## Table of Contents

- Index Pipeline Stages ................................................................. 1
- Format Signals Index Stage .......................................................... 2
- Date/time parsing and formatting .................................................. 3
- Example Stage Specification ......................................................... 4
- Configuration .............................................................................. 5
  - Configuration Properties ........................................................... 5
- Machine Learning Index Stage ....................................................... 6
  - Configuration .............................................................................. 6
    - Configuration Properties .......................................................... 6
- OpenNLP NER Extraction Index Stage ............................................. 8
  - Example Specification ............................................................... 8
  - Configuration .............................................................................. 9
    - Configuration Properties .......................................................... 9
- Update Experiment Stage ............................................................... 11
  - Configuration .............................................................................. 11
    - Configuration Properties .......................................................... 11
- Query Pipeline Stages ................................................................. 12
- Analytics Catalog Stage ............................................................... 13
  - Configuration .............................................................................. 14
    - Configuration Properties .......................................................... 14
- Boost with Signals Stage ............................................................... 16
  - Configuration overview ............................................................ 16
  - Solr query parameters .............................................................. 18
  - FAQs ....................................................................................... 18
  - Configuration .............................................................................. 19
    - Configuration Properties .......................................................... 20
- Experiment Stage ....................................................................... 23
  - Configuration .............................................................................. 23
    - Configuration Properties .......................................................... 23
- Machine Learning Stage .............................................................. 24
  - Configuration .............................................................................. 24
    - Configuration Properties .......................................................... 24
- Parameterized Boosting Stage ....................................................... 26
  - Configuration .............................................................................. 26
    - Configuration Properties .......................................................... 26
- Recommend Items for Item Stage .................................................. 28
  - Prerequisites .............................................................................. 28
  - Configuration .............................................................................. 28
    - Configuration Properties .......................................................... 28
- Recommend Items for User Stage .................................................. 32
  - Prerequisites .............................................................................. 32
  - Configuration .............................................................................. 32
Jobs Configuration Reference

Levenshtein Spell Checking Jobs

Item Similarity Recommender Jobs

Head/Tail Analysis Jobs

Ground Truth Jobs

Document Clustering Jobs

Collection Analysis Jobs

Co-occurrence Similarity Jobs

Configuration Properties

Basic job configuration

Input/output parameters

Model tuning parameters

Advanced job configuration

Training data settings

Model settings

Grid search settings

Item metadata settings

Configuration Properties

Cluster Labeling Jobs

Configuration Properties

Co-occurrence Similarity Jobs

Collection Analysis Jobs

Configuration Properties

Document Clustering Jobs

Configuration Properties

Ground Truth Jobs

Configuration Properties

Head/Tail Analysis Jobs

Head/tail analysis configuration

Job output

Head/Tail Plot (overall_distribution)

Summary Stats (summary_stat)

Query Details (queries_ordered)

Top Tokens (tokens_ordered)

Query Length (queryLength)

Tail Reasons table and pie chart (tail_reasons)

Pre-defined tail reasons

Custom dictionary

Head Tail Similarity (tail_warding)

Configuration Properties

Item Similarity Recommender Jobs

Levenshtein Spell Checking Jobs
Index Pipeline Stages

Index Pipeline stages are used to create and modify PipelineDocument objects. Use the Index Workbench to configure stages in a pipeline and preview the results.

See these reference topics for details about each index pipeline stage:

- Format Signals
- OpenNLP NER Extraction
- Machine Learning
- NLP Annotator
- Update Experiment :leveloffset: +1
Format Signals Index Stage

The Format Signals stage normalizes both the fields and field contents of a PipelineDocument to ensure that certain pre-defined fields for signals are populated.
Date/time parsing and formatting

Timestamp data can be obtained from the following fields, in this order of precedence:

- timestamp
- timestamp_tdt
- timestamp_dt
- epoch - value in this field is treated as a number of milliseconds since epoch, and UTC zone is assumed.

It’s now possible to specify the locale to be used for parsing timestamps by setting the “timestampLocale” property in the stage configuration. If this property is null then the default platform locale will be used.

Output document will carry the following two fields:

- “timestamp” - containing the ISO8601 timestamp
- “tz_timestamp_txt” - containing the "zoned format" of the timestamp with normalized components.

<table>
<thead>
<tr>
<th>Note</th>
<th>This stage does not define a list of allowed types.</th>
</tr>
</thead>
</table>

This stage does not define a list of allowed types.
Example Stage Specification

The Format Signals stage defined as part of the default 'signals_ingest' pipeline included with Fusion:

```json
{
  "type": "format-signals",
  "id": "ingest-signals",
  "flatten": true,
  "undefinedType": "general",
  "skip": false,
  "label": "format-signals",
  "type": "format-signals"
}
```
Configuration

Tip

When entering configuration values in the UI, use *unescaped* characters, such as `\t` for the tab character. When entering configuration values in the API, use *escaped* characters, such as `\\t` for the tab character.

When using Fusion's REST-API, the ID of this stage is: `signal-formatter`.

Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>allowedTypes</strong></td>
<td>List of allowed signal types.</td>
</tr>
<tr>
<td>Allowed Types</td>
<td>type: <em>array of string</em></td>
</tr>
<tr>
<td><strong>flatten</strong></td>
<td>Flatten nested values.</td>
</tr>
<tr>
<td>Flatten</td>
<td>type: <em>boolean</em></td>
</tr>
<tr>
<td>default value: <code>true</code></td>
<td></td>
</tr>
<tr>
<td><strong>timestampLocale</strong></td>
<td>Use this locale when parsing timestamp information. Null uses platform default locale.</td>
</tr>
<tr>
<td>Timestamp Locale</td>
<td>type: <em>string</em></td>
</tr>
<tr>
<td><strong>undefinedType</strong></td>
<td>Signal type when undefined. Null discards events with undefined type.</td>
</tr>
<tr>
<td>Undefined Type</td>
<td>type: <em>string</em></td>
</tr>
</tbody>
</table>
Machine Learning Index Stage

The Fusion machine learning indexing stage uses a trained machine learning model to analyze a field or fields of a PipelineDocument and stores the results of analysis in a new field of either the PipelineDocument or Context object.

In order to use the Machine Learning Stage, you must train a machine learning model. There are two different ways to train a model:

- Use a Fusion AI job that trains a model, like Logistic Regression or Random Forest.
- Train a model using Spark's MLlib API outside of Fusion, and upload this model into Fusion's blob store. Complete details are available in Machine Learning Models in Fusion.

<table>
<thead>
<tr>
<th>Tip</th>
<th>When specifying field names, multiple field names are supported, in this format: field1:weight,field2:weight,field3:weight</th>
</tr>
</thead>
</table>

Configuration

| Tip | When entering configuration values in the UI, use unescaped characters, such as \t for the tab character. When entering configuration values in the API, use escaped characters, such as \\
t for the tab character. |
|-----|------------------------------------------------------------------------------------------------------------------|

When using Fusion’s REST-API, the ID of this stage is: machine-learning.

Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultValue</td>
<td>Value to provide if a prediction cannot be made for a document.</td>
</tr>
<tr>
<td>Default Value</td>
<td>type: string</td>
</tr>
<tr>
<td>docFeatureFieldName</td>
<td>Name of the field to extract document features from (model input).</td>
</tr>
<tr>
<td>Document Feature Field</td>
<td>type: string</td>
</tr>
<tr>
<td></td>
<td>default value: ‘body_t’</td>
</tr>
<tr>
<td></td>
<td>minLength: 1</td>
</tr>
<tr>
<td>Property</td>
<td>Description, Type</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>failOnError</td>
<td>Flag to indicate if this stage should throw an exception if an error occurs while generating a prediction for a document.</td>
</tr>
<tr>
<td></td>
<td>type: boolean</td>
</tr>
<tr>
<td></td>
<td>default value: ‘false’</td>
</tr>
<tr>
<td>includeRawPredictions</td>
<td>Flag to indicate that raw predictions and scores, in addition to the best prediction, should be set on the document.</td>
</tr>
<tr>
<td></td>
<td>type: boolean</td>
</tr>
<tr>
<td></td>
<td>default value: ‘false’</td>
</tr>
<tr>
<td>modelId</td>
<td>The ID of the ML model stored in the Fusion blob store.</td>
</tr>
<tr>
<td></td>
<td>type: string</td>
</tr>
<tr>
<td>predictionFieldName</td>
<td>Name of the field to store the prediction (model output) in the document.</td>
</tr>
<tr>
<td></td>
<td>type: string</td>
</tr>
<tr>
<td>storeInContext</td>
<td>Flag to indicate that the prediction should be set as a context property instead of setting a field on the document.</td>
</tr>
<tr>
<td></td>
<td>type: boolean</td>
</tr>
<tr>
<td></td>
<td>default value: ‘false’</td>
</tr>
</tbody>
</table>
OpenNLP NER Extraction Index Stage

Named Entity Recognition (NER) is the task of finding the names of persons, organizations, locations, and/or things in a passage of free text. The OpenNLP NER Extraction index stage (previously called the OpenNLP NER Extractor stage) uses a set of rules to find named entities in a field in the Pipeline Document (the "source") and populates a new fields (the "target") with these entities.

This stage uses Apache OpenNLP project's Named Entity Recognition tool (the Name Finder tool). The OpenNLP documentation states:

The Name Finder tool can detect named entities and numbers in text. To be able to detect entities the Name Finder needs a model. The model is dependent on the language and entity type it was trained for. The OpenNLP projects offers a number of pre-trained name finder models which are trained on various freely available corpora. They can be downloaded at our model download page. To find names in raw text the text must be segmented into tokens and sentences.

See this video tutorial for a demonstration of how to configure this stage:

Models are available from the OpenNLP models SourceForge repository.

The Fusion directory fusion/4.2.x/data/nlp contains a set of NER models for English, as well as sentence, token, and part-of-speech models.

Before they can be used, model files must be uploaded to the blob store. Here is an example of how to upload the sentence model file using the curl command-line utility, where "admin" is the name of a user with admin privileges, and "pass" is the password:

```
```

See also the Fusion AI topic on Natural Language Processing.

Example Specification

Specification of a stage which extracts names of people and places from field named 'body':


Configuration

Tip
When entering configuration values in the UI, use *unescaped* characters, such as `\t` for the tab character. When entering configuration values in the API, use *escaped* characters, such as `\\t` for the tab character.

