Copyright

Copyright © 1997-2009 by Fast Search & Transfer ASA (“FAST”). Some portions may be copyrighted by FAST’s licensors. All rights reserved. The documentation is protected by the copyright laws of Norway, the United States, and other countries and international treaties. No copyright notices may be removed from the documentation. No part of this document may be reproduced, modified, copied, stored in a retrieval system, or transmitted in any form or any means, electronic or mechanical, including photocopying and recording, for any purpose other than the purchaser’s use, without the written permission of FAST. Information in this documentation is subject to change without notice. The software described in this document is furnished under a license agreement and may be used only in accordance with the terms of the agreement.

Trademarks

FAST ESP, the FAST logos, FAST Personal Search, FAST mSearch, FAST InStream, FAST AdVisor, FAST Marketrac, FAST ProPublish, FAST Sentimeter, FAST Scope Search, FAST Live Analytics, FAST Contextual Insight, FAST Dynamic Merchandising, FAST SDA, FAST MetaWeb, FAST InPerspective, NXT, FAST Unity, FAST Radar, RetrievalWare, AdMomentum, and all other FAST product names contained herein are either registered trademarks or trademarks of Fast Search & Transfer ASA in Norway, the United States and/or other countries. All rights reserved. This documentation is published in the United States and/or other countries.

Sun, Sun Microsystems, the Sun Logo, all SPARC trademarks, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Netscape is a registered trademark of Netscape Communications Corporation in the United States and other countries.

Microsoft, Windows, Visual Basic, and Internet Explorer are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Red Hat is a registered trademark of Red Hat, Inc.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

AIX and IBM Classes for Unicode are registered trademarks or trademarks of International Business Machines Corporation in the United States, other countries, or both.

HP and the names of HP products referenced herein are either registered trademarks or service marks, or trademarks or service marks, of Hewlett-Packard Company in the United States and/or other countries.

Remedy is a registered trademark, and Magic is a trademark, of BMC Software, Inc. in the United States and/or other countries.

XML Parser is a trademark of The Apache Software Foundation.

All other company, product, and service names are the property of their respective holders and may be registered trademarks or trademarks in the United States and/or other countries.

Restricted Rights Legend

The documentation and accompanying software are provided to the U.S. government in a transaction subject to the Federal Acquisition Regulations with Restricted Rights. Use, duplication, or disclosure of the documentation and software by the government is subject to restrictions as set forth in FAR 52.227-19 Commercial Computer Software-Restricted Rights (June 1987).
Contact Us

Web Site
Please visit us at: http://www.fastsearch.com/

Contacting FAST

FAST
Cutler Lake Corporate Center
117 Kendrick Street, Suite 100
Needham, MA 02492 USA
Tel: +1 (781) 304-2400 (8:30am - 5:30pm EST)
Fax: +1 (781) 304-2410

Technical Support and Licensing Procedures

Technical support for customers with active FAST Maintenance and Support agreements, e-mail: fasttech@microsoft.com

For obtaining FAST licenses or software, contact your FAST Account Manager or e-mail: fastcsrv@microsoft.com

For evaluations, contact your FAST Sales Representative or FAST Sales Engineer.

Product Training

E-mail: fastuniv@microsoft.com

To access the FAST University Learning Portal, go to: http://www.fastuniversity.com/

Sales

E-mail: sales@fastsearch.com
Contents

Preface........................................................................................................................................ii
Copyright....................................................................................................................................ii
Contact Us...................................................................................................................................iii

Chapter 1: Query parameters overview...................................................................................9
Query parameters.........................................................................................................................10
Basic Query Parameters............................................................................................................13
  Define query elements...............................................................................................................13
  Define character encoding........................................................................................................13
  Working with other formats than utf-8....................................................................................14
  Maximum number of results displayed on a page.................................................................14
  Using offsets.............................................................................................................................15
  Result view................................................................................................................................16
Language and linguistics query features....................................................................................16
  Define language for the query string.......................................................................................16
  Define language scope for query..............................................................................................17
  Lemmatization for queries against non-default composite fields..........................................18
  Enable synonym processing......................................................................................................18
  Phrase detection and spell check..............................................................................................19
  Simple spell check and phrases.................................................................................................20
  Anti-phrasing.............................................................................................................................20
  Extended wildcard characters....................................................................................................21
  Pattern search parameters..........................................................................................................22
  Automatic query re-submit if no hits..........................................................................................24
Find similar features.....................................................................................................................25
  Type of find similar query...........................................................................................................25
  Document similarity vector reference.......................................................................................26
  Query by example.......................................................................................................................26
  Sorting of find similar query results............................................................................................27
Geo search...................................................................................................................................27
  Sorting by geographical distance (geo-sorting).........................................................................28
  Sorting/filtering by geographical distance (geo-filtering)...........................................................28
  Geo search query parameters.......................................................................................................28
  Handling query results with geo information...........................................................................30
  Enable geo search in the back-end.............................................................................................31
  Enable geo search in SFE............................................................................................................31
  Performing a search for geo data..............................................................................................31
Search within collection...................................................................................................................31
Search within category....................................................................................................................32
Categorization and navigation.................................................................32
Result clustering........................................................................................32
Navigation parameters...............................................................................35
Filter Navigation Content........................................................................37
Control navigator performance..................................................................37
Navigator display and performance parameters........................................38
Navigation on Matching Scopes (Scope Navigation).................................42
Result Sorting and Ranking features..........................................................43
Sort by field, geographical distance or rank profile.....................................43
Multi-level Sorting......................................................................................44
Single-level field Sorting............................................................................45
Random Sorting..........................................................................................46
Sort Direction..............................................................................................47
Rank Profile Sorting Limitations.................................................................48
The Rank value returned in query results....................................................48
Field Collapsing..........................................................................................49
Field Collapsing (without document removal)............................................49
Field Collapsing (including document removal)...........................................49
Dynamic Duplicate Removal.......................................................................51
Proximity relevance features......................................................................54
Index Side Proximity boost................................................................--------54
Enable Result Side Proximity boost.........................................................54
Number of hits for proximity boost...........................................................54
Document freshness boost.........................................................................55
Select the time base for calculating the freshness boost............................55
Query and result processing parameters....................................................56
Select Query Processing Pipeline...............................................................56
Select Result Processing Pipeline...............................................................57
Enable document level security..................................................................57
Cache lines.................................................................................................59

Chapter 2: FQL - FAST Query Language.................................................61
Terminology...............................................................................................62
The Field/Scope Specification.................................................................62
Default Field/Scope Specification..............................................................62
Addressing Fields and Composite Fields....................................................63
Addressing scope fields............................................................................63
Addressing text scopes............................................................................64
Addressing numeric scopes......................................................................65
Addressing Scope and Entity Attributes....................................................65
Returning Scope.......................................................................................66
Search for Scope Existence.......................................................................66
Querying recursive scopes.......................................................................66
Scope specification examples....................................................................67
Chapter 3: Query Transformations ................................................. 101
  Query Transformations Principles.......................................................... 102
  Query format considerations........................................................................ 102
  Query Transformation Feedback.............................................................. 102
    Query Transformation for Anti Phrasing................................................. 103
    Query Transformations for Find Similar................................................ 103
    Query Transformations for Synonyms.................................................... 104
    Query Transformations for Lemmatization............................................. 104
  Phrase detection and Spell check.............................................................. 105

Chapter 4: Query Evaluation Optimizations and Approximations. 107
  Proximity Operator Evaluation............................................................... 108
  Lemmatization......................................................................................... 108
  Wildcard Query Evaluation..................................................................... 108
  Complex Scope Queries.......................................................................... 108
    Typing of Numeric Scopes.................................................................... 108
    Querying recursive scopes................................................................... 108
    Implicit phrasing.................................................................................. 109
  Increase QRserver stack size.................................................................. 109
A query typically consists of two main components, a query string and a set of query parameters.

The query string consists of query terms that comprise the FAST Query Language. You can combine the query terms with boolean expressions, FQL operators and parentheses. You can also specify the scope of the query by means of fields, scope fields or composite fields. The query string is also defined as one of the query parameters. Use the query parameters to specify how the query and its result should be processed and transformed, for example by applying spell check.

**Note:** The programmatic APIs and the HTTP GET query interface support the same set of query parameters, but there exist naming differences between the two API implementations.

**Note:** Specifying an option or parameter not recognized by FAST ESP will not return an error. Incorrect input syntax must be dealt with by custom made error messages.
Query parameters

Use these query parameters to control the processing and transformation of the query and its result.

The APIs provide separate namespaces for query parameters. In the Java/.NET APIs these are provided through the use of the BaseParameter class. The C++ API uses a constant value namespace. By using the API namespace you will get a parameter/value syntax check in the API methods.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>query.setParameter(new SearchParameter(BaseParameter.NAVIGATORS, &quot;price_nav,name_nav,date_nav&quot;));</td>
</tr>
<tr>
<td>.NET</td>
<td>query.setParameter(BaseParameter.NAVIGATORS, &quot;price_nav,name_nav,date_nav&quot;);</td>
</tr>
<tr>
<td>C++</td>
<td>par = my_query_fac-&gt;create_parameter (esp::search::parameters::NAVIGATORS, &quot;price_nav,name_nav,date_nav&quot;);</td>
</tr>
</tbody>
</table>

You can also use the parameters as defined on the HTTP Query Interface. In that case the parameter names and values are transparent to the API and copied directly as query parameters in the HTTP query string.

Note: Some query parameters are currently not supported within the API namespace. These are indicated with lowercase names; use the HTTP parameter syntax instead. For example: `query.setParameter(new SearchParameter("rpf_clustering:threshold", "0.5");` (Java).

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLUSTERING</td>
<td>rpf_clustering:enabled</td>
<td>Enables result-side clustering (supervised and unsupervised).</td>
</tr>
<tr>
<td></td>
<td>rpf_clustering:root</td>
<td>'rpf_clustering:root' is only applicable for the HTTP query interface (implicit set by the API). See section on Define clustering root.</td>
</tr>
<tr>
<td></td>
<td>rpf_clustering:labels</td>
<td>Refer to section Advanced Clustering Parameters for details on the individual clustering parameters.</td>
</tr>
<tr>
<td></td>
<td>rpf_clustering:size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rpf_clustering:sorting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rpf_clustering:threshold</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rpf_clustering:treesize</td>
<td></td>
</tr>
<tr>
<td>No corresponding base parameter</td>
<td>clines</td>
<td>Indicates an additional number of cache lines to cache. See section on cache lines for more information.</td>
</tr>
<tr>
<td>COLLAPSING</td>
<td>collapse</td>
<td>Enables field collapsing. See section on Field Collapsing.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>qtf_freshnessboost:datetime</td>
<td>Controls the Freshness Relevance Boost feature. See section on Freshness Boost.</td>
</tr>
<tr>
<td>DUPREM_SLOT1</td>
<td>rff_ddr:slot1</td>
<td></td>
</tr>
<tr>
<td>DUPREM_SLOT2</td>
<td>rff_ddr:slot2</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>HTTP Interface</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ENCODING</td>
<td>encoding</td>
<td>Defines character encoding used to encode the query string, and also desired for the result fields. See section on Define Character Encoding.</td>
</tr>
<tr>
<td>FUZZY</td>
<td>qtf_tokenize:fuzzy</td>
<td>Enables pattern matching and fuzzy spelling capabilities on queries. See section on Pattern search parameters.</td>
</tr>
<tr>
<td>FUZZY_THRESHOLD</td>
<td>qtf_tokenize:fuzzythreshold</td>
<td></td>
</tr>
<tr>
<td>FUZZYCUTOFF</td>
<td>qtf_tokenize:fuzzycutoff</td>
<td></td>
</tr>
<tr>
<td>GENERIC_PARAMETER</td>
<td></td>
<td>Specifies query parameters that are not available in the query object, i.e., query parameters that are not directly exposed in the FAST Web Services. <strong>Note:</strong> This parameter only applies to FAST Web Services.</td>
</tr>
<tr>
<td>HITS</td>
<td>hits</td>
<td>Specifies the maximum number of result hits returned per query. See section on Maximum Records (hits) Displayed on Each Page.</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>language</td>
<td>Defines the target language for the query (ISO 639-1 coding). See section on Define Language for the query string.</td>
</tr>
<tr>
<td>LEMMATIZE</td>
<td>qtf_lemmatize</td>
<td>Enables lemmatization. See section on Enable Lemmatization and Spelling Variations.</td>
</tr>
<tr>
<td>NAVIGATORS</td>
<td>rpf_navigation:navigators</td>
<td></td>
</tr>
<tr>
<td>NAVIGATION_HITS</td>
<td>rpf_navigation:hits</td>
<td></td>
</tr>
<tr>
<td>NAVIGATION_RESULTVIEW</td>
<td>rpf_navigation:resul tview</td>
<td></td>
</tr>
<tr>
<td>No corresponding base parameter</td>
<td>navigation</td>
<td>The APIs submit resulting field (not scope) level navigation using this HTTP query parameter. It is also possible to apply the navigation filtering conditions from the modifiers directly in this parameter. Note the difference between the NAVIGATION base parameter (corresponds to rpf_navigation) and this parameter, which does not have a corresponding base parameter. <strong>Note:</strong> The syntax for this parameter is equal to the FAST Data Search 4.x Simple Query Language.</td>
</tr>
<tr>
<td>No corresponding base parameter</td>
<td>scopenavigation</td>
<td>The modifiers for scope aware navigation should be contained in this query parameter. The parameter syntax is according to the FAST Query Language (FQL). Details on how to use Scope Navigators can be found in the Scope Navigation section.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>offset</td>
<td>Specifies the offset for the first result hit to return. See section on Using Offsets.</td>
</tr>
<tr>
<td>API</td>
<td>HTTP Interface</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PROXIMITYBOOST</td>
<td>rpf_proximityboost:enabled</td>
<td>Enables result-side proximity boost (for backwards compatibility only). See section on Proximity relevance features.</td>
</tr>
<tr>
<td>PROXIMITYBOOST_HITS</td>
<td>rpf_proximityboost:hits</td>
<td></td>
</tr>
<tr>
<td>PROXIMITYBOOST_PARAMS</td>
<td>juniper</td>
<td></td>
</tr>
<tr>
<td>QR_ESC_NEWL</td>
<td>tvmargin9=1</td>
<td>Ensures that the formatting characters (newlines, tabs etc) prevails in the returned QueryResult object. Not that this parameter is only applicable if the document summary from document processing preserves the formatting characters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This parameter is currently only supported by the Java Search API.</td>
</tr>
<tr>
<td>QTF_SECURITY_ENABLED</td>
<td></td>
<td>Enables security. Note that you have to use the FAST Security Access Module in FAST ESP to set up document level security.</td>
</tr>
<tr>
<td>QUERY</td>
<td>query</td>
<td>Defines the query terms according to FAST Query Language. See section on Define Query Elements.</td>
</tr>
<tr>
<td>qt_queriesynonyms</td>
<td>qt_queriesynonyms</td>
<td>Enables query-side synonym processing. See section on Enable Synonym Processing.</td>
</tr>
<tr>
<td>QT_PIPELINE</td>
<td>qtpipeline</td>
<td>Select alternative Query Transformation Pipeline. See section Select Query Processing Pipeline.</td>
</tr>
<tr>
<td>RP_PIPELINE</td>
<td>rppipeline</td>
<td>Select alternative Result Transformation Pipeline. See section Select Result Processing Pipeline.</td>
</tr>
<tr>
<td>RESUBMIT_FLAGS</td>
<td>resubmitflags</td>
<td>Determines how a query is resubmitted if no hits are returned. See section on Automatic query re-submit if no hits.</td>
</tr>
<tr>
<td>RESULT_VIEW</td>
<td>cgi-bin/[xml-&lt;resultview name&gt;</td>
<td>Specify a Result View different from the default one. See section on Result View.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> On the HTTP interface this is indicated in the URI (cgi-bin/...)</td>
</tr>
<tr>
<td>QTF_SECURITY_UID</td>
<td></td>
<td>Specifies the user ID to use for the query. Note that you have to use the FAST ESP Security Access Module in FAST ESP to set up document level security.</td>
</tr>
<tr>
<td>SIMILAR_TO</td>
<td>similarto</td>
<td>Enables Find Similar functionality for the query. See section on Find Similar features.</td>
</tr>
<tr>
<td>SIMILAR_TYPE</td>
<td>similartype</td>
<td></td>
</tr>
<tr>
<td>SORTSIMILAR</td>
<td>rpf_sortsimilar:enabled</td>
<td></td>
</tr>
<tr>
<td>SORT_BY</td>
<td>sortby</td>
<td>Defines sorting by field and in which direction (ascending, descending). See section on Result Sorting and Ranking features.</td>
</tr>
<tr>
<td>SORT_DIRECTION</td>
<td>sortdirection</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>HTTP Interface</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SPELL</td>
<td>spell</td>
<td>Enables query spell check and phrase detection. Both part of the DidYouMean Query Transformer. See section on Phrase Detection and Spell Check.</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>timeout</td>
<td>Specifies the timeout (in milliseconds) to use on the query between the web service and the QR Server. The default value is 0 (unlimited).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the timeout expires during search, the following error message will be returned in the result string (instead of XML): ERROR: Timeout expired while executing search with full url &lt;complete QR-Server request (including all parameters)&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This parameter only applies to FAST Web Services.</td>
</tr>
<tr>
<td>WILDCARD</td>
<td>qtf_tokenize:wildcard</td>
<td>Enables an extended set of wildcard features. See section on Extended wildcard characters.</td>
</tr>
</tbody>
</table>

### Basic Query Parameters

This section describes basic query parameters, such as parameters defining the query term string format, encoding and basic result presentation.

#### Define query elements

The query parameter defines the query as a string of one or more query terms based on one of the FAST ESP query languages.

The terms are assigned or represent values which contribute to the overall query result. Options, specifying specific conditions, can also be applied to a query prior to query submission.

#### Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERY</td>
<td>query</td>
<td>Any valid query language string</td>
</tr>
</tbody>
</table>

**Default: None**

#### Example

**Java API:**
```java
IQueryResult result = searchView.search("soccer");
```

**HTTP:**
```http
query=soccer
```

The queries above will search for the term `soccer` in the default composite field specified in the index-profile.

#### Define character encoding

The encoding parameter allows you to specify content in multiple languages.
Content will usually be interpreted by different layers of a software application. The layers might consist of applications that interpret identical fragments of text differently. To make sure the fragment of text is interpreted as the author intended it to be, by all layers, it is necessary that content is encoded in a format that is universally agreed upon.

The character encoding search parameter describes the character set encoding used to encode the query string and the result fields. FAST Enterprise Search Platform enables you to use UCS transformation format (UTF) encoding based on the ISO 10646 standard.

### Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCODING</td>
<td>encoding</td>
<td>• utf-8: UCS transformation format encoding based on the ISO 10646 standard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ISO 8859 character formats</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Most other formats as supported by the <code>libiconv</code> open source program</td>
</tr>
</tbody>
</table>

**Default: utf-8**

**Example**

```java
Java API: query.setParameter(new SearchParameter(BaseParameter.ENCODING,"utf-8");
HTTP: encoding=utf-8
```

**Working with other formats than utf-8**

When working with other formats than utf-8, these must be set explicitly.

FAST ESP assumes the query sent to the Query & Result Server is encoded in utf-8. Meaning, if you work with other formats than utf-8, these must be explicitly set using the ENCODING Base Parameter. The query string submitted via the API must be checked if in correct format. If not, it must be processed through an encoder.

When interfacing directly via the HTTP interface, a more low-level approach must be taken to ensure that characters and URL encoding conforms to the format standard you wish to use. The QRServer will provide a response encoding that is equal to the encoding parameter (Content-type : text/plain;encoding=$encoding).

**Maximum number of results displayed on a page**

The hits parameter allows you to adjust the number of results displayed on each page.

The default maximum value limits this parameter from exceeding 4100.

When using Result Clustering the Search APIs will initially retrieve all documents used as basis for the clustering algorithm (default 100). See also Result Clustering.

**Note:**

This parameter is normally only applicable for the HTTP GET interface, as it is transparent to the client when using the programmatic APIs. However, a different value than the default (10) may also be used in association with the API. For example, if the API client wants to present 12 results on each page, the API usage is optimized if ‘hits’ is set to 12. With the default value the API may need to perform 2 queries in order to return the first result page. This means that the functionality is the same, but performance is optimized. See also the example in the *Using offsets* section.
Using offsets

The offset parameter is the number of results to skip, before presenting the first result.

The offset is most often used to navigate through the search results page by page. There are some considerations for this feature when using Result Clustering. See section Iterating through the result set when using Result Clustering.

This parameter is not required for the programmatic APIs if you have a stateful query application, as the APIs provide methods to iterate through the result set. Under the hood the APIs will submit new queries to the Query & Result Server whenever needed during the iteration. For stateless applications using the APIs it may still be considered in order to optimize the number of queries submitted from the API when iterating through the result set. Otherwise the APIs will always retrieve results from offset 0 and may need to submit multiple queries in order to fetch the result page the client actually needs.

Note: As a maximum the QRserver returns 4020 documents. Thus, using offsets greater than 4020 will result in an invalid query.

Java example

In this example a stateless client requests the second page of the result set, where 12 results are presented per page:
HTTP example

```
query=string("faust goethe",mode="AND")&offset=10&hits=12
```

Result view

The result view parameter enables the definition of an alternative result view for the query, as defined within the index-profile.

This parameter is implicitly indicated with the `cgi-bin/...` parameter in the HTTP request. Refer to Using the HTTP GET Query Interface in the Query Integration Guide for details.

ℹ️ **Note:** Specifying this parameter programmatically overrides settings specified in the search profile. It is recommended to either specify all settings using the search profile or using one of the APIs, not a mix of both.

Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESULT_VIEW</td>
<td>N/A</td>
<td>result view name as defined in index-profile</td>
</tr>
</tbody>
</table>

Default

The default result view contains all fields that have been specified in the index-profile with the result attribute not equal to no.

Example

The default index-profiles include the following alternative result view which can be used as an example:

```
<result-view name="urls">
  <field-ref name="url"/>
  <field-ref name="urls"/>
</result-view>
```

This will only return the `url` (main url for the document) and `urls` (all URLs within the document) fields in the result. You enable this result view in the query as follows:

- **Java API**: `query.setParameter(new SearchParameter(BaseParameter.RESULT_VIEW,"urls"));`
- **HTTP (text template)**: `http://mynode.example.com:15100/cgi-bin/urls?query=hello`
- **HTTP (xml template)**: `http://mynode.example.com:15100/cgi-bin/xml-urls?query=hello`

Language and linguistics query features

Define language for the query string

The language parameter defines the language of the query string, in order to apply language specific query processing such as spell check and lemmatization.
The value you set for the language parameter must be a string code for the language used to express the query. FAST Enterprise Search Platform uses the ISO-639 standard for language abbreviations. For a list of languages that FAST Enterprise Search Platform supports along with their respective abbreviations, see the Advanced Linguistics Processing section in the Product Overview documentation.

Lemmatization is applied on documents prior to indexing, and the relation with the target language parameter is as follows:

- In order for Japanese lemmatization to be applied, Japanese must be defined as the Target Language (or Japanese is configured as the default language) and Lemmatization is applied for the query (see section Enable Lemmatization and Spelling Variations).
- Lemmatization is not activated if lemmatization is not supported for the language specified with this parameter.

Simple Spell Check also relies on the query language. The language specific dictionaries are selected based on the language of the query.

**Tip:** Defining the language scope for the query (only returning documents of a given set of languages) is handled by applying language filtering, see section Define Language scope for query.

**Format**

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANGUAGE</td>
<td>language</td>
<td>ISO 639-1</td>
</tr>
</tbody>
</table>

**Default:** en (English)

**Tip:** It is possible to change the default language settings.

**Example**

Java API: `query.setParameter(new SearchParameter(BaseParameter.LANGUAGE, "de");`

HTTP: `query=string(%22fast%20goethe%22,%20mode=%22and%22)&language=de`

**Define language scope for query**

Combine the language parameter with the filter parameter to limit the query scope to documents within a given set of languages.

The language of the document is specified in the `language` field and is coded according to ISO-639.

**Format**

`language:<acronym for language>`

**Example**

HTTP: `and(europe,%20filter(language:string(%22en%20es%20fr%22,%20mode=%22OR%22)))`

The example defines a filter query operator within the FAST Query Language that limits the result to English, French and Spanish documents for the query term 'europe'.
Lemmatization for queries against non-default composite fields

If you want to use another composite field for your query you can either specify composite field per term/phrase, or you can apply an alternative composite field for the entire query.

