Fast Spatio-Symbolic Searching in Huge Geo Databases

Jörg Roth
Department of Computer Science, Nuremberg Institute of Technology,
Kesslerplatz 12, 90489 Nuremberg, Germany
Joerg.Roth@th-nuernberg.de

Abstract: Searching suitable entries in large amounts of geo objects is a typical function for all kinds of location-based applications. In contrast to traditional search functions in databases or text document repositories, geo object search combines symbolic, spatial and navigational conditions. Different types search engines perform the actual search, each optimized for a specific search task. In this paper we present the HomeRun search environment that integrates different search engines to perform typical search tasks in the area of location-based services and applications. The application developer can express complex combined queries and the desired output. The search environment computes an optimized execution plan and supervises the execution. The application developer can tailor the environment to the application's search tasks and runtime environment.

1 Introduction

Geo databases form the basic resources for location-based services and applications. The amount of geo data usually is very large and easily counts many millions of records for an area of a country. For many applications, a basic operation is to find a certain record or a small set of hits from these millions of records. As examples: a route planning tool requests a user to define a route destination, e.g. by address or by search of nearby points of interests; a tour guide provides a search facility for touristic sites that are displayed on a map. We strongly believe, the underlying search function should not be implemented from the scratch for every new application. In contrast, the application should only configure the specific search task and the actual search should be executed automatically. As different applications may have different demands on the search function, we want to support the following degrees of freedom:

- *Offline vs. online search*: an application may search in locally stored data or may use a network search service.
- *Different search engines*: the developer can choose different inner search engines that perform the actual search tasks.

- *Support of different search types*: we support *spatial* queries (inside geometries, conditions on area size, line length or altitude), *symbolic* queries (any type of textual search), *nearby* and *containment* search to other objects (in a country, nearby a city), *postal addresses* and *navigational* searches (e.g., within reach in *x* min by car) and all combinations of these.
- *Support of different result types*: an application may ask for a single object (e.g. '*Pizzeria La Luna in Nuremberg*') or a list of objects (e.g. '*all Pizzerias in eastern Nuremberg*'). In addition, an application may only request the number of hits, query for the compound geometry of all results or the joint bounding rectangle of all result geometries.

Important: for a certain application, not all degrees of freedom are appropriate. E.g. a specific search engine may not be supported by an execution platform or is not required for the certain search function. Also the respective search type could request either too much memory or execution time. The application developer thus can tailor the search environment to the respective application scenario.

2 Problem Statement

In contrast to traditional search (e.g. in the World-Wide Web) that mainly support symbolic full text queries, geo data search also has to incorporate spatial conditions. A user or application that looks up geo objects may express, e.g.

- conditions related to objects names, object types or addresses;
- where objects geometrically should reside (e.g. *in* a certain rectangle or *above* a certain altitude);
- where objects should reside in relation to larger structures (e.g. *in* a city, *nearby* a city, *in* a state);
- where objects should reside in relation to other objects (e.g. *nearby a railway station*, *not nearby a landfill*);
- additional spatial conditions that these objects should fulfil, not related to locations (e.g. a size *larger than x* m^2 or a length *smaller than x* m);
- conditions that are related to path planning (e.g. *reachable within 10 minutes by car*).

We have to consider combinations of these and also complex queries such as: *give me all bistros that are in walking distance of 5 min from my currently location*. Or: *give me all hotels in Nuremberg that are in walking distance (5min) of any metro station, which connects me to the conference site*. Such complex searches request the expression of spatial joins or sub-queries.

Queries can be defined in different ways. For a *strongly structured* search a user expresses a query field by field. The application may explicitly ask for object type

(e.g. hotel) and all postal address fields. In contrast, a free search allows the user to enter an arbitrary *unstructured* query text, e.g. '*Nuremberg University*'. The search system then tries to identify the type of words and to resolve ambiguity. For unstructured queries, not all attributes of an object may be provided by a user. As a consequence, more hits than expected usually are presented and a ranking system has to order the results.

A caller can also request different results. The most obvious result is the *list of objects that match the query*. But lists may vary in how objects are described. They may, e.g. contain only object IDs, basic fields or even all object geometries. A caller also may only be interested in the count of results or the bounding box of all result.

Even though search request are considered as a single action from the viewpoint of the caller, they actually base on multiple *inner searches*. Inner searches are usually modelled by inner search engines that are highly optimized for the specific search tasks, e.g.

