Location-aware search with Apache Lucene and Solr

Combine unstructured text and spatial data to enhance your search applications

Grant Ingersoll January 12, 2010

Whether looking for the nearest coffee shop on a GPS-enabled smartphone, nearby friends via a social-networking site, or all trucks within the city delivering a certain product, more and more people and businesses are using location-aware search services. Creating such services has often been the domain of expensive proprietary solutions and geospatial experts. Recently, however, the popular open source search library, Apache Lucene, and the powerful Lucene-powered search server, Apache Solr, have added spatial capabilities. Lucene and Solr committer Grant Ingersoll walks you through the basics of spatial search and shows you how to leverage its capabilities to power your next location-aware application.

Location, location, location! Just as location is king in real estate, search plus location awareness can yield significant dividends in helping users effectively and efficiently find information. For instance, if you are a business-directory provider (such as a "Yellow Pages" site), when a user searches for a plumber, the site should return results for plumbers near the user's home. If you run a tourism site, you might want to allow travelers to search for places of interest near their destination in order to locate services that will help them enjoy their visit. If you're building a social-networking site, you might use location information to help users connect with friends. And the ubiquity of location-aware devices, such as in-car navigation systems and GPS-enabled cameras — and the large volume of publicly available map data — provide a wide variety of opportunities for building out Geographical Information Systems (GIS) that incorporate search to produce superior results for end users.

Use of spatial information goes well beyond search, but in this article I'll focus on how you can take advantage of spatial information to enhance your search application by leveraging the capabilities of Apache Lucene and Apache Solr. Why use a search engine? By no means is it a requirement, given that many good (even free) GIS tools are available. However, basing your application on search offers several powerful features where many other approaches traditionally fail. Search systems are effective at combining structured and unstructured data, enabling users to enter free-form queries that search free text such as descriptions and titles while restricting or altering the results based on geographical data. For instance, a travel site could implement a feature whereby...
a user can ask to find all hotels in Boston, Mass. with a 4-star rating and the phrase *comfortable bed* in the review text and *24-hour room service* in the description — all with subsecond response. A search system such as Apache Solr can also provide *faceting* (see Related topics for information on Solr and facets), highlighting, and spell checking on the result set so that the application can help users find what they are looking for more effectively.

I'll start with a brief review of some key Lucene concepts, leaving the deeper details to the reader to research. Next, I'll cover some of the basic concepts of geospatial search. GIS is a large field that could easily consume this entire article and many more, so I will instead focus on some basic concepts that should be fairly intuitive given the need to find services, people, and other items of interest on a daily basis. I'll round out the article with some discussion of the approaches available for indexing and searching spatial information using Lucene and Solr. I'll ground these concepts in a real, albeit simple, example using data from the OpenStreetMap (OSM) project (see Related topics).

### Recalling key Lucene concepts

Apache Lucene is a Java™-based, high performance search library. Apache Solr is a search server that uses Lucene to provide search, faceting, and many more capabilities over HTTP. Both are licensed under the commercial-friendly Apache Software License. See Related topics for more information on the features and APIs that each offers.

At their heart, Solr and Lucene both represent content as a *document*. A document is then made up of one or more *fields* plus an optional *boost value* indicating the document's importance. A field is made up of the actual content to be indexed or stored plus metadata telling Lucene how to handle that content and a boost value indicating that field's importance. It is up to you to decide how to represent your content as documents and fields, depending on how you wish to search and otherwise access the information in the document. You can have a one-to-one relationship between a single unit of your content or a one-to-many relationship. For instance, I might choose to represent a single Web page as a document with several fields such as title, keywords, and body. Or, in the case of a book, I may choose to represent each page of the book as a separate document. As you will see later, this distinction is important when it comes to encoding spatial data for search. The content in a field can be indexed or just stored as is for use by an application. If the content is indexed, then the application can search it. Indexed content can also be analyzed to produce *terms*, often called *tokens*. A term is the basis for lookup and use in the search process. A term is often a word, but it need not be. I encourage you to explore the Related topics to learn more about all of these concepts.

On the query side of the coin, Lucene and Solr offer rich capabilities for expressing user queries, ranging from basic keyword (term) queries, to phrase and wildcard queries. Both Lucene and Solr also offer the ability to restrict the space being searched by applying one or more filters, which are key to spatial search. *Range queries* and *range filters* are among the essential mechanisms for restricting the space. In a range query (or filter), the user states the need to restrict all searched documents to be between two values that have a natural sorting. For instance, it is common to use a range query to find all documents that occurred last year or last month. Underneath the hood, Lucene must enumerate the terms in the documents to identify documents that fall within the
range. As I'll show later, setting this up effectively is one of the keys to performance for filtering in spatial-search applications.