- **Specify composite field per term/phrase using the following syntax:**

  ```
  query=lem<compositename>:<term or phrase>
  ```

  In this case you do not use the 'LEMMATIZE' parameter, but instead address the lemmatized composite fields directly by prefixing the composite field name with 'lem'. For example:

  ```
  query=lemcomp1:sport%20AND%20lemcomp2:news
  ```

- **Apply an alternative composite field for the entire query using the following syntax:**

  ```
  HTTP: qtf_lemmatize=0|1&defidx=<compositename>
  HTTP: qtf_lemmatize=1&defidx=compl1&query=sport
  ```

  This option (defidx) is more convenient if there are multiple terms/phrases that shall be searched against the same (non-default) composite field. E.g. input from a search box.

  **Note:** The restrictions and alternative syntax described in this subsection does only apply when you use the simple/advanced query language.

  **Note:** The described options only works if you use lemmatization by expansion (default lemmatization mode). Lemmatization by reduction can only be applied to the default composite field.

Enable synonym processing

Use the query synonyms parameter to analyze a query term, compare it to a synonym dictionary, and alternatively modify the query automatically or provide suggestions.

If the Search View, through which search settings are specified, include synonym processing (`qtf-config.xml`), synonym processing will be performed. The synonym query transformer, `qt_synonym`, analyzes the query terms, compares it to a synonym dictionary, and either modifies the query automatically or provides suggestions.

**Note:**

- The order in which lemmatization and synonym processing is specified in the query pipeline, will affect whether synonyms are gathered for lemmatized terms or not.
- If switching from disabled to enabled state, or vice versa, the QRserver must be restarted in order for it to take affect.
- The Search View will determine which query pipeline is used, and also which instance of `qt_synonym` . The Search View can be configured through the Search Business Center .

Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qtf_querysynonyms</code></td>
<td><code>qtf_querysynonyms</code></td>
<td><code>false</code></td>
</tr>
</tbody>
</table>
Phrase detection and spell check

FAST ESP implements spell checking and phrase detection using two individual query transformation stages enabled or disabled using the same parameter.

The spelling correction is based on the `edit distance' between the mis-spelled query terms and the terms/phrases in a set of spell check dictionaries.

Simple spell check

Using simple spell check, individual terms are spell checked against language specific dictionaries. This spell check stage will only detect mis-spelling of single words.

Advanced spell check

Using advanced spell check, the query terms are spell checked against a customer defined list of words and phrases. This enables the following query transformation capabilities:

- Detect implicit phrases and proper names in the query. Found phrases will be explicitly phrased in the query ("the phrase") which means that the detected phrase is protected from further query transformation and the query will return phrase matches only. By creating a list of e.g. product names, the customer may ensure that queries are directed to the desired pages that match the implicit product name phrase.
- Detect and correct mis-spelled phrases and words. Implicit phrases in the query will be spell corrected. If the dictionary contains the phrase "nissan micra", the queries `nissan macra' will be spell corrected to "nissan micra". The spell check will even detect the phrase if both terms are mis-spelled, e.g. `nisan macra' is corrected to "nissan micra".
- Detect and correct query terms with alternative tokenization. If the dictionary contains the term "thinkpad", a query `think pad' will be corrected to "thinkpad". If the dictionary contains the term "alpha server", a query `alphaserver' will be corrected to "alpha server".

The spell checking algorithm operates on individual query segments. A query segment is a portion of the query that forms a syntactical entity of some kind. For example, if tokens within the query is put in quotes, that quoted part forms a query segment. Additionally, the - and + modifiers in the Simple Query Language also introduce segment breaks. The content within double quotes inside a string() operator is an individual query segment. This means that implicit phrases will not be detected across query segments.

Advanced spell check applies per string() operator when using the FAST Query Language (FQL). If you want to correct the spelling of "new yurk" to "new york" you must ensure that the entire mis-spelled phrase is within a string() operator, for example:

```
string(%22new%20yurk%22,%20mode=%22and%22)
```

The following queries will **not** get the proper spell checking:

```
and(string(%22new%22,%20mode=%22or%22),string(%22yurk%22,%20mode=%22or%22))new%20yurk
```

The second query will be treated as two individual string() operators, thus will not get the proper linguistics transformation.

**Note:** The anti-phrasing feature is also controlled using the SPELL parameter.
Simple spell check and phrases

Using simple spell check, individual terms are spell checked against language specific dictionaries, detecting mis-spelling of single words.

Simple spell check will by default also be applied to explicit phrases in the query when using the FAST Query Language (FQL). However, this behavior can be controlled by the LINGUISTICS parameter in FQL.

For the four following queries, simple spell check will be applied. This means that the term "hello worlg" will be corrected/suggested to "hello world":

"hello worlg"
string("hello worlg")
string("hello worlg", mode="phrase")
string("hello worlg", mode="phrase", linguistics="on")

In this case the three first queries are equal to the fourth one due to default parameter settings in FQL.

For the following two queries, the simple spell check will be applied due to the linguistics parameter setting:

string(%22hello%20worlg%22,%20linguistics=%22off%22)
string(%22hello%20worlg%22,%20mode=%22phrase%22,%20linguistics=%22off%22)

Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
<th>Description</th>
</tr>
</thead>
</table>
| SPELL | spell | on|off|suggest | • on: Apply the query with automatic spell correction.  
• off: No spell check performed.  
• suggest: Suggest transformed (spell corrected) query in the result, but do not apply spell check to the actual query performed. This enables Did you mean-type search tips in the result pages. |

Default: suggest

Example

In the following example, the mis-spelled query term `thesarus' will automatically be translated to the query "thesaurus" before being processed.

Java API:

```java
IQuery query = new Query("thesarus");
query.setParameter (new SearchParameter(BaseParameter.SPELL,"on"));
```

HTTP: query=thesarus&spell=on

Anti-phrasing

Anti-Phrasing is controlled as part of the spell check framework, using the SPELL query parameter.

Anti-phrasing applies per string() operator within the FAST Query Language (FQL). If you want to remove the anti-phrase "where is" from the query "where is jon", you must ensure that the entire string "where is jon" is within a string() operator, for example:

```sql
string("where is jon", mode="or")xml:title:string("where is jon", mode="or")
```
The following queries will not get the proper anti-phrasing:

```
and(string("where", mode="or") string("is jon", mode="or")) where is jon
```

The second query will be treated as three individual string() operators and will thus not get the proper linguistics transformation.

**Extended wildcard characters**

An extended set of wildcard characters is available to provide additional wildcard searching beyond the * and ? characters.

**Query parameters**

By default, this feature is enabled at the system level, so that query users can use FQL string() operator parameters to enable extended wildcards. Alternatively, query level parameters can be used to enable and control this feature across an entire query.

**Note:** Extended wildcards only work on text that was indexed with full wildcard support enabled.

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
<th>Description</th>
</tr>
</thead>
</table>
| WILDCARD    | qtf_tokenize: wildcard    | off|basic|ext | basic: Use default wildcard characters * and ?
|             |                          |              | ext: Enable extended wildcard character set     |
|             |                          |              | Default=basic                                   |

**Wildcard characters**

Extended wildcard characters include the default ESP operators of * and ?, plus the following wildcards:

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Match exactly one alphabetic character</td>
<td>c@er</td>
</tr>
<tr>
<td></td>
<td></td>
<td>do@</td>
</tr>
<tr>
<td>#</td>
<td>Match exactly one numeric character</td>
<td>#600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f-##</td>
</tr>
<tr>
<td>-</td>
<td>Match one or no character</td>
<td>colo_r</td>
</tr>
<tr>
<td>[]</td>
<td>Match a single character from the set; can include a hyphen to indicate a range of letters or numbers</td>
<td>A[1-5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>la[sz]er</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AC[a-z][0-9][0-9][0-9]</td>
</tr>
<tr>
<td>^</td>
<td>When ^ is the first character inside a set, match any character not found in the set</td>
<td>199[^78]</td>
</tr>
<tr>
<td>\</td>
<td>Take the next character literally and search for it; i.e., do not interpret the next character as an operator or wildcard (functions only if the</td>
<td>joe@home</td>
</tr>
</tbody>
</table>
## Examples

- **Java API:**
  ```java
  IQuery query = new Query("string("colo_r")");
  query.setParameter(new SearchParameter(BaseParameter.WILDCARD,"ext"));
  ```

- **HTTP:**
  ```plaintext
  query=colo_r&qtf_tokenize:wildcard="ext"
  ```

### Configuration parameters

The following configuration parameters apply to extended wildcards, just as they apply to the basic wildcards:

- hardwildcardcutoff
- softwildcardcutoff

## Pattern search parameters

Pattern searching compensates for more general spelling problems than does Simple Spell Check. Pattern searching provides fuzzy spelling and pattern matching capabilities. Simple Spell Check only corrects query spelling errors, and only if a spell check dictionary is available for the query language. Pattern searching works with any language that has more than a few characters per word, and finds any words in documents which closely resemble the query words, regardless of spelling errors, alternate spellings, typos, OCR errors (from scanning paper documents), or transmission errors.

By default, pattern searching is enabled at the system level, so that query users can use FQL string() operator parameters to control pattern expansion. Alternatively, query level parameters can be used to enable and control these features across an entire query.

**Note:** Pattern search only works on text that was indexed with full wildcard support enabled.

## Query parameters

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
<th>Description</th>
</tr>
</thead>
</table>
| FUZZY        | qtf_tokenize:fuzzy  | off|on      | **off:** Fuzzy expansion is disabled, except when it is enabled using the string() operator.  
**on:** Fuzzy expansion is enabled for all query terms, except when it is disabled using the string() operator. Each non-wildcard query term is expanded to include spelling errors/variants, typos, OCR errors, and other garbled words.  
*Default=off* |
### Configuration parameters

Pattern search is also affected by several cluster configuration parameters. To prevent overloading system resources, set `searchtimeout` and `hardfuzzycutoff` in the $FASTSEARCH/etc/config_data/RTSearch/*/fsearch.addon configuration file to impose limits.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Valid values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hardfuzzycutoff</td>
<td>-1 or any greater integer</td>
<td>The maximum number of terms a pattern search can expand to before a query error is returned. -1 represents no limit 0 turns off pattern expansions  Default=100</td>
</tr>
<tr>
<td>softfuzzycutoff</td>
<td>unlimited, or any positive integer</td>
<td>The silent cutoff on the maximum number of pattern expansions per query term. Default=unlimited</td>
</tr>
</tbody>
</table>

For example:

```
hardfuzzycutoff 600
```
Examples

Java API:

```java
query.setParameter(new SearchParameter(BaseParameter.FUZZY,"on"));
```

HTTP:

In the following HTTP example, a query on the word "kadaffi" could return documents containing the words:

- kadafi
- khadafi
- kada4i
- kaddafis
- qadaffi
- quadaffi
- quaddafi
- gaddafi

```http
query=kadaffi&qtf_tokenize:fuzzy=on&qtf_tokenize:fuzzythreshold=70
```

FQL:

```sql
string("galloping gormet",mode="PHRASE",fuzzy="on")
```

Automatic query re-submit if no hits

Use the resubmit flags parameter to indicate the desired transformation for automatic resubmitting the original query - given the case that no hits result from the original query.

Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESUBMITFLAGS</td>
<td>resubmittflags</td>
<td>An integer representing a bitmask as defined in the table.</td>
</tr>
</tbody>
</table>

Bitmask values

<table>
<thead>
<tr>
<th>Bitmask Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (0x00000001 or any other odd value)</td>
<td>Do not resubmit. This setting overrides any server-side methods configured to resubmit.</td>
</tr>
<tr>
<td>7984 (0x00001F30)</td>
<td>Enable Spell Check feature for the resubmit (SPELL=&quot;on&quot;).</td>
</tr>
<tr>
<td>8192 (0x00002000)</td>
<td>Enable spell check in suggest mode only - to be used together with the spell check related flags. For example, 7984+8192=16176 will apply spell check in suggest mode when resubmitting (assuming no spellcheck applied for original query).</td>
</tr>
<tr>
<td>64 (0x00000040)</td>
<td>Enable lemmatization when resubmitting.</td>
</tr>
<tr>
<td>134217728 (0x080000000)</td>
<td>Add synonyms automatically</td>
</tr>
</tbody>
</table>
Default: 1 (off)

Find similar features

The find similar features enable you to search for documents that are similar to already retrieved query results. FAST Enterprise Search Platform automatically creates a similarity component that is added to the query (as described in section Type of Find Similar query). You can control the find similar feature using the following set of parameters:

- SIMILAR_TYPE - Type of Find Similar query
- SIMILAR_TO - Document Similarity Vector reference
- SORTSIMILAR - Sorting of Find Similar query results

The SIMILAR_TO parameter can be used by the "traditional" find similar feature, or by the Query By Example feature. The latter implies a parameter value containing far more text than the former. Thus, the HTTP GET method is no longer sufficient so the HTTP POST method must be used instead.

Type of find similar query

Use the similar type parameter to specify the type of ‘find similar’ you want to apply.

The document vectors for each document, sorted by decreasing weights, can be used to build three types of similarity searches for a document d, given an original query Q. Within FAST ESP these similarity search requests are transformed to a new unique query, using the following rewrite of the query (shown using a symbolic representation, not the exact query language):

- **Find similar**: Query = Q OR <s1,w1> [OR <sm,wm>]* This means that the similarity vectors are added to the query using an 'OR' operator. This means that the original query is included in the rewritten query, but the new query may match similar documents even if the original query is not met.
- **Refine similar**: Query = Q AND (<s1,w1> [OR <sm,wm>]* This means that the query will match if the original query conditions and the similarity vector conditions are met. I.e. refine original query with similar documents as the indicated.
- **Exclude similar**: Query = Q ANDNOT (<s1,w1> [OR <sm,wm>]* This means that the query will match if the original query conditions are met but not the similarity conditions. <s,w> indicates the document's similarity vector as computed during document processing.

**Note:**

The similarity component that is added to the query (<s1,w1> [OR <sm,wm>]* is targeted towards the default Composite Field by default.

It is possible to configure the find similar feature to use another field than the default composite field, for example an alternative composite field or a scope field. You can change this behaviour by adding `qt.similar.target="scope_or_composite_field_name"` in the `qtf-config.xml` configuration file.

Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMILAR_TYPE</td>
<td>similartype</td>
<td>find</td>
</tr>
</tbody>
</table>
Default: none

Document similarity vector reference

The similarity vector is created during document processing and indicates the most important terms/concepts in the document and the corresponding weight.

When performing a find similar query, the similar to parameter contains a string parameter with the 'docvector' (similarity vector) field of the document that is to be used as template for 'find similar'. The similarity vector consists of a set of 'term,weight' expressions, indicating the most important terms/concepts in the document and the corresponding weight. Terms may be single words or phrases.

The weight is a float value between 0 and 1, where 1 indicates highest relevance. The similarity vector is created during document processing by the Vectorizer document processor. This information is returned from the preceding query. This function is currently apparent in the Search Front End, visible through the "Find" link below each result.

The result field docvector has the following format:

```
[term1,weight1][term2,weight2]...[termN,weightN]
```

Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMILAR_TO</td>
<td>similarto</td>
<td>[term,weight][term,weight]... where `weight' is a float value.</td>
</tr>
</tbody>
</table>

**Note:** The square brackets ([]) are part of the format for this query parameter.

Default: None

**Example**

Java API: `query.setParameter(new SearchParameter(BaseParameter.SIMILAR_TO, summary.getSummaryField("docvector")));`

HTTP: `query=audi&similarto=[s8,0.9][a4,0.7]`

The HTTP example will return documents containing the name of the car manufacturer Audi. However, documents including references to the S8 model are considered more relevant than the A4 model.

Query by example

The query by example feature allows you to submit a complete paragraph (or even a complete document) as a query to FAST ESP and retrieve documents that are similar to this input text.

The query by example realizes a find similar functionality, but the input text/document does not have to be indexed by FAST ESP.

A query by example request is submitted to the qrs-server as an HTTP POST request, not an HTTP GET request. The reference text/document may be very long, and thus not suited to be part of a GET URL. POST requests are only processed by the qrs-server if the QBE feature is enabled. If the query by example feature is disabled, then POST requests are not processed and the qrs-server returns an error message.

The input reference text/document is submitted via the similarto parameter. For example:

```
similarto=this+is+the+reference+text,+dude
```
The POST request should also contain a similartypem parameter, and optionally query, language and encoding parameters. The semantics of these are the same as with GET requests.

The input reference text/document text is vectorized on-the-fly by the qrserver, to yield a document vector representation of the submitted input reference text/document:

```
[reference text, 1.0][dude, 0.5]
```

The values of the language and encoding parameters are used by the vectorizer to select stopword lists. After computing the document vector, the value of the similarto parameter is substituted to include the vector instead of the raw text:

```
similarto=[reference+text,+1.0][dude,+0.5]
```

If the POST request contains a query parameter, then this is left unchanged. Otherwise, a query parameter will be inserted by the QBE handler. The value of the injected query parameter is the string in the computed vector having the highest weight (ties are resolved arbitrarily):

```
reference text
```

From here on, the POST request is treated just like a GET request. Using the parameters in the previous example, the request will function as a find/refine/exclude similar query.

**Tip:** The query by example feature is enabled by default. Refer to the configuration documentation for details about how to disable the feature.

### Sorting of find similar query results

Use the sort similar parameter to sort the query results based on similarity.

When performing a find similar query, the results may be sorted two ways:

- by relevance score (rank). This is the sorting method for normal queries, and corresponds to `SORTSIMILAR=false`
- by similarity. This is the default sorting for similarity queries, where the most similar documents are listed first. This corresponds to `SORTSIMILAR=true`.

**Note:** The sorting based on similarity is performed result-side. Only the top 100 results are used as basis for the sorting. This means that the 100 top ranked results (by relevancy) are re-sorted based on similarity. The proceeding results are sorted by relevancy.

### Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SORTSIMILAR</td>
<td>rpf_sortsimilar:enabled</td>
<td>1</td>
</tr>
</tbody>
</table>

**Default:** 1 (true)

### Geo search

Geo search sorts and/or filters query results based on geographical distance from a defined geographical location.

You must always use the sortby query parameter when performing geo queries. The sortby parameter can be used in two main ways, depending on the desired geo sort/filter operation:

- `sortby=<name of Geo Specification in Index Profile>` In this case you refer to a geo-specification as specified in the index profile. Use this format when you want to filter and/or sort on geographical distance.
- `sortby=<name of Rank Profile in Index Profile>`: In this case you refer to a rank-profile as specified in the index profile. This format is used when you want filter the result in combination with result ranking.

**Sorting by geographical distance (geo-sorting)**

Use the sort by parameter in combination with a geo specification and center coordinate to sort the result set by geographical distance.

First select the `<geo-specification>` using the sortby query parameter. The `<geo-specification>` must include `filter="no"` and `sortable="yes"`. Then select the center coordinate using the `qtf_geosearch:center` parameter.

You can only use geographical distance (geo specification) as the last sorting criteria in a multi-level sort specification. Hence, it is not possible to sort by geographical distance and then by another criteria.

If a document contains more than one coordinate, the coordinate with the shortest distance from the indicated center coordinate will be used as basis for the sorting.

**Example**

The user requests ascending sorting based on geographical distance using the following query parameters:

```
sortby=+geo_spec_3 qtf_geosearch:center=(59.56,10.41)
```

Documents without any location information will in this case not appear in the results. If you specify `keep-undefined="yes"` these documents will instead appear in the end of the result set.

**Sorting/filtering by geographical distance (geo-filtering)**

You may filter the result set based on geographical area, using a circle (indicated by center coordinate and radius), a filter box (indicated as a rectangle) or a combination of the two.

Use the sortby query parameter to identify the `<geo-specification>` as defined in the index-profile. The `<geo-specification>` must include `filter="yes"` if you apply circle-based filtering. Use `filter="no"` for rectangle-based filtering. Use the center and radius parameter to specify the center coordinate and filtering criteria.

The filtered results will be sorted by distance from the center coordinates. If a document contains more than one coordinate, the coordinate with the shortest distance from the indicated center coordinate will be used as basis for the filtering. Documents without any location information will not appear in the results (based on the default value for the `keep-undefined` attribute).

**Example: Filtering based on a circle**

```
sortby=+geo_spec_2 qtf_geosearch:center=(59.56,10.41)
qtf_geosearch:radius=150
```

The circle is indicated by center coordinate and radius.

**Example: Filtering based on a rectangle**

```
sortby=+geo_spec_3 qtf_geosearch:center=(59.56,10.41)
qtf_geosearch:filterbox=[(58.5,9.4);(60.5,11.4)]
```

The rectangle is indicated by two diagonal opposite corners.

**Geo search query parameters**

Use the following query parameteres for geo search.
**Note:** The geo search query parameters are currently not supported within the API namespace; use the HTTP query parameter syntax instead. For example, using the Java API:

```java
IQuery query = new Query("chinese restaurant");
query.setParameter(new SearchParameter("sortby", "geo_rank_2"));
query.setParameter(new SearchParameter("qtf_geosearch:center", "(59.56,10.41)");
query.setParameter(new SearchParameter("qtf_geosearch:radius", 150));
IQueryResult result = mySearchEng.search(query);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qtf_geosearch:center</td>
<td>(lat,lon)</td>
<td>Specifies the center coordinate used to calculate distances for geo sorting/filtering. The latitude/longitude values must be specified as degree coordinates, -90 to 90 for latitude and -180 to 180 for longitude. For example, to specify the center coordinate with latitude 59.56 and longitude 10.41:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>qtf_geosearch:center=(59.56,10.41)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: 0:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This parameter is mandatory when using geo search.</td>
</tr>
<tr>
<td>qtf_geosearch:radius</td>
<td>radius</td>
<td>(lat,lon)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>qtf_geosearch:radius=150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To specify the radius as the distance from the center coordinate to the specified coordinate:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>qtf_geosearch:radius=(61.56,12.41)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: The default radius as indicated in the index-profile.</td>
</tr>
<tr>
<td>qtf_geosearch:unit</td>
<td>km</td>
<td>mi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>qtf_geosearch:unit=km</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default: mi (miles)</td>
</tr>
<tr>
<td>qtf_geosearch:filterbox</td>
<td>[(lat1,lon1);(lat2,lon2)]</td>
<td>Specifies the filter rectangle used for result filtering based on an indicated rectangle (rather than a circle). Only documents with coordinates within the filter rectangle will be returned. If not specified, the radius will be used for filtering. It is possible to specify a combination of radius and filterbox. In this case the filtering is based on the area enclosed in the circle <strong>and</strong> the rectangle. For example, to specify the radius in kilometers:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>qtf_geosearch:filterbox=(59.5,10.4);(61.5,12.4)</td>
</tr>
</tbody>
</table>
**Parameter** | **Values** | **Description**
---|---|---
`rpf_geosearch:center` | (lat,lon) | Specifies an optional center coordinate C2 (see the Product Overview Guide for details) used to calculate the distance shown in the search results.

**Note:** The sorting and restricting on distance is still done using the center coordinate as given by `qtf_geosearch:center` and radius/filterbox specified by `qtf_geosearch:radius` and/or `qtf_geosearch:filterbox`.

`rpf_geosearch:unit` | km | mi | nm | The unit used in the distance presented in the results when `rpf_geosearch:center` is indicated.
Valid options are km for kilometers, mi for miles, and nm for nautical miles.
For example, to enable specification of the distance from C2 in kilometers: `rpf_geosearch:unit=km`.
Default: mi (miles)

**Note:** If no alternative center coordinate is indicated (using `rpf_geosearch:center`) the unit for distance presented in the results are according to what is specified in `qtf_geosearch:unit`.

**Handling query results with geo information**
When applying geo filtering and/or sorting, the result set will include additional result fields with the geo information for each result.

**Note:** The additional result fields are reserved field names that must not be used for normal fields in the index-profile. The value for these summary fields may be retrieved as any other result/summary field when using the Search API or the HTTP query interface.

<table>
<thead>
<tr>
<th>Type Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| `f<latitude-field>` | The latitude and longitude values associated with the document (according to the geo-specification).
For example (text template): `#flat 0.0;0.3 #flon 2.0;2.3` In this case the document contains two coordinates 0.0:2.0 and 0.3:2.3.

**Note:** The result/summary field names equals the latitude/longitude field names with a leading 'f'.

| geo_distance | The actual distance from the center coordinate (or alternative coordinate indicated by `rpf_geosearch:center`) to the nearest location in the document.
For example (text template): `#geo_distance 139.03` |

| geo_unit | The unit used for the 'distance' value.
`mi` : Miles `km` : Kilometers `nm` : Nautical miles
Example (text template): `#geo_unit mi` |

| geo_used_lat | The latitude and longitude values used for the sorting/filtering.
Example (text template): `#geo_used_lat 0.000000 #geo_used_lon 2.000000` |

**Note:** If more than one set of coordinates are defined for a document this will indicate the coordinates closest so the indicated center coordinate.
Enable geo search in the back-end

Follow this procedure to enable search in the back-end.

1. Make sure you are using an index profile that supports GEO data.
2. Feed documents containing GEO data.

Enable geo search in SFE

Follow this procedure to enable geo search in the search front end.

1. Open the `SearchServiceImpl.properties` file located in the `$FASTSEARCH/adminserver/webapps/sfe/WEB-INF/classes/com/fastsearch/espimpl/sfeapi/searchservice/` directory.
2. Add the following code to `custom_search_inputs`.

   ```java
   com.fastsearch.espimpl.sfeapi.searchservice.search.geo.LatLonGeoSearchImpl
   ```

3. Add the following to `custom_result_aspects`.

   ```java
   com.fastsearch.espimpl.sfeapi.searchservice.result.geo.GeoGraphImpl
   ```

Performing a search for geo data

Follow this procedure to search for geo data.

1. Make sure the Result Sorting drop-down box includes the rank profile and geo specification you want to use.
2. Check the Enable Geo Search box to make sure SFE is aware of this being a search for GEO data.
3. Select the appropriate geo-specification or rank-profile via the Sort By dropdown.
   The default indicates that the default rank-profile will be used.
4. Indicate your current location given by latitude/longitude.
5. You may also indicate an alternative center coordinate (rpf_geosearch:center) using the Distance ref (C2) entry.
6. In the Geo Search section you indicate a circle and/or a rectangle. The radius may be indicated as distance from center or by a radius endpoint.
7. You can add the desired geo result/summary fields in the query results by adding the names in the Data Fields field on the Setup page.

Search within collection

Limiting the scope to a set of collections is achieved by using the filter term with a query.

Filter terms can be applied using the filter() operator in the FAST Query Language. A reserved field name, `meta.collection`, identifies the collection for documents. Documents are grouped into collections based on data source and document processing.

The following example performs a query for all documents in the collections ‘web’ and ‘news’ containing the term faust:

```java
query=and(faust, filter(meta.collection:or(web, news)))
```
We may also filter out specific collections:

query=andnot(java, filter(meta.collection:news))

indicating a query for 'java', but avoid anything from the news.

Note: Limiting the scope by collection can also be performed using the API methods exposed by the search view. Refer to the query integration documentation for details.

Search within category

Use the filter terms to limit the scope of a query to a set of categories.

Categorization may be used in different ways, depending on application. However, it defines some type of pre-defined or automatically generated taxonomy info for the documents. The category field may be used to narrow the scope of a query in a similar way as for collections using filter terms. Filter terms can be applied using the filter() operator.

The following example performs a query for all documents in categories `literature` and `art` containing the terms Goethe or Faust. The category field is in this case named `taxonomy`:

query=and(string("goethe faust", mode="OR"), filter(taxonomy:or(literature, art)))

We may also filter out specific categories, for example:

query=andnot(java, filter(taxonomy:technology))

indicating a query for `java`, but avoid anything about the programming language `java`.

Categorization and navigation

Categorization and navigation browsing is mainly enabled by result clustering and navigation on taxonomy fields.

Result Clustering is applicable when you need true client-side clustering. Otherwise it is recommended to use the navigation approach instead.

Navigation on taxonomy fields provides navigation by category from query results. It is the best way to apply deep navigation (within the entire result set) into categories which occurs within the results.

Result clustering

Result clustering enables unsupervised clustering and categorization of query result.

Supervised clustering provides a clustered view based on pre-defined categories, i.e. category information that is provided for the documents prior to indexing. Unsupervised clustering provides a clustered view based on document similarities. Each document has a similarity vector assigned to it (computed at document processing time), and similar documents may be clustered in the results.

Result clustering operates on an extended result set as returned from the search engines to the query and result server. The default number of results analyzed is 100. The default number of results returned can be changed by altering the searching element in the configuration.<server>.xml file located in the $FASTSEARCH/etc/config_data/QRServer/webcluster/etc/qrserver/<cluster-server>/ directory. The basic support for result clustering feature is enabled within the index-profile.

Note: Result clustering implies a reasonable query overhead. Hence, you may experience reduced QPS within your application.
Iterating through the result set when using result clustering

The recommended approach is to iterate through the result set using a programmatic API due to built-in mechanisms, but result clustering can also be computed and delivered through the HTTP API based on any offset.

The programmatic search APIs will initially retrieve the extended result set used by the clustering algorithm (default 100 hits) in order to enable client-side browsing in the clustered results. If you use the normal result-set iteration methods in the API, it is possible to browse through the result set while still preserving the clustering info. Note, however, that a stateless client in this case will need to retrieve more and more documents per query as the user iterates beyond hit 100 (this is performed in the background, not visible for the API client). This is still reasonable as long as the client application provides capabilities for browsing within the clustered results.

There are three available approaches:

• Only return clustering info when the offset parameter is set to zero (offset=0). No clustering info available when paging through the results.
• Use one of the programmatic APIs to return clustered info of the first N results. The clustered tree will always be present, and include the same 100 results, even when paging through results. The offset parameter should not be set in this case. Refer to the query integration documentation for more details.
• Enable the alloffsets parameter and allow presentation of clustered information, starting from the result specified by the offset parameter. Thus, result clustering can be delivered based on result 2 through 2+N if the alloffsets option is enabled.

If the alloffsets option is enabled, the following query can be used to compute and deliver clustering information based on an offset greater than 0:

query=api&offset=2

If the default number of returned clustered results are 100, then the example above will return clustering info for result 2 through 102 in this case.

Tip: Not enabling the alloffsets option may in some cases be an alternative if performance is crucial and the client application does not need to provide clustering info when the end-user actively browses through the result set page by page.

Enable result clustering

Use the clustering parameter to enable result clustering.

Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLUSTERING</td>
<td>rpf_clustering:enabled</td>
<td>1/0 (true/false)</td>
</tr>
</tbody>
</table>

Default: 0 (false)

Example

Java API: query.setParameter(new SearchParameter(BaseParameter.CLUSTERING,true));
HTTP: rpf_clustering:enabled=1

Define clustering root

Use the clustering root parameter to define the root for the clustering tree.

The '*' character implies starting from top. The programmatic APIs are pre-initialized to `*`).
### Advanced clustering parameters

Use the following parameters to control result clustering.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpf_clustering:sorting</td>
<td>label</td>
<td>size</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>The clusters in the result tree are sorted according to the alphanumerical value of their labels.</td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>The clusters in the result tree are sorted according to the number of documents on the node. The cluster with the highest number of documents will appear on top.</td>
</tr>
<tr>
<td></td>
<td>rank</td>
<td>The clusters in the result tree are sorted according to the average rank of the documents in the cluster. The cluster with the highest average rank will appear on top.</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>There is no explicit sorting of the clusters. They are, however, not returned in random order since the algorithms that build the tree add clusters in an order determined by the ranks/positions of the documents in the cluster.</td>
</tr>
<tr>
<td>Default: &quot;none&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| rpf_clustering:threshold   | Numeric value between 0 and 1 | Clustering similarity threshold is a numeric parameter specifying the minimum degree of similarity two documents must have in order to be considered as members of the same unsupervised cluster. A similarity of 0 means that the vectors of the documents have no elements in common. The higher the similarity value is, the more elements the two document vectors have in common. Default: "0.3" |
|                           |                                |             |

| rpf_clustering:size       | Integer value                  | Size threshold is a numeric parameters specifying the minimum number of documents a cluster must contain before recursive clustering is applied. Recursive clustering builds up a cluster tree until the tree has reached the maximum depth or when no two documents are different enough to warrant the creation of a new cluster (i.e. when the similarity has to be below the similarity threshold). Default: 5 |
|                           |                                |             |

<p>| rpf_clustering:equality   | Numeric value between 0 and 1   | Equality threshold is a numeric similarity parameter specifying the minimum degree of similarity two documents must have in order to be considered equal. If all documents in a cluster have a similarity above this threshold, no recursive clustering will be attempted. |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Default: &quot;0.9&quot;</strong></td>
</tr>
<tr>
<td>rpf_clustering:labels</td>
<td>Integer</td>
<td>Number of labels is a numeric parameter specifying the maximum number of labels assigned to a cluster. The labels are picked from the most salient elements of the document vector. Default: 3</td>
</tr>
<tr>
<td>rpf_clustering:treesize</td>
<td>Integer</td>
<td>Treesize is a numeric parameter specifying the maximum number of top-level clusters in the result tree. The tree is pruned after sorting. Default: 100</td>
</tr>
</tbody>
</table>

**Navigation parameters**

The navigation parameters activates/deactivates and configures navigation (Live Analytics).

**Format**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>query.setParameter(new SearchParameter(BaseParameter.NAVIGATORS, &quot;price_nav,name_nav,date_nav&quot;));</td>
</tr>
<tr>
<td>.NET</td>
<td>query.setParameter(BaseParameter.NAVIGATORS, &quot;price_nav,name_nav,date_nav&quot;);</td>
</tr>
<tr>
<td>C++</td>
<td>param = my_query_fac-&gt;create_parameter(esp::search::parameters::Navigators, &quot;price_nav,name_nav,date_nav&quot;);</td>
</tr>
<tr>
<td>HTTP</td>
<td>rpf_navigation:navigators=price_nav,name_nav,date_nav</td>
</tr>
</tbody>
</table>

See also description of advanced format for NAVIGATORS and the navigator performance control documentation.

**Parameters**

**API**

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAVIGATION</td>
<td>rpf_navigation:enabled</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>NAVIGATORS</td>
<td>rpf_navigation:navigators</td>
<td>&lt;nav&gt;[,&lt;nav&gt;]*</td>
<td></td>
</tr>
</tbody>
</table>

• Simple format: A comma separated list of
<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>navigators to be included in the result. <code>&lt;nav&gt;</code> is a navigator name as defined in the index-profile.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Advanced format: This also enables more detailed control of how navigators are presented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Default: All navigators not set to passive=&quot;yes&quot; in the index-profile are returned with the query result. Passive navigators are only returned by explicitly specifying them to be included.</td>
</tr>
<tr>
<td>NAVIGATION_HITS</td>
<td>rpf_navigation:hits</td>
<td>Integer value (number of hits)</td>
<td>Used in order to override the default number of hits returned for shallow navigators.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>By default, Shallow Navigation is based on the N (default: 100) highest ranked documents within the query result.</td>
</tr>
<tr>
<td>NAVIGATION_RESULTVIEW</td>
<td>rpf_navigation:resultview</td>
<td>Valid Result View</td>
<td>This is an advanced feature that can be used to optimize search performance when using shallow navigators (only relevant for shallow navigators), but does not impact the data returned on the API/HTTP interface. It is particularly important to use when the result view consists of large text fields.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To apply this feature you should define an additional Result View consisting of only the fields that will be used to compute Navigators. This means all navigators or the number of navigators defined by the parameter NAVIGATORS.</td>
</tr>
</tbody>
</table>

**Note:** This does not impact the result view for the query (given by RESULT_VIEW)
Filter Navigation Content

ESP supports query refinement and drilling down part of navigation through its navigation and scopenavigation query parameters.

In FDS query refinement or drilling down part of navigation was done by adding filtering requirements to the filter parameter. This was problematic for reporting reasons, since it was impossible from a query log to tell if something in the filter specification originated from a navigation operation or from something else.

**Note:** There is currently no API equivalent to the ESP navigation and scopenavigation query parameters, but the HTTP version can be used instead.

Navigation Filter Syntax

Use the navigation query parameter to provide resulting field level navigation.

The search API will submit resulting field (not scope) level navigation using the navigation HTTP query parameter. The syntax for this parameter is equal to the FAST Data Search 4.x Simple Query Language.

Examples:

```
navigation=+<index profile field>:"<string to aggregate over>"
navigation=+author:"Johnson"
```

Scopenavigation Filter Syntax

Use the scopenavigation query parameter to enable scope aware navigation.

The modifiers for scope aware navigation should be contained in the scopenavigation query parameter. The parameter syntax is according to the FAST Query Language (FQL).

Example:

```
scopenavigation=xml:sentence:person:@base:clinton
```

Control navigator performance

There are several ways to control the amount of navigator data that will be returned to the API client. The following optimizations can be applied either in the index-profile or by using the advanced format of the NAVIGATOR parameter.

Computing all navigators for a search result can in some cases consume a great amount of memory. Specifying passive navigators in the index-profile allows for non-passive navigators only to always be returned, while passive navigators can be returned on request. Passive navigators can be specified to be included in the result, using the NAVIGATORS parameter.

```
query.setParameter(new SearchParameter(BaseParameter.Navigators, "name_nav,price_nav"));
```

You can use the filter parameter for application specific filtering (i.e. control which info is relevant to present in a search front end). This is a result-side filtering and will not reduce load in the search engines. The following example will return the first 100 buckets, only return buckets which contain more than the default number of

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(parameter), it is only used to optimize the data flow between the Search Engines and the Query &amp; Result Server.</td>
</tr>
</tbody>
</table>
matching documents, and perform no prefix filtering. A zero value in an argument means the default value will be applied.

```java
query.setParameter(new SearchParameter(BaseParameter.Navigators, "name_nav(filter=100/0/*)");
```

In some cases it may also be necessary to limit the amount of data that must be transferred and processed for deep string navigators. This is relevant for applications with high QPS, using navigators with large amount of unique values. You can use the cutoff parameter (as described in table ) and the corresponding index-profile parameters to apply such performance optimization.

The navigators can in this way be configured to only return the most relevant values (buckets). For instance, for a navigator on person names, only those names that occur most frequently are returned as navigator values. Even with such cutoffs, the system will return correct values for the total number of values even though the individual values are not all returned to the Search APIs. The following example will apply a cutoff value of 5000 for maxbuckets. Meaning that no more than 5000 unique values are returned for this name navigator.

```java
query.setParameter(new SearchParameter(BaseParameter.Navigators, "name_nav(cutoff=0/0/5000)" ));
```

It is possible to combine the shallow navigator filtering and the navigator cutoff in the same query. If you apply result-side filtering on number of buckets you can safely apply a corresponding bucket cutoff to optimize performance in the search engines. Note that the cutoff feature is distributed to all individual search partitions, while the filtering is performed on the final navigator set. Hence, these two features may supplement each other.

**Note:** When using the cutoff parameter, all values restrict the number of navigator values returned from individual . There are usually several search partitions per column in a system.

**Tip:** When doing cutoff on string navigators, then theoretical max. count of any buckets not displayed by the Query & Result Server is available in the navigator API interfaces.

### Example

The following Java API example provides the following filtering/cutoff for the navigator named name-nav:

- Sort buckets/modifiers descending based on modifier value frequency
- Return only the 100 first modifiers for the navigator (after the sorting)
- Perform a cutoff of 100 based on frequency, which implies that only the 100 most frequent modifiers are returned from each search partition. This can be done due to the fact that cutoff and sorting/filtering is based on the same criteria.

```java
query.setParameter(new SearchParameter(BaseParameter.NAVIGATORS, "name_nav(sort=frequency/descending,cutoff=0/0/100,filter=100/0/*")");
```

### Navigator display and performance parameters

Use the navigators query parameter to override the settings in the index-profile and specify advanced navigator configuration per navigator in a query.

The supported parameters represent a subset of the navigator attributes in the index-profile.
Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAVIGATORS</td>
<td>rpf_navigation:navigators</td>
<td>&lt;nav&gt;,[&lt;nav&gt;]*</td>
</tr>
</tbody>
</table>

<nav> indicates a parameter specification for each named navigator, and has the following format:

```
<navigator-name>([parameter=value][,parameter=value]*)
```

where <navigator-name> is the name of the navigator as defined in the index-profile. The list of `parameter=value' pairs enables you to apply a number of non-default configuration values for the named Navigator. An empty list equals using the simple format as described in the table. If a given parameter for a given navigator is not listed inside the parentheses, then the value for that parameter is taken from the index-profile.

Java example

```java
query.setParameter(new SearchParameter(BaseParameter.NAVIGATORS, 
    "price_nav()
    , name_nav(sort=frequency/descending,
    deephits=500)"));
```

The example above defines that only price and name navigators shall be returned in the query result, navigator sorting shall be descending based on bucket frequency and a maximum of 500 results is used as basis for navigator analysis.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deephits=&lt;integer value&gt;</td>
<td>Used in order to override the default number of hits that is used as the basis for Deep Navigator computation. This will impact the search performance. Example: pricenavigator(deephits=1000) Default: According to index-profile setting (deep=no). Note: This limit applies within each search partition inside the search rows/columns. Hence, the actual number of hits evaluated will be larger than this value due to the result aggregation across search partitions.</td>
</tr>
<tr>
<td>sort=&lt;property&gt;/&lt;direction&gt;</td>
<td>Defines how the buckets (modifiers) within a String Navigator are to be sorted. &lt;property&gt; - Sorting algorithm. Valid values are: frequency - orders by occurrence within the buckets, name - orders by label name, number - treats the strings as numeric and use numeric sorting. This may be useful when you need discrete values, for example when performing numeric sorting of processor speed. &lt;direction&gt; - Sorting direction. Valid values are descending and ascending Example: sort=frequency/descending Default: According to index-profile setting (sort-by/sort-order).</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>filter=&lt;buckets&gt;/freq[/&lt;minlevels&gt;/&lt;maxlevels&gt;]</td>
<td>Defines how modifiers (buckets) within a String Navigator are filtered before they are returned back to the client. The buckets, freq and prefix parameters are required; the minlevels/maxlevels parameters are optional.</td>
</tr>
</tbody>
</table>

**Note:** This parameter applies application-specific result-side filtering. This is different from the cutoff parameter which applies limitations in each individual search row/column for performance optimization.

- **<buckets>** - Maximum number of returned buckets.
  After sorting the Modifiers (buckets) within the Navigator, use this attribute to cut off any trailing buckets. Corresponds to the filter-buckets index-profile attribute. For example, if buckets=10, then only the first 10 buckets are returned, according to the indicated sorting algorithm. The value "0" implies default value according to the index-profile.

- **<freq>** - Limits the number of returned buckets.
  After modifier/bucket filtering, use this attribute to remove buckets that have lower frequency counts. Corresponds to the filter-frequency index-profile attribute. For example, freq=2 indicates that only the buckets with 2 or more members are returned. Filtering based on frequency is performed first. The value "0" implies default value according to the index-profile.

- **<prefix>** - Prefix filtering.
  Only buckets having a name that starts with this prefix are returned. The value `*` will match all names.
  Default: According to index-profile setting (filter-buckets/filter-frequency). <prefix> does not have any corresponding index-profile setting, default is no prefix filtering. For example:
  
  ```
  filter=30/2/*
  ```

  - **<minlevels> and <maxlevels>** - Minimum and maximum levels of taxonomy.
    Use these two attributes to filter taxonomy (hierarchical) navigator values based on the number of levels in a taxonomy value. In taxonomy values, levels are separated by the `/` path separator. E.g. the taxonomy value ‘Europe/France/Provence/Marseille’ has four levels.

    - **<minlevels> attribute:** The number of levels in a taxonomy value must be greater than the number specified in the <minlevels> attribute in order for the taxonomy value to be included in the navigator.
    - **<maxlevels> attribute:** The number of levels in a taxonomy value must be less than or equal to the number specified in the <maxlevels> attribute in order for the taxonomy value to be included in the navigator.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E.g. the taxonomy value ‘Europe/France/Provence/Marseille’ has four levels and so, to be included as a navigator value, <code>&lt;minlevels&gt;</code> would need to be 3 or less and <code>&lt;maxlevels&gt;</code> would need to be 4 or greater. Although these attributes are optional, you must specify <strong>either both or neither attributes</strong>; it is not valid to provide a value for only one of these attributes. The attribute should be a positive integer n. The level separator is the forward slash character (/). It is currently not possible to change this at query time or via the index profile. If you need to change the default separator character, it is possible using a file patch in the index profile. Consult FAST Global Services for further details. <strong>Note:</strong> Be aware of the performance cost of using a very large taxonomy navigator: The filtering is performed in the QR Server, not in the individual search nodes.</td>
</tr>
<tr>
<td>cutoff=&lt;frequency&gt;/ &lt;minbuckets&gt;/ &lt;maxbuckets&gt;</td>
<td>Limit the amount of data that must be transferred and processed for deep string navigators. The Navigators can be configured to only return the most relevant values (buckets). <strong>Note:</strong> This cutoff filtering is performed within each search partition inside the search rows/columns. This is different from the filter parameter which performs result-side filtering only. You may combine the two parameters. • <code>&lt;maxbuckets&gt;</code> - Limit the number of buckets This parameter limits the number of unique values (buckets) that will be returned for a navigator. The default value is given by the index-profile configuration. This is the preferred way to enhance search performance when string navigators with large number of buckets are returned. &quot;0&quot; implies default value according to the index-profile. <strong>Note:</strong> The navigator sort algorithm (sort=...) does not impact the cutoff sorting algorithm. This means that if you sort buckets by name and combine it with a maxbucket based cutoff, the sorting will be based on the buckets with the highest frequency since the sorting prior to cutoff is based on frequency. • <code>&lt;frequency&gt;</code> - Limit the number of buckets by frequency. If the number of occurrences of a navigator value in a result set is less than or equal to the frequency value, then the navigator value is not returned. The default value is given by the index-profile configuration. &quot;0&quot; implies default value according to the index-profile.</td>
</tr>
</tbody>
</table>
### Table

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;minbuckets&gt;</td>
<td>Min value for frequency cutoff. If the number of unique navigator values for the query is less than this value, then no frequency cutoff is done, and all navigator values are returned from that search partition. When cutoff-frequency is used, this parameter can be used to specify a minimum number of unique navigator values that will be returned regardless of number of occurrences. The default value of this attribute is &quot;0&quot;, meaning that frequency cutoff is done regardless of the number of unique navigator values. &quot;0&quot; implies default value according to the index-profile. The parameters &lt;frequency&gt; and &lt;minbuckets&gt; should be used with care, it is normally recommended to use &lt;maxbuckets&gt; only. Default: According to index-profile setting.</td>
</tr>
</tbody>
</table>

### Navigation on Matching Scopes (Scope Navigation)

**Scope Navigation** (also referred to as Contextual Navigation) is applying navigation to scope fields.

Scope fields represent the content in a hierarchical structure as opposed to a flat field. The advantage of using scope fields is that it is not necessary to know the index schema in advance. Applying navigation on scopes lets you limit your search results by narrowing in on a scope such as a paragraph or sentence. The values that are shown in the navigators come from the scopes matching the query and not the full document.

Scope navigators are shallow. This means Scope navigators will aggregate over the N top ranked results within the result set. It is possible to create navigators over structural elements in the matching scope, as well as on scope attribute values and the content of scopes. When using a scope navigator you will always need to specify the aggregation patterns as part of the query, using advanced query parameter navigator syntax. Also note that the drill-down is performed in a different way than normal field-level navigators. For field-level (string/numeric) navigators you may use the query API interfaces for iterating over the returned navigators and drilling down. For scope navigators you need to apply the drill-down criteria directly into a dedicated query parameter `scopenavigation`.

### Matching Scopes

The ESP hit highlighting module computes the matching scopes per query, based on the configuration of the dynamic-type field attribute in the index profile. By default, the xml summary field will contain the scopes in the xml field that match the FQL query, embedded in an XML "envelope".

Here is an example teaser for the FQL query `xml:sentence:"bill"`:

```xml
<matches>
  <match>
    <sentence id="3">My name is <person base="Bill Clinton">Bill Clinton</person> and I have been to <location base="Norway">Norway</location>.</sentence>
  </match>
  <match>
    <sentence id="9"><person base="Bill Smith">Bill</person> lives in <location base="Clinton, MS">Clinton, Missouri</location>.</sentence>
  </match>
</matches>
```

**Tip:** We can have term highlighting in combination with matching scopes.

The user can specify which scopes to have returned in the result (in addition to the scopes actually matching), by setting return="yes" as a parameter of a selected scope node in the FQL expression.
If we want to search for documents where the words foo and bar occur within the same sentence, but want to have returned the paragraphs that contain these sentences, we can express this by the following FQL query:

```
xml:scope(paragraph, return="yes"):sentence:and("foo", "bar")
```

### Aggregation

At query time, the client specifies which navigators to use. This is done using the dedicated NAVIGATORS query parameter (API base parameter NAVIGATORS or HTTP query parameter rpf_navigation:navigators) mechanism. An example fragment of the URL passed to the QRServer might look like:

```
...&rpf_navigation:navigators=scopenavigator1(context=person@base,display="People"),
   scopenavigator2(context=location@base,display="Places")&...
```

As can be seen from the example above, it is only now that we actually tell the system what the placeholder navigators should actually do, in this case aggregate over the base XML attribute of the person and location XML tags, respectively.

The context of the desired aggregation is specified using the context argument to the navigator within the NAVIGATORS query parameter. In the above example "person@base" and "location@base", respectively.

Let's call the value of context for the context specification. Currently allowed context specifications are:

- Aggregate over surface forms
- Aggregate over base forms (extracted entities)
- Aggregate over XML (scope) structure
- Aggregate over XML structure, including attributes

### Result Sorting and Ranking features

The sorting/ranking of the results may be controlled by the index-profile and by a set of query parameters.

**Sort by field, geographical distance or rank profile**

Normally queries applied to a Composite Field are sorted by relevance score (rank) based on the default rank profile associated with this composite field. Alternatively queries may be sorted by the value of one or more fields and/or by an alternative rank profile.

In order to apply an alternative field/rank sorting, the following query parameters can be used:

- `SORT_BY` - Define one or more fields or a rank profile that the results shall be sorted by
- `SORT_DIRECTION` - Defines ascending or descending sort sequence. Only applies when sorting by field or geo specification. The direction can also be set using + or - in the SORT_BY parameter. In that case SORT_DIRECTION only defines the default sort direction.

**Tip:** Field sorting can only be applied to individual fields or geo specifications (not composite fields or scope fields)

Result sorting by field value requires a corresponding configuration in the index-profile. The following two index-profile options are available as attributes to the field specification:

- `fullsort="yes"` enables full multi-level sorting. This means that more than one field may be specified for SORT_BY, and true string sorting is applied. This is the recommended mode and can be used for all types of sorting.
- `sort="yes"` enables basic, single-level sorting. Only one field or rank profile can be specified for SORT_BY, and simplified string sorting is applied. This mode is included for backwards compatibility, and may provide slightly better query performance than fullsort="yes".
Multi-level Sorting

In this mode one or more index-profile fields and/or a Rank Profile can be used for sorting. Multi-level sorting requires that fullsort="yes" is set for this field in the index-profile. This sorting mode is recommended except for very high QPS applications where Single-level field Sorting may give somewhat better performance at the cost of less sorting flexibility.

Specifying a rank profile implies that an alternative rank profile is used to calculate the dynamic rank (relevance score) for this query.

Sorting by geographical data can be applied by specifying a rank-profile tuned for geo-data.

Tip: The default text sorting is based on case sensitive ASCII alphanumeric sorting.

Format

- API:

  Parameter name: SORT_BY Valid values: sortlevel> [sortlevel]*

- HTTP:

  sortby=<sortlevel> [sortlevel]*

The query parameter specifies a set of fields and/or a rank profile for sorting. If more than one is given, this implies multi-level sorting with the given sequence of elements. The fields may be text fields (type="string") or numeric fields (not composite fields or scope fields).

<sortlevel> indicates one sort level for the query. One of the sort levels specified can be a rank profile.

The format for <sortlevel> is:

[+|−]<sortbyfield>

The table describes the detailed format for the <sortlevel> components:

<table>
<thead>
<tr>
<th>Sort level component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[+</td>
<td>−]</td>
</tr>
<tr>
<td>&lt;sortbyfield&gt;</td>
<td>&lt;sortbyfield&gt; may be one of the following: n &lt;field&gt;: A valid field-name from the index-profile that includes the attribute fullsort=&quot;yes&quot;. n &lt;rank-profile&gt;: The name of a rank profile as defined in the index-profile. This sort level will then be based on the rank value for the document according to the given rank profile. n [rank]: Can be used to fetch a document's rank value. A rank profile can only be applied as one level within the sort specification.</td>
</tr>
</tbody>
</table>

Default

If SORT_BY is not specified, rank sorting is based on the first occurring rank profile within the default composite field as defined in the index-profile.
Example
The following example will sort the results by the price field (ascending numeric), then by the rank profile named 'fresh':

- Java API:
  ```java
  query.setParameter(new SearchParameter(BaseParameter.SORT_BY, "+price fresh");
  ```

- HTTP:
  ```
  sortby=+price%20fresh
  ```

Single-level field Sorting
In this mode a single numeric field or a rank profile can be used for sorting. Single-level field sorting is somewhat faster than multi-level sorting, and requires that sort="yes" is set for this field in the index-profile. Specifying a rank profile implies that an alternative rank profile is used to calculate the dynamic rank (relevance score) for this query.

Sorting by geographical distance is also based on the single level sorting parameter format, but also requires a user coordinate.

Format
- API:
  ```
  Parameter name: SORT_BY Valid values: field | rank-profile
  ```

- HTTP:
  ```
  sortby=field|rank-profile
  ```

The query parameter specifies an index-profile field or rank-profile for sorting. The field must be a numeric field (not within a scope field).

Sort direction is specified using + or - prefix in the SORT_BY parameter or using the SORT_DIRECTION query parameter as described in section Sort Direction.

A `+' prefix indicates ascending sort direction. A `-' prefix indicates descending sort direction.

Default
If SORT_BY is not specified, rank sorting is based on the rank profile with default="yes" in the index-profile.

Example
The following example will sort the results by the price field in ascending order:

- Java API:
  ```java
  query.setParameter(new SearchParameter(BaseParameter.SORT_BY, 
  ```
Random Sorting

FAST ESP provides two approaches to randomize results:

Simple random sort

The query parameter named `random` can be used to achieve random sorting as part of a single level sorting specification. If you use this parameter only the 4000 highest ranked documents will be sorted randomly. Therefore this parameter has less impact on performance than the second approach.

```
&random=<numeric seed value>
```

By providing the same seed for another query, documents will be presented in the same "random" order, given there have been no index updates. This enables you to preserve the same random order within a search user session.

The specified seed value will be input to a function that generates a random number based on the seed. This random number will be assigned to the document. The document results will then be sorted based on this value, achieving a random sorting.

Advanced random sort

If you want a more strict random sorting behaviour the second approach should be used. This approach will provide a random result sorting across the entire result set, not only the 4000 highest ranked, and can be part of a multi-level sorting expression.

Note: This approach has a negative effect on performance, so you may experience drop in QPS or increases in search latency when using this feature instead of normal ranking. Using the hashfield option has an additional negative effect.

```
[random:seed=<seed>:hashfield=<fullsort field>:addtorankmax=<max random value>]
```

By providing the same seed for another query, documents will be presented in the same "random" order, given there have been no index updates. This enables you to preserve the same random order within a search user session.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>Using only the seed option will give a random ordering of the search results, which is not stable between index updates.</td>
<td>Yes</td>
</tr>
<tr>
<td>hashfield</td>
<td>The field specified as input to the hashfield option must be an int32 index profile field specified with the fullsort=&quot;yes&quot; attribute. Fill this field with random or unique values (for example a sequence number populated by a document processing stage). This guarantees that the results are properly randomized even after an index update.</td>
<td>No</td>
</tr>
</tbody>
</table>
### Option Description

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>addtorankmax</td>
<td>A number between 0(zero) and the value specified for this option will be added to the documents rank. This allows you to further randomize results within an interval. It is the original (pre-sorting) value that is presented in the document summary, not the rank including the random value.</td>
<td>No</td>
</tr>
</tbody>
</table>

### Simple random sort

This example will randomly sort the 4000 highest ranked documents in the result set.

```
query=airplanes&random=12345
```

### Randomly sort entire result set

This example will randomly sort the documents in the entire result set.

```
query=airplanes&sortby=-[rank] +[random:seed=5432]
```

**Note:** The square brackets in [rank] are part of the syntax.

### Strict random sorting behaviour

This example will randomly sort the entire result set even after an index update has occurred.

```
query=airplanes&sortby=-[rank] +[random:seed=5432:hashfield=hashField]
```

### Randomly sort entire result set within interval

This example will assign a random value between 0 and 200 to the documents rank. It is not a big probability that a document at the end of the interval will be assigned a random value big enough to be ranked at the top, but the results will be shuffled. Thus, achieving a certain degree of randomization. The entire result set will then be randomly sorted according to the seed value.

```
query=airplanes&sortby=-[rank] +[random:seed=5432:addtorankmax=200]
```

### Sort Direction

Defines result sorting direction (ascending/descending).

This parameter only defines the default sort direction when +/- is not specified for the individual levels of the sort specification.
This parameter does only impact the sort direction for fields. A rank profile will always be sorted descending (highest ranked document first).

**Format**

- **API:**
  
  Parameter name: SORT_DIRECTION Valid values: "ASCENDING"|"DESCENDING"

- **HTTP:**
  
  sortdirection=ascending|descending

**Default**

descending

**Example**

- **Java API:**
  
  ```java
  query.setParameter(new SearchParameter(BaseParameter.SORT_DIRECTION, "ASCENDING");
  ```

- **HTTP:**
  
  ```
  sortdirection=ascending
  ```

**Rank Profile Sorting Limitations**

In order to achieve dynamic ranking for terms in the query (proximity, context), the following two conditions must be met:

- The selected (or default) rank profile must refer to the composite field(s) referred in the query
- At least one `<context>` element in one of the composite fields referred from the rank profile must match the query terms

Other terms in the query will only contribute to recall (i.e. filtering results).

**The Rank value returned in query results**

The query results includes a result field named `rank`. This result field provides the resulting rank value based on the different components of the rank profile. Note that this value is not normalized, but can be used to indicate the relative difference in relevance within one result set.

**Tip:** The `rank` value in the query result cannot be used for this purpose when you sort the results based on field value rather than rank profile. In this case the `rank` value is just an internal number and should be discarded.
Field Collapsing

Field Collapsing allows a folding of results with the same value for a given result field. This can be used in order to collapse results with given attributes. There are two field collapsing options. Field collapsing with and without document removal.

- A use case might be site collapse, similar as found on popular Web search engines. All results from the same Domain may in this way be collapsed into one result, typically presenting the result from this domain with the highest relevance score (rank).
- Another use case might be product field collapse, e.g. collapse all results with the same product code in the result set.

Field Collapsing (without document removal)

This feature allows you to collapse results, but is dependent on index-profile configuration. Collapsing can only be applied to one field in the Index, and the field must be of type int32.

This feature is, despite the name, not a full collapse but rather a re-sort that will show the top ranked result for each value of the collapse field before all the rest.

The ID field must be configured for Field Collapsing in the index-profile.

When the feature is activated in the index-profile, the query result returned will include an additional field named morehits. This result field is generated by the Query & Result Server and can have the values '0' or '1'. '1' indicates that there exists more hits for this collapsed field. This may typically be used to generate a 'more hits' link below this document in the result set.

Format

- API:

  Parameter name: COLLAPSING  Valid values: true|false

- HTTP:

  collapse=1|0  (collapse without argument equals collapse=1)

Default

- 0/false

Example

If you wish to find all documents concerning FAST or IBM or HP, but want to group these sites and present one result per company.

http: query=string("IBM FAST HP",mode="OR")&collapse

Field Collapsing (including document removal)

This feature can be controlled at query time, making it possible to select different collapse fields on a per query basis. Unlike the more traditional field collapsing feature, this feature allows for documents to be
removed from the result. This feature offers the following options/features (collapsed documents are always removed):

- Collapse on specified numeric field
- Collapse, but keep N number of collapsed documents

**Collapse on specified numeric field**

The field used for collapsing results can be determined at query time. Thus, it is not dependent on the index-profile. Although a numeric field is required, the most convenient type is uint32. The fieldname must be prefixed with batv.

**Format:**

- **HTTP:**

  ```
  &collapseon=batv<fieldname>
  ```

**Example**

- **HTTP:**

  This example will collapse all similar documents based on size

  ```
  query=string(%22IBM%20FAST%20HP%22,mode=%22OR%22)&collapseon=batvsize
  ```

  **Tip:** Make sure that the field is enabled for full/multi-level sorting (fullsort="yes")

**Collapse and keep N number of collapsed documents**

It is possible to keep a specified number of documents for each collapsed group. N specifies the number of documents to keep and these will be sorted according to decreasing rank within the group.

**Format:**

- **HTTP:**

  ```
  &collapseon=batv<fieldname>&collapsenum=N
  ```

**Example**

- **API:**

  When using the XML result template the results will be returned as follows (using N=3):

  ```xml
  <HIT NO="1" RANK="2514117" SITEID="1" MOREHITS="1" FCOCOUNT="3">
  <HIT NO="2" RANK="8662071" SITEID="1" MOREHITS="1" FCOCOUNT="3">
  <HIT NO="3" RANK="9874012" SITEID="1" MOREHITS="0" FCOCOUNT="3">
  ```

  For the text template the same values will be returned as:

  ```
  #fcoid #morehits #fcocount
  ```
Via the API you can retrieve fcoid, morehits and fcocount as summary fields. SITEID/fcoid provides the numeric value of the collapse field.

**Tip:** The value returned will only be equal to the field value if you use uint32 as the collapse field.

MOREHITS/morehits indicates if there are more hits for a given collapsed group in the COLLAPSED result set.

FCOCOUNT/fcocount gives the total number of hits for a given duplicate id in the ORIGINAL result set (before collapsing). This value might be used to activate a "more hits from this site" link. If collapsenum=1, then the COUNT field will be > 1 if there are duplicates. If collapsenum > 1 then the SITEID field should be used too in combination with the COUNT field.

When submitting the query, use the collapsenum option:

```java
Query.setParameter(new SearchParameter("collapseon","batvsize"));
Query.setParameter(new SearchParameter("collapsenum",3));
```

When receiving the results, the following should be done in order to determine whether a "more hits for this site" link should be presented:

```java
IDocumentSummaryField totalHitsForId = IDocumentSummary.getSummaryField("fcocount");
//Check if there are additional hits for this collapse ID that is removed from the result set
//number of hits presented for a collapse ID = collapsenum
if(totalHitsForId > collapsenum) {
    //present a "more hits from this site" link
}
// number of hits presented for a collapse ID = FCOCOUNT
else if(totalHitsForId > 1) {
    //Do something
}
//collapsing not performed
else
```

### Hit and Navigator Count

When using the new query-side field collapsing feature (collapse=remove) the count for the total number of returned results will reflect the actual number of returned hits after removing the collapsed entries.

The modifier values returned by navigators does not take collapsed documents into account. It is not possible to provide navigator count reflecting the collapsed result set.

### Dynamic Duplicate Removal

A result-side duplicate removal filter can be applied using the result-filter element within the index-profile in combination with the query parameters as described below.

A dynamic duplicate removal can be applied in addition to the normal duplicate removal that is performed by content connectors like the Enterprise Crawler. It may be used for several purposes, such as:
• Detect duplicates (same document) that resides in different collections
• Detect duplicates (similar documents but with different document ID/URL) that may originate from different content sources (e.g. a web server and a file server) and therefore not detected by the duplicate detection algorithm of the connector
• Detect perceived duplicates. This can be documents that are not identical, but should be treated as identical within a query application. The criteria may be that a set of fields are identical.

Tip: The dynamic duplicate removal may be expensive from a performance perspective, especially for large/multiple fields and/or large number of removed duplicates.

The duplicate filter applies to one or more fields. Only documents with different values for these fields will be returned for the query. If the referenced field has an empty value for a specific result and the field has a fallback-ref specified, then duplicate filtering will be attempted using the fallback.

Tip: In order to use this feature for a field, the indicated field must be defined with result="yes" or result="dynamic" in the index-profile.

The relation between the configuration in the index-profile and the query parameters is as follows:

• The result-filter configuration in the index-profile defines the default setting (when not using the query parameter). This means you can enable dynamic duplicate removal for all queries on a given set of fields. Refer to Chapter Index Profile Features Management in the Configuration Guide for details. If you use the query parameter DUPLICATIONREMOVAL=1 it is not required to configure the feature in the index-profile.
• Dynamic duplicate removal can be activated using the query parameter DUPLICATIONREMOVAL=1. In this case the query parameter setting overrides the index-profile configuration
• If you have defined dynamic duplicate removal using the result-filter configuration in the index-profile, the duplicate removal can be disabled on a per query basis using the query parameter DUPLICATIONREMOVAL=0

The duplicate removal function will ensure that the entry with highest rank among the duplicates is returned in the result set. The duplicate removal function is supported on text fields only.

The duplicate removal supports a maximum of two slots. A slot indicates an individual duplicate removal criteria. If two slots are defined, at least one of the slots must be identical in order to define two documents as duplicates.

Each slot can consist of more than one field, i.e slot1 can be the concatenation of f1 and f2, while slot2 can be the concatenation of f3, f4 and f5. In this case all of the indicated fields of a slot must be equal for two documents to be treated as duplicates.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API: DUPICATIONREMOVAL HTTP: rff_ddr:enabled</td>
<td>true</td>
<td>This parameter enables Dynamic Duplicate Removal on a per query basis. The parameter overrides the default setting in the index-profile. Default: Dynamic Duplicate Removal according to the index-profile configuration.</td>
</tr>
<tr>
<td>API: DUPREM_SLOT1 HTTP: rff_ddr:slot1</td>
<td>field_spec[,field_spec]</td>
<td>A specification of fields that will be the basis for duplicate removal within slot 1. The concatenated set of fields (each given by field_spec) indicated will be the basis for the duplicate check. If one or both of the slots are specified in the query, this slot specification overrides the slot configuration in the index-profile. The table describes the components of the field specification (field_spec). Default: The slot specification as defined in the index-profile.</td>
</tr>
<tr>
<td>API: DUPREM_SLOT2 HTTP: rff_ddr:slot2</td>
<td>field_spec[,field_spec]</td>
<td>A specification of fields that will be the basis for duplicate removal within slot 2. If 2 slots are defined, two documents</td>
</tr>
</tbody>
</table>
Table 3: Field specification (field_spec) for Dynamic Duplicate Removal

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsum&lt;field&gt;</td>
<td>bsum&lt;field&gt; indicates a given field. The field name as defined in the index-profile must be preceded with `bsum'.</td>
</tr>
<tr>
<td>bsum&lt;field&gt;/[bsum&lt;field&gt;]</td>
<td>An optional `bsum&lt;field&gt;' indicates a fallback field that is used in case the first field is empty.</td>
</tr>
</tbody>
</table>

**Tip:** If you use the dynamic duplicate removal feature the total number of hits count in the result set may not be accurate. The hit count will be based on an estimate, taking into consideration the number of duplicates detected in the presented result. This also means that the hit count may change when browsing through the result set.

Table 4: The following table shows possible values for one slot within the specification:

<table>
<thead>
<tr>
<th>Parameter setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsumproduct</td>
<td>The duplicate removal will be based on the value of product field</td>
</tr>
<tr>
<td>bsumtitle,bsumbody</td>
<td>The duplicate removal will be based on the values of title and body (both must be equal)</td>
</tr>
<tr>
<td>bsumbody/bsumteaser</td>
<td>body is used as a basis for the duplicate removal unless empty. If body is empty, the algorithm falls back to teaser</td>
</tr>
<tr>
<td>f1/f2,f3/f4,f5</td>
<td>Use f1 unless empty then use f2, concat f3 unless empty then use f4, concat f5</td>
</tr>
</tbody>
</table>

**Tip:** If you define dynamic duplicate removal for a field with result="dynamic", including a fallback-ref="<fieldname>", you will need to use the fallback syntax as described above. E.g. bsumbody/bsumteaser. If the body field is configured with result="dynamic" and fallback-ref="teaser".

The following example defines the duplicates as the set of document which have the same value for title and body:

- Java API:

```java
query.setParameter(new SearchParameter(BaseParameter.DUPLICATIONREMOVAL, true);
query.setParameter(new SearchParameter(BaseParameter.DUPREM_SLOT1,"bsumtitle,bsumbody");
```

- HTTP:

```
rff_ddr:enabled=1 rff_ddr:slot1=bsumtitle,bsumbody
```
Proximity relevance features

Proximity relevance implies that the distance between query terms in matching documents impacts the query results. FAST Enterprise Search Platform supports the following proximity mechanisms:

- Explicit Proximity Operators. It is possible to apply proximity operators (NEAR/ONEAR) in a similar way as the boolean query operators. This may increase the precision by returning only documents where a number of given terms occur near each other (e.g. in the same paragraph).

- Implicit Proximity Relevance Boosting. This is a set of configurable options that enables rank (relevance score in result set) boost for documents with good proximity match. The implicit proximity relevance boost is applied regardless of query language. Implicit proximity is only applied to query expression using AND and OR query operators.

FAST Enterprise Search Platform proximity features is applied as a part of the core search engine (Index-side proximity boost).

Index Side Proximity boost

The index side proximity boost feature is the preferred mechanism for performance and precision reasons, as it is performed on indexed content and can be applied across all results matching a query. Index side proximity uses a position index that will imply a slight increase of disk usage. Using result-side proximity may therefore be considered if it is vital to save disk space, the query volume is moderate and the limitation (only applied to the n top ranked hits) is acceptable.

Enable Result Side Proximity boost

You may use this parameter if you want to use the Result Side Proximity Boost instead of the Index Side (deep) Proximity Boost (for backwards compatibility). FAST Enterprise Search Platform supports proximity rank score within an extended result set. This implies that within e.g. the 200 top ranked documents, a result-side processing is applied to determine the best proximity match. Proximity in this context means that documents where the query terms appear closer to each other will get a rank boost.

This feature is included for backwards compatibility only.

Tip: Result Proximity boost feature must be enabled in the index-profile.

Format

- API:

  \[ PROXIMITYBOOST=[true|false] \]

- HTTP:

  \[ rpf_proximityboost:enabled=[1|0] \]

Default

1/true - Result proximity boost on if enabled in the index-profile

Number of hits for proximity boost

Determine the number of hits that is basis for the result proximity calculation.
**Format**

- **API:**

  ```
  PROXIMITYBOOST_HITS=[integer]
  ```

- **HTTP:**

  ```
  rpf_proximityboost:hits=[integer]
  ```

**Default**

200

**Tip:** For more information on this feature, please contact your FAST Account Manager.

---

**Document freshness boost**

Document Freshness Boost controls to what extent relative age of the documents impacts the rank (relevance score). If enabled, newer documents will appear higher up in the result set.

The freshness boost is controlled using the Rank Profile feature of the index-profile. There are two ways of controlling this feature on a per query basis:

- Select rank profile for the query by using the `SORT_BY` parameter. Multiple rank profiles (defined in the index-profile) may have different weight on the freshness boost within the total rank. Refer to section *Sort by Field, Geographical distance or Rank Profile*.
- Select the time base for calculating the freshness boost. The freshness boost is calculated based on the relative age of each document compared to the given time base. This is controlled by the parameter `DATETIME` as described below.

**Select the time base for calculating the freshness boost**

A string parameter specifying a search datetime value to use when calculating the document freshness boost.

If the parameter is not specified, then the current system time on the node running the Query and Result Server will be used.

**Format**

- **API:**

  ```
  Parameter name: DATETIME Valid values: <datetime-string>
  ```

- **HTTP:**

  ```
  qtf_freshnessboost:datetime=<datetime-string>
  ```

<datetime-string> must be according to one of the following formats:

- `YYYY-MM-DD`
- `YYYY-MM-DDThh:mm:ss`
- `YYYY-MM-DDThh:mm:ssZ`
### Date/Time

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>Four-digit year</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit month (01=January, etc.)</td>
</tr>
<tr>
<td>DD</td>
<td>Two-digit day of month (01 through 31)</td>
</tr>
<tr>
<td>T</td>
<td>The letter 'T'</td>
</tr>
<tr>
<td>hh</td>
<td>Two digits of hour (00 through 23) (am/pm NOT allowed)</td>
</tr>
<tr>
<td>mm</td>
<td>Two digits of minute (00 through 59)</td>
</tr>
<tr>
<td>ss</td>
<td>Two digits of second (00 through 59)</td>
</tr>
</tbody>
</table>

**Z Optional UTC time zone identifier**

All date/time values must be specified according to UTC (Coordinated Universal Time), also known as GMT time zone.

### Default

System time on the Query and Result Server.

### Examples:

- **Java API:**