- databases: optimized for scalar or interval conditions that can be checked by exact match and that may result huge result lists; searches are speed up by index column structures such as B-Trees;
- full text search engines: fast textual search in documents, prefix or fuzzy searches, complex ranking that indicates the similarity and number of matches;
- spatial indexes: optimized for geometric searches with the help of index structures (e.g. R-Trees), conditions either compare geo data to a fix geometry or compare two lists of geo data geometries (spatial join);
- path planning: optimized for conditions that express the reachability by a means of transportation;
- in addition, some conditions can be computed internally by algorithms directly linked to the runtime systems.

As these search engines have their specific focus, it is not reasonable to execute all parts of a query by a single search engine. In a usual search, multiple engines have to be asked to generate the final result. Based on these considerations, we can formulate our requirements as follows:

- Queries, results and inner search engines have to be formally modelled.
- Inner search engines have to be integrated by a wrapper into the runtime system. The respective interface must reflect all formal properties and runtime issues.
- The search environment automatically generates an execution plan that transfers the queries to the requested results, using available inner search engines.
- The execution plan should be optimal according to given criteria (usually execution time).
- The search environment executes the plan, supervises the inner search engine and collects the results.

In the following we present an environment that meets these requirements.

3 The Spatio-Symbolic Search in the HomeRun Environment

The HomeRun environment [3] provides basic tools and building blocks for location-based services and applications, such as mastering of geo data, map display and route planning. As a new important service we identified the geo object search. Not every application should implement its own search infrastructure; instead, a framework should support different types of searching. An application ideally can integrate search functions with a few lines of code.

The HomeRun search environment supports all types of searches mentioned above, whereas the developer can restrict the search capabilities if either they are not desired or cannot be executed in the specific runtime environment. As the superset of possible inner search engines we identified:

- *Lucene* and an own textual search mechanism [7] for all kinds of textual conditions;
- the *Extended Split Index* [1, 2] for spatial conditions that also supports mobile platforms [5, 6];
- *donavio* [9] for all navigational conditions, also available for mobile platforms [6];
- the *HomeRun classification environment* [8] for all conditions concerning geo object types;
- SQL databases for all exact matches or interval conditions;
- additional internal functions for further query execution (e.g. set intersection).

It is easy to integrate more search technologies and engines later. We only have to formalize the search engine (see next section).

The HomeRun search environment is bound to the respective application as a software library. During instantiation, the application developer can decide which inner search engines are used inside the application. Often not all available search capabilities are really used inside a certain application:

- A specific search function may not be required in the application. An example is the navigational search that is often too unique for many applications.
- Due to memory and runtime issues, the application developer decided to exclude some search engines. Search engines usually come along with large index structures that have to be deployed, accessed or stored. In addition, search engines require a certain amount of execution time.
- Some search engines are not accessible in a certain runtime environment. As an example: Lucene is not available for all smart phone platforms.

An application developer carefully has to decide which inner search engines to use to meet the application's and user's demands for the specific search tasks.

3.1 Modelling Queries, Results and Search Engines

A certain search tasks is defined by *queries*. Each query indicates a type of conditions, e.g. concerning the geo object's name, address or location as presented in table 1. Queries are combined by logical **AND** as usually additional queries should restrict the number of results. Possible **NOT** and **OR** conditions have to be modelled by the respective query. E.g. if the caller searches all objects

in Nuremberg **AND** (*type is Bistro* **OR** *type is Restaurant*)

the required **OR** conditions for types is provided the by type query, that always accepts a list of possible geo object types, what implicitly provides a logical **OR**.

More complex query types, known from relational database queries are joins and sub-queries that are strongly related. We decided to express both by sub-queries, whereas the sub-query output must either result in sum geometry or in an objects list. As a result, we can express all supported searches by a nested set of queries, whereas nesting is used to express sub-queries.