Lucene and Solr also provide the notion of *function queries*, which allow you to use the value of a field, such as a latitude and longitude, as part of the scoring mechanism, instead of simply the internal collection statistics that comprise the primary scoring mechanism. This will also come into play later in the article, when I demonstrate using some of Solr's distance-based functions.

**Geospatial search concepts**

First and foremost, when building a spatial search application, you need to identify the spatial data to be included in the application. This data often is in the form of some geocoded system such as latitude, longitude, and elevation, or as a ZIP code or a particular street address. The more formalized the system for encoding, the easier it will be to work within your system. For example, the old folk song "Over the River and Through the Woods" (you know the words: "to Grandmother's house we go") encodes a fair amount of spatial information into the lyrics (see Related topics). But it isn't all that useful in a GIS system, because we have no clue where the woods or the river are. Contrast that with detailed directions to Grandma's that include a starting address and an ending address, and you get the picture of why properly encoded data is important. (Interestingly, systems that can extract and codify more-generic directions and geographic entities — for example, *over the river* or *near the brown house* — and reason about them can also be quite useful, but that discussion is beyond this article's scope.)

Going beyond the raw, geocoded data used to identify locations, many GIS systems can add information that occurs relative to the physical locations. For instance, a navigation system may use a series of locations laid out in order on a map to create a route of travel from point A to point B. Or a meteorologist might overlay rainfall or severe-weather data across a map of an area and allow people to search for rainfall amounts by places of interest. People living near one another often group together areas to form ZIP codes, area codes, or even towns, cities, and states. In the case of OSM, users are allowed to edit and overlay information on top of the base map, such as places of interest or even streets. Combining these overlays to form relationships and tracking them over time can further produce incredibly dynamic and capable applications.

**Representing spatial data**

Regardless of the overlays or other information associated with one or more locations, search applications need a way to represent this data in an efficient manner. Although location information can be represented in several ways, I'll focus on the ones pertinent to Lucene. First and foremost, many types of geospatial data can be represented in their "raw" format and work just fine with a search application. For example, *Syracuse* is a perfectly fine way of representing the city of Syracuse, and users typing *Syracuse* in the search box will find the documents containing *Syracuse* just like any other search term. The raw representation is, in fact, the most common way people represent named locations, such as cities, states, and ZIP codes. Do note, however, that although I use the phrase *raw representation*, you can still transform or normalize the data first. For instance, converting all mentions of *New York* to *NY* is often a perfectly reasonable thing to do.
Before I tell you about the representations that Lucene can use, it is important to understand that all representations must take into account the spatial reference that produced them (see Related topics). In the United States, the most common is the World Geodetic System, often abbreviated as WGS 84 (see Related topics). Although transformations are possible between some systems, it is best to have all of your data represented in a single system. From here on, I will assume a single system.

Numeric spatial information such as latitude and longitude (lat/lon for short) is where the more interesting representations come into play in terms of search with Lucene and Solr. Latitude and longitude are usually expressed in degrees, minutes, and seconds from the Prime Meridian (located in Greenwich, England) and usually require using a double (or more) for precision purposes. For instance, for the data included in my example — the city of Syracuse, N.Y., United States — is located around 76.150026 East (or -76.150026 if East is not specified) and 43.049648 North.

Depending on the application, encoding every latitude and longitude can result in a large number of unique terms to be indexed. This isn't necessarily a show stopper, but it can slow down search significantly, and, as you'll see later in this article, it isn't always needed. In fact, many mapping applications only care about bounding the search within a specific area, so storing approximate information about the location often yields far fewer terms without adversely affecting the search results. This precision-tradeoff approach often captures latitude and longitude into (Cartesian) tiers. You can think of each tier as a zoom level on a specific part of a map, such that tier 2 centered over the United States likely encompasses all of North America, while tier 19 is likely in someone's backyard. More specifically, each tier divides the map into $2^{tier\#}$ boxes, or grids. Each box can then be given a number and indexed onto the document. I'll explain in a later section how to leverage this information for faster searches.

