Spatial Solr Documentation

Version 1.0-RC2

Introduction iii
1. Searching with Spatial Lucene 1
    1.1. Using the Spatial Filter 1
    1.2. Filtering with the CartesianShapeFilter 2
        1.2.1. Indexing Cartesian Tier Information 2
        1.2.2. Creating & Using CartesianShapeFilter 3
    1.3. Distance Filtering 3
        1.3.1. Multi-Threaded Distance Filtering 4
    1.4. Retrieving the location of a Document 5
    1.5. Calculating distances with the GeoDistanceCalculator 5
    1.6. Retrieving the Calculated Distances 6
2. Solr enabled Spatial Search 7
    2.1. Indexing documents with the *SpatialTierUpdateProcessor* 7
    2.2. Searching for documents with the *SpatialTierQueryParser* 8
        2.2.1. Query Syntax 8
        2.2.2. Configuration 8
    2.3. Adding the calculated distances to the results 9
        2.3.1. Retrieving the Calculated Distances 10
        2.3.2. *GeoDistanceComponent* 10
A. Example Configuration 11
    A.1. solrconfig.xml 11
    A.2. schema.xml 12

Introduction

Spatial search has come to the fore of search technologies in the last few years. Through Google maps we have seen how useful it can be to find pizza stores near our house. Of course, this functionality has until recently, been most found in closed source proprietary search applications. But in the last 12 months there has been considerable work done on bringing spatial search to Apache Lucene and Solr.

Currently Lucene has a contrib called Spatial Lucene which supports constructing a Lucene Filter which will filter out those documents which are outside of a radius of a user specified point. It does this by applying two levels of filtering, the first using Cartesian Tier information, and the second by calculating the actual distance between documents and the point.

Solr does not currently have committed support for spatial search, however in SOLR-773 there has been considerable work done on integrating Solr with the Spatial Lucene contrib. Existing patches support adding cartesian tier information to documents, searching for documents through a custom query format, and adding the calculated distances to the results.

Unfortunately both code bases suffer from many problems, leaving them short of enterprise level. While these problems will be addressed, it will take some time. For the mean time, to fill this void, the Spatial Solr plugin will serve as a standalone plugin that will bring enterprise level spatial search to Apache Solr and Lucene.

Chapter 1. Searching with Spatial Lucene

Inside the plugin there are two separate packages, one providing spatial search support for Lucene, and one for Solr. The package for Lucene is based on the existing Spatial Lucene contrib. Consequently we will refer to this as Spatial Lucene in the remainder of this document. This chapter will describe in detail the main components of Spatial Lucene.

1.1. Using the Spatial Filter

At the heart of Spatial Lucene is the `SpatialFilter`, a proper Lucene `Filter` [1]. `SpatialFilter` can be seen as the public API of Spatial Lucene. It manages all the components described below, and for most users is the only class that needs to be included in their code.

`SpatialFilter` has a number of different constructors intended for different uses depending on whether you want to use geohashes or traditional latitude/longitude fields and whether you want multi-threaded distance filtering or not. Rather than going through each constructor, lets examine the parameters:

- **lat (double) / lng (double)** - These arguments are present in all the constructors and define the latitude and longitude of the centre of the search
- **radius (double)** - Also present in all the constructors, this is the radius documents must be within from the centre of the search, in order to not be filtered out
- **unit (DistanceUnit)** - Spatial Lucene supports calculating distances in multiple units, though currently only miles and kilometres are supported. This argument defines which should be used.
- **tierFieldPrefix (String)** - As described below, the `CartesianShapeFilter` reads values from fields with a tier level appended to the prefix. For maximum flexibility, the SpatialFilter allows you to specify whatever prefix you have chosen when indexing your documents
- **latField (String) / lngField (String)** - If you have indexed the latitude/longitude of your documents as separate fields, then you must specify the field names through these arguments.
- **geoHashFieldPrefix (String)** - If you have indexed the latitude/longitude of your documents as a single geohash field, then you must specify the field name through this argument.
- **distanceCalculator (GeoDistanceCalculator)** - Spatial Lucene provides multiple ways of calculating the distance between points (see below). You must provide the implementation you chose through this argument.
- **executorService (ExecutorService)** - If you have chosen to use multi-threaded distance filtering (see below), then you must provide your own ExecutorService which you are also responsible for managing. The distance filter will consume the number of threads provided through the threadCount argument.
- **threadCount (int)** - Complements the *executorService* parameter decribed above. This parameter defines the number of thread consumed by the distance filter.