When using Fusion’s REST-API, the ID of this stage is: `nlp-extractor`.

Configuration Properties
<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>rules</td>
<td>type: array of object</td>
</tr>
<tr>
<td></td>
<td>minimum number of items (minItems): 1</td>
</tr>
</tbody>
</table>
|                                  | object attributes: \{
|                                  |   entityTypes : \{
|                                  |     display name: Entity Types
|                                  |     type: array of object
|                                  |   }
|                                  | sentenceModelLocation (required) : \{
|                                  |     display name: Sentence Model
|                                  |     type: string
|                                  |     blobType : model:open-nlp
|                                  |     reference : blob
|                                  |   }
|                                  | source (required) : \{
|                                  |     display name: Source Fields
|                                  |     type: array of string
|                                  |     minItems : 1
|                                  |   }
|                                  | target (required) : \{
|                                  |     display name: Target Field
|                                  |     type: string
|                                  |   }
|                                  | tokenizerModelLocation (required) : \{
|                                  |     display name: Tokenizer Model
|                                  |     type: string
|                                  |     blobType : model:open-nlp
|                                  |     reference : blob
|                                  |   }
|                                  | writeMode : \{
|                                  |     display name: Write Mode
|                                  |     type: string
|                                  |     default value: ‘append’
|                                  |     description : What to do if document has target field already
|                                  |     enum: \{
|                                  |       overwrite append
|                                  |     } |
|                                  |   } |
|                                  |   } |
Update Experiment Stage

This stage is part of Fusion’s Machine Learning framework. It is highly experimental and subject to change.

The Machine Learning framework provides tools to evaluate alternative methods of computing and displaying search results. For example, an experiment may consist of a system which has two or more different query pipelines running, and users are randomly served search results using some variant of the system via instrumented pages that capture user response. In such a system, an Update Experiment stage would be used to feed these responses directly back into the running experiment.

The Experiment Update stage checks each document for an experiment ID, a variant ID and a value and updates the experiment accordingly. A default experiment ID can be provided if none is found in the document. By default these documents are silently discarded after processing, but the stage can be configured to forward them down the pipeline to the next stages.

Configuration

<table>
<thead>
<tr>
<th>Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>When entering configuration values in the UI, use unescaped characters, such as \t for the tab character. When entering configuration values in the API, use escaped characters, such as \	 for the tab character.</td>
</tr>
</tbody>
</table>

When using Fusion’s REST-API, the ID of this stage is: experiment-update.

Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
</table>
Query Pipeline Stages

A query pipeline is made up of a series of query stages that process incoming search queries.

A pipeline stage definition associates a unique ID with a set of properties. These definitions are registered with the Fusion API service and stored in ZooKeeper for re-use across pipelines and search applications.

Fusion includes a number of specialized query stages as well as a JavaScript stage that allows advanced processing via a JavaScript program.

Use the Query Workbench to configure stages in a query pipeline.

See these reference topics for details about each query pipeline stage:

- Analytics Catalog Query
- Boost Documents
- Boost with Signals
- Experiment
- Machine Learning
- NLP Annotator
- Machine Learning (Responses)
- Parameterized Boosting
- Solr MoreLikeThis
- Recommend Items for User
- Recommend Items for Item
- Response Document Exclusion
- Response Document Field Redaction
- Response Pairwise Swap
- Response Shuffle :leveloffset: +1
Analytics Catalog Stage

The Analytics Catalog query pipeline stage lets you define views of data stored in Fusion collections. You can query the objects in the Analytics Catalog using SQL, Solr Streaming Expressions, or regular Solr queries. See also the Catalog API.
### Configuration

**Tip**

When entering configuration values in the UI, use *unescaped* characters, such as `\t` for the tab character. When entering configuration values in the API, use *escaped* characters, such as `\\t` for the tab character.

When using Fusion's REST-API, the ID of this stage is: `analytics-catalog-query`.

#### Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>catalogProjectId</td>
<td><strong>Analytics catalog project ID.</strong> type: <em>string</em> default value: ‘fusion’</td>
</tr>
<tr>
<td>contextKey</td>
<td><strong>The key used to bind query results into the pipeline context; if not set, the query results are returned to the client.</strong> type: <em>string</em></td>
</tr>
</tbody>
</table>
| params         | **Additional Params to be Included in the Solr Query Request. Only Applies if the Query Type is 'solr'.** type: array of object object attributes: \{
<p>| queryText      | <strong>Query to execute; accepts any valid Solr query, SQL, or Solr streaming expression. Replaceable parameters will be applied before query execution using values pulled from request parameters and the pipeline context.</strong> type: <em>string</em> |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>queryType</strong></td>
<td>Query type: solr, sql, or streaming_expression</td>
</tr>
<tr>
<td>Query Type</td>
<td>type: <em>string</em></td>
</tr>
<tr>
<td></td>
<td>default value: ‘sql’</td>
</tr>
<tr>
<td></td>
<td>enum: { solr sql streaming_expression }</td>
</tr>
<tr>
<td><strong>requestHandler</strong></td>
<td>The Solr request handler to send a query to; only applies if query type is ‘solr’ or ‘streaming_expression’. Defaults to /select for solr queries and /stream for streaming expressions.</td>
</tr>
<tr>
<td>Solr Request Handler.</td>
<td>type: <em>string</em></td>
</tr>
</tbody>
</table>
Boost with Signals Stage

The Boost with Signals query pipeline stage uses aggregated signals to selectively boost items in the set of search results.

