Generating Meaningful Location Descriptions

Jörg Roth
Faculty of Computer Science
Nuremberg Institute of Technology
Nuremberg, Germany
Joerg.Roth@th-nuernberg.de

***Abstract*—The user's location is an important information to describe the current situation or context. In some scenarios, we have to rely on a purely textual description when a digital map is not available. Our approach efficiently generates a meaningful text that describes the current location. As an appropriate text is highly application-dependent, a formalism supports applications to configure the desired description. The underlying geo data may be stored with different structures and techniques, thus we incorporated the idea of *Engines* into the approach that form an interface to the underlying query mechanisms. To efficiently perform the required spatial computations even on small devices, we introduced a novel spatial index, the *Near Index*.**

Keywords—Location, Context, Reverse Geocoding, Spatial Data

I. INTRODUCTION

The user's current location is an important information for several applications and services. A location may not only be processed by applications (e.g. for route planning), but also presented to the user in a meaningful way, typically with the help of a digital map. However, sometimes we have scenarios, where a map is either not available or a purely textual presentation of the current location is more suitable:

- The location may be transferred by a message service that only supports texts, e.g. SMS.
- The location could be presented on small devices, e.g. on smart watches that are not able to display a map.
- We could use text-to-speech output. E.g. an artificial voice could read a location description for blind people.
- Metadata of videos and images can usually only store a line of text to specify the location in addition to coordinates. E.g. a digital camera with built-in GPS may create such a description for every taken picture.
- People following the *Quantified Self* movement [4] try to log everything of their life, e.g. also their location. Besides the physical location, a meaningful textual description could be logged, e.g. by a fitness tracking device.

An appropriate service could be summarized as follows: given a physical position (i.e. latitude/longitude coordinate); map it to a human readable string that is a *meaningful* characterization of this position. Important: the string is optimized for people, not for further processing by software.

978-1-4673-7328-9/15/$31.00 ©2015 IEEE

As this mapping is surprisingly complex, our main approach is to shift this task to a general mapping framework. A first observation: the text does not necessarily have to precisely and uniquely identify the current position. This means, it is not required to derive the exact original coordinate from a certain description in order to be useful. In addition, a single description is not necessarily suitable for all applications. I.e. for one application '*Market place Nuremberg*' may be sufficient, whereas another application may require descriptions such as '*Market place Nuremberg, 10 meters west of the entrance Frauenkirche*'. As a consequence, a certain application should express the requirements to the description and pass it the mapping framework.

As a second observation: an appropriate mapping framework has to run in different scenarios, ranging from server environments with huge geo databases to small devices such as smart phones or activity tracker wristbands. They significantly differ how to deal with the underlying geo data. Crucial: even though small devices nowadays may have large flash memories, the runtime memory usually is very limited. On the other hand, servers are more efficient if they load as much as possible into runtime memory. These differences highly influence geo data access and *spatial indexing* – the mechanisms to quickly process geometric queries.

In this paper we present the *HomeRun Reverse Geocoding* framework:

- It efficiently generates a descriptive string from a position. Access to geo data is modeled by *Query Engines* that shield the access to the geo data repository (e.g. GIS, database, or file) and contain the spatial indexing mechanism. Engines can, e.g., be optimized for execution speed or memory consumption.
- Applications can configure the required descriptions and tailor it to their demands. For this, a script-like formal language is defined that is able to generate even complex structured text output. The formal language can express control structures and supports different national languages and measurement units (miles, km). Even though the formal language is very expressive, a typical script only has 10 to 20 lines of code.
- We identified two major descriptive properties of a position: to be *inside a location* (e.g. '*in the railway station*'), and to be *near an object* (e.g. '*in front of the University*'). For the first property, there exist several spatial

indexing mechanisms. For the second property, we introduced a new mechanism called the *Near Index*.

- If an application asks for a description, usually multiple underlying queries are required. An *Execution Planner* tries to identify the most efficient sequence of queries according to two concepts: *Lazy Evaluation* and *Earliest Failure First*.

Our approach is fully implemented and tested on server, PC and smart phone environments.

II. Related Work

Our desired task can be considered as a mapping between *physical* and *symbolic* (or *semantic*) locations. The classification of different location representations has a long tradition. Pradhan [10] distinguished three types of locations: *physical* locations such as GPS coordinates, *geographical* locations such as '*City of Nuremberg*' and *semantic* locations such as '*Central railway station Nuremberg*'. Meanwhile, geographic locations are usually considered as semantic. In short, physical locations can be expressed by numbers, semantic locations by texts [11].

Semantic locations form the basis to define *contexts* and *situations* as discussed in [4]. Physical coordinates, e.g. specified by latitude, longitude coordinates have no meaning to users without the knowledge of objects in the nearer area. E.g. the physical coordinate 49.453914/11.077314 is meaningless for most people, whereas 'Central Market Nuremberg' may be useful in a certain situation. Thus, a user's context has to take into account objects and areas at that location that may influence the user's situation. Such objects can be traffic objects, natural objects, buildings or even virtual objects such as borderlines. As different objects may have different meanings for users, a context may take into account the respective user profile. In [14] a platform only takes user-specific locations into account such as '*home*', '*work*' or '*gym*', that are different for different users. Even though such locations describe a user's situation more appropriate, the user first has to define such locations beforehand.