  ```java
  query.setParameter(new SearchParameter(BaseParameter.DATETIME, "2003-08-27");
  
  query.setParameter(new SearchParameter(BaseParameter.DATETIME, "2003-08-27T20:16:00");
  ```

- **HTTP:**

  ```
  qtf_freshnessboost:datetime=2003-08-27
  qtf_freshnessboost:datetime=2003-08-27T20:16:00
  ```

### Query and result processing parameters

FAST Enterprise Search Platform supports customization of the query and result processing, by means of a Query Processing Pipeline and Result Processing Pipeline concept. The FAST Search Business Center allows for a Search View to be created. The Search View specifies which processing pipelines are used at query time. However, these settings can be overridden by using the presented parameters.

**Tip:** Consult FAST Technical Support for help to create custom query processing pipelines.

When custom pipelines are defined, they can be selected on a per query basis by means of the query parameters described below.

### Select Query Processing Pipeline

Select non-default Query Processing Pipeline.
Format
  • API:

    | Parameter name: QTPIPELINE Valid values: <pipeline-name> |

  • HTTP:

    qtppipeline=<pipeline-name>

Default
Default Query Processing Pipeline.

In FAST Enterprise Search Platform the default query processing pipeline is the pipeline supporting the FAST Query Language (FQL). In order to use alternative query language parsers an alternative query processing pipeline must be selected.

Select Result Processing Pipeline
Select non-default Result Processing Pipeline.

Format
  • API:

    | Parameter name: RPIPELINE Valid values: <pipeline-name> |

  • HTTP:

    rppipeline= <pipeline-name>

Default
scopesearch

The API query result methods can only be used if the custom result processing pipeline does not alter the result format as returned from the Query and Result Server.

Enable document level security
Document level security ensures that searchers only receive results containing documents that they have permission to access. Several parameters are available to configure document level security in ESP. Note that you must first install the FAST Security Access Module (SAM) before configuring document level security in ESP.

Document level security is implemented with the FAST Security Access Module (SAM). SAM does not provide user authentication, so your ESP search application must supply user identification with the following query parameters.

The `QTF_SECURITY_ENABLED` parameter enables document level security. The `QTF_SECURITY_UID` parameter specifies the user ID for a specific query.

Use the results-side equivalent parameters of `RPF_SECURITY_ENABLED` and `RPF_SECURITY_UID` to enable post query results removal. This is a secondary means of enforcing security in SAM that works on documents in a user's search results window, removing documents that the user does not have access to at the actual time that the query is performed.
The search front end passes the appropriate user identification information to SAM automatically.

**Note:** The `qtpipeline` parameter specifies the QRServer query transformation pipeline used to process a user's query. The pipeline stage contains a list of modifications the user's query goes through before having the actual query done. Make sure that your QRServer query transformation pipeline includes the appropriate SAM query transformation stage: `securityfql`. See the section on "Select Query Processing Pipeline" for more information.

See the *FAST ESP Security Access Module (SAM) User Guide* for more information about implementing document level security.

### Format

<table>
<thead>
<tr>
<th>API</th>
<th>HTTP</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QTF_SECURITY_ENABLED</td>
<td>qtf_security:enabled</td>
<td>Enables document level security. Valid values: true/false</td>
</tr>
<tr>
<td>QTF_SECURITY_UID</td>
<td>qtf_security:uid</td>
<td>Specifies the security identity of the user performing the query; use one of two forms:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Security identifier – A security identifier in the primary SAM domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>&lt;domain id&gt;:&lt;security identifier&gt;</code> – A combination of a valid SAM domain ID and a security identifier in that SAM domain separated by a colon</td>
</tr>
<tr>
<td>QTF_SECURITYFQL_ENABLED</td>
<td>qtf_securityfql:enabled</td>
<td>Enables document level security. Valid values: true/false</td>
</tr>
<tr>
<td>QTF_SECURITYFQL_UID</td>
<td>qtf_securityfql:uid</td>
<td>Specifies the security identity of the user performing the query; use one of two forms:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Security identifier – A security identifier in the primary SAM domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>&lt;domain id&gt;:&lt;security identifier&gt;</code> – A combination of a valid SAM domain ID and a security identifier in that SAM domain separated by a colon</td>
</tr>
<tr>
<td>RPF_SECURITY_ENABLED</td>
<td>rpf_security:enabled</td>
<td>Enables post query results removal (last minute access rights). Valid values: true/false</td>
</tr>
<tr>
<td>RPF_SECURITY_UID</td>
<td>rpf_security:uid</td>
<td>Specifies the security identity used to handle post query results removal, in one of two forms:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Security identifier – A security identifier in the primary SAM domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>&lt;domain id&gt;:&lt;security identifier&gt;</code> – A combination of a valid SAM domain ID and a security identifier in that SAM domain separated by a colon</td>
</tr>
</tbody>
</table>

**Note:** You only have to specify this parameter when you have enabled post query results removal.

### Examples:

**Java API:**
In a query, enter the user ID as a query parameter (QTF_SECURITY_UID), and set QTF_SECURITY_ENABLE to true:

```java
query.QTF_SECURITY_ENABLE = true;
query.QTF_SECURITY_UID = "aaearkgjshtttttsa"
```

**HTTP:**

```javascript
query=anthrax&qtf_security:enabled=true&qtf_security:uid="aaearkgjshtttttsa"
```

**FQL:**

```javascript
string("adams",qtf_securityfql:enabled="true",qtf_securityfql:uid="aaearkgjshtttttsa")
```

### Cache lines

Query results are cached in groups of ranked results called cache lines.

In use cases where the offset parameter is used increasingly to retrieve all (or at least many thousands) results, it can be beneficial for search latency to avoid the query to be reevaluated in the search core for every new identical query just with different offset setting. The cachelines setting in a query can be used to prepare the search core for subsequent high offset setting queries. This can help reduce average search latency, and reduce CPU and disk I/O load on the search servers.

The number of cache lines required to hold the results is normally determined by fsearch. The fsearch process determines the minimum number of cache lines to cache based on the requested hits and offset values. The number of results in a cache line is the same for all cache lines, and the default size is 200.

For example, assuming that the cache line size is 200:

<table>
<thead>
<tr>
<th>Hits</th>
<th>Offset</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>hits=10</td>
<td>offset=20</td>
<td>The user requests result number 21-30. Since the first cache line holds results 1-200, only the first cache line is cached.</td>
</tr>
<tr>
<td>hits=10</td>
<td>offset=195</td>
<td>The user requests results ranked 196-205. In this case the system will cache the first 2 cache lines (1-200, 201-400).</td>
</tr>
</tbody>
</table>
| hits=10 | offset=205 | The user requests result number 206-215. In this case, there are 2 possibilities:  
  - If there are multiple fsearch processes running (this is the case for most installations), the system will cache the first 2 cache lines (1-200, 201-400).  
  - If there is only one single fsearch process running, the system will cache only the second cache line (201-400). |

The cache line size can be altered by setting the `cacheLineSize` variable in the `fsearch.addon` file. For example:

```
cacheLineSize 400
```

Alternatively, you can override the default cache behavior at query time using the clines query parameter.

The clines query parameter specifies an additional number of cache lines to cache. For example, if the user requests:

```
hits=10, offset=20, clines=3
```
the system will cache the first 4 cache lines. The first one due to the specified number of hits and offset, the additional 3 as indicated by the clines query parameter. In this case, the first 800 results are cached assuming the cache line size is 200 (1-200, 201-400, 401-600, 601-800).

**Note:**

- Each fsearch process maintains its own result cache.
- Once a result set is cached, it will remain cached until that cache element is invalidated and flushed from the cache. Cache element invalidation will happen either when the result cache is full and the periodic LRU (Least Recently Used) cache removal is run or due to result cache element lifetime timeout (by default one to two hours).
- If the clines query parameter is set to 0, the system ignores its value.
- If the requested hits + offset requires more cache lines than what is indicated in the clines parameter, the system ignores the clines value.
- The query will not be rerun as long as the requested hits + offset is within the existing cache. Not even when the user changes the clines value. This reduces unnecessary querying.
You can use the FAST Query Language (FQL) to perform exact searches and to narrow the scope of your search to values belonging to a specific FAST Enterprise Search Platform field, composite field or scope field.

A query language expression can contain a number of nested sub-expressions which includes query terms, scope specifications and operators (such as boolean operators):

- **Query Terms** - One or more words, strings or numeric values to search for in a query.
- **Scope specification** - A Scope Specification limits the possible matching sections of the documents to a specific field, composite field or a scope structure within the field. For individual fields and composite fields the scope specification indicates a specific field as the scope of the query. For scope fields the scope specification indicates a given scope structure within a scope field.
- **Operators** - Different operators may apply boolean operations (AND, OR, etc.), may define certain constraints to the operands (e.g. filter()) or may specify data types and attributes to the data (such as linguistics operations).

The following FQL query example will search for the terms `hello` and `world` in the body of an indexed document.

```
body: ("hello" and "world")
```

- `body:` limits the scope of the query to the body field within the document.
- "hello world" is the primary argument to the string() operator, indicating the textual terms to search for.
- `and` indicates that the logical query operator `AND` shall be applied to "hello world".

**Tip:** For Boost and Blocks to function properly, the query must be annotated with the annotation_class attribute. The attribute value must be set to `user`. For example:

```
string("hello world", mode="and" annotation_class="user")
```
Terminology

The query syntax and expressions use the following terminology:

Table 5: Terminology used in this Section

<table>
<thead>
<tr>
<th>This terminology...</th>
<th>denotes...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-expression</td>
<td>A sub-expression is used to indicate a sub-part of a query expression. Unless otherwise stated this indicates either an operator expression, e.g. <code>NEAR(a, b)</code> or a parenthesis expression, i.e. any sub-part of a query enclosed in parentheses. E.g. <code>(a and string(&quot;hello world&quot;))</code></td>
</tr>
<tr>
<td>Prefix Operator</td>
<td>A Prefix Operator applies to one or more operands and optional parameters enclosed in parentheses. A prefix operator may for instance indicate matching conditions such as <code>AND(term1, term2)</code></td>
</tr>
<tr>
<td>Infix Operator</td>
<td>An Infix Operator exists between its left and right operands, as in A AND B. Refer to section Query Operators for syntax details.</td>
</tr>
<tr>
<td>In Operator (:)</td>
<td>The In Operator is a colon and indicates that the expression following the colon must be found within the scope specification preceding the colon. E.g. <code>content:body:sentence:&quot;hello world&quot;</code></td>
</tr>
<tr>
<td>Parameter</td>
<td>Operators may accept one or more parameters. A parameter may be a text string, a numeric value (e.g. the word distance for near()), or a true/false value.</td>
</tr>
</tbody>
</table>

The Field/Scope Specification

The Field/Scope Specification limits the content scope to given regions of the indexed content. Such a region may be identified by:

- A Composite Field containing a set of text fields within a document
- A single non-scope Field (of any data type supported in FAST Enterprise Search Platform)
- A Scope (element or sub-path) within a Scope Field

Default Field/Scope Specification

- If you do not include a field/scope specification, your query takes effect on the default composite field or default scope field (identified by the `default` attribute in the index-profile definition).
- The field/scope specification must always end with a colon (In operator). Expressions including numeric query terms must always include a Field/Scope Specification.
- Scope names are case sensitive. Meaning, in the syntax `xml:book`, the book element is case sensitive. Queries targeting scope fields can be treated with or without case sensitivity. This is determined by the tokenizer and more information can be found in the Configuration Guide. Thus, this functionality can not be steered at query time.
- Default scope type is String.
- Default mode is phrase
Addressing Fields and Composite Fields

In the FQL syntax a Field Specification (the In Operator) may be applied to:

- A single term or string, e.g.
  
  ```
  title:shakespeare body:"to be or not to be"
  ```

- An operator, e.g. the string() operator:
  
  ```
  body:("to be or not to be")
  ```

In this case the scope specification applies to the entire operator expression.

- A sub-expression enclosed in parentheses, e.g.
  
  ```
  keywords:(play and shakespeare)
  ```

In this case the scope specification applies to everything within the parentheses.

Addressing scope fields

For queries against a scope field, the top level scope specification in the query must always start with the name of the scope field (as defined in the index-profile). A Scope Field is an index entity that holds a hierarchical content structure, typically derived from XML. The scope field is also sometimes named the root scope. An index-profile may contain more than one Scope Field.

The FAST ESP feature named Entity Extraction can extract entities such as person names, countries, acronyms etc. FAST ESP is shipped with a range of predefined entities that, provided they occur in a document, can be addressed using scope search.

Tip: There is no support for auto-typing on scope fields. Meaning, numeric values will not be converted to string values if a string scope is addressed. Nor will string values be converted to numeric values.

In the following example the addressed Scope Field is named `xml`. This is equal to the scope field defined in the default index-profiles. Each element of the scope specification is followed by a colon. The element(s) following the scope field will address scope nodes within the hierarchical content structure.

```xml
xml:book:shakespeare
```

In the FQL syntax a Scope Specification may be applied to:

- A single term or string, e.g.
  
  ```
  xml:book:"shakespeare"
  xml:book:"to be or not to be"
  ```

- An FQL operator, e.g. the string() operator:
  
  ```
  xml:paragraph:string("to be or not to be")
  ```

In this case the scope specification applies to the entire operator expression.
• A sub-expression enclosed in parentheses, e.g.

\[
\text{xml:paragraph:}("\text{play}\ and\ \"\text{shakespeare}\")
\]

In this case the scope specification applies to everything within the parentheses.

Scope specifications may be nested:

\[
\text{xml:paragraph:}("\text{shakespeare}\ and\ \text{sentence:} \"\text{to\ be\ or\ not\ to\ be}\")
\]

In this case `sentence' must be a sub-scope to the scope `paragraph' within the scope field `xml'. Furthermore, if there are multiple scopes named `paragraph' in the document, the expression within the parentheses must match the same `paragraph' scope.

The depth of the FQL tree is limited by the available stack space in the QRserver. Running out of stack space causes the qrserver to crash. Avoid using nested FQL expressions of excessive depths (1000 or more). For example, instead of using binary \texttt{or} operators as in

\[
\text{or(t1,or(t2, or(...))}
\]

use

\[
\text{or(t1, t2, ...)}
\]

The first example is limited by the stack space to some thousand ti terms (one level for each term). The second form allows more terms.

\textbf{Tip}: The In operator (:) only indicates that the following expression must be found within the indicated scope or any of its descendants. It does not support the immediate child axis. This is similar to using `//\' in XPath. For more information concerning XPath, please see the World Wide Web Consortium's pages, www.w3c.org.

\section*{Addressing text scopes}

The following description applies when querying textual scopes (of type `string').

A Scope Specification always address the indicated scope and all sub-scopes. Given that a scope field named xml is defined in the index-profile, the simple FQL expression

\[
\text{xml:book:shakespeare}
\]

will match any document that contains the term `shakespeare' within a scope named `book' or any sub-scopes descending from `book'.

The full path of a scope is not required. I.e.

\[
\text{xml:book:shakespeare} \\
\text{xml:book:chapter:shakespeare} \\
\text{xml:book:chapter:sentence:shakespeare} \\
\text{xml:book:sentence:shakespeare}
\]

will all match the term `shakespeare' in any of the sentences within chapters of the book.

Let's assume a scope structure where a scope name appears on two different levels in the scope structure, e.g. the following XML example:
The query expression
xml:a:c:winnie
will match `winnie' both in document 1 and 2. If you want to match `winnie' in document 1 only, you can in fact do that by specifying:

xml:a:andnot(c:winnie, b:c:winnie)

Addressing numeric scopes

The following description applies when querying numeric scopes (of type int32, uint32, float, double or datetime).

A Scope Specification addressing a numeric scope must include the name of the numeric scope where the match may occur. This is different from textual scopes where a match may occur in any sub-scope descending from the scope addressed by the scope specification. Given that a scope field named xml is defined in the index-profile, the simple FQL expression

xml:price:500.50

will match any document that contains the float value 500.50 within a scope named `price'.

The complete path of a numeric scope is not required. I.e.

xml:price:500.50
xml:product:price:500.50
xml:products:product:price:500.50
xml:products:price:500.50

will all match if the scope `price' contains the value `500.50'.

The difference from addressing textual scopes is that the expression

xml:products:product:500.50

will not match the numeric scope named `price'.

Addressing Scope and Entity Attributes

You address scope attributes by using the @ prefix. Which attributes are available is determined by which Entity Extractors have been used.

Searching for attribute values can be performed like this:

<slope-field>:<scope>:@<attribute>:string("<queryterm>")
<slope-field>:<scope>:@<attribute>:string("<terms to search for>", mode="and")
Tip: Attribute values can only be referred to using strings

The pre-defined entity, date, has an attribute named base. This can be referred to using the following syntax.

```
xmldate:@base:"2001-01-22"
```

The example will return all documents that have this date specified.

Returning Scope

Queries may have a return="yes" specification as a parameter to a scope, which specifies the scope to return. Each scope field should have one return="yes" specification. If no return="yes" specification is found, then qt_fease will automatically add one. If more than one return="yes" specification is given for a scope field, one is chosen.

Search for Scope Existence

Search for scope existence is supported, using the scope() operator:

```
xmla:scope(c)
```

will return all documents that contain at least one scope `c' within `a' in the scope field named xml.

Tip: Such an expression will match if any scope exists, even if the scope is empty.

It is possible to use the scope() operator in proximity expressions. The following query will match if there exists a sub-scope named 'date' within a proximity distance of 20 words from the string "winter olympics":

```
xmlnear("winter olympics", scope(date), N=20)
```

Querying recursive scopes

Recursive scopes means that the same scope name appears in different places in the scope tree, e.g.

```
<A>
  hello
  <B>
    <A>world</A>
  </B>
</A>
```

A scope expression that is an argument of a query operator evaluates to the largest region of the document that satisfies the scope specification. This region might not be contained in the scope specification surrounding the operator expression and the whole expression will in this case return different results from a similar XPath expression.

The query:

```
B:A:world
```

will thus not return the document example above.

Tip: This limitation only occurs when querying recursive scope structures .
Scope specification examples

Consider the following XML example, mapped to the scope field `xml' within FAST Enterprise Search Platform:

```xml
<book>
  <title>Winnie the pooh</title>
  <chapter>
    <num type="int32">1</num>
    <sentence>Winnie came out of the forest</sentence>
  </chapter>
</book>

<book>
  <title>Murder in the forest</title>
  <chapter>
    <num type="int32">4</num>
    <sentence>Peter was running in the forest</sentence>
  </chapter>
</book>
```

The table explains how scope queries match this scope structure:

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) xml:winnie 2) xml:book:winnie 3) xml:book:title:winnie 4) xml:book:chapter:winnie 5) xml:sentence:winnie 6) xml:appendix:running</td>
<td>All queries will match the first document. (1) and (2) will match both the title and the first sentence of the first document. (3) will match the title of the first document. (4) and (5) will match the first sentence of the first document. (6) will match the sentence in the appendix of the first document</td>
</tr>
<tr>
<td>1) xml:chapter:running 2) xml:book:and(forest, num:4)</td>
<td>All queries will match the second document only. (2) also requires that the chapter number where 'forest' appears must be 4.</td>
</tr>
<tr>
<td>xml:appendix:winnie</td>
<td>This query will not match any of the documents</td>
</tr>
</tbody>
</table>

Handling Reserved Words and Special Characters

The following words are reserved within the query language:

- and, or, any, andnot, rank, near, onear, int, float, double, datetime, max, min, range, phrase, filter, not, string, starts-with, ends-with, equals, count

**Tip:** Reserved words (operators) and characters are not case sensitive, but the use of lower case is recommended for future compatibility.

If you want to express any of these words as terms or scope names in a query expression, they must be quoted.
Search for reserved words:

```
"any" OR "and" OR "rank" or("any", "and", "rank") string("any and rank", mode="OR") phrase(this, is, a, "phrase")
```

A scope is named `range`. This must be quoted in the scope specification. The two expressions are equivalent.

```
xml:"range":string("hello world", mode="OR")
xml:scope(range):string("hello world", mode="OR")
```

Characters such as double quotes must be escaped when used inside a double quoted string. This may appear when using double quotes within the simple query language syntax:

```
string("tigger \"winnie the pooh\"", mode="simpleall")
```

Table 7: Characters to escape inside double quoted strings

<table>
<thead>
<tr>
<th>Reserved character</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newline</td>
<td>\n</td>
</tr>
<tr>
<td>Carriage return</td>
<td>\r</td>
</tr>
<tr>
<td>Tab</td>
<td>\t</td>
</tr>
<tr>
<td>Backspace</td>
<td>\b</td>
</tr>
<tr>
<td>Form feed</td>
<td>\f</td>
</tr>
<tr>
<td>Double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>Single quote</td>
<td>'</td>
</tr>
<tr>
<td>Backslash</td>
<td>\</td>
</tr>
</tbody>
</table>

FQL does not always require a double quoted string. For example, `and(cat, dog)` is valid FQL even though `cat` and `dog` are not in double quotes. In this scenario, the following table lists characters that cannot be the first character or a subsequent character in a search term:

Table 8: Reserved characters in non-quoted query terms

<table>
<thead>
<tr>
<th>Reserved character</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab</td>
<td>These characters cannot be used anywhere in a non-quoted search term.</td>
</tr>
<tr>
<td>Newline</td>
<td></td>
</tr>
<tr>
<td>Carriage return</td>
<td></td>
</tr>
<tr>
<td>Space</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
<tr>
<td>]</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>]</td>
<td></td>
</tr>
</tbody>
</table>

These characters can occur anywhere in a non-quoted search term except as the first character.
FQL splits a query into query operators (AND, OR etc.) and query strings (not individual words). The query strings are then tokenized as applicable to your installation and language. Note that the tokenizer may remove other special characters as part of the tokenization process. For example, in a generic ESP installation with the query `and("[king]", "<queen>")`, although the `, `, and `>` characters are successfully parsed and the search terms are sent to the tokenizer as `king` and `<queen>`, the default tokenizer removes the special characters, leaving the final search terms as `king` and `queen`. Escaping the special characters doesn't help, since they are not characters which are eligible for escaping within a double quoted string.

When handling query terms from an end-user input (or another application) it is recommended to use the `string("<query terms>", mode="AND|OR|PHRASE")` operator to represent the query terms into the total query expression. In this way there is no possible conflict with reserved words in the query language.

Scope names (except the name of the scope field as defined in the index-profile) may contain special characters. When using special characters in a scope specification in queries, the special characters must be escaped using backslash, or the full scope name must be enclosed in double quotes. If the query application does not have full control of the sub-scope names, it is recommended always to double quote scope names, e.g.:

```plaintext
xml:"Scope_1":"Scope-2":("hello" or "world")
```

**XML mapping example**

The following examples present how XML can be mapped to scopes, and how to query the content. The examples assume that the scope field is named xml:

```xml
<book> ...
    <chapter>
        <num type=int32>5</num>
        <heading>Tigger and Winnie</heading>
        <sentence>Tigger came out of the forest</sentence>
        <sentence>Winnie the Pooh was waiting for him</sentence>
    </chapter> ...
</book>
```

Assuming that the scope field is named 'xml', this will be mapped to the following scope structure:

- book: (String)
- chapter: (String)
- num:5 (type=int32)
- heading:"Tigger and Winnie" (String)
- sentence:"Tigger came out of the forest" (String)
- sentence:"Winnie the Pooh was waiting for him" (String)

You may search for the phrase "winnie the pooh" in chapter 5 with one of the following queries:

- `xml:book:chapter:and(num:int(5), sentence:("winnie pooh"))`
- `xml:book:chapter:and(num:5, sentence:("winnie pooh"))`

The first query use an explicit typing (int(5) indicates that 5 is an integer). The second query use implicit typing (5 is implicit typed as an integer).
xml:book:chapter: indicates that the following and() expression must be within the same chapter of the book. In this example the chapter number is mapped to a numeric scope (int32). Hence, it is possible to search for the terms in any of the five first chapters of the book:

```xml
xml:book:chapter:and(num:range(min, 6), sentence:("winnie pooh"))
```

The following alternative XML representation may also be used, where the chapter number is represented as an attribute instead:

```xml
t <book> ...
  <chapter num="5">
    <heading>Tigger and Winnie</heading>
    <sentence>Tigger came out of the forest</sentence>
    <sentence>Winnie the Pooh was waiting for him</sentence>
  </chapter>
 </book>
```

You may then search for the phrase “winnie the pooh” in chapter 5 with the following query:

```xml
xml:book:chapter:and(@num:("5"), sentence:("winnie pooh"))
```

@num: will address the attribute named `num'.

You would not be able to search for the sentence within the five first chapters as described above, as the attribute value `5' will be represented as a string in the index, not an integer.

### XML Name Space

FAST ESP ignores name space information in XML element names when converting to scopes during document processing. Hence, you must not include name space in the queries (only use the element name).

Given the XML fragment:

```xml
<MY_NAMESPACE:body> hello world
```

You will match this with the query:

```xml
xml:body:string("hello world", mode="and")
```

but not with the query:

```xml
xml:MY_NAMESPACE:body:string("hello world", mode="and")
```

### Wildcard Expression

A wildcard expression indicates a single term or phrase that includes one or more of the wildcard characters `*` and `?`.

The wildcard=ext query parameter supports several wildcard characters in addition to the basic `*` and `?` characters.

FAST Enterprise Search Platform supports wildcard search for individual string-type Fields, Scope Fields and Composite Fields. The capabilities are slightly different, as described below.
**Tip:** Wildcard queries will return a syntax error if the target is a composite field and this composite field is not configured for wildcard.

### Wildcard queries for Fields and Composite Fields

In the index-profile it is possible to define two different modes for field and composite field wildcard support. The table describes the alternative wildcard formats for these two modes:

The following limitation applies:

- A wildcard term will not match lemmatized forms, only the original term in the document

#### Table 9: Wildcard Query Term Expression for Composite Fields

<table>
<thead>
<tr>
<th>Wildcard format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>text*</code></td>
<td>Full wildcard query on single terms and phrases, supporting <code>*</code> and <code>?</code> according to regular expression syntax. The target field or composite field for this query item must be configured for full wildcard search in the index-profile (wildcard=&quot;full&quot;).</td>
</tr>
<tr>
<td><code>*text</code></td>
<td><code>*ak?sp*r*</code> will find documents containing the word 'Shakespeare'. In most cases this is the recommended wildcard feature to use. may, however, be used in case of extreme query performance requirements.</td>
</tr>
<tr>
<td><code>*text*</code></td>
<td></td>
</tr>
<tr>
<td><code>te?t</code></td>
<td></td>
</tr>
<tr>
<td><code>*te?t</code></td>
<td></td>
</tr>
<tr>
<td><code>string(&quot;this exam*&quot;)</code></td>
<td></td>
</tr>
<tr>
<td><code>string(&quot;this *ample&quot;)</code></td>
<td></td>
</tr>
<tr>
<td><code>string(&quot;this *ampl*&quot;)</code></td>
<td></td>
</tr>
<tr>
<td><code>string(&quot;this exam?le&quot;)</code></td>
<td></td>
</tr>
<tr>
<td><code>string(&quot;thi* exam?le&quot;)</code></td>
<td></td>
</tr>
</tbody>
</table>

#### Prefix wildcard query item, matching all words starting with the given text. The target field or composite field for this query item must be configured for prefix wildcard search in the index-profile (wildcard="prefix").

- `paramet*` will match all terms that starts with 'paramet', e.g. 'parameter' and 'parameters'. Prefix queries may be applied on composite fields.

### Wildcard queries on Scope Fields

For Scope fields it is not necessary to configure wildcard support in the index-profile. The table describes the alternative wildcard formats for Scope Fields:

**Tip:** The **Any token** operator can not exist by itself in a query. E.g. `xml:*"**` is not allowed.

#### Table 10: Wildcard Query Term Expression for Scope Fields

<table>
<thead>
<tr>
<th>Wildcard format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>text* *text *text* te?t *te?t ? *?</code></td>
<td>Full wildcard query on single terms and phrases, supporting <code>*</code> and <code>?</code> according to regular expression syntax. <code>*ak?sp*r*</code> will find documents containing the word 'Shakespeare'.</td>
</tr>
<tr>
<td><code>xml:body:(&quot;this exam*&quot;) xml:body:(&quot;this *ample&quot;)</code></td>
<td>Full wildcard query on phrases, supporting <code>*</code> and <code>?</code> according to regular expression syntax.</td>
</tr>
</tbody>
</table>
### Wildcard format

<table>
<thead>
<tr>
<th>Description</th>
<th>XML Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml:body:(&quot;this <em>ampl</em>&quot;) xml:body:(&quot;this exam?le&quot;)</td>
<td></td>
</tr>
<tr>
<td>xml:body:(&quot;thi* exam?le&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

### Example:

The query:

```plaintext
fast search transf*
```

will get proximity and context ranking for the terms ‘fast’ and ‘search’, but not for terms in the document matching ‘transf*’. This means that you will get higher rank if ‘fast’ and ‘search’ appears close to each other, but it will not impact the rank where the terms matching ‘transf*’ appears in the documents.

### Execution Speed and Memory Usage of Wildcard Queries on Scope Fields

Use the minchunksize and maxchunksize parameters to control the memory usage when querying scope fields with wildcards.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxchunksize</td>
<td>128x1024</td>
<td>The maxchunksize parameter specifies the size of the read buffers used for each of the terms found in the wildcard expansion. If you are running out of memory due to wildcards, you can try to lower this value. If you want to try to speed up execution, and have no problems with memory usage, you can increase this value.</td>
</tr>
<tr>
<td>minchunksize</td>
<td>128x1024</td>
<td>The minchunksize parameter controls the number of disk reads. If you have little memory you can try to set the minchunksize parameter to low values, 0 is the lowest. This will give more disk reads, and the system might get slower.</td>
</tr>
</tbody>
</table>

### Any Token

This operator can be used to match arbitrary tokens. The notation used is a **`.`**.

The query matches sentence scopes that contain some arbitrary token, followed by a "hello" token, two arbitrary tokens, then "my world" and at least another two arbitrary tokens.

```plaintext
xml:sentence:"* hello * * my world * *
```

This query matches all documents where the author is presented with an arbitrary token in front of the name, and where the last part of the name can be anything.

```plaintext
author:equals("* M. Smith-*")
```

### Wildcard Cutoff

Wildcard cutoff reduces the query performance impact of wide wildcard queries.

Using very wide wildcard queries (returning too many results) may reduce the query performance substantially. It is possible to avoid such a reduction in query performance by configuring a cutoff limit for wildcard processing.
Two cutoff options are available: Hard cutoff makes the query return a query error and no hits are returned, while soft cutoff makes the query return an incomplete result. The latter means that all documents matching the wildcard expression may not be found.

The wildcard cutoff options apply to normal, composite and scope fields.

**Note:** Hard wildcard cutoff is automatically chosen over soft wildcard cutoff if the value of hard wildcard cutoff is smaller.

**Hard Wildcard Cutoff**
Define hard wildcard cutoff to make the query return a query error when exceeding the given limit, without returning any hits.

The hard wildcard cutoff is a value for the maximum number of terms a wildcard can expand to before a query error is returned. The default value is 500. The value -1 represents an unlimited number of terms.

**Example**
Submitting the query string "t*" forces FAST Enterprise Search Platform to expand the letter "t" to all found terms starting with "t" and to return all matching documents. For example, documents containing template, templates, tea, tiles, etc. Setting the hardwildcardcutoff in fsearch.addon, for the correct cluster, limits ESP to expand the query to 800 different terms. If more terms are found, a query error will be returned.

```
hardwildcardcutoff=800
```

**Soft Wildcard Cutoff**
Define a soft wildcard cutoff to make the query return a result within the given limit.

Note that the result will be incomplete; all documents matching the wildcard expression may not be found. The soft wildcard cutoff is a value for the maximum number of terms a wildcard can expand to. Terms beyond this limit are silently ignored. The default value is -1. The value -1 represents an unlimited number of terms.

**Example**
Submitting the query string "t*" forces FAST Enterprise Search Platform to expand the letter "t" to all found terms starting with "t" and return all matching documents. For example, documents containing template, templates, tea, tiles, etc. Setting the softwildcardcutoff in fsearch.addon, for the correct cluster to 500, limits ESP to expand the query to 500 different terms.

```
softwildcardcutoff=500
```

**Wildcard matching and Tokenization**
Wildcard matching is performed on the tokenized content, and performed on token level, i.e. it is not possible to match white spaces with wildcard characters.

The table lists a set of not supported wildcard matching:

<table>
<thead>
<tr>
<th>The query...</th>
<th>...will not match the content</th>
</tr>
</thead>
<tbody>
<tr>
<td>string(&quot;this*example&quot;, mode=&quot;PHRASE&quot;)</td>
<td>&quot;This example text shows wildcard capabilities&quot; (The wildcard term will not match the white space).</td>
</tr>
<tr>
<td>string(&quot;this?example&quot;, mode=&quot;PHRASE&quot;)</td>
<td>&quot;Example:This is some text of a document&quot; (The wildcard term will not match the colon. The tokenizer will replace the colon with a white space).</td>
</tr>
<tr>
<td>string(&quot;Example*This&quot;, mode=&quot;PHRASE&quot;)</td>
<td></td>
</tr>
</tbody>
</table>
**Minimum and Maximum Wildcard Expansion**

Use the string operator's minexpansion and maxexpansion parameters to specify the range of the wildcard expansion.

Syntax:

```
string(S, minexpansion=<n1>, maxexpansion=<n2>)
```

The string 'S' must be a wildcard with a single star: "prefix*suffix". The operator will match all terms starting with "prefix" and ending with "suffix", and where the substring between prefix and suffix has a length of minimum 'n1' and maximum 'n2' characters.

For example, queries containing subexpressions on the form:

```
any("prefix","prefix?","prefix??","prefix???","prefix????")
```

can be written as

```
string("prefix*",maxexpansion=4)
```

The default values are 0 for minexpansion and infinite for maxexpansion.

ℹ️ **Note:** The minexpansion and maxexpansion parameters are optional.

---

**Numeric term expressions**

The following table lists the numeric data types that can be used in FQL.

<table>
<thead>
<tr>
<th>This query component</th>
<th>denotes...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>32 bit signed integer. For non-scope fields, the target (indicated with a fieldname: prefix) must be a field that is defined as int32 (signed int) or uint32 (unsigned int) in the index-profile. For scope fields, the target scope (indicated with the scope_spc: prefix) will only match a numeric scope of type int32. This is due to no auto-typing support for scope fields.</td>
</tr>
<tr>
<td>Floating point value</td>
<td>Floating point value where 23 bits are used for the mantissa and 8 bits used for the exponent. The target field or scope (indicated with the scope_spc: prefix) will only match a scope of type float (using 2-base for the exponent) or double (using 10-base for the exponent). Values are indicated using <code>.</code> as the separator, e.g. 256.23 -256.23</td>
</tr>
<tr>
<td>Date/time</td>
<td>The date/time support in FAST Enterprise Search Platform enables the same numeric operations on date/time values as on other numeric values.</td>
</tr>
</tbody>
</table>

ℹ️ **Tip:** Refer to *Numeric query operators* for further details on query syntax for numeric values.
Hard Numeric Range Cutoff

Numeric range cutoff reduces the query performance impact of wide numeric range queries. Using very wide numeric range queries within scopes (returning too many results) may reduce the query performance substantially. It is possible to avoid such a reduction in query performance by configuring a cutoff limit for numeric range processing. The `hardnumericrangecutoff` is a value for the maximum number of terms a numeric range expression in scope fields can expand to before a query error is returned. The default value is -1. The value -1 represents an unlimited number of terms.

The `hardnumericrangecutoff` parameter can be set in the $FASTSEARCH/etc/config_data/RTSearch/*/fsearch.addon configuration file.

**Note:** Unlike wildcard cutoffs, there is no soft cutoff for numeric ranges.

Date & Time query expressions

FAST Enterprise Search Platform provides a dedicated data type for date & time, named datetime. Query expressions against a datetime field/scope must be formatted according to the following rules:

- The date/time value in the query must be formatted according to the ISO 8601 formats described below
- Date/time values are handled in a similar way as numeric values in queries. This means that you can apply the query operators as defined in section NEAR/ONEAR with NOT
- The resolution specified for a scope field in the index-profile will apply for all its sub-scopes of type datetime.

**Tip:** It is recommended to use explicit type conversion for datetime values. When using default typing you will not get any syntax error if the format is wrong, it will instead be treated as a text string.

Supported date/time formats in queries:

- YYYY-MM-DD
- YYYY-MM-DDThh:mm:ss
- YYYY-MM-DDThh:mm:ssZ

**Note:** Only four-digit years are supported in content and query input.

The actual resolution for indexed content may be configured in the index-profile. The resolution can be configured for a number of ranges. If the resolution is set to minutes (default), the ss value is discarded when parsing the date/time query format.

The following table should be read as follows. “The earliest date you can represent using **seconds**, is 1970-01-01T00:00:00Z. Using this resolution, dates can be represented until the year of 2106.”

<table>
<thead>
<tr>
<th>Date/Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY</td>
<td>Four-digit year. Earliest representation is year 0000.</td>
</tr>
<tr>
<td>MM</td>
<td>Two-digit month. (01=January, etc.)</td>
</tr>
<tr>
<td>DD</td>
<td>Two-digit day of month (01 through 31).</td>
</tr>
<tr>
<td>T</td>
<td>The letter <code>T</code></td>
</tr>
<tr>
<td>hh</td>
<td>Two digits of hour (00 through 23) (am/pm NOT allowed).</td>
</tr>
<tr>
<td>mm</td>
<td>Two digits of minute (00 through 59).</td>
</tr>
<tr>
<td>Description</td>
<td>Date/Time</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Two digits of second (00 through 59).</td>
<td>ss</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Resolution</th>
<th>Until year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970-01-01T00:00:00Z</td>
<td>seconds</td>
<td>2 106 AD</td>
</tr>
<tr>
<td>5000-01-01T00:00:00Z BC</td>
<td>minutes (default)</td>
<td>3 000 AD</td>
</tr>
<tr>
<td>1900-01-01T00:00:00Z AD</td>
<td>minutes from 1900</td>
<td>6 000 AD</td>
</tr>
<tr>
<td>400000-01-01T00:00:00Z BC</td>
<td>hours</td>
<td>89 191 AD</td>
</tr>
<tr>
<td>11000000-01-01 BC</td>
<td>days</td>
<td>736 891 AD</td>
</tr>
<tr>
<td>2147483648-01-01 BC</td>
<td>years</td>
<td>2 147 483 647 AD</td>
</tr>
</tbody>
</table>

All date/time values must be specified according to UTC (Coordinated Universal Time), also known as GMT (Greenwich Mean Time) time zone. The UTC time zone identifier - a trailing 'Z' character - is optional.

<table>
<thead>
<tr>
<th>This query component</th>
<th>denotes...</th>
</tr>
</thead>
<tbody>
<tr>
<td>time:2003-08-27</td>
<td>Return all documents from this day.</td>
</tr>
<tr>
<td>time:datetime(2003-08-27)</td>
<td>Return all documents from this day. This format implies an explicit type conversion to datetime type. The main difference will be the handling of syntax errors. When using this format an illegal datetime format will result in a syntax error.</td>
</tr>
<tr>
<td>time:range(min, 2003-08-27T20:16:00)</td>
<td>Return documents older than 2003-08-27 20:16:00. The first example refers to a non-scope datetime field (named 'time'). The second example refers to the datetime scope 'date' within the xml scope field.</td>
</tr>
<tr>
<td>xml:item:date:range(min, 2003-08-27T20:16:00)</td>
<td></td>
</tr>
<tr>
<td>and(soccer, time:range(2003-08-27, max))</td>
<td>Return all documents matching 'soccer' (within the default composite field) more recent than the date 2003-08-27</td>
</tr>
<tr>
<td>and(&quot;egypt&quot;, xml:time:range(1000-01-01, 1100-12-31))</td>
<td>Return all documents including information about Egypt between the year of 1000 and 1100. xml is the root scope in this example</td>
</tr>
</tbody>
</table>

Tip: Refer to Numeric Query Operators for more details about the range operator.

There are also other useful ways of utilizing date/time fields (only applicable for non-scope fields):

- Navigation on a datetime field
- Freshness Boost feature, which enables freshness to be a component of the result ranking
- Sorting by date

Natural Language Queries

In FAST ESP, it is possible to use natural language to express queries.
natural language query -> matches a pattern -> is rewritten into a FQL expression -> scope search is performed

- when (is|was|did|...) X? xml:sentence:(X' and (scope(date) or scope(time)))
- what does X stand for? xml:acronym:(@base:X and scope(@definition))
• who works (at|for) X? ? xml:sentence:(company:X and scope(person) and scope(jobtitle))
• why (is|are|do|does|have|has|...) X? ? xml:sentence:(X' and ("because" or "due to" or ...))

These patterns are configurable.

Enable Natural Language Queries

Natural Language Query processing currently consists of three stages. These stages are currently found in the customlinguistics pipeline.

$FASTSEARCH/etc/config_data/QRServer/webcluster/etc/qrserver/qtf-config.xml

The default configuration supports the following example.

1. Make sure the query is sent through the correct query pipeline. Several approaches exist.
   a) Specify pipeline for Search View in Search Business Center.
   b) Specify pipeline in
      $FASTSEARCH/etc/config_data/QRServer/webcluster/etc/qrserver/sources.xml
   c) Specify pipeline query-time by setting the QTPIPELINE parameter.

2. Restart the Query & Result server if necessary.

3. Make sure there is at least one collection available and that the specified pipeline includes stages for entity extraction and scope search, E.g. the Semantic pipeline.

4. Paste the following example into a document and feed it to FAST ESP:
   <xml>
   <body>
   <sentence>On putin@russia.com you can contact Vladimir Putin, the president of Russia and owner of black belt in karate.</sentence>
   <sentence>Other karate enthusiast was Franklin Delano Roosevelt.</sentence>
   </body>
   </xml>

5. Run the following query:
   http://<esp installation>:15100/cgi-bin/xsearch?query=nlquery:(what+is+the+email+of+vladimir+putin) &qtpipeline=customlinguistics

   Note: Uppercase and lowercase characters are supported.

   The following lines should be visible if successful:

   <QUERYTRANSFORM NAME="FastQT_NL2FQL" ACTION="Modified the query"
QUERY="xml:sentence:and(person:"vladimir putin", scope(email))" CUSTOM=""
MESSAGE="Rewrote query" MESSAGEID="1"/>

Query Operators

This chapter describes the operators that can be applied to the query terms.

FQL provides the following main expression formats:

• Infix operator syntax:

Infix operators supported are AND, OR, ANDNOT, RANK and ANY. Each sub-expression may be a single term (with or without a scope specification), a prefix operator sub-expression or any other supported sub expression enclosed in parentheses.
• Prefix operator syntax:
• Scoped sub-expression:

In this case any query sub-expression may be included within the parentheses, but all terms within the
sub-expression must match within the scope specification. Such expressions may be nested. In that case, a
scope specification inside the sub-expression must address a sub-scope within the main scope specification
indicated.

**Using Infix operators**

Infix operator syntax is supported for AND, OR, RANK, ANDNOT and ANY operators. It is possible to mix
prefix and infix operators within the same query expression, e.g.

```
term1 AND term2 AND price:float(3.14) AND near(term3, term4, 4)
```

Operator names are case insensitive, i.e. `AND` and `and` are equivalent.

Mixing different infix boolean operators yields the following precedence sequence. Read from left, an operator
N has precedence over N+1.

IN, NOT, NEAR, ONEAR, RANK, ANDNOT, AND, ANY, OR

Example in infix syntax:

```
Coffee and Mocha near Cup equals Coffee and (Mocha near Cup)
Coffee and Mocha or Cup equals (Coffee and Mocha) or Cup
```

**Using prefix operators**

```
[scope-spec]:operator(operand [,operand] * [, parameter="value"]*)
```

• Operator names, parameter names and parameter text values are case insensitive.
• Within the prefix operator body, white spaces are ignored (i.e. allowed) except within double quotes.
• A text string operand (enclosed in double quotes) is a set of text terms that is handled according to the
operator and given parameter(s). A single word text string operand does not need double quotes.

**Boolean query operators**

The **FAST Query Language** supports the following basic boolean query operators:

**AND operator**

Only documents matching all AND operands will be returned.

Prefix syntax:

```
and(operand, operand [, operand] *)
```

Infix syntax:

```
a AND b
```

**OR operator**

Only documents matching at least one of the OR operands will be returned. Matching documents will get
higher dynamic rank (relevance score in the result set) if more of the OR operands match.
Prefix syntax:

or(operand, operand [,operand]*)

Infix syntax:

a OR b

**ANY operator**

Similar to `OR' except that the dynamic rank (relevance score in the result set) is not affected by the number of OR operands that match or the implicit proximity between the terms.

The dynamic ranking component for this part of the query will be based on the best matching term within the `ANY' expression. Note that the difference from `OR' is only related to the ranking within the result set. The same total set of documents will match the query.

Prefix syntax:

any(operand, operand [,operand]*)

Infix syntax:

a ANY b

**RANK operator**

This operator takes two or more operands. The first operand is the main query term/expression that is matched and contributes to recall/precision as any other term/expression. This may be any valid FQL sub-expression.

The following operands must be single text terms or phrases which contribute to precision (rank) but not recall. These operands do not affect the total number of documents that match, but affect the dynamic ranking in the way that documents matching terms within the following operands will get higher rank within the result set. Hence, this operator may be used in order to boost dynamic rank according to certain text terms found, but where these terms do not need to be found to give a match.

**Tip:** For a successful result using scope fields, the RANK operator should only be used on root level.

Prefix syntax:

rank(operand, rank-operand [, rank-operand]*)

Infix syntax:

a RANK b

**Tip:** RANK operators in sub-scopes ignore the ranking operands, i.e. reduce to the first operand only. For example, xml:object:RANK(foo, bar) reduces to xml:object:foo. A message that this approximation is performed is added to the query feedback.

**XRANK operator**

This operator is identical to the RANK operator in the sense that both operators provide rank.

RANK and XRANK differ in that XRANK:

- is much more efficient than RANK when terms in B generate many hits.
- the boost level can be set through the `boost` parameter, at query-time.
- adds a constant rank value, while RANK adds rank value from the operands.
- will add rank to unranked documents.
Prefix syntax:

```plaintext
xrank(s, [a, b, c..., boost=n, boostall=yes])
```

**Note:** The default value for `boost` is 100, and for `boostall` it’s ‘yes’.

Infix syntax:

```plaintext
s XRANK b
```

where `s`, `a` and `b` can consist of any valid FQL expression. The boost value is specified with the parameter `boost=n` where `n` is some signed integer value. This operator also supports a "boostall" mode. Documents that are not in ranked form (have not been assigned a rank value) cannot receive boost from xrank. However, if the boostall parameter is ‘yes’ even these documents can receive boost. If boostall mode parameter is ‘no’ then only those documents in `s` that are in ranked form (have been assigned rank) will be boosted.

The XRANK operator only supports boost weight when using prefix FQL syntax. The only infix syntax supported is `s XRANK b` where the boost weight is fixed to 100.

- Negative boost is supported. This will reduce the dynamic rank component of the query result correspondingly. Note that the dynamic rank component will not be lower than 0 (negative resulting rank not supported). Also note that you may still get a positive resulting rank due to the static rank component from SBC Boost&Block or application-specific quality-rank added after xrank is evaluated. When using Boost&Block the resulting rank will not get below 10000.
- Overflow is not checked and boosting with a large value might lead to wraparound.
- xrank supports unary boosting. xrank(a, boost=n) will boost every document in `a` with the value `n`.

---

An e-commerce site allows its customers to search for various products. The site owners are very customer centric and want to boost products in their result set that are cheap. However, at the same time they want to get rid of the older models first and therefore boost the oldest models the most.

- `s` (set of documents to search in): `xml:category:"cameras"
- `a` (cheap cameras): `xml:price:range(min,100)
- `b` (old models): `xml:date:range(min,2006-08-27:16:00)

The xrank expression that allows you to achieve this can be written differently, depending on whether you want to assign different boost values to the different sub-expressions.

This expression will boost all the documents in `s` that are cheaper than 100 and older than 2006-08-27:16:00 with a value of 100.

```plaintext
xrank(s, a, b)
```

This expression will boost all the documents in `s` that are cheaper than 100 with a value of 1000 and products older than 2006-08-27:16:00 with a value of 200. Our result set will therefore list the cheap products before the older ones.

