Lab 6. Advanced Queries -- Part I

In the previous lab, we learned how to build indexes using various methods. In this lab, we will see how Solr's search works. Solr comes with a large searching kit; by configuring elements from this kit, it provides users with an extensive search experience and returns impressive results with a helpful interface.

Here is a list of search functionalities provided by Solr, that put Solr in the list of desirable search engines:

- Highlighting
- Spell checking
- Reranking
- Transformation of results
- Suggested words
- Pagination on results
- Expand and collapse
- Grouping and clustering
- Spatial search
- More like this word
- Autocomplete

We will look at some of these functions in detail later in this lab, but first Let's understand every component that performs an important role during searches and generates impressive results.

Search relevance

Relevance is a measurement of the user's satisfaction with the response to their search query. It completely depends on the context of the search. Sometimes, the same document can be searched by different classes of people for different context. For example, the search query [*higher tax payer in India*] can be searched by:

- An income tax department in the context of their duty
- Chartered accountants in the context of their professional interest
- Students in the context of gaining knowledge

The comprehensiveness of any response depends on the context of the search. Sometimes, the context is high, such as searching for legal information; sometimes, it is low, when someone is searching for context such as specific dance steps. So, during Solr configuration, we need to take care of this too.

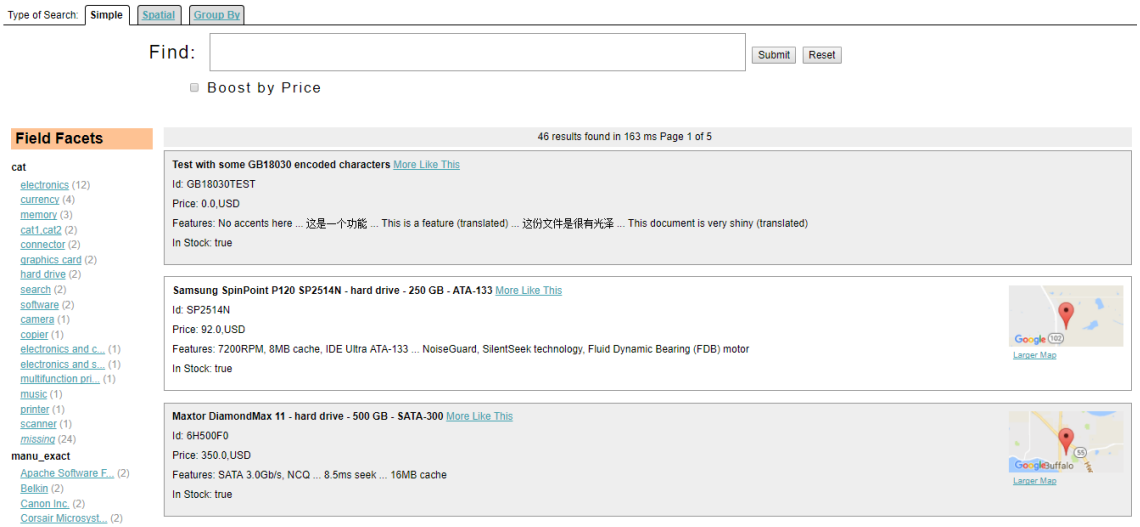
There are two terms that play an important role in relevance:

- **Precision**: Precision is the percentage of documents in the returned results that are relevant.
- **Recall**: Recall is the percentage of relevant results returned out of all relevant results in the system. Retrieving perfect recall is insignificant, for example, returning every document for every query.

From this example, we can conclude that precision and recall totally depend on the context of the search. Sometimes, we need 100% recall, say when searching for legal information. Here, all the relevant documents should be returned in the response. While in other scenarios, there is no need to return all documents. For example, when searching for dance steps, returning all the documents will overwhelm the application.

Through faceting, query filters, and other search components, the application can be configured with the flexibility to help end users get their searches, in order to return the most relevant results for users. We can configure Solr to balance precision and recall to meet the needs of a particular user community. Velocity search UI

Solr provides a user interface through which we can easily understand the Solr search mechanism. Using velocity search UI, we can explore search features such as faceting, highlighting, autocomplete, and geospatial searching. Previously we have seen an example of `techproducts`; Let's browse its products through velocity UI. You can access the UI through `http://localhost:8983/solr/techproducts/browse`, as shown in the following screenshot:



Solr uses response writer to generate an organized response. Here velocity UI uses velocity response writer. We will explore response writer later in this lab. Query parsing and syntax

---

In this section, we will explore some query parsers, their features, and how to configure them with Solr. Solr supports some query parsers. Here is the list of parsers supported by Solr:

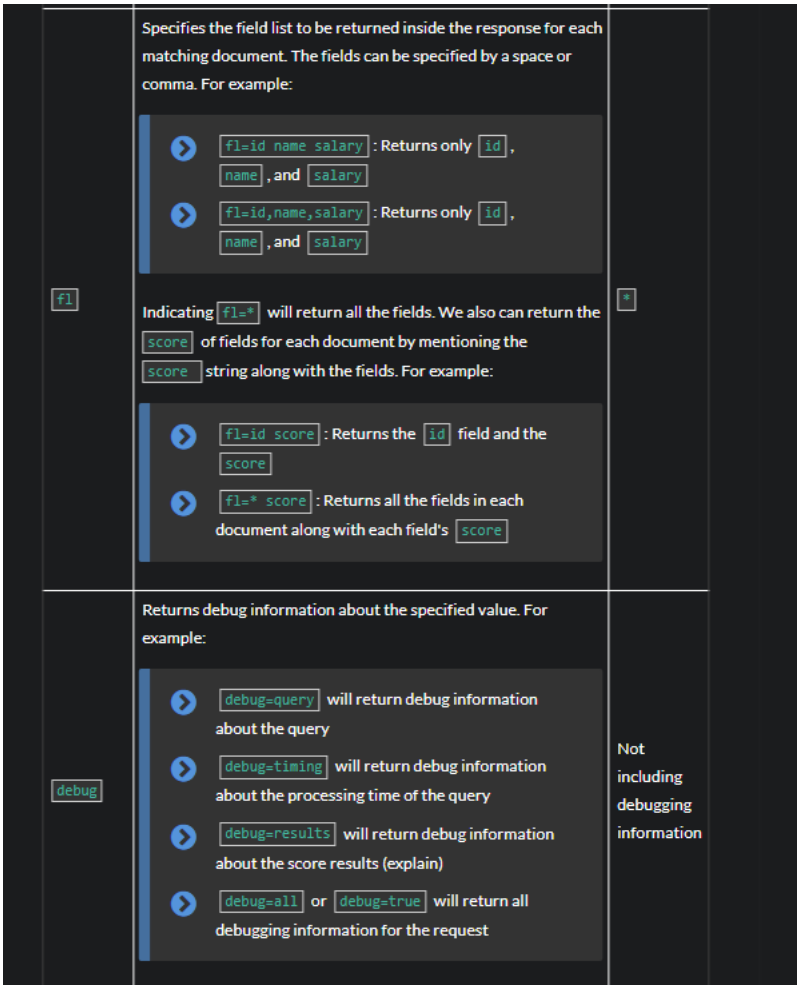
- **Standard query parser**
- **DisMax query parser**
- **Extended DisMax (eDisMax) query parser**

Each parser has its own configuration parameters for clubbing with Solr. However, there are some common parameters required by all parsers. First Let's take a look at these common parameters.

Common query parameters

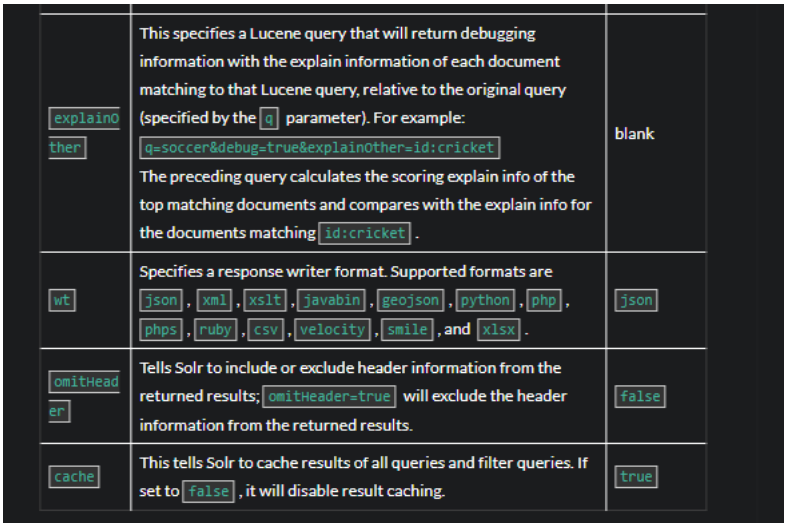
The following are the common query parameters supported by standard query parser, DisMax query parser, and extended DisMax query parser:

Parameter	Behavior	Default value
`defType`	Selects the query parser: `defType=dismax`	`Lucene` (standard query parser)
`sort`	Sorts the search results in either ascending or descending order. The value can be specified as `asc` or `ASC` and `desc` or `DESC`. Sorting is supported by numerical or alphabetical content. Solr supports sorting by field clones. **Example**: `salary asc` : Sorts based on salary (high to low). `name desc` : Sorts based on names (z → a). `salary asc name desc` : First sorts by salary high to low. Within that, it sorts the result set again sorts by name (z → a).	`desc`
`start`	Specifies the starting point from where the results should begin displaying.	`0`
`rows`	Specifies the maximum number of documents to be returned to the client from the complete result set at a time.	`10`
`fq`	Limits the result set to the documents matched by the filter query (`fq`) without affecting the score.	