A basic mapping from physical to semantic locations is the so-called *reverse geocoding*. The usual notion of reverse geocoding is: get *all* geo objects at a certain position. As such, it is a basic function of GIS (geographic information system) [8], where a user points on a map to get the geo objects that are related to the respective GIS function (e.g. power lines, or water pipes). ArcGIS provides another definition: '*Reverse geocoding determines the address of a given point on a map.*' [2].

In the last decade, *location-based applications* are also made available in mobile environments. As a result, reverse geocoding services are incorporated in APIs of mobile platforms. E.g. Android offers the *Geocoder* API [1] that mainly provides a list of address fragments for the current location. It requires an online connection to a geocoder service. With this API, an app is able to identify the current country, city and in urban areas, the postal address.

A related term to reverse geocoding is *geotagging*: add location information to media such as videos or images. E.g. some digital cameras are equipped with a GPS receiver. When taking a picture, the physical location is stored in the image's meta data. Some new cameras (e.g. Lumix DMC-TZ31) in addition perform a reverse geocoding and store a text with country, state, city and a nearby touristic sight in the image.

Some platforms provide geo data search functions (e.g. [15][18]): a user enters some text and the system tries to find geo objects that best matches the text. Even though this function has some similarities to reverse geocoding, it can be viewed as an opposite function to reverse geocoding. Especially, the underlying mapping process is different: whereas this type of search is mainly based on text search mechanisms, reverse geocoding mainly uses geometric queries. As a consequence, the type of indexing is completely different.

Even though there exist a huge variety of services that perform a mapping of physical to symbolic locations, we can identify requirements for an ideal platform that usually are not addressed:

- Current platforms do not take into account the respective application. The application should specify its demands to the output in a fine granular manner.
- Current platforms only provide a list of objects. To generate a useful text, however, requires an additional step.
- Current platforms mainly search for geo objects that cover a position. We strongly believe that also objects in the area are important to specify a location.
- Current platforms are not flexible concerning the runtime environment. An ideal platform can run on a variety of execution environments ranging from servers to small offline devices.

Even though, user-defined locations may be useful for some applications, our approach only considers user-independent geo objects from large sources such as Open Street Map [3]. As a benefit: we can start our mapping instantly and do not have to learn user-specific locations.

III. The HomeRun Reverse Geocoding Framework

The result of our considerations is a novel reverse geocoding framework created as component of the *HomeRun* platform [13]. HomeRun provides a general basis for different tasks and services in the area of location-based services and geo data processing, especially of small applications outside the mass market. It covers the area *mastering and import of geo data*, *spatial services and algorithms*, *spatial indexing* and *support for mobile platforms*. The most important current components are the routing service *donavio* [17], the map rendering service *dorenda* and the spatio-symbolic search environment *HomeRun Explorer* [15].

Fig. 1 presents the architecture of the *Reverse Geocoding Framework*. Before an application can compute location descriptions, an application-dependent *Geo Data Repository* has to be generated. It covers both the actual geo object data and spatial indexing structures. The repository is created once from a geo data source (e.g. Open Street Map) with the help of *As-*

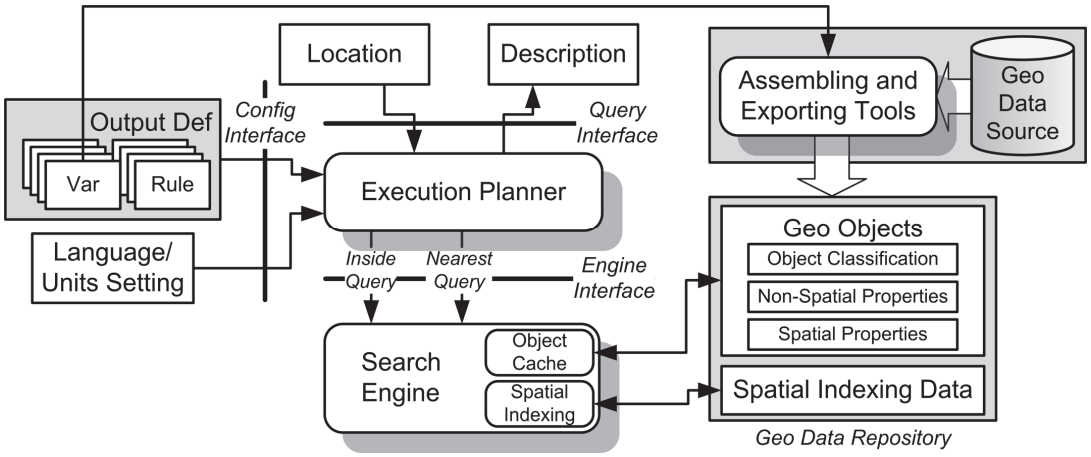

Fig. 1. Framework Architecture

sembling and Exporting Tools. The repository may be stored in e.g., a relational database or structured file.

