocoding service for postal address on that data free of charge through a REST based web service (Behr, Rimayanti and Li, 2008). The application does not rely only on its internal database filled by the community but also on comprehensive lists of POI (Point of Interest) from other publicly available data services which are integrated in its database. If no relevant results are found from its database, the request will be forwarded to Google's geocoding service (http://code.google.com/apis/maps/documentation/-geocoding/), forwarding the results to the client. The address database is stored across two database table, one for the address data submitted by the community and one for the POI data.

The structure of the table storing address data collected by various users is shown below. It mainly has columns storing address information and columns storing its geographical coordinates. 'the_geom' storer the geometry information populated using 'Longitude' and 'Latitude' fields required for a Web Map Service (WMS) service using Geoserver.

Column name	Data type
country	character varying(60)
province	character varying(30)
city	character varying(60)
name	character varying(160)
district	character varying(60)
street	character varying(60)
housenumber	character varying(60)
postcode	character varying(60)
latitude	numeric
longitude	numeric

username	character varying(60)
source	character varying(20)
addressaccuracy	integer
geometricalaccuracy	integer
version	integer
modtime	timestamp
comment	text
the_geom	geometry

Table 2: Data structure of a table storing address information collected from users

3 Integration of FTS

The main purpose of integrating FTS the provision of intelligent suggestions for the user to find a closely matched address as he/she types in a field to find a specific address. Providing such a mechanism has the advantage that the user does not have to type whole address information which sometimes is pretty long. Consider an address *'United States California San Diego Meadowrun Wy 9157'* which is a 53 character long address. A user will obviously not want to type such long address every time. In addition a user can search the above address in number of different ways such as *'Meadowrun Wy 9157'*, *'9157 Meadowrun Wy 9157 San Diego'*, '*United States Meadowrun Wy 9157*'or '*Meadowrun Wy 9157 United States*'. The conventional pattern matching operator 'LIKE' which is very commonly used in both MySQL and PostgreSQL will fail miserably in above mentioned circumstance because 'LIKE' operator is highly dependent on positioning of string to be searched. The FTS eliminates such shortcomings and provides an efficient way for a search where a user can look for an address in different word form and in different word order.

3.1 Parsing address to lexemes

The address information in the database is structured as shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**. The individual columns have to be concatenated to form a complete address in the sense of a 'full text' address. In many instances certain columns are either empty or NULL. Therefore use of string concatenation operator ('||') is not suitable in this case as it would yield NULL if any of the column used in concatenation operation is empty. Therefore, string concatenation is done in conjunction with PostgreSQL's *coalesce* function so that if any of the columns is NULL, then it will not be used in concatenation procedure. In addition, POSIX regular expression which provides more powerful means for pattern matching was used to remove space among the concatenated string.

Depending upon the dictionary used, the result of parsing a string yields different results (cf. **Fehler! Verweisquelle konnte nicht gefunden werden.**). When parsing, each dictionary excludes certain sets of common words (such as a, the, and) known as STOP words in PostgreSQL. It is up to the specific dictionary how it treats stop words.

SELECT to_tsvector('english', 'Flughafen stuttgart of germany'); >>>'germani':4 'flughafen':1 'stuttgart':2	SELECT to_tsvector('german', 'Flughafen stuttgart of germany'); >>>'of':3 'flughaf':1 'germany':4 'stuttgart':2

Table 3: Influence of language configuration in *ts_vector*

Since, the address information stored in the database is collected from all over the world, it is apparent that addresses stored have many language specific words which would incorrectly parsed if an incorrect language dictionary is used. For example, in the above table, string *'Flughafen'* was parsed in stem word *'flughaf'* correctly when used *'German'* dictionary but *'English'* dictionary could not recognize the word and hence returned whole word 'flughafen'. Therefore, it is necessary to use correct dictionary when parsing an address.

But then again, it is very difficult to predict in which language a user is entering an address to search. Therefore, it was decided to use a 'simple' dictionary configuration which is a language independent dictionary which just converts the input string into lower case and returns it as stem words.

3.2 Storing of stem vectors as separate columns

As shown in Fehler! Verweisquelle konnte nicht gefunden werden., the addresses are converted to stems by using *to_tsvector* function in the PostgreSQL. There are two possible ways in which *to_tsvector* function can be used. In the first method, the results of the *to_tsvector* are stored in an another column and stem matching is done on that new column. This has the advantage that a person can see an address string and its stem variation side by side which is very helpful to understand and debug FTS. Alternatively, *to_tsvector* can be directly used in the query without storing the stem words. This would save some space in the database but needs more computation time afterwards. For our case, we have chosen the first method of storing stem vector in a separate column in a table, as suggested by Scherbaum (2009). Besides stem words can be seen and also, it is faster compare to latter method because once the stem vectors are stored, only matching has to be done where as in second method parsing of address in lexeme stems occurs in every query (see **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Fehler! Verweisquelle konnte nicht gefunden werden. also shows the difference between the ways *to_tsvector* can be used. In query 1, *address_vector* is a column populated with stem vectors resulted from *to_tsvector* and *search_string* is the string to be matched. For matching stem vectors, PostgreSQL provides the function *to_tsquery* and *plainto_tsquery* for converting a query *to_tsvector* type. In our case, *plainto_tsquery* is more appropriated as it transforms search string into normalized stems and inserts "&" operator within the resulted string.