`fl`	Specifies the field list to be returned inside the response for each matching document. The fields can be specified by a space or comma. For example:
	`fl=id name salary` : Returns only `id` , `name` , and `salary`
	`fl=id,name,salary` : Returns only `id` , `name` , and `salary`
	Indicating `fl=*` will return all the fields. We also can return the `score` of fields for each document by mentioning the `score` string along with the fields. For example:
	`fl=id score` : Returns the `id` field and the `score`
	`fl=* score` : Returns all the fields in each document along with each field's `score`
		`*`

`debug`	Returns debug information about the specified value. For example: `debug=query` will return debug information about the query `debug=timing` will return debug information about the processing time of the query `debug=results` will return debug information about the score results (explain) `debug=all` or `debug=true` will return all debugging information for the request	Not including debugging information



`explainOther`	This specifies a Lucene query that will return debugging information with the explain information of each document matching to that Lucene query, relative to the original query (specified by the `q` parameter). For example: `q=soccer&debug=true&explainOther=id:cricket` The preceding query calculates the scoring explain info of the top matching documents and compares with the explain info for the documents matching `id:cricket` .	blank
`wt`	Specifies a response writer format. Supported formats are `json` , `xml` , `xslt` , `javabin` , `geojson` , `python` , `php` , `phps` , `ruby` , `csv` , `velocity` , `smile` , and `xlsx` .	`json`
`omitHeader`	Tells Solr to include or exclude header information from the returned results; `omitHeader=true` will exclude the header information from the returned results.	`false`
`cache`	This tells Solr to cache results of all queries and filter queries. If set to `false` , it will disable result caching.	`true`

Apart from the preceding common parameters for parsers, `timeAllowed` , `segmentTerminateEarly` , `logParamsList` , and `echoParams` are also common parameters which are used by all the parsers. We are not detailing these parameters here.

Standard query parser

Standard query parser, also known as **Lucene query parser**, is the default query parser for Solr.

Advantage

The syntax is easy and differently structured queries can easily be created using standard query parser.

Disadvantage

Standard query parser does not throw any syntax error. So, identifying syntax errors is a little tough.

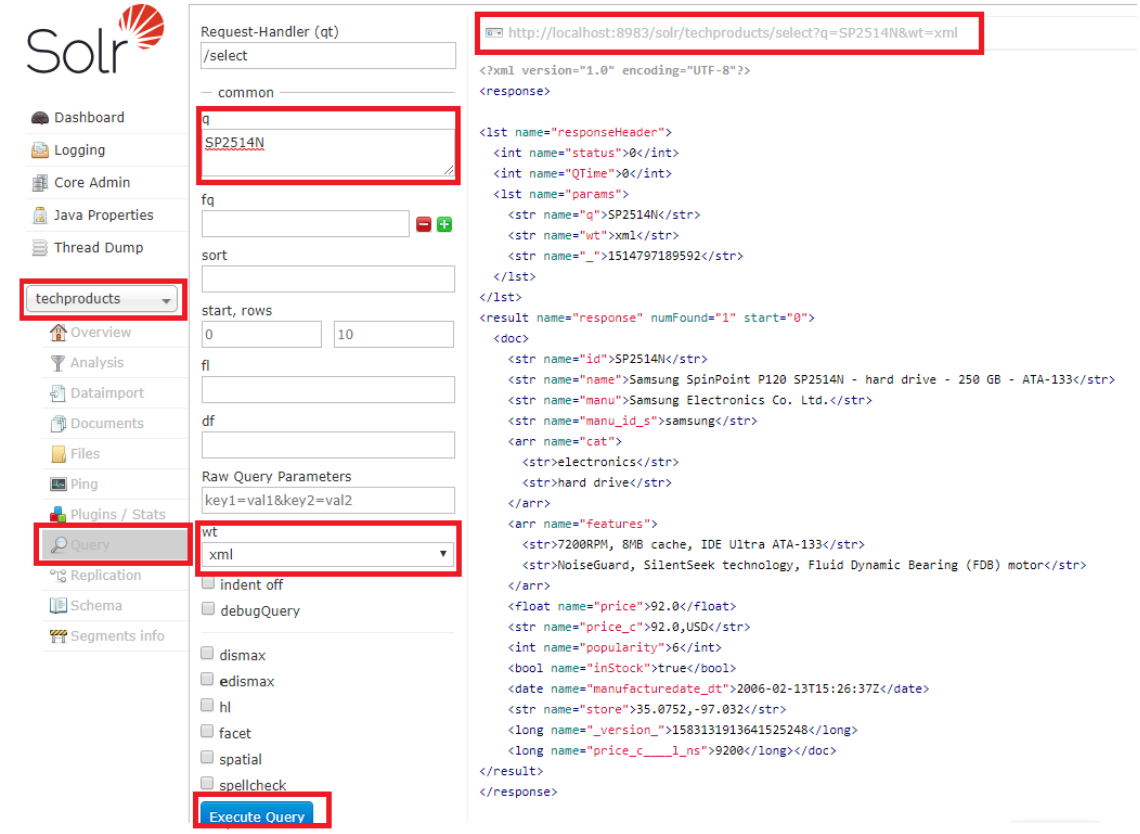
The following parameters are supported by standard query parser. We can configure them in `solrconfig.xml` :

Parameter	Behavior	Default value
`q`	Specifies a query using standard query syntax. This is a mandatory parameter for any request.	
`q.op`	Specifies the default operator for query expressions, which overrides the default operator configured inside the schema. Possible values are `AND` or `OR`.	
`df`	Specifies a default field, which overrides the default field definition inside the schema.	
`sow`	If this is set to `true`, it splits on white spaces.	`false`

Standard query parser response: The following is the sample response provided by the standard query parser when we search for `field id=SP2514N`

URL: `http://localhost:8983/solr/techproducts/select?q=SP2514N&wt=json`

In Solr admin console, while running a query to search a product with `id=SP2514N` displays the response as follows:



The response code is as follows:

```
{
 "responseHeader":{
 "status":0,
 "QTime":0,
 "params":{
 "q":"SP2514N",
 "wt":"json",
 "_":"1515346597997"}},
```

```
"response":{"numFound":1,"start":0,"docs":[
{
"id":"SP2514N",
"name":"Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133",
"manu":"Samsung Electronics Co. Ltd.",
"manu_id_s":"samsung",
"cat":["electronics",
"hard drive"],
"features":["7200RPM, 8MB cache, IDE Ultra ATA-133",
"NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing (FDB) motor"],
"price":92.0,
"price_c":"92.0,USD",
"popularity":6,
"inStock":true,
"manufacturedate_dt":"2006-02-13T15:26:37Z",
"store":"35.0752,-97.032",
"_version_":1583131913641525248,
"price_c____l_ns":9200}]
}}
```


In the same way, now the query `id=SP2514N`; and we need only two fields, `id` and `name`, in `response`.

URL: `http://localhost:8983/solr/techproducts/select?fl=id,name&q=SP2514N&wt=json`.

Response:


```
{
 "responseHeader":{
 "status":0,
 "QTime":17,
 "params":{
 "q":"SP2514N",
 "fl":"id,name",
 "wt":"json",
 "_":"1515346597997"}},
 "response":{"numFound":1,"start":0,"docs":[
 {
 "id":"SP2514N",
 "name":"Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133"}]
 }}
```


We can format the response by setting the `wt` parameter as `json`, `xml`, `xslt`, `javabin`, `geojson`, `python`, `php`, `phps`, `ruby`, `csv`, `velocity`, `smile`, or `xlsx`.

Searching terms for standard query parser

A query string to standard query parser contains terms and operators. There are two types of terms:

- **Single term**: A single word, such as `soccer` or `volleyball`
- **A phrase**: A group of words surrounded by double quotes, such as `apache solr`

We can combine multiple terms with Boolean operators to form complex queries.

Term modifiers

Solr supports many term modifiers that add flexibility or precision during searching. These term modifiers are wildcard characters, characters for making a search fuzzy, and so on.

Wildcard searches

The standard query parser supports two types of wildcard searches within a single term. They are single (`?`) and multiple (`*`) characters. They can be applied to single terms only, and not to search phrases. For example:

Wildcard search type	Special character	Search example
Single character (matches a single character)	`?`	Searching for a string `te?t` will match `test` and `text`.
Multiple characters (matches zero or more sequential characters)	`*`	Searching for string `tes*` will match `test`, `testing`, and `tester`. The wildcard characters can be used at the beginning, middle, or end of a term. For example, the string `te*t` will match `test` and `text`, and `*est` will match `pest` and `test`.

Fuzzy searches

In fuzzy searching, instead of matching exact terms, Solr searches terms that are likely similar to a specified term. The tilde (`~`) symbol is used at the end of a single word in fuzzy search. For example, to search for a term similar in spelling to roam, use a fuzzy search; `roam~` will match terms like roam, roams, and foam.

The `distance` parameter (optional) specifies the maximum number of modifications that take place between `0` and `2` . The default value is `2` . For example, searching for `roam~1` will search the terms such as roams and foam but not foams because it has a modification distance of `2` .

Proximity searching

Proximity searching searches for terms within a specific distance of each other. To implement a proximity search, specify a tilde (`~`) symbol with a numeric value at the end of a search phrase. For example, to search for `soccer` and `volleyball` within `20` words of each other in a document, do this:

```
"soccer volleyball"~20
```

The distance value specifies the term movements needed to match the specified phrase.