Latitude and longitude in Lucene terms are often represented as two different fields, but this can, in some applications, have implications for performance. If a single field is desired, lat/lon can be encoded into a single String using the Geohash encoding (see Related topics). Geohashes have the benefit of arbitrary precision by stripping off characters from the end of the hash. In many cases, locations that are near each other have common prefixes. For instance, entering Syracuse, NY into geohash.org yields the hash dr9ughxjkt4b, whereas entering the Syracuse suburb of Cicero, NY yields the hash dr9veggs4ptd3, both sharing the dr9 prefix.

So far, I've dealt specifically with points, but many geospatial applications are also interested in shapes, routes, and other relationships within the data. These are beyond the scope of what is available in Lucene and Solr; see Related topics for more information on these concepts.

Combining spatial data with text in search

Once data is represented in an index, search applications typically have at least five basic needs when it comes to interacting with the data:

- **Distance calculation**: Given a point, calculate the distance to one or more other points.
- **Bounding-box filter**: Find all matches (documents) that occur within some specific area.
• **Sorting**: Sort the results for a search by the distance from some fixed point.

• **Relevancy enhancement**: Use the distance as a boost factor in the score while allowing other factors to play a role too.

• **Query parsing**: Given an address or some other user specification of a location, create an encoded representation that can be used to search against the indexed data.

Each of these pieces can play an important role in location-based applications, but for now I'll focus on distance calculations, bounding-box filtering, and query parsing. Sorting and relevancy enhancement (boosting) just use the distance calculations, and I'll show how they play out in practice later in the article.

### Distance calculations

When calculating distances for use in GIS applications, it is important to understand there are many different approaches, each with its own merits and demerits. Distance calculations can be split into three groups, depending on how an application chooses to model the Earth. In some cases, it is perfectly acceptable to assume a flat model of the Earth and lose some accuracy in exchange for speed. In the flat-model approach, most distance calculations come down to variations on the Pythagorean theorem. The flat-model approach is often good enough when distances are short and locations are not near the poles. In other cases, a spherical model is used, and the primary distance calculation used is the **great-circle distance** (see [Related topics](#)). The great-circle distance calculates the shortest distance between any two points on the surface of a sphere. The spherical model is better-suited when distances are further apart and more accuracy is needed. Finally, an ellipsoidal model of the Earth can be used along with the Vincenty formula (see [Related topics](#)) to obtain highly accurate distances (down to 0.5 millimeters) on the ellipsoid, but for most applications the complexity of this calculation is probably not worth it.

**Give an inch, take a mile**

In many local-search applications, the need for accuracy depends on the application. In some cases, being off by a mile is no big deal, while in other cases being off by even a few millimeters can matter. For example, the Euclidean distance is often not accurate enough for spanning larger distances (say between two states), and even the Haversine (great-circle) approach may not be accurate enough in some cases, because the Earth is better modeled as an ellipsoid and not a sphere. In those cases, using Vincenty's formula may make more sense. In other applications, the only thing that matters is the ordering of the results, so something like the Squared Euclidean Distance (which isn't really a distance) can be used, thus saving a square-root calculation.

Of course, other distances are often useful as well, such as the Manhattan distance, which reflects the distance when one travels through a city laid out in blocks (as in a cab traveling through Manhattan in New York City). For the purposes of this article, I will demonstrate distances using the flat-Earth model and the great-circle distance and leave the rest to you to explore. Also note, I am disregarding elevation as a factor here, but some applications may need to account for it. For more information on geographic distances, see [Related topics](#).

### Bounding-box filter

In many location-based applications, millions of pieces of location data are available to be searched. Iterating through all of this data just to find the set of documents that both contain
the keywords of interest and are within some distance of the user’s specified location would be extremely time-consuming. Naturally, it makes sense to narrow down the set of documents first and then evaluate the relevant subset. If only latitude and longitude information was stored, then the primary option for narrowing down the document set is by passing in the ranges that encompass the area around the location. This can be visualized as in Figure 1, where the slightly opaque box represents a bounding box around the downtown area of Charleston, S.C.:

**Figure 1. Example of a bounding box centered on downtown Charleston, S.C.**

If the application also uses tier information or Geohash information, then these values can be used for better narrowing down the number of documents to search. I will demonstrate this later when discussing the specifics of indexing and searching with Lucene and Solr.