[1] Lucene Filter [http://lucene.apache.org/java/2_9_0/api/core/org/apache/lucene/search/Filter.html]

- **query (Query)** - To improve the performance of Spatial Lucene by minimising the number of documents that must be examined, the Spatial Filter executes the Query that the filter is to be associated with, as part of its filtering (see discussion below).

Despite its name, `SpatialFilter` does not do any of the filtering itself, instead it delegates it to other filters. Because calculating the distance between two points is so expensive, the overall goal of the filter is to reduce the number of documents whose distance from the centre of the search, must be calculated. It does this by first executing the Query that the Filter is associated with. This means that only those documents found by the Query are to be considered by the remainder of the filtering process. Those remaining documents are then passed into the second filtering step, the `CartesianShapeFilter`.

Example

```
ExecutorService executorService = ....
IndexReader indexReader = ....
Query query = new TermQuery(new Term("title", "pizza");

DistanceCalculator distanceCalculator = new ArcGeoDistanceCalculator();
SpatialFilter spatialFilter =
    new SpatialFilter(4.56, 54.32, DistanceUnit.KILOMETRES, "lat", "lng", "cartesianTier", distanceCalculator, query);

DocIdSet results = spatialFilter.getDocIdSet(indexReader);
System.out.println("Results");
DocIdSetIterator iterator = docIdSet.iterator();
int docId = 0;
while ((docId = iterator.nextDoc()) != DocIdSetIterator.NO_MORE_DOCS) {
    Document document = indexReader.document(docId);
    System.out.println(document);
}
```

1.2. Filtering with the CartesianShapeFilter

The `CartesianShapeFilter` is a highly efficient Lucene Filter that can quickly filter out documents that are outside of a simplified version of the search area. Much of the information about how the Filter works can be found at http://www.nsshutdown.com/projects/lucene/whitepaper/locallucene_v2.html, but put simply, the Cartesian Tier filtering involves adding information to each document about what grid squares the document is when its plotted on a grid that covers the whole planet, and then finding those documents that are in the grid squares that the search area overlaps. This is very efficient since the adding of the grid information can be done once when the document is indexed, and finding those documents which are in certain grid locations can be done by basic Lucene `TermQuery`s

1.2.1. Indexing Cartesian Tier Information

To add the grid information to your documents you must use the `CartesianTierPlotter`, which has the following constructor: `CartesianTierPlotter(int tierLevel, Projector projector)`. Here you are able to specify what is referred to as the *tierLevel*, and the implementation of the `Projector` that will project the document locations onto the grid. Currently Spatial Lucene provides only one implementation of Projector - *SinusoidalProjector*.

The *tierLevel* is a little more complicated. If the size of the grid squares created by the `CartesianTierPlotter` were very small, and your search area was very large, then the `CartesianShapeFilter` would have to be search for documents in lots of different grid locations. At the same time, if the squares were very large and your search areas was small, then the `CartesianShapeFilter` would not filter out many of the documents that are in the search area. Therefore the Cartesian tier filtering process incorporates each document being plotted on multiple grids referred to as tiers, each with its own grid square size. This means when indexing your documents you will want to use the information calculated by multiple `CartesianTierPlotters` (the recommended levels are from 9 – 17).

To retrieve the ID of the grid square that your document is in at a certain level, you must call `CartesianTierPlotter.getTierBoxId(double latitude, double longitude)`, which accepts the latitude and longitude of the document.

