Lab 7. Advanced Queries -- Part II

We started understanding the concept of relevance and its terms precision and recall in the previous lab. Then we learned about various query parsers, their parameters, and how we can configure them. In the same way, we explored various response writers, their parameters, and how we can configure them. We also looked at velocity search UI. Then we learned about various faceting parameters and faceting types, such as range faceting, pivot faceting, and interval faceting. At the end, we saw the Solr highlighting mechanism, parameters, various highlighters, and boundary scanners.

In this lab, we will learn about more search functionalities such as spellchecking, suggester, pagination, result grouping and clustering, and spatial search. Let's start with the spellchecking feature of Solr. Spellchecking

---

We have seen that Solr provides magical support for searching. Solr provides a strong index building mechanism, unifiable search configurations, and providing interesting and expected formatted results by executing various transformation steps on the query output. Spellchecking is an advantageous feature provided by Solr for those who make mistakes while typing a query or may enter an incorrect or inappropriate input. Sometimes, we have this experience while searching on Google. If we enter `sokcer`, then Google provides a hint: **`Did you mean: soccer?`** Or sometimes, typing `socer` will directly show results for soccer rather than displaying any hints.

Likewise, there are some scenarios where we need to be careful about the input word:

- If a user enters input search terms with incorrect spelling and there is no matching document available, we use the Solr spellcheck feature, displaying a message that searching for `soccer` instead of `socer` will give the user a hassle-free experience of searching without worrying much about the spelling.
- A user enters less terms for search which is not sufficient to fetch more or sufficient matching documents at that time if any suggestion terms available which contains more matching documents then we can instruct the user by giving a message like `Did you mean xxxxx`. But if the suggestion terms have the same or lesser-matching documents than the query terms, then no message should be shown.
- When no index is available for the entered search terms, no suggestions should be given to the user.

To take advantage of the Solr spellchecking feature, we need to tell the request handle to check spelling during processing. Here is the configuration of the Solr default request handler to enable spellchecking while processing a request:

```
<requestHandler name="/select" class="solr.SearchHandler">
 <lst name="defaults">
 <str name="echoParams">explicit</str>
 <int name="rows">10</int>
 <str name="spellcheck.dictionary">default</str>
 <str name="spellcheck.dictionary">wordbreak</str>
 <str name="spellcheck">on</str>
 <str name="spellcheck.extendedResults">true</str>
 <str name="spellcheck.count">10</str>
 <str name="spellcheck.alternativeTermCount">5</str>
 <str name="spellcheck.maxResultsForSuggest">5</str>
 <str name="spellcheck.collate">true</str>
 <str name="spellcheck.collateExtendedResults">true</str>
 <str name="spellcheck.maxCollationTries">10</str>
 <str name="spellcheck.maxCollations">5</str>
 </lst>
 <arr name="last-components">
 <str>spellcheck</str>
 </arr>
</requestHandler>
```

The preceding configuration is sufficient for enabling spellchecking and performs spellchecking for all queries processed through this request handler, for example, searching for `cemera` instead of `camera`.

URL: `http://localhost:8983/solr/techproducts/select?q=cemera`:

```
{
 "responseHeader":{
 "status":0,
 "QTime":7,
 "params":{
 "q":"cemera"}},
 "response":{"numFound":0,"start":0,"docs":[]
 },
 "spellcheck":{
```

```
 "suggestions":[
 "cemera",{
 "numFound":1,
 "startOffset":0,
 "endOffset":6,
 "origFreq":0,
 "suggestion":[{
 "word":"camera",
 "freq":1}]}],
 "correctlySpelled":false,
 "collations":[
 "collation",{
 "collationQuery":"camera",
 "hits":1,
 "misspellingsAndCorrections":[
 "cemera","camera"]}]}
}
```


From the preceding response, we can see that Solr returns the spellcheck container, along with suggested words and the correct spelling in response. Here we have searched for an incorrect word (`cemera`) but in the response, the correct spelling has been returned as a spellcheck-suggested word.

However, you don't need to provide any spellchecking-related parameters to the query string. Still, if you want to disable spellchecking for any specific query, you can use `spellcheck=false` and disable spellchecking for that particular query. For example:

URL: `http://localhost:8983/solr/techproducts/select?q=cemera&spellcheck=false` :


```
{
 "responseHeader":{
 "status":0,
 "QTime":0,
 "params":{
 "q":"cemera",
 "spellcheck":"false"}},
 "response":{"numFound":0,"start":0,"docs":[]
 }}
```


Spellchecking is not executed for this query though we have searched for an incorrect spelling.

It is always advisable to perform spellchecking last because we still want the default search components, such as query, facet, and debug, to execute during query processing. This can be done easily by setting `spellcheck.collate=true` . This is a collation parameter that tells Solr to run spellchecking last because generating the collation query requires an already executed query.

Spellcheck parameters

As we discussed earlier in this lab, we do not need to pass any parameter to enable or configure a parameter with a query string. Configuring them with any request handler in the `solrconfig.xml` file will enable them. Let's understand all our parameters:

Parameter	Behavior	Default value
`spellcheck`	Enables or disables spellchecking.	`false`
`spellcheck.q` or `spellcheck.q`	Specifies the query for spellchecking. It will be used if `spellcheck.q` is specified, or else the original input query will be used.	
`spellcheck.build`	Boolean parameter that tells Solr to generate a dictionary if it does not exist. Dictionary building will take additional time in query processing, so it is advisable not to pass this parameter with every request.	`false`
`spellcheck.reload`	A Boolean parameter that tells Solr to reload the spellchecker implementation.	
`spellcheck.count`	Specifies the maximum number of suggestions a spellchecker should return in the response for a specific query term. The default value is `1` if the parameter is not set and is `5` if the parameter is set but no value is assigned.	`1`
`spellcheck.onlyMorePopular`	Specifying this as true will tell Solr to return only those suggestions that have more hits than the original query.	
`spellcheck.maxResultsForSuggest`	Specifies a threshold count based on the number of documents matched from the user's original query. For example, if we have set this to `5` and a user's original query returns **>= 5** matching documents, Solr will disable the suggestion automatically.	
`spellcheck.alternativeTermCount`	Specifies the number of suggestions to be returned from the index and/or dictionary for each query term. It will also enable context-sensitive spelling suggestions.	