**Query parsing**

Query parsing comes down to determining which part of the query contains keywords to search for and which part contains location information. The latter part of this process is called geocoding (see Related topics). While I will discuss geocoding here in the context of query parsing, it is also useful during indexing. Consider the following examples of user queries:

- 1600 Pennsylvania Ave. Washington, DC
- 1 Washington Av. Philadelphia Pennsylvania
- Mall of America, 60 East Broadway Bloomington, MN 55425
- Restaurants near Mall of America
- Restaurants in the Mall of America

Examining the first two queries raises a few interesting points:

- Order of terms is almost always important, whereas in pure text-based search, order may not be important.
- Gazetteers and other spatial resources like GeoNames (see Related topics) can be quite useful in converting addresses to locations. These resources often contain lists of places of interest — for example, landmarks such as the White House.
• It is important either to normalize abbreviations such as Ave. and DC or to use synonyms to cover the variety of ways users input address information.

The remaining queries illustrate several other subtleties. For instance, in the third query, the user has specified a full address; care must be taken to parse out the name, address, city, state, and ZIP properly if you are going to search against fields for each of those properties. In the last two queries, the subtle difference between the user’s choice of near versus in is likely important. Any restaurant within some distance of the Mall would suit the fourth query’s user, whereas the last query’s user is only interested in results inside the Mall. Query parsing can be quite difficult because of the complexity of describing topics in relation to location, not to mention spelling mistakes, language ambiguity, and bad data.

Despite all the complexities of geocoding, services are available that translate addresses to location. Two commonly used services are the Google Maps public API and GeoNames (see Related topics). Unfortunately, using such Web services subjects you to their terms of use (which often include usage limits) and Internet traffic concerns. For real production systems, you are likely better off implementing your own capabilities. Although such an implementation is beyond this article’s scope, keep in mind that the GeoNames data is all freely downloadable, as are many other spatial resources, such as the CIA Factbook (see Related topics). Given good resources, it is best to build up from the basics (address, city, state) and then include places of interest and robust exception handling. Over time, your query logs will prove invaluable in creating a robust query parser that gracefully handles whatever users throw at it. As with any search application, keep in mind it is reasonable to make a best guess while also asking the user for clarification, as can be seen in the Google Maps screen capture in Figure 2:

**Figure 2. Best guesses and a request for user clarification on Google Maps**
For this article, I will demonstrate a basic query parser that uses the GeoNames service and has a few other features, but I'll leave it to the reader to implement a production-ready version. At this point, you should have enough background to get started. Thus, the remainder of the article will focus on how to index and search spatial information using Lucene and Solr.

**Installing the sample code**

In order to run the sample code, you need installations of the following:

- JDK 1.5 or higher
- Ant 1.7 or higher
- A modern Web browser such as Firefox

You also need this article's sample code (see Download), which includes a copy of Apache Solr and its dependencies. Follow these steps to install the sample code:

1. `unzip sample.zip`
2. `cd geospatial-examples`
3. `ant install`
4. Launch Solr: `ant start-solr` (to stop Solr later, run `ant stop-solr`)
5. Point your browser at `http://localhost:8983/solr/admin` and confirm that Solr started up properly. You should see a basic admin screen with a box to run queries in.

Once you have Solr up and running, you're ready to take the first steps with spatial data in Lucene. Running the install step will download some sample data generated from the OSM project, which I've set aside at `http://people.apache.org/~gsingers/spatial/`. For this article, I've included sample OSM data from four locations within the United States that are of interest to me (permalinks to OSM are listed in the files):

- Syracuse, N.Y.
- Downtown Minneapolis, Minn.
- Around the Mall of America in Bloomington, Minn.
- Downtown Charleston, S.C.

To demonstrate many of the concepts in this article, I've written code to index the OSM data into Solr, and to associate some simple facts with specific locations (for example, see the `syracuse.facts` file in the data directory.) The goal of this effort is to show how unstructured text and spatial data can be combined to create effective search applications. Note also that I am using the Solr 1.5-dev version (the current development trunk for Solr), not the recently released Solr 1.4.

**Indexing spatial information in Lucene**

Lucene 2.9 added two new features that play an important role in spatial search. First, Lucene has implemented better numeric-range querying and filtering capabilities, which are often used for bounding-box approaches. Second, Lucene has a new contrib module containing the stand-alone project formerly called Local Lucene (see Related topics). (The code lives in contrib/spatial of Lucene; I've included the JAR file with the sample code.) The spatial contrib offers tools for
Location-aware search with Apache Lucene and Solr

Creating Cartesian tiers and Geohash codes, as well as tools for creating Lucene query and filter objects.