Range searches

In range searches, documents are searched based on a provided range (upper and lower bound) for a specific field. All the documents whose values for the specified field fall in a given range will be returned. The range search can be inclusive or exclusive of the range. For example, here the range query matches all documents whose `price` field has a value between `1000` and `50000` , both inclusive:

```
price:[1000 TO 50000]
```

Along with date and numerical fields, we can specify the range as words as well. For example:

```
title:{apache TO lucene}
```

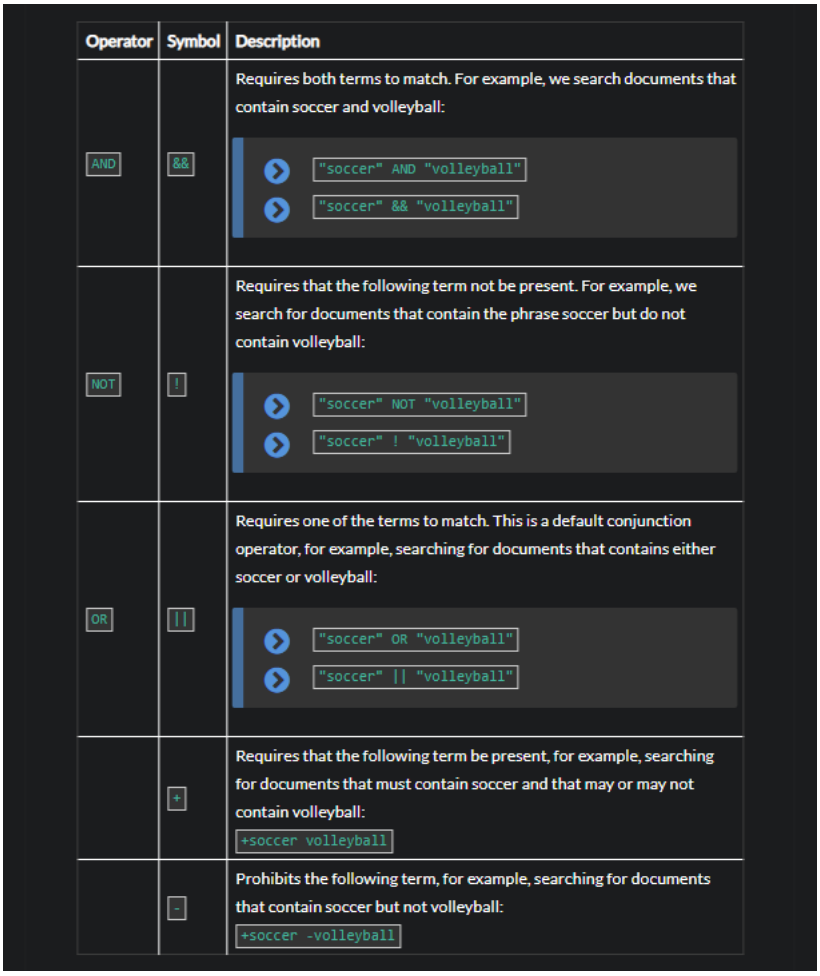
The preceding range configuration will search all documents whose titles are between `apache` and `lucene` , but not including `apache` and `lucene` :

- **Inclusive**: Square brackets `[ & ]` . Documents are searched by including the upper and lower bound.
- **Exclusive**: Curly brackets `{ & }` . Documents are searched between the upper and lower bound, but excluding the bounds.

Combining inclusive and exclusive is also possible, where one end is inclusive and the other is exclusive, for example, `price:[5 TO 20}` .

Boolean operators

Here is a list of Boolean operators supported by standard query parser:



Operator	Symbol	Description

AND	&&	Requires both terms to match. For example, we search documents that contain soccer and volleyball:
		`"soccer" AND "volleyball"`
		`"soccer" && "volleyball"`

NOT	!	Requires that the following term not be present. For example, we search for documents that contain the phrase soccer but do not contain volleyball:
		`"soccer" NOT "volleyball"`
		`"soccer" ! "volleyball"`

OR	||	Requires one of the terms to match. This is a default conjunction operator, for example, searching for documents that contains either soccer or volleyball:
		`"soccer" OR "volleyball"`
		`"soccer" || "volleyball"`

	+	Requires that the following term be present, for example, searching for documents that must contain soccer and that may or may not contain volleyball: `+soccer volleyball`
	-	Prohibits the following term, for example, searching for documents that contain soccer but not volleyball: `+soccer -volleyball`

Note

Please note that the Boolean operators `AND` and `NOT` must be specified in uppercase.

Escaping special characters

[] Solr treats these characters with a special meaning when they are used in a query:

```
+  -  &&  ||  !  ( )  { }  [ ]  ^  "  ~  *  ?  :  /
```

Using a backslash character(`\`) before the special character will notify Solr not to treat it as a special character but as a normal character. For example, for the search string `(1+1):2` the plus and parentheses, and colon can be ignored as special characters and will be treated as normal characters like this:

```
\(1\+1\)\:2
```

Grouping terms

Solr supports groups of clauses using parentheses to form subqueries that control the Boolean logic for a query.

This example will form a query that searches for either `soccer` or `volleyball` and `world cup` :

```
(soccer OR volleyball) AND "world cup"
```

Two or more Boolean operators can also be specified for a single field. Simply specify the Boolean clauses within parentheses. For example, this query will search for the field that contains both `soccer` and `volleyball` :

```
game:(+soccer +volleyball)
```

Dates and times in query strings

We need to use an appropriate date format whenever we are running a query against any date field. Search queries for exact date values will require quoting or escaping because `:` is listed as a special character for the parser:

```
createddate:2001-01-11T23\:45\:40.60Z

createddate:"2001-01-11T23:45:40.60Z"
```

```
createddate:[2001-01-11T23:45:40.60Z TO *]

createddate:[1999-12-31T23:45:40.60Z TO 2001-01-11T00:00:00Z]

timestamp:[* TO NOW]

publisheddate:[NOW-1YEAR/DAY TO NOW/DAY+1DAY]

createddate:[2001-01-11T23:45:40.60Z TO 2001-01-11T23:45:40.60Z+1YEAR]

createddate:[2001-01-11T23:45:40.60Z/YEAR TO 2001-01-11T23:45:40.60Z]
```

Adding comments to the query string

Comments can also be added to the query string. Solr supports C-style comments in the query string. Comments may be nested. For example:

```
soccer /* this is a simple comment for query string */ OR volleyball
```

The DisMax Query Parser

The DisMax query parser processes simple phrases (simple syntax). The DisMax query provides an interface that looks similar to Google. The DisMax Query Parser supports the simplified syntax of the Lucene query parser. Quotes can be used for grouping phrases. The DisMax Query Parser escapes all Boolean operators to simplify the query syntax, except the operators `AND` and `OR`, which can be used to determine mandatory and optional clauses.

Advantages

It produces syntax error messages. It also provides additional boosting queries, boosting functions, and filtering queries for search results.

DisMax query parser parameters

Apart from common parameters, the following is a list of all parameters supported by the DisMax Query Parser. All the default values for these parameters are configured in `solrconfig.xml`:

Parameter	Behavior

`q`	Specifies a query string with no special characters and treats Boolean operators `+` and `-` as term modifiers. Wildcard characters like `*` are not supported by this parameter:
	`q=apache`
	`q="Apache Lucene"`

`a.alt`	Defines an alternate query when the main query parameter `q` is not specified or blank. This parameter is mainly used to match all documents to get faceting counts.
`qf`	The `qf` parameter assigns a boost factor to a specific field to increase or decrease its importance in the query. For example, `qf="firstField^3.4 secondField thirdField^0.2"` assigns `firstField` a boost of `3.4`, keeps `secondField` with the default boost, and assigns `thirdField` a boost of `0.2`. These boost factors make matches in `firstField` much more significant than matches in `secondField`, which becomes much more significant than matches in `thirdField`.
`mm`	The minimum parameter defines the minimum number of optional clauses that must match. Words or phrases specified in the `q` parameter are considered as optional clauses unless they are preceded by a Boolean `AND` or `OR`. Possible values for the `mm` parameter are integer (positive and negative), percentage (positive and negative), simple, or multiple conditional expressions. The default value for mm is 100%, which means all clauses must match.
`pf`	The **Phrase Fields** (**pf**) parameter boosts the score of a document when all the terms in the `q` parameter appear in close proximity.
`ps`	The **Phrase Slop** (**ps**) parameter specifies the number of positions a term is required to move to match a phrase specified in a query with the `pf` parameter.
`qs`	Similar to the `ps` parameter for the `pf` parameter, the **Query Phrase Slop** (**qs**) defines the amount of slop on phrase queries explicitly included in the user's query string with the `qf` parameter.