Using the main query and the stage configuration parameters, this stage performs a secondary query to the _signals_aggr collection and returns updated boost weights for the items in the main query’s search results. Items that have received more user interaction also receive higher boost weights.

See Recommendation Methods for more information.

This stage supports asynchronous processing.

| Tip | This stage accesses the signals_aggr collection. Before using it, verify that the following permission is set: GET:/solr/<collection-name>_signals_aggr/select |

Configuration overview

The fields below are especially useful to understand when configuring this stage.

<table>
<thead>
<tr>
<th>Number of Recommendations</th>
<th>Sets the rows query param in the main query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>numRecommendations</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Signals</th>
<th>Sets the rows query param in the query that searches the _signals_aggr collection, so only the specified number of aggregated signals are retrieved and used for boosting.</th>
</tr>
</thead>
<tbody>
<tr>
<td>numSignals</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Aggregation Type</th>
<th>A filter to retrieve aggregated signals in the _signals_aggr collection per each aggregated signal's aggr_type_s field value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggrType</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solr Field to Boost On</th>
<th>The document field in the main collection on which to perform boosting. Typically it should use default field, which is id.</th>
</tr>
</thead>
<tbody>
<tr>
<td>boostId</td>
<td>This field corresponds to the Rollup Field/rollupField field. Together, these two fields act like a &lt;field&gt;::&lt;value&gt; pair in the query modification for boosting.</td>
</tr>
<tr>
<td><strong>Boost Method</strong></td>
<td><strong>boostingMethod</strong></td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>This adds a query parameter to the original query, either “query-param” or “query-parser”. The result is (&quot;query-param&quot; or &quot;query-parser&quot;) + Boost Param(&quot;boost&quot; or `bq`), as in the examples below:</td>
<td></td>
</tr>
<tr>
<td>“query-param”+&quot;boost&quot;, result boost query param</td>
<td></td>
</tr>
<tr>
<td><code>boost=&quot;map(query({field \='id\' v= '6239046,13026192'}), 0, 0, 1, 27.1705)&quot;</code></td>
<td></td>
</tr>
<tr>
<td>“query-parser”+&quot;boost&quot;, result boost query param</td>
<td></td>
</tr>
<tr>
<td><code>bp_xxx_bbqx=&quot;map(query({!field f='id' v= '6239046,13026192'}), 0, 0, 1, 27.1705)&quot;</code></td>
<td></td>
</tr>
</tbody>
</table>

---

**Tip**

When **Boost Param** uses bq, similar logic applies. When **Boost Param/boostingParam** uses “boost”, it works with both “query-param” and “query-parser”.

---

<table>
<thead>
<tr>
<th><strong>Rollup Field</strong></th>
<th><strong>rollupField</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates which aggregated signal document field the boost parameter will use for the final boosting. It works in combination with the <strong>Solr Field to Boost On/boostId</strong> field.</td>
<td></td>
</tr>
<tr>
<td>This should be set to the field in the aggregated signal collection that stores the doc list that is aggregated as one record. By default it's set to <strong>doc_id_s</strong>, because by default this is used in the click_signal_aggr SQL job.</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th><strong>Rollup Weight Field</strong></th>
<th><strong>rollupWeightField</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates the final boost weight used to calculate the new score for docs retrieved by the main query.</td>
<td></td>
</tr>
<tr>
<td>Similar to <strong>Rollup Field/rollupField</strong> above, this should be set to the field in the aggregated signal collection that stores the final weight that was calculated. By default it's <strong>weight_d</strong>, because by default this is used in the click_signal_aggr SQL job.</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th><strong>Final Boost Weight Expression</strong></th>
<th><strong>weightExpression</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculates the final weight using the weight and score retrieved from the <strong>_signals_aggr</strong> collection.</td>
<td></td>
</tr>
<tr>
<td>The default value is <code>math:log(weight_d + 1) + 10 * math:log(score+1)</code></td>
<td></td>
</tr>
</tbody>
</table>
Solr query parameters

These parameters are used in the Solr Query parameters/queryParams field for retrieving signal aggregation docs from the _signals_aggr collection. These Solr query params will affect which aggregated signals are used for producing the boosting parameter on the main query.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qf=query_t</td>
<td>Defines which field to query. In the default case, the query searches on the query_t field of aggregated signal docs.</td>
</tr>
<tr>
<td>pf=query_t^50</td>
<td>Boosts docs within the set of retrieved docs using phrase matching.</td>
</tr>
<tr>
<td>pf2=query_t^20</td>
<td>pf2 is similar to pf; the difference is that pf2 works on bigram phrases.</td>
</tr>
<tr>
<td>pf3=query_t^10</td>
<td>pf3 is similar to pf; the difference is that pf3 works on trigram phrases.</td>
</tr>
</tbody>
</table>

FAQs

If there is fq in the main query, how is it matched with the correct aggregated signal?