In order for the `CartesianShapeFilter` to be able to find those documents that are in certain grid squares at a specific level, the grid square Ids must be indexed in different fields. To simply the process, the Filter assumes the names of the fields are `prefixTierLevel`, where prefix is a consistent value you choose when indexing your documents.

Example

The following example illustrates a way to add the cartesian tier information to documents:

```
List<Document> documents = ...
Projector projector = new SinusoidalProjector();
List<CartesianTierPlotter> plotters = new ArrayList<CartesianTierPlotter>();
for (int i = 9; i <= 17; i++) {
    plotters.add(new CartesianTierPlotter(i, projector));
}

for (Document document : documents) {
    for (CartesianTierPlotter plotter : plotters) {
        double boxId = NumericUtils.doubleToPrefixCoded(plotter.getTierBoxId(54.32, 4.56));
        Field field = new Field("cartesianTier" + plotter.getTierLevelId(),
              String.valueOf(boxId), Store.NO, Index.NOT_ANALYZED);
      document.add(field);
    }
}
```

1.2.2. Creating & Using CartesianShapeFilter

Having indexed your documents, the `CartesianShapeFilter` will not be able to filter out those outside of the search area. Most users will never have to worry about how to construct instances of `CartesianShapeFilter` since the `SpatialFilter` does it for you, however if you need to then you must use the `CartesianShapeFilterBuilder`, which provides a very simple API.

The constructor `CartesianShapeFilterBuilder(String tierPrefix)` simply takes in the prefix you chose when indexing your documents. Then the method `Filter buildFilter(double latitude, double longitude, double miles)` which accepts the latitude/longitude of the centre of your search area, and the radius of the search area, will return a configured instance of `CartesianShapeFilter`.

Example

The following example illustrates how to construct and use your own instance of `CartesianShapeFilter`:

```
IndexReader indexReader = ....
CartesianShapeFilter filter = new CartesianShapeFilterBuilder("cartesianTier").buildFilter(4.56, 54.32, 30);
DocIdSet docIdSet = filter.getDocIdSet(indexReader);
```

1.3. Distance Filtering

Having executed the Query and used the `CartesianShapeFilter` to reduce the number of documents whose distance from the centre of the search area needs to be calculated, the `SpatialFilter` then delegates to the configured instance of `DistanceFilter`.

The `DistanceFilter` is not a proper Lucene Filter in that it takes in a `BitSet` indicating which documents it should calculate the distances for. This is necessary since calculating the distances for even 1 million documents can take between 1-2 seconds. The role of the `DistanceFilter` is to calculate to differing degrees of accuracy (see the discussion about the `GeoDistanceCalculator` below), the distances from each document set in the `BitSet`, to the centre of the search area. If the distance for a document is greater than the search radius, then the document is filtered out.

Spatial Lucene provides two implementations of `DistanceFilter`, `NoOpDistanceFilter` which does no filtering at all and can be used when the `CartesianShapeFilter` is sufficient, and the *ThreadedDistanceFilter* which will be examined in the remainder of this section

1.3.1. Multi-Threaded Distance Filtering

The `ThreadedDistanceFilter` calculates the distances for documents in multiple threads. By doing so, it greatly reduces the time taken to complete the filtering process. Like the `CartesianShapeFilter`, most users will not have to instantiate their own instances of the `ThreadedDistanceFilter` since the `ShapeFilter` will do it for them, however its worth while looking at its constructor:

```
public ThreadedDistanceFilter(
        double lat,
        double lng,
        double radius,
        DistanceUnit unit,
        LocationDataSetFactory dataSetFactory,
        GeoDistanceCalculator distanceCalculator,
        ExecutorService executorService,
        int threadCount) {
     ...
}
```

Here the latitude/longitude of the centre of the search area, plus the search radius, are provided as normal. Additionally the `DistanceUnit` defining which unit the distances should be calculated in is also specified, along with the `LocationDataSetFactory` which will be used to retrieve the latitude/longitude information for the documents (see below) and the `GeoDistanceCalculator` that will be used to calculate the distances between points (also see below). Most importantly however is the *executorService* and *threadCount* arguments.

