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

Geo databases form the basic resources for location-based services. 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 the user to define a route destination (e.g. by address); a tour guide provides a search facility for touristic sites that are displayed on a map.

The HomeRun environment [3] provides basic tools and building blocks for location-based services, e.g. mastering 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.

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

- where these objects geometrically should reside (e.g. in a certain rectangle or above a certain altitude);
- where these objects should reside in relation of larger structures (e.g. in a city, nearby a city, in a state);
- where these 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 fulfill, not related to locations (e.g. a size larger than x m² or a length smaller than x m).

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. 'Darmstadt 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 result.

To support a huge variety of applications, the HomeRun search allows the following degrees of freedom:

- Offline vs. online search: an application may search in locally stored data or may use a search service.
- Different search engines: the developer can choose different search engines, or may remove certain search engines that either are not supported by an execution platform or not required for the certain search function.
- Structured vs. unstructured search: an application can provide a strongly structured search facility or may provide an unstructured free search.
- 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, object properties 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.
- Support of exploratory search: that requires: 1. high search performance (query during typing) 2. informative query feedback (e.g. immediate map display) and 3. additional query support (e.g. suggestions to complete a search word or to complete the entire query).
Inside the HomeRun search different search engines are used. 1. A traditional database is used, e.g., to restrict the results to special object classes (e.g. all restaurants). Our classification approach [4, 8] allows to query for certain object type classes (including all sub classes and list classes) in a single column condition that is speed up by standard indexing mechanisms. Database queries are also used for such spatial properties that can be expressed by scalar values (e.g. area size, diameter or length). 2. A spatial search is used whenever the query contains a comparable geometry (e.g. polygon, circle or rectangle) together with a required relation (e.g. 'completely inside', 'touches the border'). Spatial search is based on a spatial index as geometrical checks usually are costly. 3. A textual search is used, whenever search words should appear in the object's name, class, address or property. Exact, fuzzy, prefix and 'sounds like' search are supported.

From the implementation point of view we support different technologies for textual search (e.g. Lucene, SQLite FTS), spatial indexing (e.g. geo database extensions) and non-spatial queries. In addition, we support our own geo-textual indexing mechanism [7] and a spatial index mechanism [1, 2] that also supports mobile platforms [5, 6].

To execute a query, the sub queries passed to the search engines could in principle be executed individually and the final result would be the intersection of all results. E.g.: we look for all hotels in a certain geographical region 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 its 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 resolve 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.

We minimize the overall execution time by three mechanisms:

- Even though each search index is optimized for a certain query type, we integrate additional (redundant) data that actually is data from another index. We e.g. add spatial data to the full text search index.
- We support simultaneous execution whenever possible.
- We avoid loading data that is not required for the final output. If, e.g. the application wants to compute the joint bounding rectangle of all geometries, it is not necessary to load the respective exact geometries.

The number of permutations of query types and result types is very high (approx. 400 000). To manage the complexity of the execution planning problem, we introduced an approach based on production rules. Each production rule is able to model a certain search engine execution. We are able to express the expected execution time and to identify independent executions. To generate an execution plan, we thus compute the sequence of production rules that map queries to the required output and minimize the expected execution time.

References