Select * from TABLE where address_vector @@ plainto_tsquery('simple', 'search_string') >> Query 1	Select *from TABLE where to_tsvector('simple', 'address') @@ plainto_tsquery('simple', 'search_string'); >>Query 2

Table 4: Storing *ts_vector* vs. without storing *ts_vector*

3.3 Creating indexes for faster search

Once the column address_vector is populated with stem vectors resulted from to_tsvector using 'simple' language configuration, it is necessary to use indexing in that column which would greatly enhance speed of execution of a query. PostgreSQL provides two types of indexes which can be used for FTS: Generalize Inverted Index (GIN) and Generalized Search Tree Index (GiST)[1]. In The PostgreSQL Global Development Group (2010) as well as in a Watermann (2010), it is stated that following performance differences exist between GiST and GIN.

- GIN index lookups are about three times faster than GiST
- GIN indexes take about three times longer to build than GiST
- GIN indexes are about ten times slower to update than GiST
- GIN indexes are two-to-three times larger than GiST

Since speed of query execution is of utmost importance for implementing FTS with autocomplete and longer build time for the GIN index is not a constrain for our application, GIN index appear to be ideal for us. The updating of indexes will be performed every day or week during night time with low user interaction. Nevertheless, time required to build both GIN and GiST indexes were compared. For a table in a database with about 600 thousands records, time required for building GIN and GiST was ~36 s and ~72 s respectively whereas for another table populated with data from Geonames.org with much higher numbers of records (~ 7 Millions), time required for building GIN and GiST was ~13 min and ~30 min. In both of the cases, the time required for building the GIN index was twice as much as for the GiST index. However, owing its superior execution speed, GIN index was selected as most appropriate and hence implemented for the FTS.

[1] For detailed description, please refer The PostgreSQL Global Development Group (2010)

3.4 Ranking search results

With above described step wise procedure, the FTS is a fully functional search feature providing near instant results as a user looks for a particular address. However, it is not yet fully optimal because it still lacks an ability of measuring relevancy of found results with a particular query when many matches are found. For example, if a user looks for *'Dee'*, the FTS implemented in the database would find '*United States California San Diego Deer Lake'*, *'Red Deer'* or *'Deer Park, Maryland'* out of many others. Since, the results of a query could be many, a common practice is to limit the results that are displayed. In such case, query results are displayed in the order it is found in the database, hence in such case only 'United States California San Diego Deer Lake' might be presented to users in spite of the fact that other results are more relevant to the query when we consider the length of the results. Such limitation can be alleviated in the FTS when some ranking for search results are integrated so that the results are displayed according to relevancy which takes into account proximity information or how often the query terms appear in the results. PostgreSQL provides functions *ts_rank* and *ts_rank_cd* for ranking search results. The latter function is used because it is considered more superior than the previous function as it computes cover density ranking for given vector and query utilizing positional information in input vectors (Bartunov and Sigaev, 2007). When ranking search results, size of the results is also taken into account as a two words result with one instance is probably more relevant than a six words result.

```
Select REQcolumns,ts_rank_cd(index, query, 32 ) AS rank from TABLE, plainto_tsquery ('simple', 'search_string') query where query @@ index
```

Table 5: Query to account for ranking search results

3.5 Incorporating provision for alternatives names

It is obvious that a same place can have different names within the same language as well as based on different language (aliases and vernacular names, see Behr 2010). For example, 'Germany' has other names such as Deutschland (common, German), Bundesrepublik Deutschland (official,German), Duitsland (common,Dutch) or Alemania (common, Spanish) among many others. It is important that FTS implementation should also incorporate provision for alternative names so that when a user residing in Spain or Portugal searches "Almania" using native regional language Spanish, the search string should be considered as " Germany ". For the same purpose, PostgreSQL provides provision of extending FTS configuration with user defined synonym dictionary and thesaurus dictionary configuration. Synonym dictionary is used to replace a word with a synonym whereas thesaurus dictionary is used to replace a collection of words with other words. These dictionaries are list of *'FROM '* to *'TO'* words which have to be created in a Notepad or similar text applications for each language configuration that will be used in FTS and placed in a certain PostgreSQL installation. In addition, FTS configuration has to be altered before these dictionaries can be used[2]. Since *simple* configuration for FTS was used, which is a language independent configuration, synonyms and thesaurus words in different languages can easily be placed in a single text file. **Fehler! Verweisquelle konnte nicht gefunden werden.** show the structure of the thesaurus dictionary and the output when the dictionary is incorporated in FTS. For example, if a user queries 'UAE' which is a short form of the country ' United Arab Emirates ', the FTS with user defined synonym and thesaurus configurations results 'arab':2'united':1 'emirates':3. The numbers in the results indicates position of strings in the parent string 'United Arab Emirates'.