Through the `ExecutorService` and *threadCount* it is possible to control how threaded the `ThreadedDistanceFilter` is. The `ThreadedDistanceFilter` partitions the `BitSet` indicating which documents need to have their distance calculated, into *threadCount* number of partitions. Each partition is then given to the distance calculating code as a task that is executed by the `ExecutorService`. Therefore by choosing an appropriate *threadCount* value, you can finely tune how much threading is necessary to achieve the kind of performance you need. Equally by providing an appropriate `ExecutorService`, you can control how the threading interacts with the other threading in your application.

While the `SpatialFilter` does not choose the values for these arguments, the following are rules of thumb for the *threadCount* and `ExecutorService`:

- *threadCount* should probably not be larger than the number of physical processors you have available (including cores)

- ExecutorService: `ThreadPoolExecutor` implementation which *corePoolSize* = *threadCount* and *maxPoolSize* = *corePoolSize* * (number of intended concurrent requests)

Example

The following example illustrates how to create and use an instance of `ThreadedDistanceFilter`:

```
IndexReader indexReader = ....
ExecutorService executorService = new ThreadPoolExecutor(2, 10, 10, TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>())
LocationDataSetFactory factory = ....

DistanceCalculator distanceCalc = new ArcGeoDistanceCalculator()
ThreadedDistanceFilter distanceFilter = new ThreadedDistanceFilter(4.56, 54.32, 30, DistanceUnit.KILOMETRES, factory,
        distanceCalc, executorService, 2);

BitSet bitSet = ...;

BitSet results = distanceFilter.bits(indexReader, bitSet);
System.out.println(results.cardinality() + " documents passed the distance filter");
```

1.4. Retrieving the location of a Document

Spatial Lucene does not place any requirements on how the latitude/longitude information for documents should be stored. Instead, it abstracts this away with the interfaces `LocationDataSet` / `LocationDataSetFactory`.

The `LocationDataSet` interface defines a simple API for retrieving the latitude/longitude information for a document:

```
Point getPoint(int docId)
```

Here, Point is a wrapper class for an x and y coordinate (which map to latitude and longitude) and *docId* is the index ID of the document.

`LocationDataSetFactory`, which is responsible for creating instances of `LocationDataSet`, also has a simple API with just one method:

```
LocationDataSet buildLocationDataSet(IndexReader indexReader) throws IOException
```

This means that a `LocationDataSet` instance relates to a specific `IndexReader`.

Spatial Lucene provides two implementations of the above interfaces:

- `LatLongLocationDataSet/Factory` – Implementation that assumes the latitude/longitude information are stored in two distinct fields within documents. The values are read from the Lucene `FieldCache` as doubles.

- `GeoHashLocationDataSet/Factory` – Implementation that assumes the latitude/longitude information is stored in a single field encoded using the Spatial Lucene `GeoHashUtil` class. The encoded Strings are read from the Lucene `FieldCache StringIndex`.

1.5. Calculating distances with the GeoDistanceCalculator

To calculate the distances from each document to the centre of the search area, the `ThreadedDistanceCalculator` uses the provided instance of `GeoDistanceCalculator`. `GeoDistanceCalculator` is a simple interface which abstracts away the mathematics behind calculating distances, and allows users to choose one that fits their needs in terms of accuracy and efficiency. It has a simple API with only 1 method:

```
double calculate(
        double sourceLongitude,
        double sourceLatitude,
        double targetLongitude,
        double targetLatitude,
        DistanceUnit unit)
```

Here the longitude/latitude of the two points are provided, along with the `DistanceUnit` defining the unit that the calculated distance should be in.