A central component is the *Execution Planner* (see section III.B) that takes the current location and transmits queries to the underlying *Search Engine*. Search Engines process both types of spatial queries namely *Inside Queries* and *Nearest Queries* (see section III.C). The storage, access and spatial processing are hidden inside an Engine. The framework comes along with pre-defined engines that either expect the geo data to be stored in a relational database or in structured files. Engines based on files are optimized for small devices (e.g. activity trackers) with small runtime memories: all search operations are executed in random access file manner directly on the file system. Inside our pre-defined engines, the *Extended Split Index* [12] forms the basis for spatial indexing. This index heavily makes use of column indexes of relational databases, if available.

An application developer can use one of these engines, but is free to develop a new one according to the Engine Interface.

A. The Mapping Formalism

The last important component in fig. 1 is the *Output Definition*. We assume there are a plenty of information that describe the current location in a meaningful way, e.g.

- the postal address, country, state, city, quarter;
- significant objects such as touristic sights, buildings, shops, rivers, lakes, city walls, peaks, watersides;
- streets, corner of crossing streets;
- name of the wider area or region.

For all of the categories above we can state to be *inside* (e.g. to be *in* a city) or be *nearby*. If inside, we additionally may identify the location in relation to the surrounding area (e.g. *near the city center*, *near the northern border*). If nearby, we additionally may identify the distance and direction to the nearby object (e.g. *10 km south of*).

As already mentioned: different applications may consider different information as important. If, e.g. we take pictures as a tourist, the city and the nearby touristic sight are important. If we take pictures as a natural scientist, we may find the closest mountain, glacier or lake more useful. A fundamental facility for an application to configure the output is the *Configuration Interface*. A certain application may, e.g., define the following rules to generate a useful description:

- If you are inside a city, print its name. If a greater city is only nearby, generate a description such as '*2 km south of ...*'. If there is a city quarter, proceed similar.
- If there is a postal address very close to the location, print it. If there are multiple postal addresses nearby that only differ in house numbers, only print city, Zip code and street without house number.
- If there is a lake, mountain, volcano etc. in a wider range, incorporate it into the description.
- If there is a *Point of Interest* (*POI*) such as a shop, restaurant, fuel station, train station or touristic sight nearby, incorporate it into the description.
- If the location is inside a significant area (e.g. in a forest or on an island), mention it. If the location is identifiable inside that area (e.g. '*near the center of...*', '*at the northern border of*'), use it to refine the description.
- If the location is on a significant road (e.g. a motorway), mention it. If the location is close to an intersection of two roads, mention the corner if these roads.

If the collected information is not descriptive enough, relax some rules, e.g. also try to find POIs in a wider range, or also mention the direction to a city that is farer away.

This example shows that applications may have complex demands how to generate a meaningful description. We have to structure and formalize these demands:

- A set of *variables* identify notable properties that may be useful for a meaningful description. E.g. '*nearest Point of Interest of type shopping that is not farer than 500m form the current location*' or '*postal address at the current location*'.
- A set of *rules* contain constant texts, function calls and control structures. A simple rule may be `You are in-`

TABLE I. SAMPLE OUTPUT DEFINITIONS

Source Code	Meaning	Example Output

`VAR: $City1=InsideCity`
`VAR: $Addr=Addr(20m, REQ_STREET | REQ_HOUSENUMBER)`
`RULE: $City1 $if{Addr!=null,',`
`$Addr.street $Addr.housenumber'}`
	a nearby address point; if not available, only the current city	`Nuremberg, Hohfeder-straße 40`

`VAR: $Region=InsideRegion`
`VAR: $City2=NearestCity(50km)`
`RULE: $City2.distancestr`
`$if{City2.direction,' $City2.directionfromstr'}`
`of $City2$if{Region!=null,', $Region'}`
	a nearby city inside a larger region if available	`50km south of Nuremberg, Frankonia`

`VAR: $POI1=POI(300m, INDUSTRIAL | COMMERCIAL)` `RULE: near $POI1.objecttype '$POI1'`	an industrial or commercial object nearer than 300m	`near restaurant 'la Luna'`

`VAR: $POI2=POI(2km, WATER | GEOLOGY)`
`RULE: $if{POI2.distance<10, 'near', '$POI2.distancestr'}`
`$if{POI2.direction, ' $POI2.directionfromstr'}`
`of $POI2.objecttype '$POI2'`
	a water or geological object nearer than 1km with direction, if available	`1km northwest of lake 'Brombachsee'`

`VAR: $City3=NearestCity(5km)`
`VAR: $Road=NearestRoad(50m, HIGHWAY | COUNTRYROAD)`
`RULE: $Road.objecttype '$Road'`
`$if{City!=null, ', $City3.distancestr from $City3'}`
	a large road	`highway 'A3', 3km from Nuremberg`

`VAR: $Crossing=NearestCrossing(50m, URBANROAD)` `RULE: near '$Crossing.roada' corner '$Crossing.roadb'`	a nearby crossing	`near 'Hohfederstraße' corner 'Deichslerstraße'`

`VAR: $POI3=POI(1km, POI_TYPE_GEOLOGY)`
`RULE: $if{POI3.areasize>100000,`
`'near large geological site '$POI3.objecttype'',`
`$fail}`
	a geological object with a certain size	`near large geological site 'vulcano'`

`side $City`, where **`$City`** is a variable that contains the current city name.