`spellcheck.extendedResults`	Boolean variable that tells Solr to return additional details about the spellcheck results. For example, setting this to true will return the frequency of the original terms and of suggestions from the index.	
`spellcheck.collate`	If true, this parameter tells Solr to create a new query (collation query) from the suggested spelling correction. The collation query can be executed by clicking on the link `Did you mean ... ?` Returning results is guaranteed, so Solr must execute the collation query in the background before returning results to users.	
`spellcheck.maxCollations`	The maximum number of collation queries Solr will generate. The default is `1` . It will work only if `spellcheck.collate` is true.	`1`
`spellcheck.maxCollationTries`	Specifies the number of times Solr should try for collation before giving up. Specifying a lower value will improve performance as Solr will not provide suggestions in all cases.	
`spellcheck.maxCollationEvaluations`	Specifies the maximum number of word correction combinations for evaluating a correct collation that will run against an index.	`10000`
`spellcheck.collateExtendedResults`	A boolean value that tells Solr whether to return a response in expanded format or not. Setting this to true will return the response in expanded format. It will work only if `spellcheck.collate` is true.	`false`
`spellcheck.collateMaxCollectDocs`	Specifies the maximum number of documents to be collected for testing of collations against the index. The default value for this parameter is `0` , which means that all documents should be collected.	`0`

`spellcheck.collateParam.*Prefix`	Used to specify an additional parameter that you want to be considered by the Spellchecker when internally validating collation queries.	
`spellcheck.dictionary`	Specifies the dictionary name to be used by Solr for spellchecking.	`default`
`spellcheck.accuracy`	Specifies the accuracy level to decide whether the results provided after spellchecking are sufficient or not. Possible values are float values between `0` and `1` .	`Float.MIN_VALUE`
`spellcheck.<DICT_NAME>.key`	Specifies a key/value pair for dictionary implementation of spellcheck.	

Implementation approaches

Solr provides many approaches for implementing spellchecking. Let's get a brief overview of these approaches.

IndexBasedSpellChecker

The `IndexBasedSpellChecker` builds a parallel index based on the Solr index and performs spellchecking using this parallel index. For this, a field needs to be defined as a basis for the index terms. The easiest way is to copy terms from some fields (subject, description, and so on) to another field created for spellchecking. Here is a simple example of the `IndexBasedSpellChecker` approach configured in `solrconfig.xml`:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
 <lst name="spellchecker">
 <str name="classname">solr.IndexBasedSpellChecker</str>
 <str name="spellcheckIndexDir">./spellchecker</str>
 <str name="field">content</str>
 <str name="buildOnCommit">true</str>
 <!-- optional elements with defaults
 <str name="distanceMeasure">org.apache.lucene.search.spell.LevensteinDistance</str>
 <str name="accuracy">0.5</str>
 -->
 </lst>
</searchComponent>
```

The attribute `classname` defines the approach to be used for spellchecking. If we do not define this attribute, `solr.IndexBasedSpellChecker` is the default one used for spellchecking.

DirectSolrSpellChecker

The `DirectSolrSpellChecker` performs spellchecking directly using the terms from the Solr index without building a parallel index like `IndexBasedSpellChecker`. As `DirectSolrSpellChecker` does not build any indexes, suggestion terms are always up to date as with the Solr main index. Here is an example configured in `solrconfig.xml`:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
 <lst name="spellchecker">
 <str name="name">default</str>
 <str name="field">name</str>
 <str name="classname">solr.DirectSolrSpellChecker</str>
 <str name="distanceMeasure">internal</str>
 <float name="accuracy">0.5</float>
 <int name="maxEdits">2</int>
 <int name="minPrefix">1</int>
 <int name="maxInspections">5</int>
 <int name="minQueryLength">4</int>
 <float name="maxQueryFrequency">0.01</float>
 <float name="thresholdTokenFrequency">.01</float>
 </lst>
</searchComponent>
```

FileBasedSpellChecker

The `FileBasedSpellChecker` evaluates spellings from an external file dictionary. This is useful when Solr works as a spellchecking server. This is also useful where spelling suggestions are not on the base of actual terms. To implement `FileBasedSpellChecker`, modify `solrconfig.xml` as follows:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
 <lst name="spellchecker">
 <str name="classname">solr.FileBasedSpellChecker</str>
 <str name="name">file</str>
 <str name="sourceLocation">spellings.txt</str>
 <str name="characterEncoding">UTF-8</str>
 <str name="spellcheckIndexDir">./spellcheckerFile</str>
 <!-- optional elements with defaults
 <str name="distanceMeasure">org.apache.lucene.search.spell.LevensteinDistance</str>
 <str name="accuracy">0.5</str>
 -->
 </lst>
</searchComponent>
```

The `sourceLocation` parameter holds the path of the file that contains the spelling dictionary. The `characterEncoding` parameter specifies the character-encoding algorithm.

WordBreakSolrSpellChecker

The `WordBreakSolrSpellChecker` performs spellchecking by breaking and/or combining query terms. This is useful for terms where users put whitespaces at incorrect places. For example, to search for `group dance choreographer`, a user may enter `groupdance choreographor` as a search term. It also provides shard support. Here is the configuration in `solrconfig.xml` for this approach:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
 <lst name="spellchecker">
 <str name="name">wordbreak</str>
 <str name="classname">solr.WordBreakSolrSpellChecker</str>
 <str name="field">lowerfilt</str>
 <str name="combineWords">true</str>
 <str name="breakWords">true</str>
 <int name="maxChanges">10</int>
 </lst>
</searchComponent>
```