Spatial Lucene currently has two implementations of `GeoDistanceCalculator`:

- `PlaneGeoDistanceCalculator` – Implementation that assumes the points are on a flat plane and uses basic trigonometry. Very efficient but with an error of around 7%

- `ArcGeoDistanceCalculator` – Implementation that calculates the great circle distance between the two points. Slow but very accurate.

If neither of these implementations suite your needs then it is possible to create your own and provide it as an argument to the `SpatialFilter` or `ThreadedDistanceFilter`.

1.6. Retrieving the Calculated Distances

In addition to providing a `DocIdSet` which contains only those documents that are in the search area, the `SpatialFilter` allows users to also access the distances calculated for those documents (if any where). They can be retrieved by first retrieving the `DistanceFilter` through `SpatialFilter.getDistanceFilter()`. Then, from the `DistanceFilter`, you can either retrieve a map of distances by document ID through `DistanceFilter.getDistances()`, or retrieve the distance for a specific document through `DistanceFilter.getDistances(int docId)`.

Example

The following example illustrates how to retrieve the calculated distances for the resulting documents from the `SpatialFilter`:

```
IndexReader indexReader = ....
SpatialFilter filter = ....

DistanceFilter distanceFilter = filter.getDistanceFilter();
DocIdSetIterator iterator = docIdSet.iterator();
int docId = 0;
while ((docId = iterator.nextDoc()) != DocIdSetIterator.NO_MORE_DOCS) {
    System.out.println("Distance from Map " + distanceFilter.getDistances().get(docId));
    System.out.println("Distance retrieved directly " + distanceFilter.getDistance(docId));
}
```

Chapter 2. Solr enabled Spatial Search

At the time of writing, there is no officially committed support for integrating Solr with Spatial Lucene. Hence the Spatial Solr Plugin provides its own support, developed from the ideas introduced in *SOLR-773* [1]. Unlike the patches in *SOLR-773*, the support provided by this plug-in is completely separated from the core Solr code, and integrates at the extension points provided by Solr. It is designed to be as non-intrusive as possible and uses standard Solr concepts and features.

The only compromise that had to be made in the design was to create a `GeoDistanceComponent` (discussed below) that adds the calculated distances to the search results. Ideally it would be not necessary to have a component to just do this, however for the plug-in to remain fully separated, it was necessary to create such a component.

For the purpose of this document, the support for Solr enabled Spatial Search will be referred to as Spatial Solr.

2.1. Indexing documents with the *SpatialTierUpdateProcessor*

Previously it was shown that in order to use the `CartesianShapeFilter`, cartesian tier information needed to be added to documents when they are indexed. Spatial Solr provides an `UpdateProcessor`[2], `SpatialTierUpdateProcessor`, which hooks into the `UpdateProcessorChain` and automatically adds the cartesian tier information to any document that is indexed. The following example illustrates what needs to be added to your solrconfig.xml to configure the `UpdateProcessor`:

```
<updateRequestProcessorChain>
    <processor class="nl.jteam.search.solrext.spatial.SpatialTierUpdateProcessorFactory">
        <str name="latField">lat</str>
        <str name="lngField">lng</str>
        <int name="startTier">9</int>
        <int name="endTier">17</int>
        <str name="tierPrefix">_tier_</str>
    </processor>
    <processor class="solr.LogUpdateProcessorFactory"/>
    <processor class="solr.RunUpdateProcessorFactory"/>
 </updateRequestProcessorChain>
```

Here it is defined that the fields that the latitude and longitude information in the documents can be found in the *lat* and *lng* fields respectively (geohashes is not currently supported by Spatial Solr). It is also defined that the range of tier levels should be 9 – 17.