- *Language/Unit Settings* support multi-lingual applications. Strings constants can be defined for different national languages and units (e.g. km or miles). In addition, control structures (e.g. **`if`**) can use the language or unit setting in conditions.

The actual output definition is specified as source code as presented in Table I. Each example shows a single rule. In reality, a specification consists of multiple rules, whereas the first rule that 'fires' generates the output.

Variable definitions include a variable type (e.g. POI, InsideCity) and further conditions. E.g. **`$POI3`** (Table I, last row) denotes the nearest Point of Interest of type `GEOLOGY` (i.e. mountains, volcanoes) that is closer than 1km.

Note that the distinction between different object types and classes (such as '*shop*', '*mountain*', '*city*') is not trivial. Typical geo data sources only provide a *weak classification* that allows a contributor to formulate any pairs of key value to define object classes and properties. In contrast, we need a *strong classification* with easy to distinguish object classes. HomeRun heavily relies on a strong classification concept [16], thus the variables can easily select all geo objects of, e.g., type **`WATER`**. On the other hand, the strong classification is required to print the object type as string. Expressions such as **`$POI3.objecttype`** are virtual impossible with the weak classification of e.g. Open Street Map.

The definition of rules is inspired by the **`printf`** format, known from the programming language C. The great benefit: a developer can mix constant texts and dynamic values in a single string. This makes it easy for an application developer to 'program' an output. The basic constructs of rules are (beside constant strings):

- **`$Var`** or **`$Var.query`**: The first returns the object name (e.g. city name) assigned to the variable. The second allows to query additional properties, e.g. the distance or direction to the current location.
- **`$if(`***cond, then, else*`)`**: Output can be conditionally. This allows a developer to switch texts dependent on conditions such as distance or whether a variable is assigned. As an example: if an object is very close, the distance is not printed in meters, instead '*near*...' is printed (Table I, 4th row).
- **`$fail`**: This forces an explicit failure of a rule, i.e. the next available rule will be checked. A **`$fail`** may only appear conditionally. E.g. in Table I (last row) the rule should not apply for small objects.

Language-dependent string constants (e.g. '*near*') can be stored in resource lists. The text generation process then automatically selects the appropriate national variant at runtime.

The separation of variables from rules is essential: it allows our platform to scan all dynamically generated objects beforehand. The set of variables is required both as input for the *Execution Planner* during a mapping operation and to create the *Geo Data Repository*. For the latter case, the set of variables configures, which geo objects actually have to be exported. Moreover they are required to generate spatial indexes (see section III.C).

B. The Resolution Mechanisms

The basic operation to generate an output is to *resolve* all involved variables of a rule. Resolve means: query the data repository and assign a value. A resolution can also fail. E.g. an **`InsideCity`** variable is not assigned, if the current location is outside of any city. To generate a description from a given set of variables and rules, the Execution Planner works as follows:

- Iterate through all available rules in the given order. The first rule that fires generates the output.
- A rule fires, if a) all top-level variables (i.e. not conditionally) are assigned; b) for every **`if`** all variables in the condition as well as the variables in the respective *then* or *else*-branch are assigned; c) no **`$fail`** is recognized in a conditional branch.

To resolve a variable is the most time-consuming task as it requires to query a large data repository for objects that are related to the current location. Even though we speed up a query with the help of spatial indexes, a query takes considerably long compared to other operations in the mapping process. Thus, we want to resolve as few variables as possible. Our approach is based on two mechanisms:

- *Lazy Evaluation*: a resolution is performed not before a variable is actually considered to be part of the output.
- *Earliest Failure First*: for a certain rule, resolve the variable with the highest probability *not* to get assigned first.

The benefit: if a rule fails, it is detected as early as possible. To control these mechanisms, we *estimate*, if a query will fail, before we actually execute it. Let:

$p_{fail}(\text{var}) \in [0.0 \ldots 1.0]$ be the expected ratio for this variable *not* to be assigned. This value takes into account the geometric density across the surface.

$p_{cond}(\text{cond}) \in [0.0 \ldots 1.0]$ be the expected ratio for this condition to be true. This value takes into account the nature of the condition (e.g. var.distance<…) and the density of involved geo objects.