```plaintext
xrank(xrank(s, a, boost=1000), c, boost=200)
```

---

**ANDNOT operator**

Only documents that match the first operand (or terms/conditions preceding the `ANDNOT` operator when using infix format) and NOT match the second operand will be returned. The prefix format accepts more than 2 operands, where all operands but the first one are part of the NOT condition.
Prefix syntax:

\texttt{andnot(operand, operand [,operand]*)}

Infix syntax:

\texttt{a ANDNOT b}

**NOT operator**

Only documents that do NOT match the operand will be returned. I.e. the operand specifies what should not be included in the result. The operand may be any valid FQL expression.

Syntax (prefix only):

\texttt{not(operand)}

- When addressing sub-scope fields, the NOT operator is only supported when ESP can automatically rephrase it as an ANDNOT query.

\texttt{xml:title:(not("talking")and "about")}

will work because it can be rephrased to

\texttt{xml:title:("about" andnot "talking")}

**COUNT operator**

This operator allows you to specify the number of query term occurrences a document must include, for it to be returned as a result. The operator supports single query terms, phrases and wildcards.

**Tip:** The operator can only be specified on root scopes.

Syntax:

\texttt{root:count(arg [,from=<numeric value>, to=<numeric value>])}

or

\texttt{root:count(arg) <, <=, =, >=, or > <numeric value>}

Only retrieve documents where the query term "FAST" appears more than 5 times:

\texttt{xml:count("FAST") > 5}

Only retrieve documents where the query term "FAST" appears between 5 and 10 times. This approach is encouraged when specifying range values. From is inclusive, while to is exclusive:

\texttt{xml:count("FAST", from=5, to=11)}

Wildcards can also be included:

\texttt{count("fa* se* and transfer") > 2}
**Proximity query operators**

The explicit proximity query operators (NEAR/ONEAR) can be used to restrict the document result set to documents that have N terms within a certain distance from each other.

Syntax:
- `near(arg, arg [, arg]* [, N=<numeric value>])`
- `onear(arg, arg [, arg]* [, N=<numeric value>])`

where `N` indicates the maximum number of words that is allowed to appear between the terms (explicit proximity). If NEAR/ONEAR includes more than two arguments, the maximum number of words allowed between the terms (N) is counted within the entire expression.

Default term distance value (N) is 4.

For NEAR, the order of the query terms doesn't matter for the matching, only the distance.

ONEAR is the ordered variant, and requires ordered match of the terms.

Any number of terms may be combined with NEAR and ONEAR operators.

If you apply NEAR/ONEAR to query terms against different fields, this will currently be accepted by the parser and approximated to AND. Thus

```
title:a NEAR body:b
```

is in this version of FAST Enterprise Search Platform evaluated as

```
title:a AND body:b
```

NEAR/ONEAR operands may be single terms, phrases or boolean OR/ANY sub-expressions. Wildcards are accepted.

```
near(plane, bike, boat, car, n=4)
```

returns documents that have the words plane, bike, boat, and car with no more than 4 words separating all of these four words. This means that the document text

`'I have a car, a bike, a boat, and a plane'`

will match - the four words separating are "a", "a", "and" and "a", while the following document text

`'I have a car, and a bike, a boat, and a plane'`

will not match.

```
onear(car, bike, boat, plane) onear(car, bike, boat, plane, n=4) onear(car, "a bike", "a boat", "a plane", n=1)
```

will all match the document text:

'I have a car, a bike, a boat, and a plane'

since the words appears in correct order.
Consider the following snippet of an XML document, indexed in the scope field named `xml`:

```xml
<paragraph>
  <sentence>Tigger came out of the forest</sentence>
  <sentence>Winnie the Pooh was waiting for him</sentence>
</paragraph>
```

The following query will match, since the query addresses the paragraph level:

```
xm:paragraph:near(tigger, winnie, N=5)
```

The following query will not match, since the query addresses the sentence level and the two terms are situated in different sentences.

```
xm:paragraph:sentence:near(tigger, winnie, N=5)
```

It is also possible to include wildcards.

```
xm:paragraph:near("tigg*", "winnie", N=5)
```

**NEAR/ONEAR term distance considerations**

`N` indicates the maximum number of words that is allowed to appear between the query terms within the matching segment of the document. If NEAR/ONEAR includes more than two arguments, the maximum number of words allowed between the query terms (N) is counted within the segment of the document matching all the NEAR/ONEAR terms.

NEAR/ONEAR operates on tokenized text. This means that special characters like ,.;: etc. will be treated as white spaces. Hence, the term distance as discussed below relates to tokens within the indexed text.

If you use ONEAR or NEAR with equal arguments, it will work as follows:

```
near(a, a, n=x)
```

will always return true if at least one instance of `a` appears within the context. This also means that `near()` cannot be used as a count operator. See the documentation on the `count()` operator for more information on counting term occurrences.

```
onear(a, a, n=x)
```

will only return true if different instances of `a` appears within the proximity distance `x`.

The actual NEAR/ONEAR feature evaluates the total number of token matches (T) between the NEAR/ONEAR operator expression and the matching segment of the document. This value `T` cannot always be determined from the query itself, e.g. in case an operand is a boolean sub-expression.

This approximation has the following side-effects:

- **NEAR applied to phrases will also match overlapping phrases in the text.**
- **If a token in the matching segment matches more than one operand to the NEAR/ONEAR expression, the query may match even if the number of non-matching tokens within the matching segment exceeds the value of `N` in the NEAR/ONEAR operator expression. An overlap may e.g. be overlapping phrases. If the number of token overlap matches is `O`, the query will match if not more than `N+O` non-matching tokens appear within the matching segment of the document.**
- **When using lemmatization and spelling variations on scope fields, and two inflected forms or synonyms exist within the NEAR operator expression, the expression may match although there are more than the allowed amount of terms in between.**
**NEAR/ONEAR across scope boundaries**

By default a scope boundary implies no extra word position between consecutive scopes (between the last word of the first scope and the first word of the following scope) in the index. Example 3 provides an example of this.

The `word distance' between scopes is also denoted `semantic distance' as it indicates the `semantic relation' between the scopes in a document for free-text content. This also relates to implicit proximity (dynamic ranking based on word distance).

**Tip:** If the query address a scope that specifies an attribute, an extra word position will be induced. Thus, boundary matching will not function as expected.

**Tip:** Some XML content may use XML tags for text formatting, e.g. "&lt;paragraph&gt;hello &lt;b&gt;world&lt;/b&gt;&lt;/paragraph&gt;". You may avoid adding the additional word position on the scope boundary if you define the &lt;paragraph&gt; element as a typed element.

**NEAR/ONEAR with NOT**

The NOT operator cannot be used inside the NEAR/ONEAR operator, like this:

```
NEAR(audi, not(bmw), n=2)
```

**Numeric query operators**

Numeric query operators include explicit type conversion and range operators.

**Explicit and implicit type conversion**

The following explicit type conversions provide explicit typing of numeric values:

- `float()`
- `double()`
- `int()`
- `datetime()`

For non-scope fields, the explicit type conversion is optional and normally not needed. The type of the query term is detected according to the type of the target numeric field.

For scope fields it is necessary to know the exact type of the numeric query term, as the type of the target numeric scope is not known at query processing time. If the explicit type conversion operators are not used, the following default types are assumed:

- `int32`:
  
  | 23 | -23 |

- `float`:
  
  | 23.0 | -23.0 | 23.45 |

- One of the accepted date/time formats (e.g. 2004-12-14T14:13:11): Treated as a datetime type. When using default typing you will not get any syntax error if the format is wrong, it will instead be treated as a text string. Hence, it is recommended to use explicit type conversion for datetime values.

- All other terms: Treated as a string() with default mode="phrase", i.e. equivalent to:

  ```
  string("hello world", mode="phrase")
  ```
Note the difference between the two queries:

```
rootscope:change:256
rootscope:change:"256"
```

The first query will search for the integer value 256 in a subscope named change. This must be a numeric sub-scope, otherwise there will be no match. The second query will search for the text term '256' in a subscope named change. This must be a sub-scope of type string, otherwise there will be no match.

**Numeric range operator**

The numeric range operator may be used for numeric and date/time fields. The operator enables range matching expressions.

Syntax:

```
range(start, end [,from="GE"|"GT"] [,to="LE"|"LT"])```

- `start`: Start value for the range. If no start value, use the reserved operand 'min'.
- `end`: End value for the range. If no end value, use the reserved operand 'max'.
- `from`: Optional parameter that indicates open/close start interval.
  - "GE": Greater or equal to the start value (>= start of interval)
  - "GT": Greater than the start value (> start of interval)
  - Default: "GE"
- `to`: Optional parameter that indicates open/close end interval.
  - "LE": Less or equal to the end value (<= end of interval)
  - "LT": Less than the end value (< end of interval)
  - Default: "LT"

**Examples**

```
content:adds:and(title:cars, price:range(min, 20000))
content:adds:and(title:cars, price:range(min, 20000, from="GE", to="LT"))
```

These queries will search for 'cars' in the scope 'title' (or any text sub-scopes to 'title') which must be in the scope 'adds' which must be in the scopefield 'content'.

The second required condition stipulates that the price must be less than 20000. In other words, `range(min, 20000)` indicates not that the minimum price is 20000, but that a start price value is not specified (min) and the end or top value in the price range is 20000.

```
and(time:range(2002-01-01, 2002-12-31, to="LE"), kasparov)
and(time:range(datetime(2002-01-01), datetime(2002-12-31), to="LE"), kasparov)
and(time:range(2002-01-01, 2002-12-31, from="GE", to="LE"), kasparov)
```

returning documents containing the term 'kasparov' from the year 2002.

```
and(type:laptop, disk:range(40, max))
```

returning laptops that have 40 GB (assuming field value is # of GB) or more from a product database.
**String operator**

The string operator may be used to implement a more direct mapping from a search bar to a query expression. They also include an alternative way of specifying explicit proximity constraints when used in addition to other string parameters.

\[
\text{string("<text string>") [. par="val"]}
\]

indicates a text string (one or more terms) and zero or more parameters as described below.

The FQL **string()** operator uses the syntax:

\[
\]

where:

mode="PHRASE"|"AND"|"OR"|"ANY"|"NEAR"|"ONEAR"|"SIMPLEANY"|"SIMPLEALL"

The fuzzy parameter enables or disables pattern expansion for query terms inside the <text string>. This setting overrides the FUZZY query parameter setting, allowing fuzzy expansion to be enabled or disabled on specific portions of the query. Query terms which contain wildcard characters are unaffected by this setting.

The wildcard parameter controls wildcard expansion of terms inside the <text string>. This setting overrides the WILDCARD query parameter setting, allowing extended wildcards to be enabled or disabled on specific portions of the query.

The string() operator may also be used as a type conversion, i.e.

\[
\text{string()}
\]

will treat the numeric value `24.5' as a text string.

The following string parameters are supported:

- **String mode**: mode=
- NEAR/ONEAR term distance (N) for string() operator: n=
- Relevance weight for dynamic ranking (string() operator): weight=
- Control linguistics processing: linguistics=
- Pattern searching

**String mode**

This parameter indicates how the terms in the <text string> shall be treated. Valid modes are:

\[
\text{mode="PHRASE"|"AND"|"OR"|"ANY"|"NEAR"|"ONEAR"|"SIMPLEANY"|"SIMPLEALL"}
\]

- **"PHRASE"** equals tokenized phrase match. Similar to specifying

\[
\text{phrase(term, term [,term]*)}
\]

- **"AND"** equals and(term, term [,term]*)
- **"OR"** equals or(term, term [,term]*)
- **"ANY"** equals any(term, term [,term]*)

Refer to section *Boolean query operators* for details on these boolean modes.

- **"NEAR"** equals near(term, term [,term*], N)
• "ONEAR" equals onear(term, term [.term]*, N)
• "SIMPLEANY"
• "SIMPLEALL"

The modes SIMPLEANY and SIMPLEALL enable parsing of the Simple Query Language within an FQL expression. Refer to section Scope Navigation for details.

Default : mode="PHRASE".

**NEAR/ONEAR term distance for string operator**

This parameter indicates the maximum term distance for mode="NEAR" | "ONEAR":

\[ N = \text{max word distance} \]

The string() expression:

\[
\text{string("hello world", mode="NEAR", n=5)}
\]

equals the near() expression:

\[
\text{near(hello, world, n=5)}
\]

Default : N=4

**Relevance weight for dynamic ranking using string operator**

This parameter is a positive numeric value indicating term weight for dynamic ranking:

\[ \text{weight=} \text{<relevance weight>} \]

Weight=100 is the normal weight. A lower value indicates that this term shall contribute less to the ranking. A higher value indicates that this term shall contribute more to the ranking. Weight=0 implies that this term has no rank weight.

Weight applies to all the terms in the string() expression.

Default: weight=100

**Tip:** The weight parameter will only have effect for queries towards composite fields and root-scopes. Special considerations apply when using term weight towards scope fields.

The feature has main effect for OR queries. However, it may also have some effect on AND queries. The FAST Enterprise Search Platform dynamic rank algorithm may imply that different terms gives different rank contribution depending on where in the document the term match occurs. This is the case when using the 'Weight List' concept within the index-profile. Furthermore, the difference in rank contribution may also be based on term frequency and inverse document frequency. An example may illustrate this:

Given the query 'and(string("a"), string("b", weight=200))', and we have an index-profile where the 'title' field has more weight than the 'body' field (as defined by a Weight List). In the Index document 1 includes term 'a' in the title and term 'b' in the body, while document 2 includes term 'a' in the body and term 'b' in the title. In this example document 2 will get the highest total rank, as the documents with higher dynamic rank contribution will get even more boosted.

**Tip:** The relative term boost (positive or negative) is applied to the dynamic rank component of the total rank. However, proximity boost (distance between words) rank calculations are not affected by the term weighting. Note therefore that the relative weighting does not always imply that the total rank for the document is modified according to the percentage given.
Example:
```
java or string("python perl", mode="OR", weight=50)
```

will search for the terms `java`, `python` or `perl`, where `java` will have twice as much rank contribution as the two other terms.

**Control linguistics processing**

This parameter will enable all linguistics features for the string (lemmatization, synonyms, spell check) if enabled for the query.
```
linguistics="ON" | "OFF"
```

Default: linguistics="ON"

To enable linguistics features, you need to:

- Enable Document Processing side linguistics features in the index-profile (Lemmatization)
- Activate language specific spell check dictionaries during installation
- Activate the linguistics features in the query using the corresponding global query parameters.

There are cases in which it is not desirable for linguistics to be on:

- When addressing a scope field that specifies a value you do not wish to be processed by linguistics, it is important to use the `filter()` operator. The following query will result in the value "en" being processed by for example spell-checking or lemmatization. Thus, the language value submitted with the query might be "es" (Spanish) when "en" (English) was desired.
```
xml:language:"en"
```

- Using the `filter()` operator automatically sets linguistics=off, so the value "en" will not be changed prior to query submission.
```
filter(xml:language:"en")
```

**Tokenized phrase matching**

You may search for an exact string of tokens using the `string()` operator with mode="phrase" or the `phrase()` operator.

All phrase operations, also when using `string("hello world", mode="phrase")`, implies tokenized phrase match. This means that all white spaces are treated equally, and special characters like ;:__,__/- are treated as white spaces. This is related to the tokenization process.

The `phrase()` operands may be single terms or boolean OR/ANY sub-expressions. Wildcards are accepted.

Syntax:
```
phrase(tokenexpr [operator] [, tokenexpr [operator]]*)
```

where tokenexpr can be a single token or an `or(token, token [, token]*)` expression.

The default mode for the `string()` operator is "PHRASE". Hence, the following expressions are equivalent:
```
phrase(this, is, a, "phrase")
string("this is a phrase", mode="PHRASE")
string("this is a phrase")
```
**Tip:** In the first expression the word "phrase" must be double quoted due to the fact that `phrase` is a reserved word in FQL.

A set of text terms enclosed in double quotes are by default treated as a string() operator. Hence, the following expressions are equivalent (and indicates a phrase):

```plaintext
code
string("this is a phrase")
"this is a phrase"
```

There is also support for using the OR operator within the phrase() operator. Thus, the following query is valid:

```plaintext
code
phrase("new" or "old", "york")
```

### Handling strings with special characters

The appearance of special characters like ,;:-_/ are treated as white spaces within a double quoted string expression. This is related to the tokenization process. However, these characters will also imply an implicit phrasing of the tokens separated by these characters.

The phrase query expression:

```plaintext
code
xml:book:animals/birds
```

will be evaluated equally to the query expression:

```plaintext
code
xml:book:string("animals birds", mode="phrase")
```

All these expressions implies tokenized phrase match.

On the other hand, if you use a different mode than "phrase", the query expressions

```plaintext
code
xml:book:string("animals/birds", mode="and")
xml:book:string("animals/birds", mode="or")
```

will also be treated as a tokenized phrase match expression, equal to:

```plaintext
code
xml:book:string("animals birds", mode="phrase")
```

Correspondingly, the query expression:

```plaintext
code
xml:book:string("animals/birds animals/insects", mode="or")
```

will be evaluated equally to the query expression:

```plaintext
code
xml:book:or(string("animals birds", mode="phrase"),
string("animals insects", mode="phrase"))
```

And the query expression:

```plaintext
code
xml:list:string("help@fastsearch.com")
```

will be evaluated equally to the following phrase, given default tokenization behaviour:

```plaintext
code
xml:list:string("help fastsearch com")
```
Boundary Match Operators (anchoring)

The starts-with(), ends-with() and equals() operators may be used to indicate exact token match or boundary match conditions, i.e. that the word/phrase must appear in the start and/or end of a scope, field, or string within a field.

**Tip:** Boundary matching is not possible on numeric fields. Numeric fields will always be subject to exact or value range matching.

Some applications may require that you are able to perform an exact match of a field. This may for instance be a product name field where the full name of one product is a substring of another product name.

The following boundary match operators are supported:

- **starts-with(<term or phrase>)**
  
  The term or phrase must be in the beginning of the field/string/scope. <term or phrase> may be a single term or a phrase. Examples:
  
  ```
  starts-with(shakespeare
  starts-with("william shakespeare")
  starts-with(string("william shakespeare", mode="PHRASE"))
  starts-with(phrase(william, shakespeare))
  ```

- **ends-with(<term or phrase>)**
  
  The term or phrase must be in the end of the field/string/scope.

- **equals(<term or phrase>)**
  
  The term or phrase must provide an exact token match with the field/string/scope.

Assume three documents with the following values for the author field:

- Document 1: Mr Adam Jones
- Document 2: Adam Jones sr
- Document 3: Adam Jones

The table shows how to match these documents using boundary match expressions:

<table>
<thead>
<tr>
<th>This query item</th>
<th>denotes...</th>
</tr>
</thead>
<tbody>
<tr>
<td>author:equals(&quot;adam jones&quot;)</td>
<td>Exact field or string token match. This query will only match document 3.</td>
</tr>
<tr>
<td>author:starts-with(&quot;adam jones&quot;)</td>
<td>Matching start of field or string. This query will match documents 2 and 3.</td>
</tr>
<tr>
<td>author:ends-with(&quot;adam jones&quot;)</td>
<td>Matching end of field or string. This query will match documents 1 and 3.</td>
</tr>
</tbody>
</table>

**Boundary match on Scope Fields**

Boundary match on scope fields enables you to match the full tokenized content of an individual sub-scope of type `string`.

Boundary match can be applied to unique words or phrases, also including wildcard terms. This is different from non-scope fields where wildcard terms is not possible to use in combination with boundary match.
For scope fields, there is no need for any specific boundary match configuration in the index-profile.

The following two queries will match the document:

xml:book:sentence:starts-with("tigger was running")
xml:book:starts-with("tigger was running")

The second query will match a document if any node supporting boundary match within <book> starts with the indicated phrase. This means you will not be able to search for a document where the 'book' starts with this phrase, i.e. in the first chapter in this case.

There is support for separating scopes at document processing time, based on scope element names. Meaning you can configure elements to be further away from each other, based on an elements name.

The example presents a chapter structure including sections. Although these texts are in different chapters, a phrase or explicit proximity query may match across the scope boundaries. Consider the following query:

near(ateist, member, n=10)

In the example above it may or may not be desired to have such matches across chapters. An application may therefore decide to configure the actual scope matching so that there is a "semantic barrier" (meaning no phrase/proximity match across scopes) between the chapters, but not between the sections. If you configure the document processing like this, the above query would not match, because the two terms appear in two different chapters. However, the following query would match, because the two terms appear within the same chapter:

near(member, page, n=10)
Boundary match on non-scope fields

Boundary match on non-scope fields may be applied to the entire text field, or to individual strings within a field containing a list of string values, e.g. a list of names. In this case it may be desired to be able to match the exact content of each string, and to avoid query match across string boundaries.

In order to apply boundary match queries, you will need to configure the relevant field in the index-profile:

- Apply the `boundary-match` attribute to the field specification. This enables queries that match exact field value or start/end of field.
- If you want to support multiple strings in each field, apply the `separator` attribute to the field specification. This will enable boundary match between individual string values of the field. The separator attribute indicates which character that separates the strings in the original content. This enables querying unique string values (e.g. names) within the field.

By enabling the Boundary Match feature for the field, you enable two features:

- Explicit boundary match queries, as described further below.
- Avoid phrases to match across string boundary. For fields that contains multiple strings, this feature will ensure that a string does not match words before/after a boundary indication.

Example

Given the following value for the field named `author`. The field is configured for Boundary Match with the separator attribute set to `'#':`

```
Peter Adam#Adam Jones#Peter Gordon
```

You will match the document with the query:

```
query=author:equals("adam jones")
```

but not with the query:

```
query=author:equals("jones peter")
```

because the `'#' separator implies a phrase break between Adam Jones and Peter Gordon.

Filter (metadata) operator

The filter operator is typically used when querying metadata or (not natural language) parametric content.

Using the filter operator automatically implies the following for the specified query:

- Linguistics will be set to `linguistics="OFF"`
- Ranking will be disabled
- No query highlighting in the Dynamic Document Summary (teaser)

```
filter(<any valid FQL sub-expression>)
```

Tip: If you use the `string()` operator inside `filter()`, please note that linguistics is by default switched off (the default value for the operand `linguistics` is "OFF"). It is still possible to enable linguistics processing within each `string()` expression inside `filter()` using the operand `linguistics="on"`.

Tip:

If you need to filter a large set of values (e.g., within a subscription based search solution with large number of subscription entities) you should consider using numeric values rather than string/text values.
For example, an alternative way of expressing:

```
and(string("hello world"), filter(fieldname:or(1, 20, 453, ..., 3473)))
```

is:

```
and(string("hello world"), filter(fieldname:int("1 20 453 ... 3473", mode="or")))
```

The latter approach uses the `int()` operator with a double quoted string with the set of numeric values. This will provide a substantially better query performance when filtering with a large set of numeric values. Refer to *Numeric Query Operators* for details.

**Example**

filter() is typically used to add metadata constraints to queries.

You use the specific language codes ("nn" or "nb") if you want to limit the query result to all Norwegian documents. This can be done as follows in FQL:

```
AND(FILTER(languages:OR("nn","nb")), <your query expression>)
```

---

**Ranking Considerations**

Dynamic ranking of query results works slightly different depending on the types of fields used in the query.

**Fields and Composite Fields**

Dynamic ranking related to the actual matching of query terms with fields is supported for composite fields only. Query terms against individual fields will only contribute to recall (that is which documents to be returned for a query).

**Example**

You want to search for documents containing the terms `Bob' and `Jones', and boost documents with `engineer' in the title field. Given that the default composite field also includes the title field, and that the title field has been assigned higher weight than the body, the following query expression will achieve an implicit boosting of documents where `engineer' appears in the title field:

```
rank(string("Bob Jones", mode="and"), string("engineer"))
```

However, it will also boost documents with `engineer' in the body, but the boost effect will then be lower. This may or may not be desired. If you do not want this side-effect, you will need to define a separate composite field only containing the `title' field. In this case we name it `comtitle'. Then you must define a rank profile referring to both the default composite field and the `comtitle' composite field. You will also need to select this rank profile for the query (unless it is the default rank profile).

It is also possible to explicitly set the rank weightings:

```
rank(string("Bob", weight="200"), string("Jones"))
```
This expression will double the rank value of documents containing the term Bob, compared to documents containing the term Jones. It is ranked twice as valuable because the default value is 100.