Table 1. Currently supported queries

Query	Condition
Address	ask for country, city, ZIP code, street or house number or combinations
Altitude	ask for altitude in meters (≤, ≥, *between*)
AreaSize	ask for the surface area size in m^2 (≤, ≥, *between*)
ClassByName	ask for object type (e.g. 'restaurant'), defined by a name or a synonym, supports fuzzy and prefix search
Class	ask for object type, defined by a list of internal class IDs
LineLength	ask for the length in meters of line-like objects, e.g. streets (≤, ≥, *between*)
Realname	ask for the object's name, supports fuzzy and prefix search
RelativeByName	define surrounding or nearby objects, e.g. a city where the respective objects has to reside in, supports fuzzy and prefix search
Relative	define surrounding or nearby objects, given by a list of IDs
TextLine	pass a textline query (unstructured search)
Type	define general attributes, e.g. geometric type (*point*, *line*, *polygon*)
Geometric	restrict the location by e.g., polygon, circle or rectangle or even more complex geometric structures (*inside*, *outside*, *overlapping*)

The search environment must have the possibility to optimize the execution of inner search engines to generate the specific result, but not more results as required. As an example: if the caller does not want to get a list of hit objects, but only the count of results, the environment could in principle also compute the entire list and leave it to the caller to count the entries. However, if the involved search engine is an SQL data-

base, it is much more efficient to execute a `SELECT COUNT(*)`… instead of passing the detailed result table to the application. As a consequence, the caller must express the desired output in a fine granular manner. We accept the following results:

- list of geo objects (called *domain list*), in addition we can define requested fields per object, e.g. only ID, basic fields or if we want to get the object's geometry;
- the number of results;
- the *minimal bounding rectangle* (*MBR*) of all result geometries;
- the sum geometry of all result geometries, maybe extended by a *buffer* operation;
- the list of road points (i.e. starts or targets for path planning);
- the *isochrone* geometry, i.e. a geometry that indicates a border of reachablilty using path planning under certain costs.

The caller is free to request any non-empty subset of these results.

Finally, we have to model inner search engines. First, we need a formal model of the engine as illustrated in fig. 1 to generate an optimized execution plan.

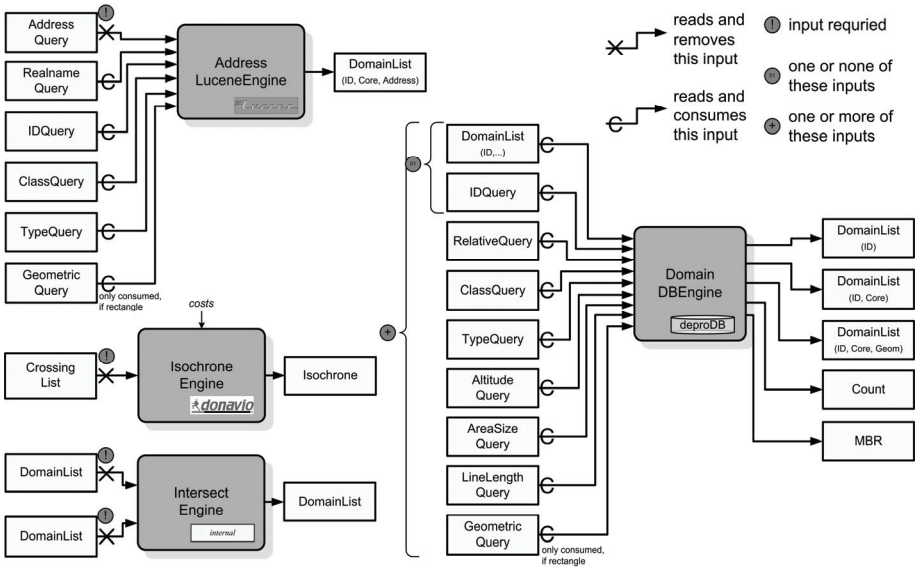

Fig. 1. Formal representation of some inner search engines

Each search engine has to indicate a mapping of queries to results and provides an estimation of the required execution time. Due to plan generation issues (next section) the formal representation specifies additional information how to deal with queries:

- *consume*: the query is considered by the engine but left in the set of queries. Other search engines can again read and process the same query.
- *remove*: the query is read and removed from the set of queries.

This distinction is used to redundantly read certain queries multiple times to address the large intersection problem (see section 3.3).

The second goal of the search engine model: we need a unified wrapper to integrate arbitrary engines into our search environment. It supports these tasks:

- the respective structures have to be accessed (e.g. indexes are loaded into memory, files or databases have to be opened);
- our queries have to be mapped to engine-specific queries (e.g. SQL or Lucene);
- results have to be mapped to HomeRun results;
- searches have to be initiated and cancel requests have to be passed to the engine;
- errors have to be reported to the caller in a platform-independent manner.