We can recursively extend the notion of p_{fail} as follows:

$$p_{fail}(\$\text{fail}) = 1.0 \quad (1)$$

$$p_{fail}(\text{cond}) = \prod p_{fail}(\text{var}_i) \quad (2)$$
(for all variables var_i of the condition)

$$p_{fail}(\text{sub-rule}) = \prod p_{fail}(\text{part}_i) \quad (3)$$
(for all parts part_i of a sub-rule)

$$p_{fail}(\$\text{if}(\text{cond}, \text{then})) = p_{fail}(\text{cond}) \cdot p_{cond}(\text{cond}) \cdot p_{fail}(\text{then}) \quad (4)$$

$$p_{fail}(\$\text{if}(\text{cond}, \text{then}, \text{else})) = p_{fail}(\text{cond}) \cdot (p_{cond}(\text{cond}) \cdot p_{fail}(\text{then}) + (1 - p_{cond}(\text{cond})) \cdot p_{fail}(\text{else})) \quad (5)$$

If the Execution Planner plans the queries of a certain rule, it first computes the p_{fail} values for each part of the rule and tries to resolve them in descending order. If a part is an **`$if`**, it recursively proceeds with all parts of the statement: first all variables of the condition, then all parts of the *then* or *else* sub-rule, again in descending ordering by p_{fail}.

This approach considers variables that probably cannot be assigned first. As a result, if a rule fails, the failure is detected as early as possible, thus superfluous queries are avoided. The crucial point is the assumed ratio for variables and conditions. E.g. if city areas cover 10% of the covered surface, an estimation of p_{fail}(InsideCity-var)=0.9 is a reasonable assignment. More complex variables and conditions however may lead to very complex considerations and may be difficult to estimate. However, even for simple estimations the overall computation time significantly benefits. It is important to note that even a naïve approach only affects the execution time, not the correctness of the result.

Some comments about optimality. If a certain rule i fires (and thus all rules $1 \ldots i-1$ do not fire) the theoretical optimal set of variables would be the smallest set

- that cause the rules $1 \ldots i-1$ to fail and
- that cause the rule i to fire.

The problem: we are able to detect, if a certain variable causes a rule to fail not before we actually resolved it, thus this theoretical minimum cannot be effectively computed beforehand. However, it turned out that our approximation is very close to the theoretical minimum (see section IV).

C. Spatial Indexing

A fundamental difference of spatial queries compared to relational database queries is the type of indexing. Well-known indexing structures such as balanced binary trees are not suitable for spatial queries. Here, a search hit is not a result of equality of keys, but of geometric relations (e.g. inside a polygon, overlapping with a line string). The corresponding structures are called *spatial indexes*. Most of them approximate exact geometries by bounding boxed that are put into a tree structure [5][6] or mapped to a space-filling curve [12].

Spatial indexes can very quickly compute a set of candidates that *probably* have a certain relation to a given fixed geometry. Very common are *Inside Queries* such as *get all geo objects that are inside a given polygon*, or *get all geo objects that enclose a certain coordinate*. Surprisingly, existing spatial indexes have difficulties to directly process *Nearest Queries* such as *get the nearest object of type xy*. Unfortunately, this query is typical for many variables in our framework. This is why we introduced a new type of spatial index, the *Near Index*.

In principle, we could *simulate* Nearest Queries by spatial indexes that only support Inside Queries: we ask for all objects inside a circle with a maximum distance; then we detect the nearest objects by iterating though all candidates. As a drawback, this approach would require to load large sets of geo objects of which only one is the 'nearest'.