In this case, you need to use the lw.rec.fq query parameter in the main query. lw.rec.fq can be parsed by the Boost with Signals stage, and therefore the filters specified in it can be added to the Solr query that is retrieving the aggregated signals.

For example, if we have filter query param fq=format:CD&fq=name:Latin, this needs to be translated into lw.rec.fq=filters_s:"format:CD $ name:Latin". The final main query should be:

```
```

Now the Boost with Signals stage will only retrieve aggregated signals that have the same filter query.

What if my aggregated signals are in a different collection?

You can point the Boost with Signals stage to a different signal collection by adding a collection parameter in the Solr Query Parameters section.
## Solr Query parameters

Parameters for querying Signal aggregation collection

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>qf</td>
<td>query_t</td>
</tr>
<tr>
<td>pf</td>
<td>query_t^50</td>
</tr>
<tr>
<td>pf</td>
<td>query_t~3^20</td>
</tr>
<tr>
<td>pf2</td>
<td>query_t^20</td>
</tr>
<tr>
<td>pf2</td>
<td>query_t~3^10</td>
</tr>
<tr>
<td>pf3</td>
<td>query_t^10</td>
</tr>
<tr>
<td>pf3</td>
<td>query_t~3^5</td>
</tr>
<tr>
<td>boost</td>
<td>map(query{</td>
</tr>
<tr>
<td>mm</td>
<td>50%</td>
</tr>
<tr>
<td>defType</td>
<td>edismax</td>
</tr>
<tr>
<td>sort</td>
<td>score desc, weight_d</td>
</tr>
<tr>
<td>collection</td>
<td>signal_aggre2</td>
</tr>
</tbody>
</table>
Tip

When entering configuration values in the UI, use *unescaped* characters, such as `\t` for the tab character. When entering configuration values in the API, use *escaped* characters, such as `\\t` for the tab character.

When using Fusion’s REST-API, the ID of this stage is: `recommendation`.

### Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aggrType</strong></td>
<td>Aggregation Type</td>
</tr>
<tr>
<td></td>
<td>type: <em>string</em></td>
</tr>
<tr>
<td></td>
<td>default value: ‘click@doc_id,filters,query’</td>
</tr>
<tr>
<td><strong>boostId</strong></td>
<td>Solr Field to Boost On</td>
</tr>
<tr>
<td></td>
<td>type: <em>string</em></td>
</tr>
<tr>
<td></td>
<td>default value: ‘id’</td>
</tr>
<tr>
<td><strong>boostingMethod</strong></td>
<td>The boost method to use. <em>query-parser</em> should be chosen if <em>defType</em>!=<em>edismax</em> for main query.</td>
</tr>
<tr>
<td></td>
<td>type: <em>string</em></td>
</tr>
<tr>
<td></td>
<td>default value: ‘query-param’</td>
</tr>
<tr>
<td></td>
<td>enum: <code>{ query-param, query-parser }</code></td>
</tr>
<tr>
<td><strong>boostingParam</strong></td>
<td>‘Boost’ multiplies scores by the boost values whereas ‘bq’ adds optional clauses to main query.</td>
</tr>
<tr>
<td></td>
<td>type: <em>string</em></td>
</tr>
<tr>
<td></td>
<td>default value: ‘boost’</td>
</tr>
<tr>
<td></td>
<td>enum: <code>{ boost, bq }</code></td>
</tr>
<tr>
<td><strong>numRecommendations</strong></td>
<td>Number of Recommendations</td>
</tr>
<tr>
<td></td>
<td>type: <em>integer</em></td>
</tr>
<tr>
<td></td>
<td>default value: ‘10’</td>
</tr>
<tr>
<td>Property</td>
<td>Description, Type</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>numSignals</td>
<td>Number of signals to process when getting recommended items.</td>
</tr>
<tr>
<td></td>
<td>type: <code>integer</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘100’</td>
</tr>
<tr>
<td>queryParams</td>
<td>Parameters for querying Signal aggregation collection</td>
</tr>
<tr>
<td></td>
<td>type: <code>array of object</code></td>
</tr>
<tr>
<td>rollupField</td>
<td>Field to use for rolling up documents that have same doc id’s</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td>rollupWeightField</td>
<td>Field to use for signal weights</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td>Property</td>
<td>Description, Type</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rollupWeightStrategy</td>
<td>Rollup weight strategy</td>
</tr>
<tr>
<td></td>
<td>Strategy to use for rolling up the weight: max or sum</td>
</tr>
<tr>
<td></td>
<td>type: string</td>
</tr>
<tr>
<td></td>
<td>enum: <code>{ sum max }</code></td>
</tr>
<tr>
<td>scaleRange</td>
<td>Scale Boosts</td>
</tr>
<tr>
<td></td>
<td>Scale the boost values to a [min,max] range</td>
</tr>
<tr>
<td></td>
<td>type: object</td>
</tr>
</tbody>
</table>
|                          | object attributes: `{                     
|                          | scaleMax: `{                     
|                          |   display name: Maximum value of the scale range                          |
|                          |   type: number                                                                |
|                          | }                             
|                          | scaleMin: `{                     
|                          |   display name: Minimum value of the scale range                           |
|                          |   type: number                                                                |
|                          | }                             |
| weightExpression         | Final Boost Weight Expression                                                    |
|                          | Optional expression to compute the final boost weight using a combination of     |
|                          | fields returned by Solr, such as score and weight_d. Set to weight_d for similar |
|                          | behavior as older versions                                                     |
|                          | type: string                                                                    |
|                          | default value: `math:log(weight_d + 1) + 10 * math:log(score+1)`               |
# Experiment Stage

The Experiment query pipeline stage is part of Fusion’s Machine Learning framework.