The required control is achieved by an object oriented class system that provides an interface structure to model search engines.

3.2 Generating an Execution Plan

Based on the available inner search engines, queries and required results, the environment has to create an execution plan. A plan is successful, if all queries are removed or consumed and the required results are generated. If more than one plan meets these requirements, the plan with the lowest runtime has to be identified.

Fig. 2 shows an execution plan for the query: *area objects in 'Nuremberg', smaller than 1000m^2, type 'University', inside a certain circle*. We request as results: list of objects and the buffer sum geometry.

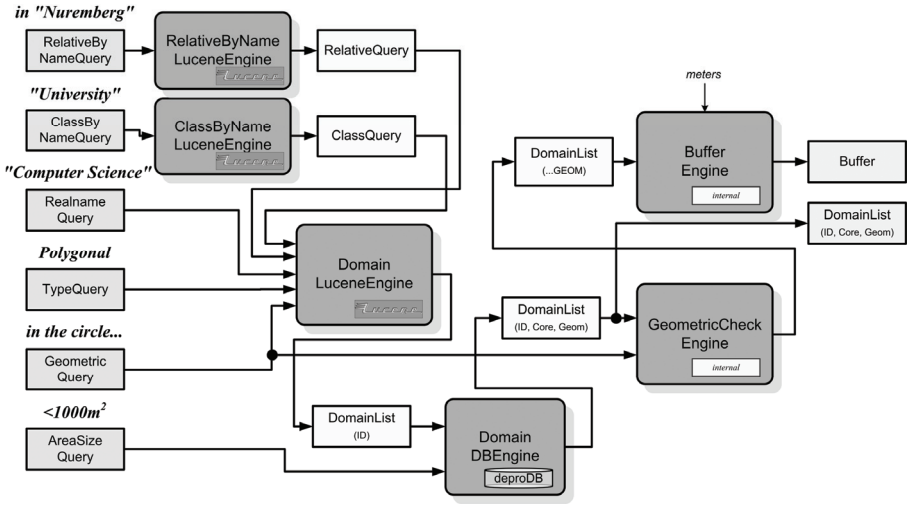

Fig. 2. A sample execution plan

To find execution plans is also a task for relational databases. However there are some important differences. In relational databases, the query explicitly formulates the intermediate results. In our case it is not clear, which intermediate results lead to the desired output. Planners for database queries have to plan more fine-grained. Decisions are, e.g. ordering of conditions, usage of indexes and caches. Most important: the database planner has full access to all internal structures. In our case, we have to consider the inner search engines as black boxes.

The resulting planning problem is very similar to the parsing problem in formal languages. We can regard the formal representation of an inner search engine as *production rule*. The goal is to derive a certain set of symbols (our *result*s) from a start set of symbols (our *queries*) with the help of available productions (our *search engines*). Unfortunately, the respective formal language behind this problem is of type-0 (unrestricted grammar). For type-0 parsing, there is not even an algorithm that can decide, if a result can be produced. Moreover in our case, we do not only want to know *if* there is a plan, we want to select the *optimal* plan.

As a solution, we introduced so called *meta rules* that significantly reduce the number of plan combinations to check:

- First, we know the maximum path length inside a plan (excluding sub-queries). This is because we can identify the most complex reasonable query that makes use of all possible inner search engines. Currently, the maximum is 8.
- Second, for most search engines only a single execution per search is reasonable. In addition, some engines can be viewed as alternatives, of which only one can be reasonably executed.
- Finally, there are certain engines that only occur on the very left or very right side of the execution chain.

The required information are incorporated into the inner search engine model, thus the execution planner can optimize the search based on these information. As a result, execution plans can effectively be created at runtime for any combination of queries and results.

3.3 Further Challenges

The large intersection problem: To execute an **`AND`** query, all parts could in principle be executed individually and the final result would be the intersection of all results. E.g.: we search *all hotels (***`AND`***) in a certain geographical region (***`AND`***) that have* '*Grand*' *in their name*. The query execution could be: *all hotels at all*, intersected with *all objects in the geographical region*, intersected with *all objects that have 'Grand' in their name*. Then however, we had to deal with very large intermediate results that cause large memory requirement and long execution time. As a second issue: to re-

solve ambiguity (especially for unstructured search), a single search index often has to queried multiple times. As a major goal we want to reduce the number of total search engine executions and to reduce the amount of intermediate results that are removed later due to a result intersection.