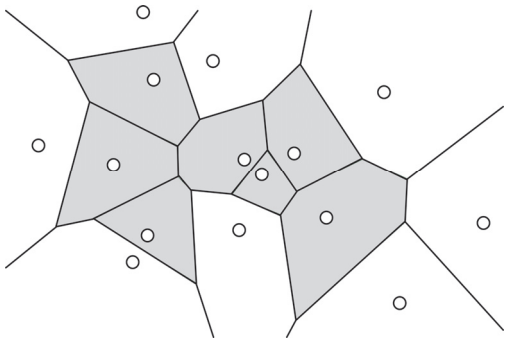

Fig. 2. Voronoi regions

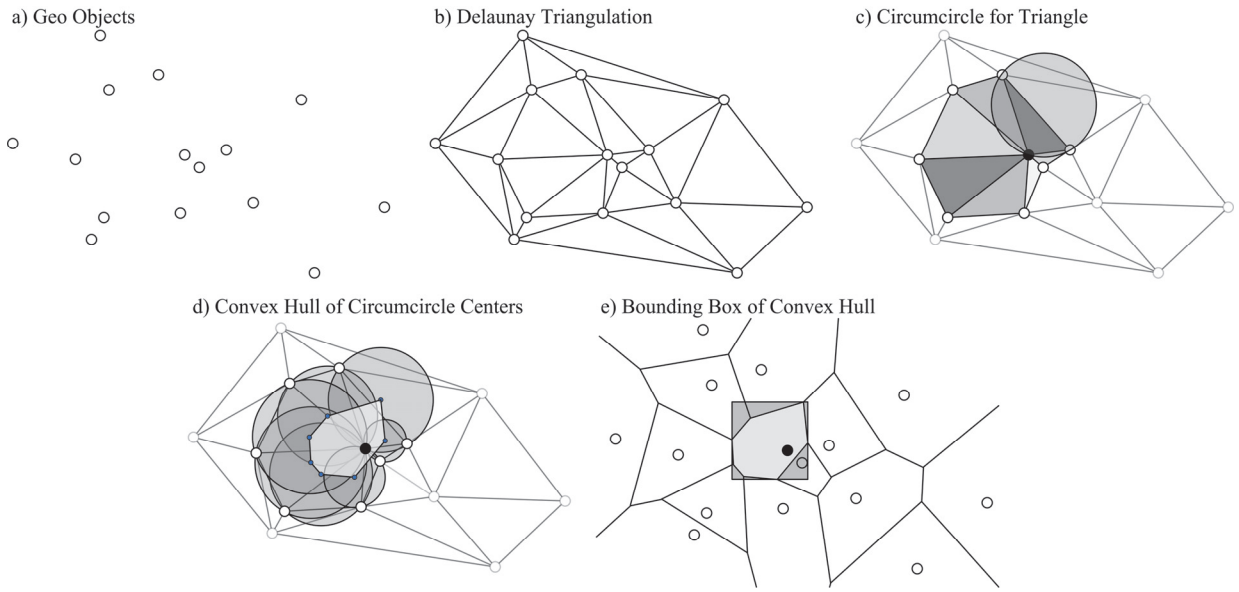

Fig. 3. Generating the bounding box of nearest points

Our approach to execute Nearest Queries is based on *Voronoi regions* (fig. 2) [9]. For a set of points (the centers of our geo objects) Voronoi regions enclose all positions that are nearest to a certain object. They can be infinite (at the 'border') or finite (gray regions in fig. 2). Finite regions are always convex.

The main idea of our Near Index is to compute Voronoi regions for all objects and store the regions instead of object geometries in a traditional spatial index. Some considerations:

1. We have to compute an own set of regions for all object subsets defined by a *variable*. If e.g. a variable denotes '*touristic sights nearer than 500m*', the center points define all touristic sights. For this, our framework automatically scans all variables and builds the respective Near Indexes beforehand. This usually is executed *once* as part of the assembling and exporting task and requires some minutes of computation for million objects of a country.

2. As a typical spatial index does not store geometries but only their bounding boxes, we do not have to compute and store the actual Voronoi regions, but only the maximum and minimum coordinates of all region points. If later more than one bounding rectangle encloses a position, it is effective to exactly check distances, as the number of candidates is very small.

3. Variables always contain a maximum distance. Beyond this distance, an object is not considered as *near*, even though it may be the nearest. E.g., it is not reasonable to describe a position as '*50km south of market place Nuremberg*'. If a Voronoi region extends the maximum distance, the respective bounding rectangle is cut. This especially is important for infinite Voronoi regions at the border.

4. Efficient algorithms for Voronoi regions require to model objects as a single point. This obviously is a simplification as geo objects have a certain extent. Usually, we can accept this simplification. The majority of objects in the geo database are already point-like. For area-like objects (e.g. lakes, cities), we actually find the object with the nearest center, not the nearest border. Even though geometrically this is a difference, it does not affect the usefulness of a description. A significant difference however appears for line-like objects (e.g. rivers, roads). For these, we have to enter multiple object points that represent the shape into the Near Index.

The actual Voronoi computation works as follows (fig. 3):

- First, from the list of objects (fig. 3a), we generate a network of triangles using the so called *Delaunay Triangulation* (fig. 3b). It holds the property: no object resides inside a circumcircle of *any* triangle.
- For an object, we compute a circle through the three triangle vertices for all connected triangles (fig. 3c). Their centers form the vertices of Voronoi regions (fig. 3d).
- For finite Voronoi regions: we compute the maximum and minimum coordinates of all vertices (fig. 3e). This is the required nearest bounding rectangle for an object.

To detect infinite Voronoi regions, we use a simple approach: if the polygon of all collected vertices contains the object, the region must be finite; if not, some borders must be infinite half-lines in order to cover the object.

A last question is when to use Euclidian and when to use spherical geometric formulas. The problem: most geometry software libraries (e.g. *JTS* [7]) are based on Euclidian geometry. This mainly affects the computation of a circle through three given points.

Fig. 4 illustrates a problem. If an algorithm considers geo coordinates as Euclidean, the center point of a circle through three points does *not* describe the position on the Earth's surface that has the same great circle distance to these three points.

An argument, why we often still use Euclidian geometry for geo computations is: in small scales, the curvature of the Earth

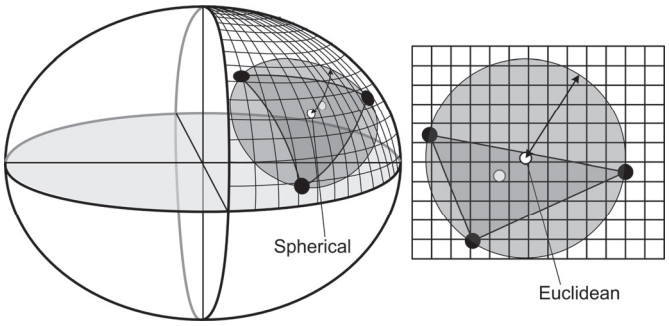

Fig. 4. Spherical vs. euclidean circle construction

can be neglected. This is why in our first approach for Voronoi regions was based on Euclidian computations. It surprisingly turned out, even in small scales (some 100m), very often the Near Index then wrongly returned the second nearest object. Thus, in our final solution:

- We replaced the computation of the Voronoi vertices by a more complex spherical formula.
- Even though the Delaunay Triangulation also has to deal with distance circles, we could still use the more simple Euclidian approach without noticeable errors.