The Machine Learning framework provides tools to evaluate alternative methods of computing and displaying search results. For example, an experiment may consist of a system which has two or more different query pipelines running, and users are randomly served search results using some variant of the system via instrumented pages that capture user response. In such a system, the Experiment query stage selects a variant from a running experiment and injects its properties into the current Context.

## Configuration

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>experimentReference</td>
<td>Use the selected experiment to route or modify the request</td>
</tr>
<tr>
<td>required</td>
<td></td>
</tr>
</tbody>
</table>

Experiment Reference

Type: `object`

Object attributes: 

```json
{  
  id: {  
    display name: Experiment ID  
    type: `string`  
    maxLength: 128  
    pattern: `^[A-Za-z0-9_\-]+$`  
  }  
  percentOfTraffic: {  
    display name: Percent of Traffic to Include In Experiment  
    type: `number`  
    default value: ‘100.0’  
    description: The percent of traffic to include in the experiment (if present)  
    exclusiveMaximum: false  
    exclusiveMinimum: false  
    maximum: 100.0  
    minimum: 0.001  
  }  
}
```

When using Fusion's REST-API, the ID of this stage is: `experiment`.

When entering configuration values in the UI, use *unescaped* characters, such as `\t` for the tab character. When entering configuration values in the API, use *escaped* characters, such as `\\t` for the tab character.
Machine Learning Stage

The Machine Learning query pipeline stage uses a trained machine learning model to analyze a field or fields of a Request object and stores the results of analysis in a new field added to either the Request or the Context object.

In order to use the Machine Learning Stage, you must train a machine learning model. There are two different ways to train a model:

- Use a Fusion job that trains a model, like Logistic Regression or Random Forest.
- Train a model using Spark’s MLlib API outside of Fusion, and upload this model into Fusion’s blob store. Complete details are available in Machine Learning Models in Fusion.

Tip

When specifying field names, multiple field names are supported, in this format: field1:weight,field2:weight,field3:weight

Configuration

Tip

When entering configuration values in the UI, use unescaped characters, such as \t for the tab character. When entering configuration values in the API, use escaped characters, such as \\\	 for the tab character.

When using Fusion’s REST-API, the ID of this stage is: machine-learning-query.

Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultValue</td>
<td>Value to provide if a prediction cannot be made for a document. type: string</td>
</tr>
<tr>
<td>failOnError</td>
<td>Flag to indicate if this stage should throw an exception if an error occurs while generating a prediction for a document. type: boolean default value: ‘false’</td>
</tr>
<tr>
<td>Property</td>
<td>Description, Type</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>modelId</strong></td>
<td>The ID of the ML model stored in the Fusion blob store.</td>
</tr>
<tr>
<td>Machine Learning Model ID</td>
<td>type: string&lt;br&gt;blobType: model:ml-model&lt;br&gt;minLength: 1&lt;br&gt;reference: blob</td>
</tr>
<tr>
<td><strong>predictionFieldName</strong></td>
<td>Name of the field to store the prediction (model output) in the document.</td>
</tr>
<tr>
<td>Prediction Field Name</td>
<td>type: string&lt;br&gt;minLength: 1</td>
</tr>
<tr>
<td><strong>queryFeatureFieldName</strong></td>
<td>Name of the request parameter to extract query features from (model input).</td>
</tr>
<tr>
<td>Query Feature Field</td>
<td>type: string&lt;br&gt;default value: ‘q’&lt;br&gt;minLength: 1</td>
</tr>
<tr>
<td><strong>storeInContext</strong></td>
<td>Flag to indicate that the prediction should be set as a context property instead of setting a field on the document.</td>
</tr>
<tr>
<td>Store the Prediction in the Context</td>
<td>type: boolean&lt;br&gt;default value: ‘false’</td>
</tr>
</tbody>
</table>
Parameterized Boosting Stage

The Parameterized Boosting query pipeline stage reads the `boostValues` (in `List<DocumentResult>` format) from the context variable (added by a Rollup Aggregation stage or a JavaScript stage), and adds boosts to the main query using `bq` or `boost` based on the stage configuration. The weights for the boost values can also be scaled.

Configuration

Tip

When entering configuration values in the UI, use unescaped characters, such as `\t` for the tab character. When entering configuration values in the API, use escaped characters, such as `\\t` for the tab character.

When using Fusion’s REST-API, the ID of this stage is: `adv-boost`.

Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boostFieldName</td>
<td>The field name to boost the values on. type: <code>string</code></td>
</tr>
<tr>
<td>boostingMethod</td>
<td>The boost method to use. query-parser should be chosen if <code>defType!=edismax</code> for main query. type: <code>string</code> default value: ‘query-param’ enum: `{ query-param query-parser }</td>
</tr>
<tr>
<td>boostingParam</td>
<td>‘Boost’ multiplies scores by the boost values whereas ‘bq’ adds optional clauses to main query. type: <code>string</code> default value: ‘boost’ enum: `{ boost bq }</td>
</tr>
<tr>
<td>key</td>
<td>The key name to read from context for boost id and values. type: <code>string</code></td>
</tr>
</tbody>
</table>

26
<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>scaleRange</td>
<td>Scale Boosts</td>
</tr>
<tr>
<td></td>
<td>Scale the boost values to a ([\text{min},\text{max}]) range</td>
</tr>
<tr>
<td></td>
<td>type: object</td>
</tr>
<tr>
<td></td>
<td>object attributes: {</td>
</tr>
<tr>
<td></td>
<td>_scaleMax : {</td>
</tr>
<tr>
<td></td>
<td>display name: Maximum value of the scale range</td>
</tr>
<tr>
<td></td>
<td>type: number</td>
</tr>
<tr>
<td></td>
<td>_scaleMin : {</td>
</tr>
<tr>
<td></td>
<td>display name: Minimum value of the scale range</td>
</tr>
<tr>
<td></td>
<td>type: number</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
**Recommend Items for Item Stage**

The Recommend Items for Item query pipeline stage uses signals about users' item choices to recommend related items based on a specific item. Relationships between items can be based on different criteria, such as click patterns, people who bought this also bought that, percentage match of document tags, and so on.

Given an item ID, this stage performs a secondary query to the `items_for_item_recommendations` collection to find related items, then retrieves those items from the main collection.

This pipeline stage uses items-for-item recommendations that have been precomputed by an ALS Recommender.

See also Items-for-item Recommendations to learn how to configure this recommender type and fetch recommendations.

**Prerequisites**

**Enable recommendations:**

Before creating a Recommend Items for Item stage, enable recommendations.

- **In the Fusion UI** – With the Query Workbench open, click **Settings > Enable Recommendations**.
- **Using the REST API** – Use this command to enable recommendations:

  ```
  curl -u admin:<password> -X PUT \http://<hostname>:<port>/api/v1/collections/<collection-name>/features/recommendations -H 'content-type: application/json' -d '{"enabled":true}
  ```

**Note**

When you enable recommendations, Fusion creates a query pipeline that already contains this stage, and that is configured for boosting. The query pipeline is `<collection>_items_for_item_recommendations`.

**Configuration**

**Tip**

When entering configuration values in the UI, use **unescaped** characters, such as \t for the tab character.

When entering configuration values in the API, use **escaped** characters, such as \\\t for the tab character.

When using Fusion's REST-API, the ID of this stage is: `item-recommendation`.

**Configuration Properties**
<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boostFieldName</td>
<td>The field name to boost the values on.</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: 'id'</td>
</tr>
<tr>
<td>boostingMethod</td>
<td>The boost method to use. query-parser should be chosen if defType!=edismax for main query.</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘query-param’</td>
</tr>
<tr>
<td></td>
<td>enum: <code>{ query-param query-parser }</code></td>
</tr>
<tr>
<td>boostingParam</td>
<td>'Boost' multiplies scores by the boost values whereas 'bq' adds optional clauses to main query.</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘boost’</td>
</tr>
<tr>
<td></td>
<td>enum: <code>{ boost bq }</code></td>
</tr>
<tr>
<td>collection</td>
<td>If left blank, the default recommendation collection for the collection being queried will be used.</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td>foldInUpdates</td>
<td>Update recommendations based on user activity that has happened since the last recommendation job run</td>
</tr>
<tr>
<td></td>
<td>type: <code>boolean</code></td>
</tr>
<tr>
<td>itemIdField</td>
<td>the name of the field in the recommendation collection where user ID is stored</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘itemId’</td>
</tr>
<tr>
<td>itemIdParam</td>
<td>The name of the request parameter containing the item ID</td>
</tr>
<tr>
<td></td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘item_id’</td>
</tr>
<tr>
<td>Property</td>
<td>Description, Type</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>modelID</td>
<td>type: string</td>
</tr>
<tr>
<td></td>
<td>default value: &quot;&quot;</td>
</tr>
<tr>
<td>modelIdField</td>
<td>the name of the field in the recommendation collection where model ID is stored</td>
</tr>
<tr>
<td></td>
<td>type: string</td>
</tr>
<tr>
<td></td>
<td>default value: 'modelId'</td>
</tr>
<tr>
<td>numRecommendations</td>
<td>type: integer</td>
</tr>
<tr>
<td></td>
<td>default value: '10'</td>
</tr>
<tr>
<td>otherItemldField</td>
<td>the name of the field in the recommendation collection where similar item recommendation is stored</td>
</tr>
<tr>
<td></td>
<td>type: string</td>
</tr>
<tr>
<td></td>
<td>default value: 'otherItemld'</td>
</tr>
<tr>
<td>resultsLocation</td>
<td>If As Response is chosen, then the result of the RPC call will be the one and only response. In all other cases, the stage will put the response from the REST/RPC call into the target location using the resultsKey.</td>
</tr>
<tr>
<td></td>
<td>type: string</td>
</tr>
<tr>
<td></td>
<td>default value: 'As Boosts'</td>
</tr>
<tr>
<td></td>
<td>enum: { As Boosts As Response }</td>
</tr>
<tr>
<td>scaleRange</td>
<td>Scale the boost values to a [min,max] range</td>
</tr>
<tr>
<td></td>
<td>type: object</td>
</tr>
<tr>
<td></td>
<td>object attributes:</td>
</tr>
<tr>
<td></td>
<td>scaleMax: {</td>
</tr>
<tr>
<td></td>
<td>display name: Maximum value of the scale range</td>
</tr>
<tr>
<td></td>
<td>type: number</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>scaleMin: {</td>
</tr>
<tr>
<td></td>
<td>display name: Minimum value of the scale range</td>
</tr>
<tr>
<td></td>
<td>type: number</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description, Type</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| similarityScoreField | the name of the field in the recommendation collection where item similarity is stored  
                     | type: `string`  
                     | default value: ‘sim’ |
Recommend Items for User Stage