The following table summarizes the configuration options for the `UpdateProcessor`:

Table 2.1. `SpatialTierUpdateProcessorFactory` Configuration

Param Name	Meaning	Required	Default Value
latField	Name of the field in the documents where the latitude information can be found	No	lat
lngField	Name of the field in the documents where the longitude information can be found	No	lng

[1]SOLR-773 [http://issues.apache.org/jira/browse/SOLR-773]
[2]More information about update processors can be found at http://wiki.apache.org/solr/UpdateRequestProcessor

startTier	ID of the first Cartesian tier that the documents should be plotted on	Yes	
endTier	ID of the last Cartesian tier that the documents should be plotted on	Yes	
tierPrefix	Name that should be used as the prefix for the fields that the Cartesian tier information will be indexed in	No	_tier_

2.2. Searching for documents with the *SpatialTierQueryParser*

Having indexed your documents with the Cartesian tier information, you can now query them out by using the `SpatialTierQueryParser`. Before examing how to configure the `QueryParser`, lets first look at the query syntax it provides.

2.2.1. Query Syntax

The following is an example of how to query for documents using Spatial Solr:

```
q={!spatial lat=4.32 long=54.32 radius=30 unit=km calc=arc threadCount=2}title:pizza
```

This will result in the a query for documents that have the value “pizza” in the field “title”, that are within 30km of latitude 4.32 longitude 54.32.

To allow the user to control the filtering process, the `QueryParser` allows most of the configuration values to provided in the local params of the query string. The following table summaries the params supported:

Table 2.2. Supported query parameters

Param Name	Meaning	Required	Default Value
lat	Latitude of the centre of the search area	Yes	
long	Longitude of the centre of the search area	Yes	
radius	Radius of the search area	Yes	
unit	Unit the distances should be calulcated in. Currently only *km* and *miles* are supported	No	miles
calc	`GeoDistanceCalculator` that will be used to calculate the distances. Currently arc for `ArchGeoDistanceCalculator` and plane for `PlaneGeoDistanceCalculator` are supported	No	arc
threadCount	Number of threads that will be used by the `ThreadedDistanceFilter`	No	1

2.2.2. Configuration

The following is an example of what to add to your *solrconfig.xml* to configure the `SpatialTierQueryParserPlugin`:

```
<queryParser name="spatial" class="nl.jteam.search.solrext.spatial.SpatialTierQueryParserPlugin"
        basedOn="dismax">
    <str name="corePoolSize">1</str>
    <str name="maxPoolSize">2</str>
    <str name="keepAlive">60</str>
    <str name="latField">lat</str>
    <str name="lngField">lng</str>
    <str name="tierPrefix">_tier_</str>
</queryParser>
```

Included in this configuration are two sets of values. The parameters *corePoolSize*, *maxPoolSize* and *keepAlive* are used by the `QueryParser` to instantiate a `ThreadPoolExecutor`, which will be used in all searches. The parameters *latField*, *lngField* and *tierPrefix* are the same as used in the `SpatialTierUpdateProcessor` and define where the latitude, longitude and cartesian tier information can be found in documents.

The following table summarizes the configuration options:

Table 2.3. `SpatialTierQueryParserPlugin` Configuration parameters

Param Name	Meaning	Required	Default Value
corePoolSize	Core pool size of the `ThreadPoolExecutor` created by the `QueryParser`	Yes	
maxPoolSize	Max pool size of the `ThreadPoolExecutor` created by the `QueryParser`	Yes	
keepAlive	Keep alive, in seconds, for the `ThreadPoolExecutor` created by the `QueryParser`	Yes	
latField	Name of the field in the documents where the latitude information can be found	No	lat
lngField	Name of the field in the documents where the longitude information can be found	No	lng
tierPrefix	Prefix of the fields that contain the Cartesian tier information	No	_tier_

In addition to the above configuration options, it is possible to define the name of another `QueryParser` through the *basedOn* attribute. This is the name of the `QueryParser` that the `SpatialTierQueryParser` will delegate to parse the user actual query, which in the previous example is title:pizza. In the above configuration this is set to *dismax* which will mean the `DisMaxQParser` will be used to parse the query.