The last point is important: the Delaunay Triangulation is the more complex part, thus, it is useful to re-use existing library function 'out of the box'.

We fully implemented the Near Index based on the Extended Split Index. The speed up compared to a traditional spatial index is significant (see next section).

IV. Evaluation

We evaluated the described mechanisms in our HomeRun platform. In detail: a) we measured execution times in different runtime environments; b) we checked the efficiency of the Execution Planner based on Lazy Evaluation and Earliest Failure First; c) we checked the efficiency of our Near Index.

We imported the geo data set 'Germany' from Open Street Map. It undergoes the import process pipeline to our HomeRun data representation that especially provides a strong classification. Some characteristic numbers:

- We have 34.6 million geo objects in an area of 357 thousand km^2.
- We have 96.8 objects per km^2 on average; 8.6 objects on average cover a certain position.
- Our output is based on 15 variables in 10 rules. We performed 10000 random queries.

Table II shows examples of generated descriptions. We measured the average runtime for a query in three runtime environments:

- *Server*: Xeon E5640 2.67 GHz, Windows Server 2008
- *PC*: AMD 9920 2,2 GHz Windows 7
- *Smart Phone*: Galaxy Nexus i9250, 1.2 GHz Cortex-A9, Android 4.3

TABLE II. Examples of Generated Descriptions

Position	Generated Description
49.469017/11.155400	Nuremberg, Laufamholz, near the castle 'Oberbürg'
49.450253/11.139154	Nuremberg, Am Tiergarten 30, Zerzabelshof, ('Am Tiergarten' corner 'Schmausenbuck'), near the bus stop 'Tiergarten'
49.450578/11.070893	Nuremberg, Ludwigsplatz, Lorenz, near the metro station 'Weißer Turm'
49.123310/10.927830	Pleinfeld, Ramsberg am Brombachsee, near the harbour 'Ramsberg'
50.463897/10.667095	Reurieth (Siegritz, 2 km northeast of center Reurieth)
50.732300/6.900087	Weilerswist, Metternich, 100 m from the conservation area 'Swistauen'
48.562963/7.547785	20 km west of Kehl
51.500781/11.535926	Country road 'K 2319' in Eisleben
49.444506/8.898446	Hirschhorn, near steakhouse 'By Debo'
50.921943/6.977095	Cologne, Deutz, 200 m from the industrial park 'Deutzer Hafen'
48.654684/13.616025	Hauzenberg, Steinberg, 300 m west of the supermarket 'Lidl'
49.669420/8.004593	Kirchheimbolanden, Kupferberg, ('Schlesienstraße' corner 'Danziger Straße') 300 m northwest of the church 'St. Peter'
50.013844/11.609961	600 m to Harsdorf, Upper Franconia

We developed our runtime library (size approx. 1MB) in Java, thus could easily create applications for different environments. We measured execution times for two engines: *DB* that uses the HomeRun SQL database, and *File* that is based on a highly compressed file format, optimized for random access. As on smart phones only the embedded database SQLite with very limited index support is available, we tested DB only on PC and Server. Both engines were tested with our Near Index and without, i.e. with a traditional spatial index that simulates nearest queries. Table III shows the results.

TABLE III. Average Time in ms to Generate a Description

	DB with near index	DB w/o near index	File with near index	File w/o near index
Server	4.0	93.8	0.9	12.2
PC	4.4	135.3	1.3	19.0
Smart Phone	N/A	N/A	146.6	858.6

Server and PC nearly perform equal. The smart phone is significantly slower due to a number of reasons: first, the general execution speed is lower. Second, the amount of RAM per app is very limited, thus we cannot benefit from caching. Third, the access to external files cannot benefit from *memory mapped files* again as a reason of low available memory. Even though 146.6 ms for a single output generation is very long compared to PC and server, it still is feasible for typical apps, as during usual operation, location descriptions will not permanently be queried.

Please note that the operation is completely performed offline (i.e. without any network service), even on the smart phone. All spatial data and index files have a size of 1.3 GB and stored on the smart phone's flash memory. Even though these files are considerably large, the library takes care only to consume a small amount of RAM during operation.

Table III also illustrates the benefit of the Near Index that is up to 30 times faster compared to a traditional spatial index. Some findings:

- For Nearest Query, the traditional index returns 19.7 candidates on average that had to undergo the exact geometric check.
- Our Near Index in contrast returned 0.155 candidates on average. This means, the index directly stated for 84.5% of all queries: *there is no nearest object within the maximum distance*.
- As a result, the traditional index has to check 127 times more candidates. Note that for every check, the geo object has to be retrieved from the database or file to get its exact geometry.

To sum up: the Near Index has not only proofed its efficiency – we could claim this new type of index is indispensable for the approach.