Bob AND Jones RANK comtitle:engineer
Bob AND Jones RANK comtitle:engineer!200

Ranking of Scope Fields

There are a number of ranking considerations/limitations when using FQL against scope fields:

Context based ranking on scope fields

Context based ranking is not supported when performing scope queries. It is, however, possible to include a scope field in a composite field in combination with other fields. In this way you can apply context based ranking by searching against this composite field or using the RANK() operator to boost matches in the title. A typical use case is that you want to boost matches in some title part of a scope field. This can be achieved in the following way:

1. Use the XMLMapper document processor to extract title information from the XML and assign that to e.g. the non-scope field named title.
2. Define a composite field scopecomp consisting of the scope field and the title field.
3. Define a rank profile scoperank for this composite field, where the context boost for title is higher than for the scope field.
4. Given that the original query is:
   xml:chapter:("winnie the pooh")
   you can boost matches in chapter titles by rewriting the FQL to:
   rank(xml:chapter:("winnie the pooh"), scopecomp:("winnie the pooh"))

Proximity based ranking on scope fields

Scope queries (not using composite fields as indicated in previous section) are not controlled by rank profiles, but will include basic proximity (word distance) based ranking except for the following cases:

- The query includes a NEAR/ONEAR expression where the NEAR/ONEAR parameters are not only single terms.
- The query includes a phrase.

Using the RANK operator on scope fields

The RANK operator will only have effect on scope queries where the scope of the rank() operator is the root-scope. Hence, the following query

xml:rank(string("hello world"), string("programming"))

will have the expected effect (boost documents containing "programming"), but the following queries will have no effect from the rank() operator:

xml:paragraph:rank(string("hello world"), string("programming"))
xml:rank(string("hello world"), paragraph:string("programming"))
The query evaluation for these two queries will be equal to:

```xml
xml:paragraph:("hello world") xml:("hello world")
```

### Filter Operator Effect on Performance

Query terms specified within the filter operator will not have any rank calculated. If you want to optimize your installation for QPS (Queries Per Second), the use of the filter operator can therefore have a big impact.

#### Large numeric filter expressions

If your application uses multiple numeric terms as part of a filter expression for a query, it is recommended to use the following syntax in FQL:

```fql
filter(int("term1 term2....termN"))
```

This is functionally equal to the expression:

```fql
filter(or("term1, term2,....,termN"))
```

but implies substantially lower overhead during query parsing and is therefore recommended to use, especially for large numeric filter expressions.

```fql
and(string("hello world"),filter(auth:int("1 20 453 3473")))
```

The expression `int("1 20 453 3473")` will be treated as an `OR(1,20,453,3473)`, where the numeric filter terms may be IDs within a subscription-based system.

### Query term weighting

#### Non-scope queries

The query term weight feature (weight=x parameter to string() operator) will only have effect when searching against composite fields.

#### Scope queries

The query

```fql
xml:or(string("hello world"), string("programming", weight=200))
```

will apply extra relevance boost for documents containing the term `programming`. Boosting can also be applied on sub-scope level:

```fql
xml:or(string("hello world"), title:string("programming", weight=200))
```
Search Timeout

Specify configurable search timeout (searchtimeout) to drop queries that would take too long time to execute.

The 'searchtimeout' parameter defines the actual timeout in the search engine in milliseconds, and can be set in the $FASTSEARCH/etc/config_data/RTSearch/*/fsearch.addon configuration file. The default timeout is 15000 (15 seconds).

Note: If a query times out, this will return the HTTP error message 1012.

Scope Navigation

Scope navigation is applying navigation to scope fields. Scope fields represent the content in a hierarchical structure as opposed to a flat field. It is not necessary to know the index schema in advance.

You specify which navigators to use and what content to aggregate over, at query time. This is done using the NAVIGATORS / rpf_navigation:navigators parameter.

```plaintext
&rpf_navigation:navigators=scopenavigator1(context=person@base, display="People"),scopenavigator2(context=location@base,display="Places")
```

The example defines two navigators that will aggregate over the XML attribute base of the XML elements person and location respectively.

When drilling down into the matching scopes you use the query parameter scopenavigation. The syntax for this parameter is according to FQL.

Error Handling

In case of FQL syntax error an HTTP status message is returned with status code 1201 and a syntax error text from the FQL parser. Refer to Chapter Using the HTTP GET Query Interface in the Query Integration Guide.

In the APIs an exception will be thrown, containing the same information.

Tip: Currently the FQL parser will not return a syntax error if you submit an operator expression with an unsupported parameter or a string() operator with an unsupported mode value. In this case the parameter will simply be ignored. This enables creation of custom extensions to the query language. Contact your FAST Account Manager for further details.

Simple Query Language support

The Simple Query Language support enables creation of search boxes with simple operators as found on popular search engines on the Web. This includes support for mixing words/phrases and mixing AND/OR/NOT in a simpler way than the more formal FQL syntax.

Tip: The supported syntax is a subset of the Simple Query Language as used in earlier versions of the product.
This format is supported within the syntax of the string() operator:

```python
string("<simple query language expression>", mode="simpleall")
```

```python
string("<simple query language expression>", mode="simpleany")
```

- **simpleall**: A logical AND is applied between the terms and/or phrases within the `<simple query language expression>`, unless the terms are preceded with `-`.
- **simpleany**: A logical OR is applied between the terms and/or phrases within the `<simple query language expression>`, unless the terms are preceded with `+` or `-`.

The terms/phrases in the `<simple query language expression>` must be separated with white space.

**Phrase support**

A phrase must be enclosed in double quotes. The double quotes indicating the phrase must be escaped, e.g.

```python
string("shakespeare \"king lear\")", mode="simpleall")
```

This example query will search for documents that includes `shakespeare' and the phrase "king lear". This query expression is equal to the following FQL query expressions:

```python
and(shakespeare, string("king lear", mode="phrase")
and(shakespeare, phrase(king, lear)
and("shakespeare", "king lear")
```

**Tip**: Make sure that your query front-end escapes double quotes written by the end-user.

**Query language expression, simpleall mode**

`mode="simpleall"` implies that all query terms/phrases (separated by white space) have to be matched in the returned documents. This equals a logical AND between the terms/phrases.

`-` operator may also be used to indicate a 'NOT' expression, and this is described in the table.

**Table 14**: Simple Query Language, mode="simpleall"

<table>
<thead>
<tr>
<th>The prefix</th>
<th>denotes...</th>
</tr>
</thead>
</table>
| +          | Allowed but ignored. The expressions:  
+faust  +goethe  
+faust goethe  
faust +goethe  
faust goethe  
are equivalent and will return all documents that include both 'goethe' and 'faust'. |
| -          | A logical NOT applied to the following term.  
goethe faust -marlowe  
returning all documents that include 'goethe' and 'faust', but not if these documents also includes 'marlowe' |
**Query language expression, simpleany mode**

mode="simpleany" implies that at least one of the query terms/phrases (separated by white space) has to be matched in the returned documents. This default matching mode can be overridden using +/- as described in the table.

**Tip:** Using this mode corresponds to using the `OR` boolean query parameter (not `ANY`) as defined in section *Boolean query operators*.

In mode="simpleany" one of the query terms/phrases has to be matched if no +/- operators are applied (only one or more terms delimited by white space). The usage of the +/- operators and parentheses is described in the table.

<table>
<thead>
<tr>
<th>The prefix</th>
<th>denotes...</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>The term/phrase following the `+' must be matched (logical AND)</td>
</tr>
<tr>
<td>+faust</td>
<td>returning all documents that include 'goethe' and 'faust'</td>
</tr>
<tr>
<td>+faust goethe</td>
<td>returning all documents that include 'faust'. The documents that also include 'goethe' will get higher dynamic rank and thus appear higher up in the result set.</td>
</tr>
<tr>
<td>&quot;king lear&quot; +shakespeare</td>
<td>returning all documents that include 'shakespeare'. The documents that also include the phrase &quot;king lear&quot; will get higher dynamic rank and thus appear higher up in the result set.</td>
</tr>
<tr>
<td>-</td>
<td>A logical NOT.</td>
</tr>
<tr>
<td>+goethe +faust</td>
<td>returning all documents that include 'goethe' AND 'faust', but not if these documents also include 'marlowe'.</td>
</tr>
<tr>
<td>+goethe faust</td>
<td>returning all documents that include 'goethe', but not if these documents also include 'marlowe'. The documents that also include 'faust' will get higher dynamic rank and thus appear higher up in the result set.</td>
</tr>
<tr>
<td>faust -marlowe</td>
<td>This is a special case that will be evaluated using an approximation, and will be evaluated as andnot(faust, marlowe)</td>
</tr>
</tbody>
</table>

**Tip:** No space is allowed between the `+/-` character and the following term/phrase.
**Advanced Query Language Support**

The FAST Query Language (FQL) supports the Advanced Query Language syntax with a few exceptions. The following table indicates the Advanced Query Language (AQL) features that are not supported in FQL, are deprecated or are not fully compatible:

<table>
<thead>
<tr>
<th>Query expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator precedence</td>
<td>For AQL the operators all have the same precedence. For FQL the following operator precedence applies: <code>n in</code> (i.e. <code>:</code> operator) <code>n not n near</code>, <code>onear n rank n andnot n and n any n or</code> This only has an impact if you mix the operators in infix format without using parentheses.</td>
</tr>
<tr>
<td>NEAR/ONEAR syntax</td>
<td>FQL uses prefix operator syntax for these operators, whereas AQL uses infix syntax. The infix syntax for NEAR/N and ONEAR/N is supported in FQL for backwards compatibility. Note that this syntax is deprecated and may not be supported in the next major release of FAST Enterprise Search Platform.</td>
</tr>
<tr>
<td>Numeric range operators</td>
<td>The numeric range operators are re-defined in FQL, using a prefix syntax. The infix syntax for numeric range operators is supported in FQL for backwards compatibility. Note that this syntax is deprecated and may not be supported in the next major release of FAST Enterprise Search Platform.</td>
</tr>
<tr>
<td>Using double quotes</td>
<td>In AQL double quotes indicates that phrase search shall be applied. In FQL double quotes represents a string of terms that not necessarily shall be treated as a phrase. However, by default FQL treats a set of tokens enclosed in double quotes as a phrase when not appearing as an argument to a string operator. This means that it is compatible with AQL when using an expression like: <code>foo AND &quot;hello world&quot;</code></td>
</tr>
<tr>
<td>Term weight</td>
<td>When using FQL, term weight has to be applied as a <code>weight</code> parameter to a <code>string()</code> operator. Hence it cannot be used as in AQL.</td>
</tr>
<tr>
<td>Spell Check</td>
<td>The Advanced Spell Check feature enables you to detect and spell check implicit phrases in queries. The AQL query: <code>nissan AND macra</code> will match the phrase &quot;nissan micra&quot; if this is found in the advanced spell check dictionary. When using FQL you will only achieve spell checking for this phrase if the phrase is enclosed in a string() operator. Hence, the AQL query above will not be properly spell checked.</td>
</tr>
<tr>
<td>Boundary match</td>
<td>The boundary match operators are re-defined in FQL, using a prefix operator syntax. Hence, the <code>^</code> and <code>^</code> are not supported in FQL.</td>
</tr>
<tr>
<td>Syntax check</td>
<td>FQL provides a more strict syntax check. Syntactically incorrect AQL queries will in many cases be approximated without any syntax error feedback: FQL will not accept incorrect syntax, but return a syntax error feedback with the query result instead.</td>
</tr>
</tbody>
</table>
Simple Query Language Compatibility

FQL provides a mechanism for enclosing simple query language constructs in a string() operator. This is described in section Simple Query Language Support.

The simple query language syntax supported within FQL supports only a subset of the Simple Query Language. The differences are:

- Parentheses are not supported within the simple query language expression.
- Field/scope specifications are not supported inside the simple query language expression.
- Numeric matching is not supported within the simple query language expression.

Table 17: FQL - Simple Query Language compatibility

<table>
<thead>
<tr>
<th>Old Query type</th>
<th>Support in FQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE=any</td>
<td>string(&quot;&lt;query string&gt;&quot;, mode=&quot;simpleany&quot;) +/- and phrases are supported. Parentheses, numeric expressions and field specifications (e.g. title:shakespeare) are not supported.</td>
</tr>
<tr>
<td>TYPE=all</td>
<td>string(&quot;&lt;query string&gt;&quot;, mode=&quot;simpleall&quot;) +/- and phrases are supported. Parentheses, numeric expressions and field specifications (e.g. title:shakespeare) are not supported.</td>
</tr>
<tr>
<td>TYPE=phrase</td>
<td>string(&quot;&lt;query string&gt;&quot;, mode=&quot;phrase&quot;)</td>
</tr>
<tr>
<td>TYPE=adv</td>
<td>With a few exceptions the old Advanced Query Language query string can be expressed in FQL syntax.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anti-phrasing</td>
<td>The anti-phrasing feature is used to remove common phrases, as these do not contribute to precision. When using FQL you will only achieve anti-phrasing for a given phrase if the phrase is enclosed in a string() operator.</td>
</tr>
</tbody>
</table>

The FILTER parameter

The query parameter named FILTER is deprecated in FAST Enterprise Search Platform 5.0. In this version of the product it may be used in association with the Simple/Advanced query language. When using FQL, filter constraints should be expressed using the filter() operator within an FQL expression.
Chapter 3

Query Transformations

Topics:

- Query Transformations Principles
- Query format considerations
- Query Transformation Feedback

This chapter presents the query transformations that provide feedback, as can be seen when submitting queries through the Search Front End.

- Query Transformations Principles
- Query Transformation Feedback
Query Transformations Principles

FAST Enterprise Search Platform is able to perform a number of automatic or suggested transformations of the user's query, based on Advanced Linguistics. This includes spell checking, phrase recognition and anti-phrasing.

There are three ways of using the query transformations:

- **Modify** - The query term string is automatically modified using the transformation parameters. The modified query and the result set is returned.

- **Suggest** - The executed query is not transformed, but a suggested transformed query is returned together with the result set. This enables search result pages using ‘search tip.’

- **Modify if no hits** - It is possible to request a query term string to be transformed in case of no returned hits from the original query. In this case the modified query term string is returned so that a result page may inform the end-user of the performed transformation.

The table lists the names of the query pipeline stages which will provide query transformation feedback as used in the APIs and returned on the HTTP interface:

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Name (API)</th>
<th>Name (HTTP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lemmatization</td>
<td>qt_lemmatizer</td>
<td>qt_lemmatize</td>
</tr>
<tr>
<td>Spell check</td>
<td>FastQT_DidYouMean</td>
<td>spell</td>
</tr>
<tr>
<td>Anti Phrasing</td>
<td>FastQT_DidYouMean</td>
<td>spell</td>
</tr>
<tr>
<td>Find Similar</td>
<td>FastQT_Similar</td>
<td>similar_to</td>
</tr>
<tr>
<td>Synonyms</td>
<td></td>
<td>qt_querisyonymys</td>
</tr>
</tbody>
</table>

**Tip:** If you change the query transformation configuration in the Query & Result Server, you must restart the engine before the new configuration is activated.

Query format considerations

Query transformations will be applied to all string() operands unless enclosed in filter() or if linguistics="off" is specified.

Query Transformation Feedback

The Search APIs provide a set of methods that operates on the query transformation feedback returned with the query. Refer to the API chapters for details.

When using the HTTP GET query interface, the feedback may be provided in the form of XML data, returned to the search client by the Query and Result Server (QRserver). The table provides the definition of the information returned for each query transformation:

| Table 19: Query transformation feedback definition |
Query Transformations

The following sub-sections describe the query transformers in more detail, including a description of the query transformation feedback. Refer to the Search APIs section for an overview on how to utilize the transformations in queries and results.

### Query Transformation for Anti Phrasing

This transformer removes common phrases from the query.

The Anti-phrasing is controlled as part of the overall spell check framework (the SPELL query parameter).

The feedback indicates that the transformer performed removal of common phrases from the query:

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>FastQT_DidYouMean</td>
</tr>
<tr>
<td>QUERY</td>
<td>Query string after rewrite/suggest transformation (anti-phrase removed)</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>Query string before rewrite/suggest transformation</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>Textual explanation of spell check operation. “Rewrote query” or “Suggested rewrite for query”</td>
</tr>
<tr>
<td>MESSAGEID</td>
<td>11: Anti-phrasing (modify) 12: Anti-phrasing (suggest)</td>
</tr>
</tbody>
</table>

### Query Transformations for Find Similar

A find similar query performed. The query transformation feedback indicates the resulting query:

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>FastQT_SimilarQuery</td>
</tr>
<tr>
<td>QUERY</td>
<td>-</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>-</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>1: Expanding query to similar documents 2: Refining query to similar documents 3: Excluding similar documents 4: Ranking based on similar documents 5: No stack to operate on</td>
</tr>
</tbody>
</table>
Query Transformations for Synonyms

Table 22: Synonym processing has been performed on the submitted query.

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>qt_synonym</td>
</tr>
<tr>
<td>QUERY</td>
<td>-</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>-</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>1 : Transformed the query</td>
</tr>
<tr>
<td></td>
<td>2 : Semantic query transformation suggested</td>
</tr>
<tr>
<td></td>
<td>3 : Suggested acronyms</td>
</tr>
<tr>
<td></td>
<td>4 : Suggested synonyms</td>
</tr>
<tr>
<td></td>
<td>5 : Semantic query transformation performed on parts of the query</td>
</tr>
<tr>
<td></td>
<td>6 : Added synonyms</td>
</tr>
<tr>
<td></td>
<td>7 : No semantic query transformation</td>
</tr>
</tbody>
</table>

Query Transformations for Lemmatization

Lemmatization is applied to the query. This means that the query will look for any forms of the query term it can find in the lemmatization index. The actual query expression will not be modified.

Table 23: Query transformation feedback for all lemmatization modes

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>&quot;nop&quot;</td>
</tr>
<tr>
<td>QUERY</td>
<td>-</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>-</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>16 : Lemmatization turned off for current query</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>&quot;Modified the query&quot;</td>
</tr>
<tr>
<td>QUERY</td>
<td>TERM</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>&quot;Modified query stack&quot; or &quot;Modified FQL tree&quot;</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>4 : Term rerouted (lemmatization by document expansion)</td>
</tr>
</tbody>
</table>

Table 24: Query transformation feedback for lemmatization by document expansion

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>&quot;Modified the query&quot;</td>
</tr>
<tr>
<td>QUERY</td>
<td>LEM1 LEM2 ...</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>&quot;Modified query stack&quot; or &quot;Modified FQL tree&quot;</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>1 : &quot;Lemmatized term (query reduction)&quot;</td>
</tr>
</tbody>
</table>
Phrase detection and Spell check

The query transformation feedback will consist of a top-level element that indicates the total transformation performed and sub-elements indicating the query transformations performed by the simple and advanced spell check stages.

All transformation feedback is returned via the FastQT_DidYouMean as described below. In the query result you will first see the individual feedback for simple and/or advanced spell check. This may be used if you want to distinguish between these two stages in the Search Front End presentation. After that, you will see the consolidated feedback for all spell check stages. This is the preferred transformation feedback to be used in the Search Front End.

The elements will provide the resulting rewritten or suggested query in the QUERY attribute.

Tip: The rewritten query only indicates that the original query terms are queried against another part of the index that is expanded with inflected forms.

### Table 26: Query transformation feedback for lemmatization by query expansion

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>&quot;Modified the query&quot;</td>
</tr>
<tr>
<td>QUERY</td>
<td>LEM1 LEM2 ...</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>&quot;Modified query stack&quot; or &quot;Modified FQL tree&quot;</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>2 : &quot;Lemmatized term (query expansion)&quot;</td>
</tr>
</tbody>
</table>

### Table 27: Query transformation feedback for simple/advanced spell check

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>FastQT_DidYouMean</td>
</tr>
<tr>
<td>QUERY</td>
<td>-</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>&quot;no change to stack&quot; or &quot;No change to FQL tree&quot;</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>&quot;No query transformation&quot;</td>
</tr>
</tbody>
</table>

### Table 28: Query transformation feedback for consolidated spell check

<table>
<thead>
<tr>
<th>Element</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>FastQT_DidYouMean</td>
</tr>
<tr>
<td>QUERY</td>
<td>inside &quot;new york&quot; (ref example below)</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>-</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>Textual explanation of spell check operation, e.g. &quot;Spellchecked phrase&quot;</td>
</tr>
<tr>
<td>MESSAGEID</td>
<td>3 : Simple Spell Check (modify) 4 : Simple Spell Check (suggest) 6 : Advanced Spell Check (modify) 7 : Advanced Spell Check (suggest)</td>
</tr>
</tbody>
</table>
If you use the API methods to retrieve the query transformation feedback, you can either iterate through all transformations and look for the transformations you want to present, or use the getModification() to retrieve a named modification. In the latter case the API method will return the last modification entity for the given name. In the case of FastQT_DidYouMean this will be the consolidated spell check feedback.

The content of the QUERY element in the transformation for FastQT_DidYouMean cannot be used as a self-contained query string to be re-submitted as FQL. The most convenient way to resubmit the query will be to retrieve the original query object and re-submit with the value of the query parameter SPELL changed to "on".

Alternatively, you will have to wrap the content of the retrieved QUERY element from the transformation object as follows:

```java
string("<content of QUERY element>", mode="simpleany")
```

or
```
<box><pre>
string("<content of QUERY element>",
mode="simpleall")
</pre></box>
```

In this case you must remember to parse the string retrieved from the QUERY element and escape double quotes (in case the Advanced Spell Check has phrased the result).

**Example**

```java
string("iside newyork", mode="and")
```

The Simple Spell Check feature will translate `iside' to `inside'
The Advanced Spell Check feature will correct to the defined phrase `new york' if it exists in the dictionary.
This appendix provides an overview of optimizations and approximations that are being done to queries in certain situations.

Topics:

- Proximity Operator Evaluation
- Lemmatization
- Wildcard Query Evaluation
- Complex Scope Queries
- Increase QRserver stack size
Proximity Operator Evaluation

For the FQL query language it is possible to use more complex arguments to proximity operators. This includes boolean OR/ANY expressions and wildcard terms. Proximity expressions with more than 4 arguments are also evaluated correctly. In this case there is a difference in rank evaluation.

Certain approximations apply when using proximity operators with arguments that overlaps (e.g. two overlapping phrases).

Lemmatization

Normal (non-scope) fields do not support lemmatization with proximity operators, i.e. the lemmatization setting is ignored and you will only get correct match for the original terms in the document. When using FQL you will also get a Query Transformation feedback in this case.

Wildcard Query Evaluation

This section includes information on how wildcards are interpreted in certain situations.

Performance optimizations for wildcard queries

A wildcard cutoff can be configured in order to avoid too long execution times for queries with wide wildcard terms (matching too many words in the index). If the cutoff limit is reached, an empty result set is returned. Note that in this case you will not get any error message.

Complex Scope Queries

This section presents complex issues when dealing with scope fields and queries.

Typing of Numeric Scopes

When querying numeric scopes you must ensure that the type used in the query and in the scope match. If the price scope is of type ‘float’ you must apply explicit typing of the scope, example:

```
xm:products:product:price:float(5)
```

Querying recursive scopes

Recursive scopes means that the same scope name appears in different places in the scope tree, e.g.

```
<A>
  hello
  <B>
  <A>world</A>
</B>
```
A scope expression that is an argument of a query operator evaluates to the largest region of the document that satisfies the scope specification. This region might not be contained in the scope specification surrounding the operator expression and the whole expression will in this case return different results from a similar XPath expression.

The query:

```
B:A:world
```

will thus not return the document example above.

† Tip: This limitation only occurs when querying recursive scope structures.

**Implicit phrasing**

In certain cases a sequence of tokens in a query will automatically be rewritten to a phrase expression. This includes:

- Advanced Spell Check where the token(s) match a proper name, e.g. "New York"
- When a word includes non-searchable characters. In this case the tokenization will remove the non-searchable characters and split the original word into separate tokens, but the tokens will be searched as a phrase.

**Increase QRserver stack size**

Submitting deeply nested or/and complex queries can cause the QRserver stack to overflow. This can be fixed by increasing the stack size of the QRserver.

1. Open `$FASTSEARCH/etc/config_data/QRserver/webcluster/etc/qrserver/qrserverrc`.
2. Add a parameter named `stacksize`.

   The parameter value is the new size of the thread stack in bytes. The default value is 131072 (128Kb). The size of the stack you need depends on the complexity of the query. For example: `stacksize=196608 (192Kb)`.
3. Restart the QRserver.
4. Validate the new stack size by looking at the configuration page at: `http://server:qrserver-port/configuration`