Distributed spellcheck

Solr supports spellchecking on distributed indexes also. The following are the two parameters that are required for the request handler (excluding request handler `/select`) to implement spellchecking on distributed indexes:

Parameter	Behavior
`shards`	Specifies the shards in distributed indexing configuration.
`shards.qt`	Specifies a request handler for requesting to shards. This parameter is not required by the `/select` request handler.

For example, to search for the word `cemera` on distributed indexing (`shard1` and `shard2`), the following is the URL:

`http://localhost:8983/solr/techproducts/spell?spellcheck=true&spellcheck.build=true&spellcheck.q=cemera&shards.qt=/spell&shards=solr-shard1:8983/solr/techproducts,solr-shard2:8983/solr/techproducts` Suggester

In the preceding section, we have seen how Solr handles incorrectly spelled terms and then returns the correct output for them. Let's move one step ahead and provide a feature wherein the user always enters correct spellings but we want to be a step ahead and provide list of suggestions using whatever the user has already typed. This can be achieved by a Solr tool called suggester. The suggester suggests terms when the user types words. During the implementation of the suggester, we need to consider these two things:

- It must be very fast as we need to display suggestions on the user's characters type
- The suggestions should be ranked and ordered by term frequency

To configure any suggester, `SuggestComponent` needs to be configured in `solrconfig.xml`. Here is a simple configuration of a suggester:

```
<searchComponent name="suggest" class="solr.SuggestComponent">
 <lst name="suggester">
 <str name="name">mySuggester</str>
 <str name="lookupImpl">FuzzyLookupFactory</str>
 <str name="dictionaryImpl">DocumentDictionaryFactory</str>
 <str name="field">cat</str>
 <str name="weightField">price</str>
 <str name="suggestAnalyzerFieldType">string</str>
 <str name="buildOnStartup">false</str>
 </lst>
 </searchComponent>
```

Now this `suggest` search component must be associated with any request handler in `solrconfig.xml`. Previously we used the `/select` request handler, where we configured many components such as spellchecking. Also, the `/select` request handler contains some built-in components: query, faceting, highlighting, and so on. Merging the suggester with other components is not the best approach at all times; sometimes we expect only suggestions and not spellchecking. So, defining a separate request handler for suggestions is the recommended approach. For that, we are defining a separate request handler here and adding the previously defined `suggest` search component to this request handler in `solrconfig.xml`:

```
<requestHandler name="/suggest" class="solr.SearchHandler" startup="lazy">
 <lst name="defaults">
 <str name="suggest">true</str>
 <str name="suggest.count">10</str>
 </lst>
 <arr name="components">
 <str>suggest</str>
 </arr>
</requestHandler>
```

Now, all requests coming at the `/suggest` request handler will be treated for suggestions only. This will work as the other request handler and allow us to configure default parameters for suggestion requests.

Suggester parameters

As we have seen, first we need to define a search component and then we need to inject that component with any request handler. Both the sections contain some parameters that fulfill the suggestion requirements. Here is a list of parameters.

Suggester search component parameters are as follows:

Parameter	Behavior	Default value
`searchComponent`	The name of the search component.	
`name`	Suggester name, which will be referred with the request handler.	
`lookupImpl`	Specifies the lookupImpl implementation algorithms used to look up terms in the suggest index. Available implementations are `AnalyzingLookupFactory`, `FuzzyLookupFactory`, `AnalyzingInfixLookupFactory`, `BlendedInfixLookupFactory`, `FreeTextLookupFactory`, `FSTLookupFactory, TSTLookupFactory`, `WFSTLookupFactory`, and `JaspellLookupFactory`.	`JaspellLookupFactory`
`dictionaryImpl`	Specifies a dictionary implementation for suggestions. Available implementations are `DocumentDictionaryFactory`, `DocumentExpressionDictionaryFactory`, `HighFrequencyDictionaryFactory`, and `FileDictionaryFactory`.	`HighFrequencyDictionaryFactory`
`sourceLocation`	Specifies the dictionary file path when a suggestion is implemented using `FileDictionaryFactory`. The main index will be used as the source of terms and weights if this parameter value is empty.	
`field`	Specifies a field from which the index is to be used as the basis of suggestion terms. The specified field must be stored in the index.	
`storeDir`	Specifies a path to store the dictionary file.	