To solve this, we introduced redundant data into search engines. Even though each search index is optimized for a certain query type, we integrated additional data that actually is data from another index. As an example: we add location data to the full text search index. If we look up the full text index in combination with a spatial query, the text index additionally considers the location. As the respective mechanism is not as effective as our spatial index, the index may only roughly test a condition, e.g. based on bounding rectangles instead of exact polygons. This may result in false positive hits in the first intermediate result. But as the search engine chain must exactly check later, this is not a problem. But as the intermediate result is significantly smaller, this approach is effective.

Support of parallel execution: After an execution plan has been generated, it will be executed using the inner search engines. The plan execution already supports parallel processing, if desired. This means: whenever all input for a specific inner search engine is available, it will be executed, even though another engine is currently running. If a runtime environment has resources to execute more than one engine, the overall execution time is therefore reduced. However, the time to finish all executions can be minimized even more, if we modify the optimization criterion for the execution planner: instead of minimizing the sum of execution times of all inner search engines, we can minimize the longest path between any pair of query and result. This however may lead to completely different execution plans with considerably more CPU load, but that earlier generates a result. The application developer can decide which type of optimization is suitable for a certain application and execution environment.

4 Conclusion

The HomeRun search environment provides an effective search tool that can be linked to an application. We believe to find geo objects is a fundamental function in the area of location-based applications, whereas the application developer should not be responsible to plan and execute the respective search tasks. As a search for geo objects may contain many different facets, we strongly rely on inner search engines that are optimized for different search types, e.g. full texts, navigation or location. New search engines and query types can be integrated into the environment using a

modelling schema. This supports a formal representation used to generate an execution plan and an execution model, used to control the execution at runtime.

The generation of execution plans was surprisingly difficult as the underlying theoretical problem was comparable to the parsing problem of type-0-languages. We solved this by meta rules that significantly simplified the problem. In addition we considered the large intersection problem and are able to generate execution plans optimized for parallel execution.

References

1. Roth J.: *Verwaltung geographischer Daten mit Hilfe eines Add-ons für Standard-Datenbanken*, Verwaltung, Analyse und Bereitstellung kontextbasierter Informationen, GI Informatik 2009, Lübeck, 29.9.2009, 2041-2055
2. Roth J.: *The Extended Split Index to Efficiently Store and Retrieve Spatial Data With Standard Databases*, IADIS International Conference Applied Computing 2009, Rom (Italy), Nov. 19-21, 2009, Vol. I, 85-92
3. Roth J.: *Die HomeRun-Plattform für ortsbezogene Dienste außerhalb des Massenmarktes*, in Zipf A., Lanig S., Bauer M. (Hrsg.) 6. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", Heidelberger Geographische Bausteine Heft 18, 2010, 1-9
4. Roth J.: *Übernahme von Geodatenbeständen aus Open Street Map und Bereitstellung einer effizienten Zugriffsmöglichkeit für ortsbezogene Dienste*, Praxis der Informationsverarbeitung und Kommunikation (PIK), Vol. 13, No. 4, 2010, 268-277
5. Roth J.: *Moving Geo Databases to Smart Phones – An Approach for Offline Location-based Applications*, Innovative Internet Computing Systems (I2CS), Berlin, June 15-17, 2011, GI Lecture Notes in Informatics, Vol. P-186, 228-238
6. Roth J.: *A Spatial Hashtable Optimized for Mobile Storage on Smart Phones*, in Werner M., Haustein M. (Hrsg.): 9. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", Sept. 13-14, 2012, TU Chemnitz, Universitätsverlag Chemnitz, 2013, 71-84
7. Roth J.: *Combining Symbolic and Spatial Exploratory Search – the Homerun Explorer*, Innovative Internet Computing Systems (I2CS), Hagen (Germany), June 19-21, 2013, Fortschritt-Berichte VDI, Reihe 10, Nr. 826, 94-108
8. Roth J.: *From Weak to Strong Geo Object Classification*, 10. Fachgespräch "Ortsbezogene Anwendungen und Dienste", Sept. 16, 17, 2013, FSU Jena, im Druck
9. Roth J.: *Modularisierte Routenplanung mit der donavio-Umgebung*, in Werner M., Haustein M. (Hrsg.): 9. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", 13.-14. Sept. 2012, TU Chemnitz, Universitätsverlag Chemnitz, 2013, 119-131