Tip

To learn more about query syntax and parsers in Solr visit the following URL's:

- http://wiki.apache.org/solr/SolrQuerySyntax
- http://wiki.apache.org/solr/SolrPlugins#QParserPlugin

2.3. Adding the calculated distances to the results

While finding those documents that are within the radius of a certain point is useful, having the calculated distances included in the search results is of key importance in Spatial Solr. Ideally it would be possible to add arbitrary information to search results, however Solr currently only supports adding such information to the search response, which goes against Spatial Solr's goal of being non-intrusive and keeping to Solr concepts. Therefore in order to add the information to the search results, Spatial Solr includes the `GeoDistanceComponent` which is a Solr SearchComponent.

2.3.1. Retrieving the Calculated Distances

Of course before the calculated distances can be included in the search results, they must first be collected, which is the role of the `FieldValueSourceRegistry`, `FieldValueSource` and the `DistanceFieldValueSource`. `FieldValueSource` is an interface that defines the source for arbitrary information that can be added to search results. Instances of implementations of the interface are registered in the `FieldValueSourceRegistry`, which can be used anywhere where a `SolrQueryRequest` is available. When the search results are then included in the `SolrQueryResponse`, the values from the `FieldValueSources` in the registry are read and added to the results.

Currently the only implementation of `FieldValueSource` is `DistanceFieldValueSource` which is created and registered by the `SpatialTierQueryParser` at query time. The `DistanceFieldValueSource` retrieves and makes available the distances calculated by the `SpatialFilter` created for the query.

2.3.2. *GeoDistanceComponent*

With the distances now available through the `DistanceFieldValueSource`, the `GeoDistanceComponent` is now able to include the calculated distance for each result. To use the `GeoDistanceComponent`, it must be configured as follows in your solrconfig.xml:

```
<searchComponent name="geodistance" class="nl.jteam.search.solrext.spatial.GeoDistanceComponent">
    <defaults>
        <str name="distanceField">distance</str>
    </defaults>
</searchComponent>
```

Here the only configuration value is the name of the field that the distances should be added under in the search results. This field can also be referenced in the fl to control whether or not the distances should be included in results, just like another other field.

Note

Because the `GeoDistanceComponent` uses the distances which are calculated at query time, it must be configured to execute after the `QueryComponent` (or whatever component executes your query) in your request handler.

Appendix A. Example Configuration

A.1. solrconfig.xml

```xml
<config>

    ...

    <requestHandler name="standard" class="solr.SearchHandler" default="true">
        <lst name="defaults">
            <str name="echoParams">explicit</str>
        </lst>
        <arr name="components">
            <str>query</str>
        </arr>
    </requestHandler>

    <requestHandler name="/update" class="solr.XmlUpdateRequestHandler"/>

    <updateRequestProcessorChain>
        <processor class="nl.jteam.search.solrext.spatial.SpatialTierUpdateProcessorFactory">
            <str name="latField">lat</str>
            <str name="lngField">lng</str>
            <int name="startTier">9</int>
            <int name="endTier">17</int>
        </processor>
        <processor class="solr.LogUpdateProcessorFactory"/>
        <processor class="solr.RunUpdateProcessorFactory"/>
    </updateRequestProcessorChain>

    <queryParser
        name="spatial"
        class="nl.jteam.search.solrext.spatial.SpatialTierQueryParserPlugin">
        <str name="corePoolSize">1</str>
        <str name="maxPoolSize">2</str>
        <str name="keepAlive">60</str>
    </queryParser>

    ...

</config>
```

A.2. schema.xml

```
<schema name="spatial" version="1.1">

    <types>

        ...

        <fieldType name="string" class="solr.StrField" sortMissingLast="true" omitNorms="true"/>
        <fieldType name="double" class="solr.TrieDoubleField" precisionStep="0" omitNorms="true"
                positionIncrementGap="0"/>

    </types>

    <fields>

        ...

        <field name="lat" type="double" indexed="true" stored="true" required="false" multiValued="false"/>
        <field name="lng" type="double" indexed="true" stored="true" required="false" multiValued="false"/>

        <dynamicField name="_tier_*" type="string" indexed="true" stored="true"/>

    </fields>

    ...

</schema>
```