`buildOnCommit` and `buildOnOptimize`	If this is true, the suggestion dictionary will be rebuilt after a soft commit. If false, the suggestion dictionary will be built only when we provide the suggest.build=true parameter in the URL. Use `buildOnCommit` to rebuild the suggestion dictionary with every soft commit or `buildOnOptimize` to build it only when the index is optimized. This parameter is not recommended if the volume of the indexes is very high; in such a scenario, `suggest.build=true` is recommended.	`false`
`buildOnStartup`	Specifying this to true will build the suggestion directory at the time of Solr start or core reloading. If this parameter is not specified, the suggester will check whether the suggestion directory is present on disk, and build one if it is not found. Enabling (`true`) is not recommended as sometimes it may take too long in build; instead, use `suggest.build=true` for building manually.	`false`

Suggester request handler parameters are as follows:

Parameter	Behavior	Default value
`suggest`	A boolean parameter to enable/disable suggestions.	`true`
`suggest.dictionary`	A mandatory parameter. It specifies the dictionary component configured inside the search component.	
`suggest.q`	Specifies a query to use for suggestions.	
`suggest.count`	Specifies the number of suggestions to be returned.	`10`
`suggest.cfq`	Specifies the **context filter query** (**CFQ**) used to filter suggestions based on the context field. CFQ is only supported by `AnalyzingInfixLookupFactory` and `BlendedInfixLookupFactory` and only when backed by a `Document*Dictionary` .	
`suggest.build`	If true, it will build the suggester index.	`false`
`suggest.reload`	If true, it will reload the suggester index.	`false`
`suggest.buildAll`	If true, it will build all suggester indexes	`false`
`suggest.reloadAll`	If true, it will reload all suggester indexes.	`false`

All of these parameters are usually configured in the request handler. However, we can override them at query time by passing parameters in the URL.

Running suggestions

We have configured the suggester and request handler for this suggester in `solrconfig.xml` . Now Let's run the configured suggester and examine the response.

Example: Let's search for `elec` and see how many suggestions are returned in response.

URL: `http://localhost:8983/solr/techproducts/suggest?suggest=true&suggest.build=true&suggest.dictionary=mySuggester&suggest.q=elec` :


```
{
 "responseHeader":{
 "status":0,
 "QTime":14},
 "command":"build",
 "suggest":{"mySuggester":{
 "elec":{
 "numFound":3,
 "suggestions":[{
 "term":"electronics and computer1",
 "weight":2199,
 "payload":""},
 {
 "term":"electronics",
 "weight":649,
 "payload":""},
 {
 "term":"electronics and stuff2",
 "weight":279,
 "payload":""}]}}}
}
```


The query `elec` has returned three suggestions under the `mySuggester` section. Here, we have provided a single dictionary by the parameter `suggest.dictionary=mySuggester`. In the same way, we can configure multiple dictionaries in `solrconfig.xml` and use them in a URL like this:

`http://localhost:8983/solr/techproducts/suggest?suggest=true&suggest.build=true&suggest.dictionary=mySuggester&suggest.dictionary=yourSuggester&suggest.q=elec`
Pagination

A query may fetch a number of results for a search. Returning all results at a time and displaying all of them on a single page is not an ideal approach for any search application. Rather, returning the top [*N*] number of matching results (sorted based on some fields) first is the ideal way for an application. Solr supports a pagination feature whereby we can return a certain number of results rather than all results and display them on the first page. If we can't find the results we are looking for on the first page, we can call the next page of results by running the subsequent request with pagination parameters. Pagination is very helpful in terms of performance because instead of returning all matching results at a time, it will return only a specific number of results; so the result is very quick. Using pagination, we also can determine how many queries are required to fulfill the expectations behind the search; so we can manage relevance accordingly.

How to implement pagination

To implement pagination, we need to configure two parameters in the query request.

- `start` : Indicates from where the results should be returned from the complete result set
- `rows` : Indicates how many results must be returned from the complete result set

We have to specify some value if we have used the start parameter in a request. Keep in mind that the value of the start parameter should be less than the total number of matching results. If we set `start` higher than the total number of matching results found, the query will not return anything.

Example: Search for a `query ``q=*:*` and get the first five results sorted by ID in ascending order.

URL: `http://localhost:8983/solr/techproducts/select?q=*:*&fl=id,name&start=0&rows=5&sort=id asc` :


```
{
 "responseHeader":{
 "status":0,
 "QTime":0,
 "params":{
 "q":"*:*",
 "fl":"id,name",
 "start":"0",
 "sort":"id asc",
 "rows":"5"}},
 "response":{"numFound":32,"start":0,"docs":[
```

```
 {
 "id":"0579B002",
 "name":"Canon PIXMA MP500 All-In-One Photo Printer"},
 {
 "id":"100-435805",
 "name":"ATI Radeon X1900 XTX 512 MB PCIE Video Card"},
 {
 "id":"3007WFP",
 "name":"Dell Widescreen UltraSharp 3007WFP"},
 {
 "id":"6H500F0",
 "name":"Maxtor DiamondMax 11 - hard drive - 500 GB - SATA-300"},
 {
 "id":"9885A004",
 "name":"Canon PowerShot SD500"}]
 }
}
```

Only five results are returned in the response. Now, if we want the next five results, we set `start=5` and `rows=5`. Please note that the response index starts with zero.

Cursor pagination

Pagination implementation using the start and rows parameters is very easy and straightforward. But when we have very large data volumes, these parameters are not sufficient to implement pagination. For example, consider this:

- During query processing, Solr first loads all the matching documents in memory; then it creates an offset by the start and rows parameters and returns that offset for that query. If the data volume is very large, Solr first loads all matching results in the memory and then applies pagination. So this will create a performance problem.
- In large volumes of data, a request for `start=0&rows=1000000` may create trouble for Solr in maintaining and sorting a collection of 1 million documents in memory.
- A request for `start=999000&rows=1000` also creates the same problem. To match the document at the 999001^th^ place, Solr has to traverse through the first 999,000 documents.
- A similar problem exists with SolrCloud, where indexes are distributed. If we have 10 shards, then Solr retrieves 10 million documents (1 million from each shard) and then sorts to find 1,000 documents matching the query parameters.

As a solution to these problems, Solr introduce a feature of pagination---cursor. Cursor does not manage caching on the server but marks the point from where the last document was returned. Now that mark point, called `cursorMark`, is supplied inside the parameters of subsequent requests to tell Solr where to continue from.