Before you look at the code that indexes the data, it is important to assess how you expect to interact with the data and how much data you have to deal with in your application. For instance, for most people with a small to moderate number of documents (say, fewer than 10 million), indexing latitude and longitude and using simple numeric range queries will likely yield sufficient performance, whereas larger applications may need to do more specific things (such as add Cartesian tiers) to reduce the number of terms and documents to be filtered and scored. It is also important to think about which format to store the information in. Many spatial distance algorithms require the numbers to be in radians, while others can work with degrees. Thus, it may be worthwhile to convert your lat/lon values to radians during indexing instead of having to convert them over and over during search time. The trade-off, of course, is more space (disk and, possibly, memory) if you need both formats. Finally, are you going to be faceting, sorting, and scoring on the location features, rather than just use them for filtering? If so, then you may need alternate representations as well.

Because this article is just demonstrating the concepts and is not intended for production use, I will show how to index Geohash, Cartesian tiers, and latitude and longitude all in one place, in some sample Java code. To get started, I've defined a number of values in the Solr schema (located in geospatial-examples/solr/conf/schema.xml) to capture the OSM data. The primary fields for location are shown in Listing 1:

Listing 1. Sample Solr schema

```
<!- Latitude -->
  <field name="lat" type="tdouble" indexed="true" stored="true"/>
<!- Longitude -->
  <field name="lon" type="tdouble" indexed="true" stored="true"/>
<!- lat/lon in radians
  In a real system, use a copy field for these instead of sending over the wire -->
  <field name="lat_rad" type="tdouble" indexed="true" stored="true"/>
  <field name="lon_rad" type="tdouble" indexed="true" stored="true"/>
<!- Hmm, what about a special field type here? -->
  <field name="geohash" type="string" indexed="true" stored="true"/>
<!- Elevation data -->
  <field name="ele" type="tfloat" indexed="true" stored="true"/>
<!- Store Cartesian tier information -->
  <dynamicField name="tier_*" type="double" indexed="true" stored="true"/>
```

Lucene and Solr

Although I am using the Solr schema to demonstrate the fields to index, all concepts here are easily available in Lucene as well. For instance, a `tdouble` is just Lucene 2.9.1 speak for a `NumericField` with a precision step of 8.

I am storing the lat/lon values as `tdouble` fields. A `tdouble` is a `double` represented internally using a Trie structure. It can then be used by Lucene to significantly reduce the number of terms that need to be evaluated during range calculations (bounding box), despite the fact that it actually adds more terms to the index. I'm storing the Geohash as a simple `string` (unanalyzed) because I just want to do exact match on it. Elevation, although strictly not needed for the kinds
of calculations I’m doing, is stored as a `tfloat`, which is just a `float` stored in the Trie structure. Finally, the `tier_*` dynamic field allows the application to add Cartesian tier fields dynamically without needing to predeclare each one. I will leave it to you to explore the other metadata fields captured by the indexing process.

The code responsible for indexing the data is located in the source tree in the sample.zip file. The `Driver` class is a command-line utility for launching the indexing process, and the actual indexing takes place as part of a SAX `ContentHandler` implementation named `OSMHandler`. Within the `OSMHandler` code, the key lines of interest are in the `startElement()` method. I’ll break it out into three sections. The first example, shown in Listing 2, indexes the latitude and longitude as doubles and also converts them to radians for indexing:

**Listing 2. Sample indexing of latitude/longitude**

```java
//... current is a SolrInputDocument
double latitude = Double.parseDouble(attributes.getValue("lat"));
double longitude = Double.parseDouble(attributes.getValue("lon"));
current.addField("lat", latitude);
current.addField("lon", longitude);
current.addField("lat_rad", latitude * TO_RADS);
current.addField("lon_rad", longitude * TO_RADS);
```

Indexing lat/lon is pretty straightforward. Next, I index the Geohash value for the lat/lon pair, as demonstrated in Listing 3:

**Listing 3. Sample indexing of Geohash**

```java
String geoHash = GeoHashUtils.encode(latitude, longitude);
current.addField("geohash", geoHash);
```

In the Geohash code in Listing 3, I am using the `GeoHashUtils.encode()` (there’s an equivalent `decode()` method) method that comes with the Lucene spatial contrib package to convert the lat/lon pair to a single Geohash string, which I then add to the Solr. Finally, to add Cartesian tier support, I did two things in the `OSMHandler` code:

- In the constructor, I set up n instances of the `CartesianTierPlotter` class, one for each tier that I wish to index.
- In the `startElement()` method, I looped over the n plotters and got the identifier for each grid element that contains the latitude and longitude of the current OSM element. The code for this looks like Listing 4:

**Listing 4. Sample indexing of Cartesian tiers**

```java
//... //Cartesian Tiers
int tier = START_TIER; //4 //Create a bunch of tiers, each deeper level has more precision for (CartesianTierPlotter plotter : plotters)
{current.addField("tier_" + tier, plotter.getTierBoxId(latitude, longitude));
tier++;
}
```
Typically, a query need only go against one tier at a time, so having multiple tiers usually does not pose a problem. You should pick the number of tiers you need based on how fine-grained you want to make your search. If you take the time to look through the rest of the indexing code, you will see I added a variety of other metadata values related to the data points in the OSM files. I am currently indexing only two OSM data types: a node and a way. A node is simply a location at a specific latitude and longitude, while a way is a combination of nodes that are all somehow related, such as a street. (See the OSM Data Primitives link in Related topics to learn more about the OSM files.)

What's a CartesianTierPlotter?

The CartesianTierPlotter's job is to take a projection of the Earth (in my case, I use a Sinusoidal projection; see Related topics) and the lat/lon information, convert it down into the grids used by the tier system, and give each grid cell a unique number. During search, an application can then specify the grid IDs to restrict the search to find results in.

Now that you have an understanding of the basics of creating a Solr document that contains spatial information, it just needs to be executed. The Driver class takes in the data and fact files plus the URL where Solr is running, and hands off the work to the OSM2Solr class. The OSM2Solr class then uses Solr's Java client, SolrJ, to take the documents created by the OSMHandler SAX parser and sends them in batches to the Solr server. You can run the Driver class on your own using the command line, or you can simply run ant index and Ant will do the necessary work to run the driver. When you are done, point your browser at http://localhost:8983/solr/select/?q=:*:* and verify that Solr found 68,945 documents. Take a moment to peruse the returned results and familiarize yourself with the content.

I have only scratched the surface of the myriad of ways you could slice and dice the OSM data, but it is time to move on to discuss how to leverage this data in an application.

Searching by location

Now that I have data in the index, it's time to revisit the different ways to use it. Namely, I am going to demonstrate how to sort, boost, and filter documents based on spatial information in the index.

Distance-related calculations

Boosting and sorting documents by distance is a common requirement for many spatial applications. To that end, Lucene and Solr come with several capabilities for calculating distances (see Related topics). Lucene includes tools for sorting and filtering based on the great circle (Haversine) formula (see DistanceUtils and DistanceFieldComparatorSource), while Solr has several FunctionQuery functions for calculating distances, including:

- Great circle (Haversine and Geohash Haversine)
- Euclidean and Squared Euclidean
- Manhattan and other p-norms

Boosting by distance is quite easy using Solr's distance functions. I'll focus on Solr's function queries because they are the easiest to use and require no programming to leverage, but they can easily be used in or ported to Lucene.
As I showed earlier, I have several fields set up to store OSM data, including lat/lon, lat_rad/lon_rad, and geohash. I can then search and boost on these values:

- hsin (great circle): http://localhost:8983/solr/select/?q=name:Minneapolis AND _val_:"recip(hsin(0.78, -1.6, lat_rad, lon_rad, 3963.205), 1, 1, 0)"^100
- dist (Euclidean, Manhattan, p-norm): http://localhost:8983/solr/select/?q=name:Minneapolis AND _val_:"recip(dist(2, lat, lon, 44.794, -93.2696), 1, 1, 0)"^100
- sqedist (Squared Euclidean): http://localhost:8983/solr/select/?q=name:Minneapolis AND _val_:"recip(sqedist(lat, lon, 44.794, -93.2696), 1, 1, 0)"^100
- ghhdist (Geohash Haversine): http://localhost:8983/solr/select/?q=_val_:"recip(ghhsin(geohash(44.79, -93), geohash, 3963.205), 1, 1, 0)"^100

In each of these cases, I’ve combined a keyword query with a distance-based FunctionQuery to produce a result set that factors in both the keyword score and the distance score. To see the effect of each of these parts, add an &debugQuery=true onto each query and take some time to examine the explanations produced by Solr. These are just examples of their usage. To see the full signatures and documentation for these and other FunctionQuery functions, see Related topics. Of course, you may choose to boost one part over the other, or otherwise change the calls to fit your needs.