File	Query	Result

University of Applied Sciences : HFT
 	
UAE : United Arab Emirates
	to_tsvector('simple','UAE');
	
	to_tsvector('simple','University of Applied Sciences');
		'arab':2'united':1 'emirates':3
		'hft':1

Table 6: Using custom thesaurus dictionary and its results

The speed of execution of a query with FTS configuration with integrated custom made dictionary configuration seems to be relatively fast up to a certain numbers of lines or records. For thesaurus configuration with about 60000 records, execution speed was in the order of less than a second. Since, the thesaurus list, that we have,

[2] For more detailed explanation about the how configuration of the FTS can be changed, please refer to The PostgreSQL Global Development Group (2010).

contains lines in the orders of millions (alternatives names collected mainly form geonames.org), the speed of query execution was considerably slow (~40 to 50 s).This was not acceptable for the FTS with autocomplete where results should be ideally displayed with less than a second. Hence, the idea of using custom dictionary configuration was totally abandoned but an alternative way was implemented. In this alternative approach, all the alternative names were also stored in the table, which was later concatenated with the main name and *tsvector* of the new column. But one difficulty was that the alternative names had many duplicate values which have to be removed before it is converted into *tsvector* otherwise, it would negatively affect the ranking of search results. Since, PostgreSQL does not have a built in function to remove duplicate values from a list, a custom functions was implemented (Fehler! Verweisquelle konnte nicht gefunden werden.**)**. For example if string 'name a, name b, name c, name a' is provided to that function, it would yield 'name a, name b, name c'. We are using PostgreSQL 8.3 which does not have *unnest* function which is available in higher versions, *unnest* function is added as custom function as well. After removing duplicates words from concatenated strings, the *tsvector* is stored and FTS is implemented as described in Section **Fehler! Verweisquelle konnte nicht gefunden werden.** to Section.**Fehler! Verweisquelle konnte nicht gefunden werden.**.

```
CREATE OR REPLACE FUNCTION uniq_words(text)
RETURNS text AS $$
SELECT array_to_string(ARRAY(SELECT DISTINCT trim(x) FROM
unnest(string_to_array($1,',')) x),', ')
$$ LANGUAGE sql;
```

Table 7 : Custom function written for replacing duplicate words from a list of strings

3.6 Incorporating provision of mistyped words

When a user types an address, there might be some spelling mistakes, most probably there would be only few letters difference in what he/she intended to type and what he/she has typed. FTS is pretty unforgiving in the case of misspelled words which means if an user types *'Uneted States'* instead of *'United States'*, the FTS would not yield any results. This is less than ideal because it is a natural tendency that most of the users tend to type fast, and spelling mistakes are likely to occur when typing in a fast speed. Therefore, to cater room for spelling mistakes, PostgreSQL's *fuzzystringstrmatch* module which provides several functions to determine similarities and distances between strings is examined for its suitability. Basically there are two main categories of matching of string similarity: the first class comprises equivalence methods which return true or false and the second similarity ranking methods which returns some sort of similarity measures (Behr, 2010). The fuzzystringstrmatch module provides functions such as *Soundex*, *Metaphone* and *Double Metaphone* which are based on methods of matching similar sounding names and falls in the second category mentioned before. The PostgreSQL Global Development Group (2010) states that these functions are not very useful for non-English names. But our database contains addresses coming from different languages with different phonology and etymology. This already makes *Soundex*, *Metaphone* and *Double Metaphone* functions less ideal. In addition, these functions are greatly dependent on ordering of words within the string. Since a user can type a particular address in a numbers of different ways, these functions are not very appropriate. For example the difference function which is based on the Soundex system gives difference of 4 (0 means different and 4 means similar) when comparing *'stutgart'* with *'stuttgart Germany'* but results 0 when comparing *'stuttgart'* with *'Germany stuttgart'*. A user could type address in the order of house number, street, city, province and country, or in the order of country, city, house number and street. Hence, these functions do not fulfill the requirements. The *fuzzystringstrmatch* module also provides another function named as *Levenshtein distance* which works differently than the phonetics matching functions like *Soundex* and *Metaphone*. *Levenshtein distance* compares strings in terms of alphabetical positions in source and target string. But it also encompasses same drawback associated with *Soundex* and *Metaphone*, i.e. it is also affected by the ordering of words within a string.