To implement pagination through cursor, we need to specify the parameter `cursorMark` with the value of `\*`. Once we implement cursor pagination in Solr, Solr returns the top [*N*] number of sorted results (where [*N*] can be specified in the rows parameter) along with an encoded string named `nextCursorMark` in the response. In subsequent requests, the `nextCursorMark` value will be passed as the `cursorMark` parameter. The process will be repeated until the expected result set is retrieved or the value of `nextCursorMark` matches `cursorMark` (which means that there are no more results).

Examples: Fetching all documents.

Using pseudo-code:

```
$params = [ q => $some_query, sort => 'id asc', rows => $r, cursorMark => '*' ]
$done = false
while (not $done) {
  $results = fetch_solr($params)
  // do something with $results
  if ($params[cursorMark] == $results[nextCursorMark]) {
    $done = true
  }
  $params[cursorMark] = $results[nextCursorMark]
}
```

Using SolrJ:

```
SolrQuery q = (new SolrQuery(some_query)).setRows(r).setSort(SortClause.asc("id"));
String cursorMark = CursorMarkParams.CURSOR_MARK_START;
boolean done = false;
while (! done) {
  q.set(CursorMarkParams.CURSOR_MARK_PARAM, cursorMark);
```

```
  QueryResponse rsp = solrServer.query(q);
  String nextCursorMark = rsp.getNextCursorMark();
  doCustomProcessingOfResults(rsp);
  if (cursorMark.equals(nextCursorMark)) {
    done = true;
  }
  cursorMark = nextCursorMark;
}
```

Using curl:

```
$ curl '...&rows=10&sort=id+asc&cursorMark=*'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 10 docs here ...
  ]},
  "nextCursorMark":"AoEjR0JQ"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEjR0JQ'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 10 more docs here ...
  ]},
  "nextCursorMark":"AoEpVkRCREIxQTE2"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEpVkRCREIxQTE2'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 10 more docs here ...
  ]},
  "nextCursorMark":"AoEmbWF4dG9y"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEmbWF4dG9y'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 2 docs here because we've reached the end.
  ]},
  "nextCursorMark":"AoEpdmlld3Nvbmlj"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEpdmlld3Nvbmlj'
{
  "response":{"numFound":32,"start":0,"docs":[
    // no more docs here, and note that the nextCursorMark
    // matches the cursorMark param we used
  ]},
  "nextCursorMark":"AoEpdmlld3Nvbmlj"}
```

Examples: Fetching [*N*] Number of documents.

SolrJ:

```
while (! done) {
 q.set(CursorMarkParams.CURSOR_MARK_PARAM, cursorMark);
 QueryResponse rsp = solrServer.query(q);
 String nextCursorMark = rsp.getNextCursorMark();
 boolean hadEnough = doCustomProcessingOfResults(rsp);
 if (hadEnough || cursorMark.equals(nextCursorMark)) {
 done = true;
 }
 cursorMark = nextCursorMark;
}
```

When implementing pagination, we need to take care of a few things:

- If we have used a start parameter in the request, we have to specify some value.
- The field on which we are applying sorting must be unique (`uniqueKey` field).
- The `cursorMark` values are calculated based on the sort values of each document, and it may be possible that multiple documents have the same sort values; that will create identical `cursorMarks` values. Now, Solr will be confused in subsequent requests as to which `cursorMark` value should be considered. To overcome this, Solr provides an additional field, `uniqueKey`, used as a clause with a

sorting parameter. This `uniqueKey` guarantees that the documents are returned in deterministic order, and that way each `cursorMark` will always point to a unique value.

- When documents are sorted based on a `Date` function, `NOW` will create confusion for cursor because `NOW` will create a new sort value for each document in each subsequent request. This will result in never-ending cursors and return the same document every time. To overcome this situation, select a fixed value for the `NOW` parameter in all requests. Result grouping

---

Result grouping is a useful feature in Solr; it returns an optimal mix of search results for a query. Result grouping can be performed based on field values, functions, or queries.

Sometimes, we have multiple similar documents for a single search term, for example, multiple locations for the same hospital, recipes for specific food, plans for term insurance, and so on. In the normal way, if we are searching for one such term, it will return all similar documents and we will have to display all of them on the same page. Through result grouping, we can display only a single document (or the top few or some limited number) for each unique value, and provide a message link with meaningful text and the number of total results found for that query. Clicking on that link will expand the full search result list. This is similar to the expand and collapse features of a search application. Result grouping is just as capable as expand and collapse; additionally it removes duplicate documents from the list. But from a performance point of view, expand and collapse is a better choice than result grouping.

We looked at faceting in the previous lab. Result grouping also looks similar to faceting, but through faceting, Solr returns a separate facet section along with the counts for each value. Through result grouping, it actually returns the unique values and their counts (like faceting) plus a number of documents that contain each of the specified values. Also, groups returned in the results section are sorted based on the sorting parameter provided inside the query.

Combining grouping with faceting is also possible. Currently, field and range faceting are included in grouped faceting, but date and pivot faceting are not included. The facet counts are calculated based on the first `group.field` parameter and other `group.field` parameters are ignored.

Result grouping parameters

Parameter	Behavior	Default value
`group`	If true, query results will be grouped.	`false`
`group.field`	Specifies the field by which the result will be grouped.	
`group.func`	Specifies a function name by which the result will be grouped.	
`group.query`	Specifies a result grouping query that will return a single group of documents.	
`rows`	Specifies the number of groups to return.	`10`
`start`	Specifies starting point from the list of groups.	
`group.limit`	Specifies the number of results to be returned for every group.	`1`
`group.offset`	Specifies starting point from the list of documents of every group.	
`sort`	Specifies the field based on groups will be sorted.	score desc
`group.sort`	Specifies the field based on documents will be sorted within each group. The default behaviour is that if group.sort is not specified, the sort parameter value will be used.	