As for sorting by distance, Solr offers one primary option, which really is a bit of a workaround to the fact that Solr does not yet have sort-by-function capabilities and the fact that no custom FieldType names are defined at this point. However, the workaround is simple. To sort by function, create your query as above, but zero out, via boost, the keyword clause, as in q=name:Minneapolis^0 AND _val_:.... This will cause the keyword score to be zero (but the matching documents will still be returned) and the function value to be the sole component of the score. Longer term, look for Solr to add FieldType names for better support of sorting without needing to zero out the main query.

With the sorting and scoring out of the way, it’s time to move on to filtering.

**Filtering**

To filter by location with Solr, the three primary mechanisms in Table 1 are available to application writers for restricting the document space:

**Table 1. Approaches to filtering**

<table>
<thead>
<tr>
<th>Filter approach</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>Create a range filter that encompasses the lat/lon of the bounding box. For performance reasons, it is important to use the TrieField (NumericField) capabilities of Solr with this approach.</td>
<td><a href="http://localhost:8983/solr/select/?q=*:*&amp;fq=lon:%5B-80">http://localhost:8983/solr/select/?q=*:*&amp;fq=lon:[-80</a> TO -78]&amp;fq=lat:[31 TO 33]</td>
</tr>
<tr>
<td>Cartesian tier</td>
<td>Given a lat/lon and a distance, identify the grid cells surrounding the center point and restrict to only those documents containing those cells. See What’s a QParserPlugin? for more details on the source implementation of this.</td>
<td><a href="http://localhost:8983/solr/select/?q=*:*&amp;fq=%7B">http://localhost:8983/solr/select/?q=*:*&amp;fq={</a>! tier x=32 y=-79 dist=50 prefix=tier_}</td>
</tr>
</tbody>
</table>
Distance | Using Solr’s frange (function range) QParserPlugin capability and a distance function (more in Distance-related calculations, above), determine the distance between the points and restrict the document space. | http://localhost:8983/solr/ select/?q=*:*&fq={!frange l=0 u=400}hsin(0.57, -1.3, lat_rad, lon_rad, 3963.205)

A few words on density

The density of data in a specific range often plays an important role in the user search experience. For example, an application providing business search for Manhattan in New York, N.Y. is much denser than one providing results for Buffalo, Minn. (see Related topics). In fact, it can be useful to incorporate this information into your filtering function such that the application picks an effective distance to ensure good results. Unfortunately, demonstrating how to do this is beyond this article's scope.

Which approach is right for you? It will depend on the density of your points (see A few words on density), but a good recommendation is to start off with the simple range approach and then move up to the tier approach if need be. The key factor is to look at the number of terms that need to be evaluated at any one time when calculating the range, because that number is what directly controls how much work Lucene must do in order to restrict the result set.

A simple geonames.org query parser

Building a full-fledged query parser for spatial is beyond the scope of this article. Instead, I'll build a simple QParserPlugin that will handle the chores of obtaining results from location information from GeoNames. This parser assumes that an application can split up the user input ahead of time into two parts: the keyword query and the spatial query. In fact, many local search applications ask the user for input in just this way via two input boxes.

What's a QParserPlugin?

A QParserPlugin is Solr-speak for a query parser plug-in module. Just as many pieces of Solr are pluggable, so to are query parser implementations. For this article, I am using three different query parser plug-ins, one that comes with Solr — FunctionRangeQParserPlugin ({!frange}) — and two I wrote: CartesianTierQParserPlugin ({!tier}) and the GeonamesQParserPlugin. The source for the two plug-ins is located under the src tree in the sample download. Both plug-ins are already configured in Solr via the solrconfig.xml file: QParserPlugins are invoked in the query by specifying {!parserName [parameters]}[query], (parameters and query may be optional), as in {!tier x=32 y=-79 dist=50 prefix=tier_} and {!frange l=0 u=400}hsin(0.57, -1.3, lat_rad, lon_rad, 3963.205).

The parser can take several parameters:

- topo: Short for toponym (see the GeoNames documentation). The location to search for in GeoNames. Required.
- rows: The number of rows to get back from GeoNames. Optional. Default is 1.
- start: The result to start on. Optional. Default is 0.
- lat: The latitude field name to use as a ValueSource for the FunctionQuery. If specified, lon must also be set.
- lon: The longitude field name to use as a ValueSource for the FunctionQuery. If specified, lat must also be set.
• `gh`: The Geohash field name to use as a ValueSource for the FunctionQuery. If specified, `lat/lon` must not be set.
• `dist`: The distance function to use. String. One of `[hsin, 0-Integer.MAX_VALUE, ghhsin]`. If a geohash field is specified, then this field is disregarded and `ghhsin` is automatic. Default is 2 for the 2-norm (Euclidean).
• `unit`: `KM|M`: The units to use, `KM` for metric, `M` for English. Default is `M`.
• `boost`: float: The amount to boost the function query by. Default is 1.