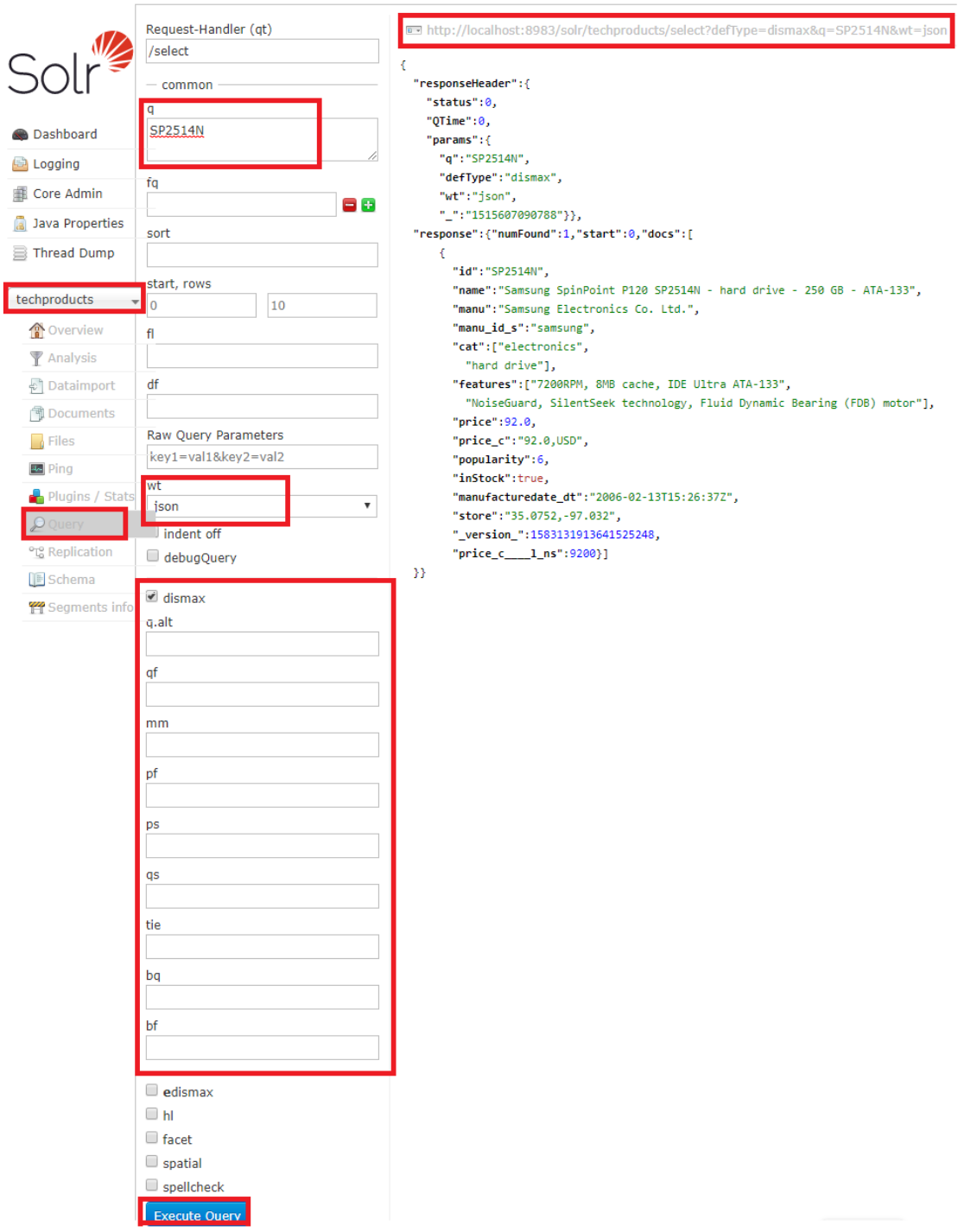
`tie`	The tie breaker parameter specifies a float value (which should be something much less than 1) to use as a tiebreaker when a query term is matched in more than one field in a document.
`bq`	The **Boost Query** (**bq**) parameter specifies an additional and optional query clause that will be added to the user's main query to influence the score. For example, if you want to add a relevancy boost for recent documents: `q=apache` `bq=date:[NOW/DAY-1YEAR TO NOW/DAY]` Multiple `bq` parameters can also be used when a query needs to be parsed as separate clauses with separate boosts.
`bf`	The **Boost Functions** (**bf**) parameter specifies functions (with optional boosts) that will be added to the user's main query to influence the score. Any function supported natively by Solr can be used along with a boost value. For example, if you want to show the most recent documents first, this is the syntax: `bf=recip(rord(createddate),1,100,100)`

We have seen all the configuration parameters for DisMax Query Parser and now we are ready to run a search using DisMax query parser.

The query ID is `SP2514N` and all DisMax Parser parameters are default, which means we are not specifying values in those parameters.

The URL is `http://localhost:8983/solr/techproducts/select?defType=dismax&q=SP2514N&wt=json`

The following shows the Solr admin console showing an example for DisMax query parser:



Response:


```
{
 "responseHeader":{
 "status":0,
 "QTime":0,
 "params":{
 "q":"SP2514N",
 "defType":"dismax",
 "wt":"json",
 "_":"1515607090788"}},
 "response":{"numFound":1,"start":0,"docs":[
 {
```

```
"id":"SP2514N",
"name":"Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133",
"manu":"Samsung Electronics Co. Ltd.",
"manu_id_s":"samsung",
"cat":["electronics",
"hard drive"],
"features":["7200RPM, 8MB cache, IDE Ultra ATA-133",
"NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing (FDB) motor"],
"price":92.0,
"price_c":"92.0,USD",
"popularity":6,
"inStock":true,
"manufacturedate_dt":"2006-02-13T15:26:37Z",
"store":"35.0752,-97.032",
"_version_":1583131913641525248,
"price_c____l_ns":9200}]
}}
```


Example: Retrieve only field `id` and `name` with `score`

URL: `http://localhost:8983/solr/techproducts/select?defType=dismax&q=SP2514N&fl=id,name,score`

Response:


```
{
 "responseHeader":{
 "status":0,
 "QTime":1,
 "params":{
 "q":"SP2514N",
 "defType":"dismax",
 "fl":"id,name,score"}},
 "response":{"numFound":1,"start":0,"maxScore":2.6953351,"docs":[
 {
 "id":"SP2514N",
 "name":"Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133",
 "score":2.6953351}]
 }}
```


Use `fl=*` to retrieve all the fields.

Example: Now we search for query `iPod`, assigning boosting to the `fields` features and `cat`.

URL: `http://localhost:8983/solr/techproducts/select?defType=dismax&q=iPod&qf=features^10.0+cat^0.5`

Response:


```
{
  "responseHeader":{
    "status":0,
    "QTime":1,
    "params":{
      "q":"iPod",
      "defType":"dismax",
      "qf":"features^10.0 cat^0.5"}},
  "response":{"numFound":1,"start":0,"docs":[
      {
        "id":"IW-02",
        "name":"iPod & iPod Mini USB 2.0 Cable",
        "manu":"Belkin",
        "manu_id_s":"belkin",
        "cat":["electronics",
          "connector"],
        "features":["car power adapter for iPod, white"],
        "weight":2.0,
        "price":11.5,
        "price_c":"11.50,USD",
        "popularity":1,
        "inStock":false,
        "store":"37.7752,-122.4232",
        "manufacturedate_dt":"2006-02-14T23:55:59Z",
        "_version_":1583131913695002624,
        "price_c____l_ns":1150}]
  }}
```


Example: Boost results that have a field that matches a specific value.

URL: `http://localhost:8983/solr/techproducts/select?defType=dismax&q=iPod&bq=cat:electronics^5.0`

In the same way, we can construct a URL for other parameters as well.

eDisMax Query Parser

The eDisMax Query Parser is an improved version of the DisMax Query Parser. Along with supporting all the features provided by DisMax Query Parser, it supports the following:

- Lucene query parser syntax
- Improved smart partial escaping in the case of syntax errors
- Improved proximity boosting by using word shingles
- Advanced stop word handling
- Improved boost function
- Pure negative nested queries
- We can specify which fields the end user is allowed to query, and specify to disallow direct fielded searches

eDisMax Query Parser Parameters: Along with the common parameters, we have these eDisMax Query Parser parameters:

Parameter	Behavior	Default Value
`sow`	Split on whitespace. Possible values are true and false. Once we set this to `true`, text analysis will be done for every individual whitespace-separated term.	False
`mm.autoRelax`	Relax the clauses in case of some of the clauses removed like stop words and search wont get impacted due to any clause removal. We need to take care when using the `mm.autoRelax` parameter because sometimes we may get unpredictable results. Possible values are `true` and `false`.	False
`boost`	A multivalued list of strings parsed as queries, with scores multiplied by the score from the main query for all matching documents.	
`lowercaseOperators`	Treats lowercase `and` and `or` the same as the operators `AND` and `OR`.	False
`pf2`	A multivalued list of fields with optional weights. It's similar to `pf`, but based on pairs of word shingles.	
`pf3`	A multivalued list of fields with optional weights, based on triplets of word shingles. It is similar to `pf`, except that instead of building a phrase per field out of all the words in the input, it builds a set of phrases for each field out of each triplet of word shingles.	
`ps`	The `ps` parameter specifies how many term positions the terms in the query can be off by to be considered a match on the phrase fields.	
`ps2`	Similar to `ps` but overrides the slop factor used for `pf2`. If not specified, `ps` is used.	
`ps3`	Similar to `ps` but overrides the slop factor used for `pf3`. If not specified, `ps` is used.	
`stopwords`	Tells Solr to disable `StopFilterFactory` configured in query analyzer. Possible values are `true` and `false`. False will disable `StopFilterFactory`.	True
`uf`	Specifies which schema fields the end user is allowed to explicitly query.	Allow all fields or `uf=*`

Examples: Searching for `music` or `camera` and boosting by popularity.