`group.format`	Specifies the response format after grouping. Possible values for this parameter are simple and grouped.	Advanced grouping format
`group.main`	A boolean variable that tells Solr to use first the field grouping command as the main result in the response using `group.format=simple` . A true value will do the same.	`false`
`group.ngroups`	A boolean variable that tells Solr to include the number of groups that have matched to the query in result. A true value will do the same.	`false`
`group.truncate`	A boolean variable that tells Solr to calculate facet counts based on the most relevant document of each group matching the query. A true value will do the same.	`false`
`group.facet`	To enable/disable grouped faceting.	`false`
`group.cache.percent`	Enables caching for result grouping by configuring a value greater than zero. The maximum value is 100.	`0`

Running result grouping

We have explored the result grouping concept and configurations. Now Let's execute some examples.

Example: Grouping by field `manu_exact"` , which specifies the manufacturers of items in the `techproducts` dataset.

URL: `http://localhost:8983/solr/techproducts/select?fl=id,name&q=solr+memory&group=true&group.field=manu_exact`


```
{
 "responseHeader":{
 "status":0,
 "QTime":5,
 "params":{
 "q":"solr memory",
 "fl":"id,name",
 "group.field":"manu_exact",
 "group":"true"}},
 "grouped":{
 "manu_exact":{
 "matches":6,
 "groups":[{
 "groupValue":"Apache Software Foundation",
 "doclist":{"numFound":1,"start":0,"docs":[
 {
 "id":"SOLR1000",
 "name":"Solr, the Enterprise Search Server"}]
 }},
 {
 "groupValue":"Corsair Microsystems Inc.",
 "doclist":{"numFound":2,"start":0,"docs":[
 {
 "id":"VS1GB400C3",
 "name":"CORSAIR ValueSelect 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC 3200) System Memory - Retail"}]
 }},
 {
 "groupValue":"A-DATA Technology Inc.",
 "doclist":{"numFound":1,"start":0,"docs":[
 {
 "id":"VDBDB1A16",
 "name":"A-DATA V-Series 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC 3200) System Memory - OEM"}]
 }},
 {
```

```
 "groupValue":"Canon Inc.",
 "doclist":{"numFound":1,"start":0,"docs":[
 {
 "id":"0579B002",
 "name":"Canon PIXMA MP500 All-In-One Photo Printer"}]
 }},
 {
 "groupValue":"ASUS Computer Inc.",
 "doclist":{"numFound":1,"start":0,"docs":[
 {
 "id":"EN7800GTX/2DHTV/256M",
 "name":"ASUS Extreme N7800GTX/2DHTV (256 MB)"}]
 }}]}},
 "spellcheck":{
 "suggestions":[],
 "correctlySpelled":true,
 "collations":[]}
}
```


In the same way, grouping can be achieved by specifying a query or queries.

Example: Retrieve the top three results for the field `memory` for two price ranges of 0.00 to 99.99 and over 100, using `group.query`.

URL: `http://localhost:8983/solr/techproducts/select?indent=true&fl=name,price&q=memory&group=true&group.query=price:[0+TO+99.99]&group.query=price:[100+TO+*]&group.limit=3`


```
{
  "responseHeader":{
    "status":0,
    "QTime":2,
    "params":{
      "q":"memory",
      "indent":"true",
      "fl":"name,price",
      "group.limit":"3",
      "group.query":["price:[0 TO 99.99]",
        "price:[100 TO *]"],
      "group":"true"}},
  "grouped":{
    "price:[0 TO 99.99]":{
      "matches":5,
      "doclist":{"numFound":1,"start":0,"docs":[
          {
            "name":"CORSAIR ValueSelect 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC 3200) System Memory - Retail",
            "price":74.99}]
      }},
    "price:[100 TO *]":{
      "matches":5,
      "doclist":{"numFound":3,"start":0,"docs":[
          {
            "name":"CORSAIR XMS 2GB (2 x 1GB) 184-Pin DDR SDRAM Unbuffered DDR 400 (PC 3200) Dual Channel Kit System Memory - Retail",
            "price":185.0},
          {
            "name":"Canon PIXMA MP500 All-In-One Photo Printer",
            "price":179.99},
          {
            "name":"ASUS Extreme N7800GTX/2DHTV (256 MB)",
            "price":479.95}]
      }}},
  "spellcheck":{
    "suggestions":[],
    "correctlySpelled":false,
    "collations":[]}}
```

Result clustering

So far, we have seen Solr searching by the keyword used in search query. Result clustering is the advanced search component of Solr; it first identifies the similarities between documents, and using these similarities, it finds related documents. It is also not necessary for the identified similarities to be present in the query or document.

The clustering component first discovers the results of a search query and identifies similar terms or phrases found within the search results. A clustering algorithm discovers relationships across all the documents from the search result and forms in a meaningful cluster label. Solr comes with several algorithms for clustering implementation.

Result clustering parameters

Parameter	Behavior	Default value
`clustering`	Enable/disable clustering.	`true`
`clustering.engine`	Specifies which clustering engine to use. If not specified, the first declared engine will become the default one.	first in a list
`clustering.results`	When true, the component will run a clustering of the search results (this should be enabled).	`true`
`clustering.collection`	When true, the component will run a clustering of the whole document index.	`false`

From the preceding list, some parameters are of the search component and some of them are of the request handler. There are some additional parameters that are specific to engine-level configuration. We can pass these parameters to the query url to modify configurations at query runtime.