We finally checked the Execution Planner.

- From the 15 variables, due to Lazy Evaluation only 12.6 variables had to be resolved on average (84%).
- As we apply Earliest Failure First in addition, we had to resolve 8.6 variables on average (57%).
- The theoretical minimum of resolved variables to find the firing rule (according to section III.B) was 8.4.

Our approach is very close to the theoretical optimum.

V. Conclusions

A textual description is an important means to specify the current location besides a map output. There exist many scenarios, where the map output is not possible and we have to rely on pure texts. We presented an approach that allows even small offline devices to generate appropriate location descriptions. Access to geo data repositories is encapsulated by Engines that can easily be exchanged.

We strongly believe there is no single suitable description for all applications, thus we allow an application to easily formalize the desired output. This formalism is based on *variables* and *rules* that are specified in a program-like syntax. We presented an efficient mechanism to check variables and rules based on Lazy Evaluation and Earliest Failure First.

Beside the information 'where am I inside' we identified different *nearest* objects as important information to describe a location. As traditional spatial indexes are not optimized to query for nearest objects, we introduced a new index type, the Near Index. We proofed its effectiveness.

In the future, we not only want to consider the user's location but also the orientation, driving direction or position history. This can improve a useful description even more. E.g. a digital camera could only consider touristic sights that may appear on a photo because of the view angle.

As a further direction, we want to extend the notion of 'nearest': in some scenarios, the human perception of nearby objects significantly differs from the geometric notion. Consider to be on one side of a highway. All nearby objects on the other road side are candidates to be geometrically nearest objects, even though a real person would not agree. Thus, in the future, we want to extend our near index to also consider path planning results.

References

[1] Android Developers, http://developer.android.com

[2] ArcGIS API for JavaScript, developers.arcgis.com

[3] Bennett, J., 2010: OpenStreetMap, Packt Publishing

[4] Dey, A., K.; Abowd, G., D., 1999: Towards a better understanding of context and context-awareness, GVU Technical Report GIT-GVU-99-22, Georgia Institute of Technology

[5] Finkel, R., Bentley, J., L., 1974: Quad Trees: A Data Structure for Retrieval on Composite Keys. Acta Informatica 4 (1): 1974, 1–9

[6] Guttman, A., 1984: R-Trees: A Dynamic Index Structure for Spatial Searching. Proc. of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, June 1984, 47-57

[7] JTS Topology Suite, http://sourceforge.net/projects/jts-topo-suite

[8] Kenneth, E. F, Lynch, M., 2014: Geographic Information Systems as an Integrating Technology: Context, Concepts, and Definitions, Geographer's Craft Project, Department of Geography, University of Colorado

[9] Liebling, T., M., Lionel, P, 2012: Voronoi Diagrams and Delaunay Triangulations: Ubiquitous Siamese Twins, Documenta Mathematica. Extra Volume ISMP, 2012, 419-431

[10] Pradhan, S., 2000: Semantic Locations, Personal Technologies, Vol. 4, No. 4, 213-216

[11] Roth, J., 2005: A Decentralized Location Service Providing Semantic Locations, habil thesis, University of Hagen

[12] Roth, J., 2009: The Extended Split Index to Efficiently Store and Retrieve Spatial Data With Standard Databases, IADIS International Conference Applied Computing 2009, Rome (Italy), 19.-21. Nov. 2009, Vol. I, 85-92

[13] Roth, J., 2010: "Die HomeRun-Plattform für ortsbezogene Dienste außerhalb des Massenmarktes", in A. Zipf, S. Lanig, M. Bauer (eds.) 6. GI/ITG KuVS Workshop Location based services and applications, Heidelberger Geographische Bausteine Heft 18, 2010 (in German)

[14] Roth, J., 2011: Context-aware Apps with the Zonezz Platform, ACM MobiHeld 2011, Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds, Cascais (Portugal), Oct 23 2011

[15] Roth, J., 2013: Combining Symbolic and Spatial Exploratory Search – the Homerun Explorer, Innovative Internet Computing Systems (I^2CS), Hagen, June 19-21 2013, Fortschritt-Berichte VDI, Reihe 10, Nr. 826, 94-108

[16] Roth, J., 2014: From Weak to Strong Geo Object Classification, in Schau V., Eichler G., Roth J. (eds): Proc of the 10th Workshop Location-based application and Services (LBAS) Sept. 16-17 2013, University of Jena, Germany, Logos Verlag Berlin, 3-12

[17] Roth, J, 2014: Predicting Route Targets Based on Optimality Considerations, Intern. Conf. on Innovations for Community Services (I4CS), Reims (France) June 4-6 2014, IEEE xplore, 61-68

[18] Roth, J., 2014: Fast Spatio-Symbolic Searching in Huge Geo Databases, Proc. of the 11th Workshop Location-based application and Services (LBAS), Darmstadt, Germany, to appear

[19] Swan, M., 2012: Sensor Mania! The Internet of Things, Wearable Computing, Objective Metrics, and the Quantified Self 2.0, Journal of Sensor and Actuator Networks, 2012, 1(3), 217-253