URL: `http://localhost:8983/solr/techproducts/select?defType=edismax&q=music+OR+camera&boost=popularity`

Response:

```
{
 "responseHeader":{
 "status":0,
 "QTime":1,
 "params":{
```

```
 "q":"music OR camera",
 "defType":"edismax",
 "boost":"popularity"}},
 "response":{"numFound":2,"start":0,"docs":[
 {
 "id":"9885A004",
 "name":"Canon PowerShot SD500",
 "manu":"Canon Inc.",
 "manu_id_s":"canon",
 "cat":["electronics",
 "camera"],
 "features":["3x zoop, 7.1 megapixel Digital ELPH",
 "movie clips up to 640x480 @30 fps",
 "2.0\" TFT LCD, 118,000 pixels",
 "built in flash, red-eye reduction"],
 "includes":"32MB SD card, USB cable, AV cable, battery",
 "weight":6.4,
 "price":329.95,
 "price_c":"329.95,USD",
 "popularity":7,
 "inStock":true,
 "manufacturedate_dt":"2006-02-13T15:26:37Z",
 "store":"45.19614,-93.90341",
 "_version_":1583131913780985856,
 "price_c____l_ns":32995},
 {
 "id":"MA147LL/A",
 "name":"Apple 60 GB iPod with Video Playback Black",
 "manu":"Apple Computer Inc.",
 "manu_id_s":"apple",
 "cat":["electronics",
 "music"],
 "features":["iTunes, Podcasts, Audiobooks",
 "Stores up to 15,000 songs, 25,000 photos, or 150 hours of video",
 "2.5-inch, 320x240 color TFT LCD display with LED backlight",
 "Up to 20 hours of battery life",
 "Plays AAC, MP3, WAV, AIFF, Audible, Apple Lossless, H.264 video",
 "Notes, Calendar, Phone book, Hold button, Date display, Photo wallet, Built-in games, JPEG photo playback, Upgradeable firmware,
USB 2.0 compatibility, Playback speed control, Rechargeable capability, Battery level indication"],
 "includes":"earbud headphones, USB cable",
 "weight":5.5,
 "price":399.0,
 "price_c":"399.00,USD",
 "popularity":10,
 "inStock":true,
 "store":"37.7752,-100.0232",
 "manufacturedate_dt":"2005-10-12T08:00:00Z",
 "_version_":1583131913706536960,
 "price_c____l_ns":39900}]
 }}
```


In the same way, we can configure all the eDisMax parser parameters and explore the search functionality. Response writer

---

The user who is searching is mainly interested in the search output/response. Rather than providing output in only a single format, if we allow them to select their choice of output/response format and return a response in that format, it will really make the user happy. The good news is that Solr provides various response writers for the end user's convenience.

Once the user runs a search, along with providing matching results, Solr provides a formatted and well-organized output result that becomes easy and attractive for the end user. Solr handles this through a response writer. Solr supports these response writers:

- JSON (default)
- Standard XML
- XSLT
- Binary
- GeoJSON
- Python
- PHP
- PHP serialized
- Ruby
- CSV
- Velocity
- Smile
- XLSX

We can select the response writer by providing an appropriate value to the `wt` parameter. These are the response writer values for `wt`:

Response writer	wt parameter value
JSON	`json`
Standard XML	`xml`
XSLT	`xslt`
Binary	`javabin`
GeoJSON	`geojson`
Python	`python`
PHP	`php`
PHP serialized	`phps`
Ruby	`ruby`
CSV	`csv`
Velocity	`velocity`
Smile	`smile`
XLSX	`xlsx`

Let's explore some of these response writers in detail.

JSON

JSON response writer converts results into JSON format. This is a default response writer for Solr, so if we do not set the `wt` parameter, the default output will be in JSON format. We can configure the `MIME type` for any response writer in the `solrconfig.xml` file. The default `MIME type` for JSON response writer is `application/json`; however, we can override it as per our search needs. For example, we can override the `MIME type` configuration in `techproducts solrconfig.xml`:

```
<queryResponseWriter name="json" class="solr.JSONResponseWriter">
 <!-- For the purposes of the tutorial, JSON responses are written as
 plain text so that they are easy to read in *any* browser.
 If you expect a MIME type of "application/json" just remove this override.
 -->
 <str name="content-type">text/plain; charset=UTF-8</str>
</queryResponseWriter>
```

JSON response writer parameters

This is a list of JSON response writer parameters that we need to configure to get a response in the expected format:

Parameter	Behavior	Value	Description
`json.nl`	Controls the output format of `NamedList`, where the order is more important than access by name. `NamedList` is currently used for field faceting data.	`flat` (default)	`NamedList` is represented as a flat array, with alternating names and values. **Input:** `NamedList("a"=1, "bar"="foo", null=3, null=null)` **Output:** `["a",1, "bar","foo", null,3, null,null]`
		`map`	`NamedList` can have optional keys and repeated keys. It preserves the order. **Input:** `NamedList("a"=1, "bar"="foo", null=3, null=null)` **Output:** `{"a":1, "bar":"foo", "":3, "":null}`
		`arrarr`	`NamedList` is represented as an array of two element arrays. **Input:** `NamedList("a"=1, "bar"="foo", null=3, null=null)` **Output:** `[["a",1], ["bar","foo"], [null,3], [null,null]]`
		`arrmap`	`NamedList` is represented as an array of JSON objects. **Input:** `NamedList("a"=1, "bar"="foo", null=3, null=null)` **Output:** `[{"a":1}, {"b":2}, 3, null]`
		`arrnv`	`NamedList` is represented as an array of name type value JSON objects. **Input:** `NamedList("a"=1, "bar"="foo", null=3, null=null)` **Output:** `[{"name":"a","type":"int","value":1}, {"name":"bar","type":"str","value":"foo"}, {"name":null,"type":"int","value":3}, {"name":null,"type":"null","value":null}]`
`json.wrf`	Adds a wrapper function around the JSON response. Useful in AJAX with dynamic script tags for specifying a JavaScript callback function.	`function`	

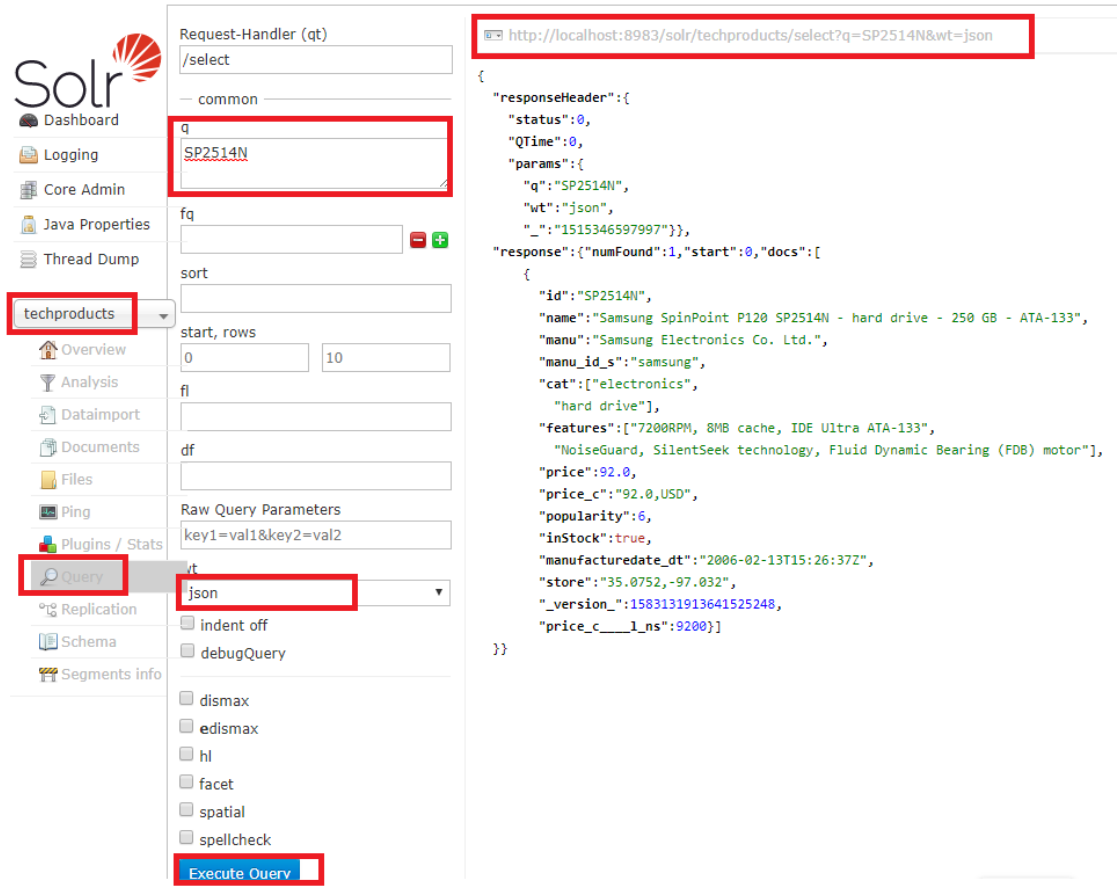
Example: Searching for `id=SP2514N`.