Result clustering implementation

Solr's built-in example `techproducts` contains preconfigured components for result clustering, but by default, the configurations are disabled.

The clustering implementation requires the following configurations.

Install the clustering contrib

The clustering contrib extension requires `solr-clustering-*.jar` under `/dist` and all JARs under `contrib/clustering/lib`. Configure the file path according to your system file path in `solrconfig.xml`:

```
<lib dir="${solr.install.dir:../../../..}/contrib/clustering/lib/" regex=".*\.jar" />
 <lib dir="${solr.install.dir:../../../..}/dist/" regex="solr-clustering-\d.*\.jar" />
```

Declare the cluster search component

** **Define the search component for clustering in `solrconfig.xml`:

```
<searchComponent name="clustering" enable="true"
 class="solr.clustering.ClusteringComponent" >
 <lst name="engine">
 <str name="name">lingo</str>
 <str name="carrot.algorithm">org.carrot2.clustering.lingo.LingoClusteringAlgorithm</str>
 <str name="carrot.resourcesDir">clustering/carrot2</str>
 </lst>
 <lst name="engine">
 <str name="name">stc</str>
 <str name="carrot.algorithm">org.carrot2.clustering.stc.STCClusteringAlgorithm</str>
 <str name="carrot.resourcesDir">clustering/carrot2</str>
 </lst>
</searchComponent>
```

We can also declare cluster components as engines or a multiple clustering pipeline, which can be selected by specifying the parameter `clustering.engine=<engine name>` in the query URL.

Declare the request handler and include the cluster search component

Define the request handler for the cluster in `solrconfig.xml` :

```
<requestHandler name="/clustering" startup="lazy" enable="true" class="solr.SearchHandler">
 <lst name="defaults">
 <bool name="clustering">true</bool>
 <bool name="clustering.results">true</bool>
 <!-- Field name with the logical "title" of a each document (optional) -->
 <str name="carrot.title">name</str>
 <!-- Field name with the logical "URL" of a each document (optional) -->
 <str name="carrot.url">id</str>
 <!-- Field name with the logical "content" of a each document (optional) -->
 <str name="carrot.snippet">features</str>
 <!-- Apply highlighter to the title/ content and use this for clustering. -->
 <bool name="carrot.produceSummary">true</bool>
 <!-- the maximum number of labels per cluster -->
 <!--<int name="carrot.numDescriptions">5</int>-->
 <!-- produce sub clusters -->
 <bool name="carrot.outputSubClusters">false</bool>
 <!-- Configure the remaining request handler parameters. -->
 <str name="defType">edismax</str>
 <str name="qf">
 text^0.5 features^1.0 name^1.2 sku^1.5 id^10.0 manu^1.1 cat^1.4
 </str>
 <str name="q.alt">*:*</str>
 <str name="rows">100</str>
 <str name="fl">*,score</str>
 </lst>
 <arr name="last-components">
 <str>clustering</str>
 </arr>
</requestHandler>
```

Now we are done with cluster configurations. Let's run a cluster for the built-in example `techproducts` by setting `enable="true"` in the `searchComponent` and `requestHandler` configuration. We can enable the same by specifying a JVM system property using the following command:

```
solr start -e techproducts -Dsolr.clustering.enabled=true
```

Let's run a query for electronics using the configured request handler `/clustering` and see the cluster response.

URL: `http://localhost:8983/solr/techproducts/clustering?q=electronics&rows=100` :

```
{
  "responseHeader":{
    "status":0,
    "QTime":32},
  "response":{"numFound":14,"start":0,"maxScore":2.9029632,"docs":[
      ....
    ....
      ]
  },
  "clusters":[{
      "labels":["DDR"],
      "score":3.037927435185717,
      "docs":["TWINX2048-3200PRO",
        "VS1GB400C3",
        "VDBDB1A16"]},
    {
      "labels":["iPod"],
      "score":7.317758461138239,
      "docs":["F8V7067-APL-KIT",
```

```
        "IW-02",
        "MA147LL/A"]},
    {
      "labels":["Canon"],
      "score":6.785392802370259,
      "docs":["0579B002",
        "9885A004"]},
    {
      "labels":["Hard Drive"],
      "score":10.460153088070832,
      "docs":["SP2514N",
        "6H500F0"]},
    {
      "labels":["Retail"],
      "score":1.629540936033123,
      "docs":["TWINX2048-3200PRO",
        "VS1GB400C3"]},
    {
      "labels":["Video"],
      "score":10.060361253597023,
      "docs":["MA147LL/A",
        "100-435805"]},
    {
      "labels":["Other Topics"],
      "score":0.0,
      "other-topics":true,
      "docs":["EN7800GTX/2DHTV/256M",
        "3007WFP",
        "VA902B"]}]
}
```


Here we can see a few clusters discovered for the query (`q=electronics`). Each cluster has a label and the score shows the kindness of the cluster. The score is specific to an algorithm and meaningful only in relation to the scores of other clusters in the same set. A score with a higher value is better. Spatial search

---

Location-based data search is a very important requirement nowadays, such as searching for distances from a place, searching a house within a radius, and so on. Solr supports searching for location-based data called spatial or geospatial searches.

This can be implemented by indexing a field in each document that contains a geographical point (a latitude and longitude); and then at query time, we can find and sort documents by distance from a geolocation (latitude and longitude). The matching listings (results) can be displayed on an interactive map, in which we can zoom in/out and move the map center point to find nearby listings using spatial search. Like latitude and longitude, Solr also allows us to index geographical shapes (polygons), which are used to find a document that intersects geographical regions. Spatial search implemented by indexing a field (latitude and longitude) helps to search from a specific point, while implementing by indexing geographical shapes helps to search all documents within a given range. Solr also supports returning the distances of matching documents in a response. We can also apply sorting on distances to provide more useful results to the user.