The code for this example is in `GeonamesQParserPlugin.java` in the sample download. (The Solr server is already configured in the Solr version included in the download.) Invoking it is much like invoking the `CartesianTierQParserPlugin` outlined above. For instance, to search for malls in the index relative to Bloomington, Minn., I did `http://localhost:8983/solr/select/?q=text:mall AND _query_:{!geo topo='Bloomington, MN' lat=lat_rad lon=lon_rad dist=hsin}"`. By taking the `QParserPlugin` approach, I was able to focus in on handling just the specific syntax that was important to me while location-wise still allowing all the text-based query parsing capabilities to proceed as usual.

From here, the `GeonamesQParserPlugin` could be expanded significantly to work with postal codes and many other location specifications. Naturally, it also needs more error handling and likely needs to be converted to use the GeoNames dataset (see Related topics) so that it doesn't rely on Web services calls. Solr also has an open issue in its issue tracker for adding spatial query parser support (see Related topics).

**Where to next?**

I've demonstrated Lucene and Solr's capabilities for searching, sorting, and filtering text documents based on a point-based model of location. From here, a real local-search application will need to investigate how best to scale, handle user queries, and visualize the results. For scaling, some of the question is answered by how many terms need to be enumerated when the bounding-box filter is created. Beyond the filter question for scaling, the usual search-related factors will apply, such as whether to distribute the index or simply replicate. See the Lucene and Solr Related topics.

If you are interested in building out a more significant GIS application, you will need to add much more sophisticated capabilities for route finding, shape intersection, and a great deal more. But if you need to build out a solid search application that combines the structure of point-based locations with unstructured text, then look no further than Lucene and Solr.

**Acknowledgments**

Special thanks to fellow Lucene/Solr committers Ryan McKinley and Yonik Seeley for their insights and review of this article.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial examples</td>
<td>j-spatial.zip</td>
<td>52.4 MB</td>
</tr>
</tbody>
</table>
Related topics

- Lucene and Solr:
  - **Apache Lucene**: Learn more about Lucene.
  - **Apache Lucene on developerWorks**: Explore Lucene in these articles.
  - *Lucene In Action*, 2nd ed. (Erik Hatcher, Otis Gospodnetic, and Michael McCandless, Manning, 2008): Read this book to learn all the details of Lucene.
  - **Apache Solr**: Learn more about Solr.
  - **Apache Solr on developerWorks**: Explore Solr in these articles.
  - **FunctionQuery in the Solr Wiki**: See all of Solr's FunctionQuery capabilities.
  - **SOLR-1578**: Learn about Solr's efforts to add a spatial query parser.

- Geography and GIS:
  - **Geographic information system**: Visit this Wikipedia page for some starting information on GIS.
  - **Spatial reference (Map projection), Geodetic system, and World Geodetic System**: Check out the Wikipedia articles on these topics.
  - **Latitude** and **Longitude**: Read Wikipedia's explanations of latitude and longitude.
  - **Geographical distance**: Learn more from Wikipedia about calculating geographic distances, including the great-circle distance calculation.
  - **Geohash**: Wikipedia explains what a Geohash is and how it can be used.
  - **OpenStreetMap**: Visit the OSM site and get more spatial data to try out, check out the OSM Wiki’s Map Features list, and learn about the OSM Data Primitives.
  - **Minimum bounding box**: Read about bounding boxes on Wikipedia.
  - **The World Factbook**: Browse the CIA World Factbook.
  - **Sinusoidal projection**: Wikipedia explains what this type of map projection is used for.

- \(L^p\) space and p-norms: Check Wikipedia's page on these mathematical concepts.

- Public geocoding services:
  - **Google Maps**
  - **GeoNames**
  - **Java client for GeoNames Webservices**: Download a Java client for accessing the geonames.org Web services API.
  - **GeoNames data**: Download the full GeoNames dataset.
  - "Over the River and Through the Woods": Learn all the lyrics of Lydia Maria Child's song.

- Visit (virtually):
  - **Buffalo, Minn.**
  - **New York, N.Y.**