URL: `http://localhost:8983/solr/techproducts/select?q=SP2514N&wt=json.`

Response:


```
{
 "responseHeader":{
 "status":0,
 "QTime":1,
 "params":{
 "q":"SP2514N",
 "wt":"json",
 "_":"1514797189592"}},
 "response":{"numFound":1,"start":0,"docs":[
 {
 "id":"SP2514N",
 "name":"Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133",
 "manu":"Samsung Electronics Co. Ltd.",
 "manu_id_s":"samsung",
 "cat":["electronics",
 "hard drive"],
 "features":["7200RPM, 8MB cache, IDE Ultra ATA-133",
 "NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing (FDB) motor"],
 "price":92.0,
 "price_c":"92.0,USD",
 "popularity":6,
 "inStock":true,
 "manufacturedate_dt":"2006-02-13T15:26:37Z",
 "store":"35.0752,-97.032",
 "_version_":1583131913641525248,
 "price_c____l_ns":9200}]
 }
}
```


We can analyze response writer using the Solr console admin as well. Go to the Solr console admin, select **`techproducts`**, and click on the **`Query`** tab. Insert your text query in the `q` field, select the `wt` parameter as `json` and click on the **`Execute Query`** button at the bottom. The query URL and response output will be displayed as follows:



Solr console admin, response writer configuration, and output

Standard XML

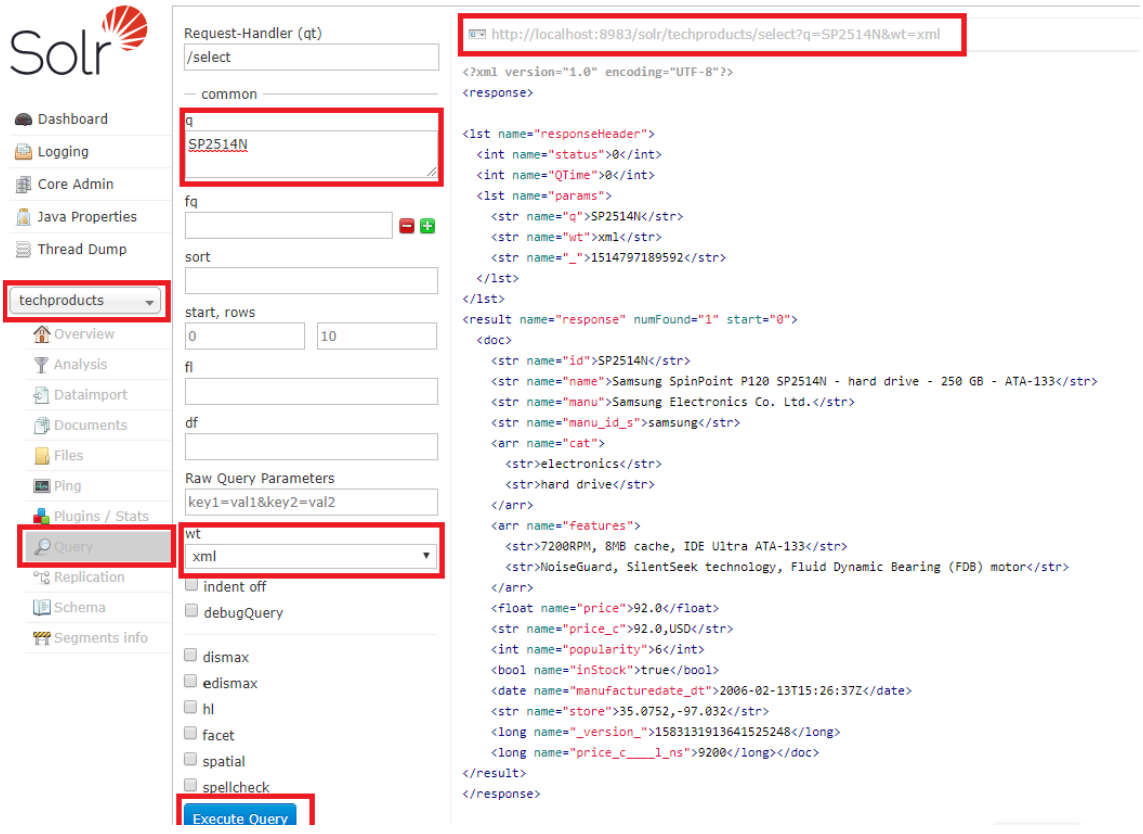
The standard XML response writer is the most common and usable response writer in Solr.

Standard XML response writer parameters:

Parameter	Behavior	Default Value
`version`	The `version` parameter determines the XML protocol used in the response. The advantage of setting this parameter is that the response format remains the same even if the Solr version gets an upgrade.	The default value is the latest supported one. The only currently supported version value is 2.2.
`stylesheet`	It includes a `<?xml-stylesheet type="text/xsl" href="..."?>` declaration in the XML response.	Solr does not return any style sheet declaration by default.
`indent`	Indenting the XML response for a more human-readable format.	By default, Solr will not indent the XML response.

Example: Searching for query `id=SP2514N` and retrieving a response in XML format.

Solr admin console, searching for product `id=SP2514N`, and retrieving the response in XML format:



URL: `http://localhost:8983/solr/techproducts/select?q=SP2514N&wt=xml`

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
 <int name="status">0</int>
 <int name="QTime">0</int>
 <lst name="params">
 <str name="q">SP2514N</str>
 <str name="wt">xml</str>
```

```
 <str name="_">1514797189592</str>
 </lst>
</lst>
<result name="response" numFound="1" start="0">
 <doc>
 <str name="id">SP2514N</str>
 <str name="name">Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133</str>
 <str name="manu">Samsung Electronics Co. Ltd.</str>
 <str name="manu_id_s">samsung</str>
 <arr name="cat">
 <str>electronics</str>
 <str>hard drive</str>
 </arr>
 <arr name="features">
 <str>7200RPM, 8MB cache, IDE Ultra ATA-133</str>
 <str>NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing (FDB) motor</str>
 </arr>
 <float name="price">92.0</float>
 <str name="price_c">92.0,USD</str>
 <int name="popularity">6</int>
 <bool name="inStock">true</bool>
 <date name="manufacturedate_dt">2006-02-13T15:26:37Z</date>
 <str name="store">35.0752,-97.032</str>
 <long name="_version_">1583131913641525248</long>
 <long name="price_c____l_ns">9200</long></doc>
</result>
</response>
```

CSV

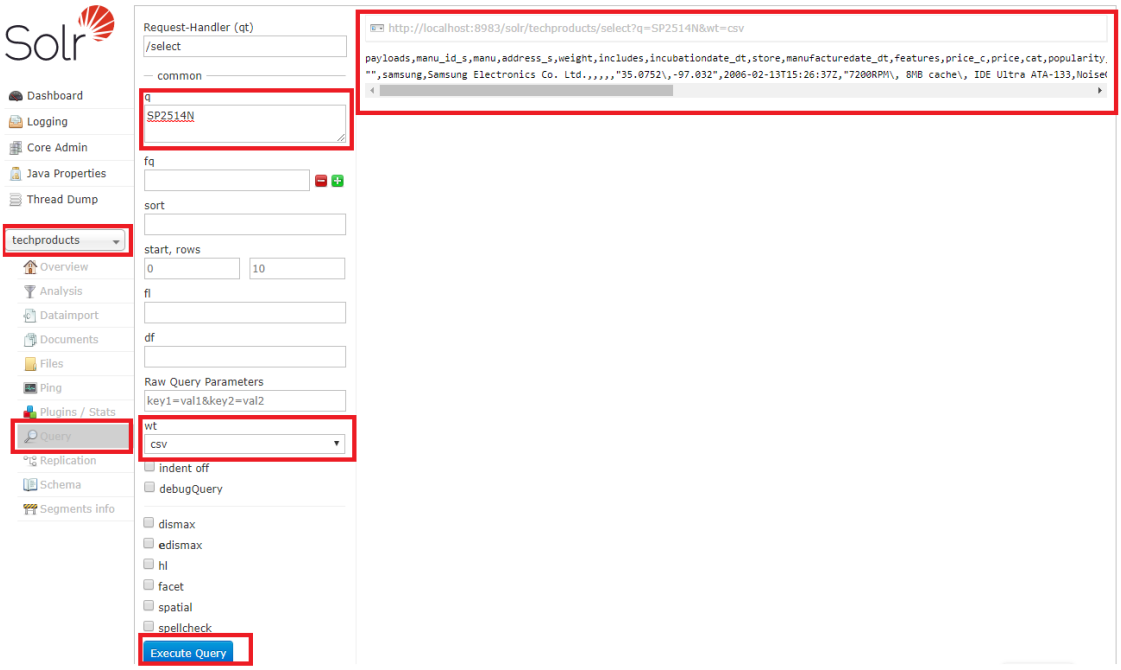
This returns the results in CSV format. Some information (like facet) will be excluded from the CSV response. The CSV response writer supports multi-valued fields as well as pseudo-fields, and the output of this CSV format is compatible with Solr's CSV update format.

CSV response writer parameters:

Parameter	Behavior	Default value
`csv.encapsulator`	Specifies a character to be used as an encapsulator in the response	`"`
`csv.escape`	Specifies the items to be escaped from the response	`None`
`csv.separator`	Specifies the separator for the CSV response	`,`
`csv.header`	Indicates whether to print header information in the CSV response or not	`true`
`csv.newline`	Used to start a new line from this parameter value	`\n`
`csv.null`	Specifies the value to be returned in the response instead of returning null	Zero-length string

Example: Searching for query `id=SP2514N` and retrieving the response in `.csv` format.

Solr admin console, searching for product `id=SP2514N`, and retrieving the response in `.csv` format:



URL: `http://localhost:8983/solr/techproducts/select?q=SP2514N&wt=csv`

Response:

```
payloads,manu_id_s,manu,address_s,weight,includes,incubationdate_dt,store,manufacturedate_dt,features,price_c,price,cat,popularity,r

"",samsung,Samsung Electronics Co. Ltd.,,,,,"35.0752\,-97.032",2006-02-13T15:26:37Z,"7200RPM\, 8MB cache\, IDE Ultra ATA-
133,NoiseGuard\, SilentSeek technology\, Fluid Dynamic Bearing (FDB) motor","92.0\,USD",92.0,"electronics,hard drive",6,Samsung
SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133,true,SP2514N,
```

Velocity

Solr supports a velocity response writer, which is used in Velocity UI to demonstrate some core search features: **faceting**, **highlighting**, **autocomplete**, and **geospatial** searching. velocity response writer is an optional plugin available in the `contrib/velocity` directory.

To use velocity response writer, we must include its `.jar` file and all dependencies in the `lib` folder, and configure in `solrconfig.xml` as follows:

```
<queryResponseWriter name="velocity" class="solr.VelocityResponseWriter">
 <str name="template.base.dir">${velocity.template.base.dir:}</str>
<!--
 <str name="init.properties.file">velocity-init.properties</str>
 <bool name="params.resource.loader.enabled">true</bool>
 <bool name="solr.resource.loader.enabled">false</bool>
 <lst name="tools">
 <str name="mytool">com.example.MyCustomTool</str>
 </lst>
-->
</queryResponseWriter>
```

Here, we have not configured the initialization and request parameters but as per our needs, we can configure them.

We have almost finished looking at search configuration components such as relevance, various query parsers, and response writers. Now Let's take a deep dive and and explore various search result operations: faceting, clustering, highlighting, and so on. Faceting

---

Faceting is the mechanism provided by Solr to categorize results in a meaningful arrangement on indexed fields. Using faceting, the end user will be provided with categorized results, along with a matching count for that search. Now the user can explore the search results, drill down to any result, and thus find an exactly matching result in which they are interested.

There are many types of faceting provided by Solr. Here is a list of faceting types that Solr currently supports:

- Range faceting
- Pivot (decision tree) faceting
- Interval faceting

We will explore these later in this lab. But to configure any faceting in Solr, first we have to configure the related parameters. So Let's understand faceting parameters first.

Common parameters

These are the common parameters for all types of faceting:

Parameter	Behavior	Default value
`facet`	Enable or disable faceting.	`false`
`facet.query`	Specifies a faceting query, which overrides Solr's default faceting query and returns a faceting count.	

Field-value faceting parameters

Field-value parameters are used to trigger faceting based on the indexed terms in a field. By default, all field-value faceting parameters can be specified on a per field basis with the syntax of `f.<fieldname>.facet.<parameter>`:

Parameter	Behavior	Default value
`facet.field`	Identifies a field that should be treated as a facet. At least one field must have this parameter; otherwise, none of the other field-value faceting parameters will have any effect.	
`facet.prefix`	Limits facet values to terms beginning with the string specified.	
`facet.contains`	Limits facet values to terms containing the string specified.	
`facet.contains.ignoreCase`	If `facet.contains` is used, the `facet.contains.ignoreCase` parameter causes cases to be ignored when matching the given substring against the candidate facet terms.	
`facet.limit`	Specifies the maximum number of constraint counts that should be returned for the facet fields. The possible values are positive and negative. Providing any negative value indicates that Solr will return an unlimited number of constraint counts.	`100`
`facet.sort`	Determines the ordering of the facet field constraints. Possible values are: `count`: Sorts constraints based on the count (high to low) `index`: Sorts constraints based on their index order The default sorting is based on the index, but if the limit parameter (`facet.limit`) is greater than zero, the default sorting will be the count.	
`facet.offset`	Allows paging through facet values. The offset defines how many of the top values to skip instead of returning later facet values.	`0`
`facet.mincount`	Specifies the minimum counts required for a facet field to be included in the response. If a field's counts are less than the minimum, the field's facet is not returned.	`0`
`facet.missing`	Specifies whether or not the count of all matching documents that do not have any values is to be returned in the facet's field.	`false`

`facet.method`	Specifies the type of algorithm or method Solr should use when faceting a field. The available methods in Solr are: `enum` : Iterates over all the terms in the index, calculating a set intersection with those terms and the query. This method is faster for fields that contain fewer values. `fc` : Iterates over documents that match the query and finds the terms within those documents. The fc method is faster for fields that contain many unique values. `fcs` : Performs per-segment field faceting for single-valued string fields. This method performs better faceting if the index is changing constantly. It also accepts a threads local param, which can speed up faceting.	`fc`
`facet.enum.cache.minDf`	Specifies the minimum number of documents required to match a term before `filterCache` should be used for that term. The default is `0` , which means `filterCache` should always be used.	`0`
`facet.exists`	To cap facet counts by 1, specify `facet.exists=true` . This parameter can be used with `facet.method=enum` or when it's omitted. It can be used only on on-trie fields (such as strings). It may speed up facet counting on large indices and/or high-cardinality facet values.	`false`
`facet.excludeTerms`	Removes the specified terms from facet counts but keeps them in the index.	
`facet.threads`	Specifies the number of threads to execute for faceting the fields in parallel. Specifying the thread count as 0 will not create any threads, and only the main request thread will be used. Specifying a negative number of threads will create up to `Integer.MAX_VALUE` threads.	

Range faceting

Range faceting can be done on date fields and numeric fields.

Range faceting parameters:

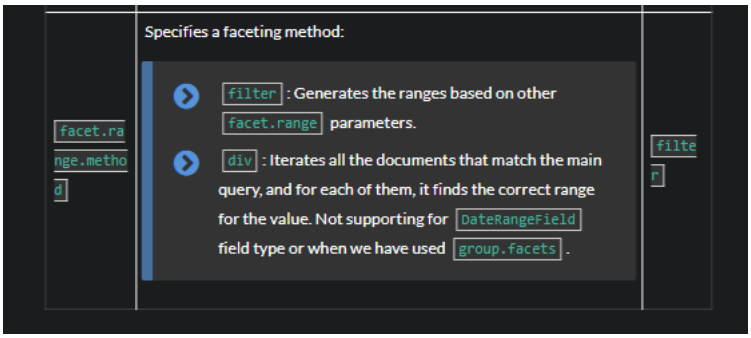
Parameter	Behavior	Default value

`facet.range`	Specifies the field for which Solr should create range facets. For example:
	`facet.range=salary&facet.range=rank`
	`facet.range=createdDate`	

`facet.range.start`	Specifies from where (lower bound) the range starts. For example:
	`f.salary.facet.range.start=10000.0&f.rank.facet.range.start=1`
	`f.createdDate.facet.range.start=NOW/DAY-30DAYS`	

`facet.range.end`	Specifies where (upper bound) the range ends. For example:
	`f.salary.facet.range.end=100000.0&f.rank.facet.range.end=50`
	`f.createdDate.facet.range.end=NOW/DAY+30DAYS`	

`facet.range.gap`	The size of each range will be added to the `lower` bound successively until the `upper` bound is reached.	
`facet.range.hardend`	A Boolean parameter that specifies how Solr should handle cases where `facet.range.gap` does not divide evenly between `lower` bound and `upper` bound. If it is `true`, the last range constraint will have the `facet.range.end` value as an `upper` bound. If `false`, the last range will have the smallest possible `upper` bound greater than `facet.range.end` such that the range is the exact width of the specified range gap.	`false`
`facet.range.include`	Determines how to compute range faceting between the lower bound and upper bound. Possible values are: `lower`: All gap-based ranges include their `lower` bound `upper`: All gap-based ranges include their `upper` bound `edge`: The first and last gap ranges include their `edge` bounds (`lower` for the first one, `upper` for the last one) even if the corresponding `upper` / `lower` option is not specified `outer`: The `before` and `after` ranges will be inclusive of their bounds even if the first or last ranges already include those boundaries `all`: Includes all options— `lower`, `upper`, `edge`, and `outer`.	
`facet.range.other`	Specifies that, in addition to the counts for each range between `lower` bound and `upper` bound, counts should be computed for these options as well: `before`: All records with field values lower than `lower` bound of the first range `after`: All records with field values greater than the `upper` bound of the last range `between`: All records with field values between the start and end bounds of all ranges `none`: Do not compute any counts `all`: Compute counts for `before`, `between`, and `after`	



Example: Search query for `iPod` with faceting enabled and range for the field price from `1000` to `100000`

URL: `http://localhost:8983/solr/techproducts/select?defType=edismax&q=ipod&fl=id,name,price&facet=true&facet.range=price&facet.range.start=10&facet.range.end=20&facet.range.gap=5`

Response:


```
{
 "responseHeader":{
 "status":0,
 "QTime":1,
 "params":{
 "facet.range":"price",
 "q":"ipod",
 "defType":"edismax",
 "facet.range.gap":"5",
 "fl":"id,name,price",
 "facet":"true",
 "facet.range.start":"10",
 "facet.range.end":"20"}},
 "response":{"numFound":3,"start":0,"docs":[
 {
 "id":"IW-02",
 "name":"iPod & iPod Mini USB 2.0 Cable",
 "price":11.5},
 {
 "id":"F8V7067-APL-KIT",
 "name":"Belkin Mobile Power Cord for iPod w/ Dock",
 "price":19.95},
 {
 "id":"MA147LL/A",
 "name":"Apple 60 GB iPod with Video Playback Black",
 "price":399.0}]
 },
 "facet_counts":{
 "facet_queries":{},
 "facet_fields":{},
 "facet_ranges":{
 "price":{
 "counts":[
 "10.0",1,
 "15.0",1],
 "gap":5.0,
 "start":10.0,
 "end":20.0}},
 "facet_intervals":{},
 "facet_heatmaps":{}}}
```


Pivot faceting

Pivot faceting is similar to pivot tables in the latest spreadsheets. Pivot faceting provides a facility to generate an aggregate summary from fetched faceting results on multiple fields:

Parameter	Behavior	Default value
`facet.pivot`	Specify the field on which you want to apply pivoting	
`facet.pivot.mincount`	Specify the minimum number of documents that need to match in order for the facet to be included in the results	`1`

Example: In our `techproducts`, we need the stock availability based on the popularity of a category

URL: `http://localhost:8983/solr/techproducts/select?q=*:*&facet.pivot=cat,popularity,inStock&facet.pivot=popularity,cat&facet=true&facet.field=cat&facet.limit=5&rows=0&facet.pivot.minco`

Response:


```
{
"facet_counts":{
 "facet_queries":{},
 "facet_fields":{
 "cat":[
 "electronics",14,
 "currency",4,
 "memory",3,
 "connector",2,
 "graphics card",2]},
 "facet_dates":{},
 "facet_ranges":{},
 "facet_pivot":{
 "cat,popularity,inStock":[{
 "field":"cat",
 "value":"electronics",
 "count":14,
 "pivot":[{
 "field":"popularity",
 "value":6,
 "count":5,
 "pivot":[{
 "field":"inStock",
 "value":true,
 "count":5}]}]
}]}}}
```


Interval faceting

Interval faceting is similar to range faceting, but it allows us to set variable intervals and count the number of documents that have values within those intervals in the specified field. Interval faceting is likely to be better with multiple intervals for the same fields, while a facet query is likely to be better in environments where a filter cache is more effective:

Parameter	Behavior	Default value
`facet.interval`	To specify a field where we want to apply the interval. It can be used multiple times for multiple fields in a single request. For example: `facet.interval=price&facet.interval=popularity`	
`facet.interval.set`	To specify a set of intervals for the field. It can be specified multiple times to indicate multiple intervals. For example: `f.price.facet.interval.set=[0,10]&f.price.facet.interval.set=(10,100]` (1,100) -> include values greater than 1 and lower than 100 [1,100) -> include values greater or equal to 1 and lower than 100 [1,100] -> include values greater or equal to 1 and lower or equal to 100	

Example: Faceting query for field `price >=10` and `price < 20`

URL: `http://localhost:8983/solr/techproducts/select?q=*:*&facet=true&facet.interval=price&f.price.facet.interval.set=[10,20)`

Response:

```
"facet_counts":{
 "facet_queries":{},
 "facet_fields":{},
 "facet_ranges":{},
 "facet_intervals":{
 "price":{
 "[10,20)":2}},
 "facet_heatmaps":{}
 }
```

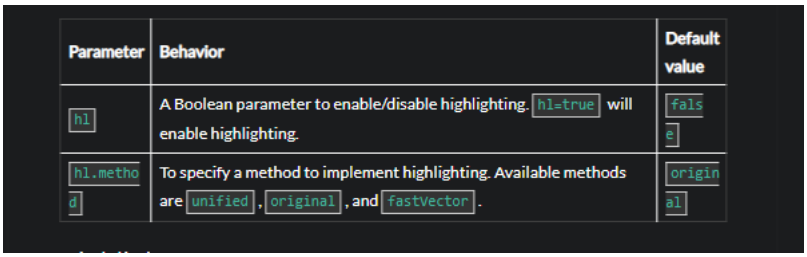
Highlighting

Solr supports a feature called **highlighting** that helps end users who are running a query to scan results quickly. Providing a matching term in bold and highlighted the format makes it an extremely satisfying experience for the user. With highlighting, the user can quickly determine the terms they are searching for or make a decision that the provided results do not match their expectations, and lets them move to next query.

Solr comes with a great configuration for highlighting. There are many parameters for **fragment sizing**, **formatting**, **ordering**, **backup.alternate behavior**, and **categorization**. Fragments or snippets are parts of the response that contain matching terms.

Highlighting parameters

Solr provides a large list for highlighting fragments. The following are the basic parameters required to start highlighting:



Parameter	Behavior	Default value
`hl`	A Boolean parameter to enable/disable highlighting. `hl=true` will enable highlighting.	`false`
`hl.method`	To specify a method to implement highlighting. Available methods are `unified`, `original`, and `fastVector`.	`original`

Highlighter

Highlighter is nothing but a highlighting implemented method that actually performs the activity. There are three methods available for highlighting. They are `unified`, `original`, and `fastVector`. To implement highlighting, first we need to specify one method to `hl.method`. If we do not select any method, the default original method performs the activity.

There are many parameters supported by highlighters. Sometimes, the implementation details and semantics will be a bit different, so we can't expect identical results when switching highlighters. Normally, highlighter selection is done via the `hl.method` parameter, but we can also explicitly configure an implementation by class name in `solrconfig.xml`. Let's explore highlighters in detail.

Unified highlighter (hl.method=unified)

Unified highlighter is the new highlighter from Solr 6.4. This is the most flexible highlighter and supports the most common highlighting parameters. It can handle any query accurately, even `SpanQueries`. The greatest benefit of using this highlighter is that we can add more configurations to speed up highlighting on large data documents. We can also add multiple configurations on a per field basis.

Original highlighter (hl.method=original)

Original highlighter is the default highlighter, also known as **standard highlighter** or **default highlighter**. The advantage of this highlighter is its capability of highlighting any query accurately and efficiently, like `unified` highlighter, but it is very slow compared to `unified` highlighter.

The `original` highlighter is much slower at highlighting on large text fields or complex text analysis because it reanalyzes the original text at query time. It supports full-term vectors, but compared to `unified` highlighter and `fastVector` highlighter, it is very slow. Also it does not have a breakiterator-based fragmenter, which can cause problems in some languages.

FastVector highlighter (hl.method=fastVector)

FastVector Highlighter (**FVH**) is faster than `original` highlighter because it skips the analysis step when generating fragments. Sometimes, FVH is not able to highlight some of the fields; in such cases, it will do a conjunction with the `original` highlighter to match the requirement. For such cases, we need to set `hl.method=original` and `f.yourTermVecField.hl.method=fastVector` for all fields that should use the FVH.

Boundary scanners

Sometimes, `fastVector` highlighter will truncate highlighted words, so the output after highlighting may be incomplete or improper. To resolve this issue, we need to configure a boundary scanner in `solrconfig.xml`. There are two types of boundary scanners available in Solr. We have to specify a boundary scanner using the parameter `hl.boundaryScanner`.

The breakIterator boundary scanner

The `breakIterator` boundary scanner scans term boundaries by considering the language (`hl.bs.language`) and boundary type (`hl.bs.type`) and provides expected, accurate, and complete output without any loss of characters. It is used most often. To implement the `breakIterator` boundary scanner, we need to add the following code snippet to the highlighting section in the `solrconfig.xml` file:

```
<boundaryScanner name="breakIterator" class="solr.highlight.BreakIteratorBoundaryScanner">
 <lst name="defaults">
 <str name="hl.bs.type">WORD</str>
 <str name="hl.bs.language">en</str>
 <str name="hl.bs.country">US</str>
 </lst>
</boundaryScanner>
```

Possible values for the `hl.bs.type` parameter are `WORD`, `LINE`, `SENTENCE`, and `CHARACTER`.

The simple boundary scanner

The simple boundary scanner scans term boundaries by the specified maximum character value (`hl.bs.maxScan`) and common delimiters such as punctuation marks (`hl.bs.chars`). To implement it, we need to add the following code snippet to the highlighting section in `solrconfig.xml` :

```
<boundaryScanner name="simple" class="solr.highlight.SimpleBoundaryScanner" default="true">
 <lst name="defaults">
 <str name="hl.bs.maxScan">10</str >
 <str name="hl.bs.chars">.,!?\t\n</str &gt;
 </lst >
</boundaryScanner>
```

Example: Querying for `ipod` , highlighting for the field `name` using `fastVector` highlighter

URL: `http://localhost:8983/solr/techproducts/select?hl=true&hl.method=fastVector&q=ipod&hl.fl=name&fl=id,name,cat`

Response:

```
{
 "responseHeader":{
 "status":0,
 "QTime":4,
 "params":{
 "q":"ipod",
 "hl":"true",
 "fl":"id,name,cat",
 "hl.method":"fastVector",
 "hl.fl":"name"}},
 "response":{"numFound":3,"start":0,"docs":[
 {
 "id":"IW-02",
 "name":"iPod & iPod Mini USB 2.0 Cable",
 "cat":["electronics",
 "connector"]},
 {
 "id":"F8V7067-APL-KIT",
 "name":"Belkin Mobile Power Cord for iPod w/ Dock",
 "cat":["electronics",
 "connector"]},
 {
 "id":"MA147LL/A",
 "name":"Apple 60 GB iPod with Video Playback Black",
 "cat":["electronics",
 "music"]}]
 },
 "highlighting":{
 "IW-02":{
 "name":["<em>iPod</em> & <em>iPod</em> Mini USB 2.0 Cable"]},
 "F8V7067-APL-KIT":{
 "name":["Belkin Mobile Power Cord for <em>iPod</em> w/ Dock"]},
 "MA147LL/A":{
 "name":["Apple 60 GB <em>iPod</em> with Video Playback Black"]}}}
```

The highlighting section includes the ID of each document and the field that contains the highlighted portion. Here we have used the `hl.fl` parameter to say that we want query terms highlighted in the `name` field. When there is a match to the query term in that field, it will be included for each document ID in the list. In the same way, we can explore highlighting more by configuring different parameters.

Summary

In this lab, we learned the concept of relevance and its terms: Precision and Recall. Then we looked at the velocity search UI. We saw the common parameters for various query parsers and explored each query parser (standard, DisMax, and eDisMax) in detail. After that, we looked at various response writers in detail: JSON, standard XML, CSV, and velocity response writer. We also explored Solr term modifiers, wildcard parameters, fuzzy search, proximity search, and range search.

We looked at all Boolean operators. Then we learned about various faceting parameters and faceting types such as range, pivot, and interval faceting. At the end, we saw Solr highlighting mechanisms, parameters, highlighters, and boundary scanners.

In the next lab, or rather the second part of this lab, we will learn more search functionalities such as spell checking, suggester, pagination, result grouping and clustering, and spatial search.