Spatial search implementation

There are various sequential steps we need to execute in order to implement spatial search.

Field types

First we need to configure field type in `managed-schema.xml` or `schema.xml` :

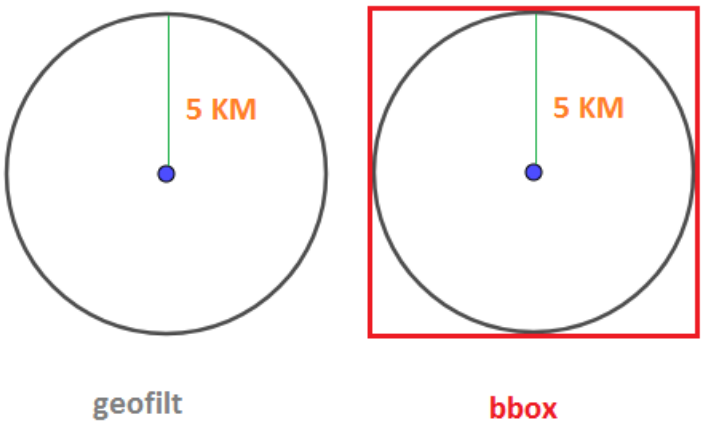
```
<fieldType name="location" class="solr.LatLonPointSpatialField" docValues="true"/>
```

There are mainly four field types available in Solr that support spatial search:

- `LatLonPointSpatialField` : The most commonly used field type and an alternative to `LatLonType` (deprecated) for latitude---and longitude-based search.
- `LatLonType` : Its non-geodetic twin `PointType` field type, and this one is deprecated now.
- `SpatialRecursivePrefixTreeFieldType` : RPT supports more advanced search features, such as polygons and heatmaps. `RptWithGeometrySpatialField` is derived from RPT. It is used for geography-based searches. Sorting and boosting are not supported.
- `BBoxField` : Used to search a bounding box. It supports sorting and boosting.

Query parser

Previously, we explored various query parsers that support normal searching. But when it comes to spatial search, we need to configure a special parser that supports spatial search:



There are two types of spatial search parsers available in Solr.

- `geofilt` : The `geofilt` parser is used to retrieve results based on the geospatial distance from a given point. Please look at the preceding image. Another way of looking at it is that it creates a circular shape filter. For example, to find all documents within 5 kilometers of a given latitude-longitude point, the query will be:

```
&q=:&fq={!geofilt sfield=location}&pt=45.15,-93.85&d=5
```

This filter returns all results within a circle of the given radius around the initial point.

- `bbox` *: *The `bbox` parser works like `geofilt` , except that it uses the bounding box of the calculated circle. Please look at the preceding diagram. For the same example, the query will be:

```
&q=:&fq={!bbox sfield=location}&pt=45.15,-93.85&d=5
```

Spatial search query parser parameters

Both the parsers support the following parameters:

Parameter	Behavior	Default value
`d`	Specifies a radius distance (usually in kilometers) within the document that should be considered for matching.	
`pt`	Specifies latitude and longitude for the format as `lat,lon` , `x,y` for `PointType` , or `x, y` for RPT field types.	
`sfield`	Specifies a spatial field defined in `managed-schema.xml` or `schema.xml` .	
`score`	If the query is used in a scoring context, this determines what type of scores will be produced. It is not supported by `LatLonType` or `PointType` . Possible values are `none` , `kilometers` , `miles` , `degrees` , `distance` , and `recipDistance` .	`none`
`filter`	If set to `false` , parser scoring will be disabled. This is not supported by `LatLonType` or `PointType` .	`true`

Function queries

In spatial search, along with returning matching documents within a given range, Solr also supports calculation of matching document distances from the search point and merging them in the returning response. Solr even supports sorting and boosting on the calculated distance. To calculate distances, the following are the functions available in Solr:

- `geodist` : This calculates the distance between too points (latitude and longitude)
- `dist` : Calculates the p-norm distance between multidimensional vectors
- `hsin` : This calculates the distance between two points on a sphere
- `sqedist` : Calculates the squared Euclidean distance between two points

From the preceding list, `geodist` is the most commonly used function for most search cases. It takes three optional parameters: `sfield` (spatial field defined in `managed-schema.xml`), latitude, and longitude. For example, we find all cities from the searching point (`45.15` , `-93.85`) within a radius of 50 kilometers, calculate their distance from the searched point, and return in the response:

```
&q=:&fq={!geofilt}&sfield=location&pt=45.15,-93.85&d=50&fl=id,city,distance:geodist()
```

Additionally, we sort by distance ascending. The following is the query:

```
&q=:&fq={!geofilt}&sfield=location&pt=45.15,-93.85&d=50&fl=id,city,distance:geodist()
&sort=geodist() asc
```

This is the basic overview of spatial search. There are many more spatial searching features available in Solr. Exploring every feature here is not possible. However, we can take this lab as a reference and explore more spatial search features. Rather than going into more details of spatial search, Let's move on to the next lab. Summary

---

In this lab, we explored and understood various searching functionalities such as spellchecking, suggester, pagination, result grouping, and result clustering. Finally, we looked at spatial search.

So far, we have seen configurations for each one and executed examples by configuring various functionality parameters. Now Let's move to the next lab, where we will see how to configure Solr for production and learn fine-tuning methodologies for better performance. We will explore how to secure Solr and how to take backups. We will configure logging and get an overview of SolrCloud.

{.mb15 .ng-hide}