Hence, it turns out that *fuzzystringstrmatch* module functions contain drawbacks and therefore could not be used for incorporating provision for mistyped words. PostgreSQL provides another module named *pg_trgm*, which provides functions and operators for determining similarity of string based on trigram matching. Trigram matching is a technique where trigrams (groups of three consecutive characters taken from a string) from source and target are matched against each other and similarity is measured by counting the number of trigrams source and target share. Afterwards the similarity measure is calculated which is a ratio of intersection size of trigrams divided by union size of trigrams. When comparing, trigram first converts strings to lower case. It also supports indexes such as GiST and GIN for very fast similarity searches. Therefore, trigram matching function *similarity* was used to make room for mistyped words because of the advantage that it is not affected by word ordering within the string as shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**.

select similarity ('Germany Munich', 'Munich')	0.466667
select similarity ('munich germany ', 'Munich')	0.466667

Table 8 : Similarity score does not affected by ordering of words in a string

However, it was discovered that it has a slight disadvantage that it cannot identify non-English characters while forming trigram and instead considers them as a space. This has been illustrated in **Fehler! Verweisquelle konnte nicht gefunden werden.**. Notice that when forming trigram of a string 'Germany Münich' with non-English letter 'ü', the letter is replaced as space and hence is present in the trigram. This drawback does however not affect the similarity score if the non-English letter is present in source *and* target string because it is considered as space. But it affects the similarity score for place names like Münich (in English it is known as Munich). If a source string is stored as *'Germany Münich'* and target string is *'Germany Munich'* then the similarity score is quite different than what it would have been if the source string was stored as *'Germany Munich'*(c.f. **Fehler! Verweisquelle konnte nicht gefunden werden.**). This is not an ideal solution for the database with many non-English characters; nevertheless there is no other alternative available in PostgreSQL than to accept trigram with its shortcoming and to use it.

select show_trgm('Germany Munich')	{" g"," m"," ge"," mu",any,"ch ",erm,ger,ich,man,mun,nic,"ny ",rma,uni}
select show_trgm('Germany Münich')	{" g"," m"," n"," ge"," m "," ni",any,"ch ",erm,ger,ich,man,nic,"ny ",rma}
select similarity (lower('Germany Münich'), 'münich')	0.466667
select similarity (lower('Germany Münich'), 'munich')	0.466667

Table 9 : Similarity function takes non-English characters as space

4 Conclusions

The paper describes a stepwise procedure to implement full text search (FTS) using the open-source database management system PostgreSQL and the jQuery autocomplete plugin which is a great way to provide an intelligent suggestion of the address already stored in the database as a user type in a field to find a specific address. With this features, address retrieval from the database is performed in a more intuitive manner providing intelligent suggestion for users as they type, in nearly real time. Alternative names were also incorporated in FTS when such information is available for places. But the drawback of FTS is that it is pretty unforgiving in the case of mistyped words. If there is even a single mistyped letter in a word, FTS would not result in any suggestion list. It was attempted to provide room of mistyped with *fuzzy string matching* and *trigram* module. It can be said that the function provided in *fuzzy string matching* was not appropriate for the purpose, but trigram module was found to be suitable to a certain extent. But due to its slow speed and the huge database, it could not be coupled with autocomplete feature of FTS.

References

Bartunov, O. and Sigaev, T., 2007. Full-Text Search in PostgreSQL: A Gentle Introduction, PGCon-PostgreSQL Conference for Users and Developers, Ottawa.

Behr, F. J., 2010. Geocoding: Fundamentals, Techniques, Commercial and Open Services. Proceedings Applied Geoinformatics for Society and Environment (AGSE). Publications of the Stuttgart University of Applied Sciences, Hochschule für Technik Stuttgart, Volume 109 .

Behr, F. J., Rimayanti, A. and Li, H., 2008. Opengeocoding.org – A Free, Participatory, Community Oriented Geocoding Service. ISPRS, Commission IV, WG IV/5, The International Archives of the International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. Vol. XXXVII. Part B4. Beijing, 2008

Scherbaum, A., 2009. PostgreSQL: Datenbankpraxis für Anwender, Administratoren und Entwickler. Open Source Press, München.

The jQuery Project, 2010. www.jquery.org, (last accessed: 27/06/2011)

The PostgreSQL Global Development Group, 2010.PostgreSQL 9.0.04 Documentation. The PostgreSQL Global Development Group.

Watermann, J. , 2010. PostgreSQL: Installation, Grundlagen, Praxis. Galileo Press, Bonn,