The Recommend Items for User query pipeline stage uses signals about item choices to recommend other similar items for a specific user. Personalization for the user can be based on the user's search history, browsing history, or purchase history, and so on.

This pipeline stage uses items-for-user recommendations that have been precomputed by an ALS Recommender job.

See also Items-for-user Recommendations to learn how to configure this recommender type and fetch recommendations.

Prerequisites

Enable recommendations:

Before creating a Recommend Items for Item stage, enable recommendations.

- In the Fusion UI – With Query Workbench open, click Settings > Enable Recommendations.
- Using the REST API – Use this command to enable recommendations:
  ```
  ```

Note

When you enable recommendations, Fusion creates a query pipeline that already contains this stage, and that is configured for boosting. The query pipeline is `<collection>_items_for_user_recommendations`.

Configuration

Tip

When entering configuration values in the UI, use unescaped characters, such as `\t` for the tab character. When entering configuration values in the API, use escaped characters, such as `\\t` for the tab character.

When using Fusion’s REST-API, the ID of this stage is: `user-recommendation`.

Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boostFieldName</td>
<td>The field name to boost the values on. type: <code>string</code> default value: <code>id</code></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>boostingMethod</td>
<td>The boost method to use. query-parser should be chosen if defType!-edismax for main query.</td>
</tr>
<tr>
<td>boostingParam</td>
<td>'Boost' multiplies scores by the boost values whereas 'bq' adds optional clauses to main query.</td>
</tr>
<tr>
<td>collection</td>
<td>If left blank, the default recommendation collection for the collection being queried will be used.</td>
</tr>
<tr>
<td>foldInUpdates</td>
<td>Update recommendations based on user activity that has happened since the last recommendation job run</td>
</tr>
<tr>
<td>itemIdField</td>
<td>the name of the field in the recommendation collection where item ID is stored</td>
</tr>
<tr>
<td>modelCollection</td>
<td>The name of the collection where models are stored. By default this is {app_name}_recommender_models</td>
</tr>
<tr>
<td>modelID</td>
<td>Model ID</td>
</tr>
<tr>
<td>Property</td>
<td>Description, Type</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>modelIdField</strong></td>
<td>the name of the field in the recommendation collection where model ID is stored</td>
</tr>
<tr>
<td>Model ID Field</td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘modelId’</td>
</tr>
<tr>
<td><strong>numRecommendations</strong></td>
<td>Number of Recommendations</td>
</tr>
<tr>
<td></td>
<td>type: <code>integer</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘10’</td>
</tr>
<tr>
<td><strong>rawSignalsCollection</strong></td>
<td>The collection to use to fetch recent user interactions, if 'Estimate Recent Results' is true.</td>
</tr>
<tr>
<td>Signals Collection</td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td><strong>resultsLocation</strong></td>
<td>If As Response is chosen, then the result of the RPC call will be the one and only response. In all other cases, the stage will put the response from the REST/RPC call into the target location using the resultsKey.</td>
</tr>
<tr>
<td>Results Location</td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘As Boosts’</td>
</tr>
<tr>
<td></td>
<td>enum: { As Boosts As Response }</td>
</tr>
<tr>
<td><strong>scaleRange</strong></td>
<td>Scale the boost values to a [min,max] range</td>
</tr>
<tr>
<td>Scale Boosts</td>
<td>type: <code>object</code></td>
</tr>
</tbody>
</table>
|                        | object attributes: \{
|                        | scaleMax : \{
|                        |   display name: Maximum value of the scale range
|                        |   type: `number`
|                        | \}
|                        | scaleMin : \{
|                        |   display name: Minimum value of the scale range
|                        |   type: `number`
|                        | \}
<p>|                        | }                                                                                                                                              |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description, Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>userIdField</td>
<td>the name of the field in the recommendation collection where user ID is stored</td>
</tr>
<tr>
<td>User ID Field</td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘userId’</td>
</tr>
<tr>
<td>userIdParam</td>
<td>The name of the request parameter containing the user ID</td>
</tr>
<tr>
<td>User ID Request Parameter</td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘user_id’</td>
</tr>
<tr>
<td>weightField</td>
<td>the name of the field in the recommendation collection where weight of the recommendation is stored</td>
</tr>
<tr>
<td>Weight Field</td>
<td>type: <code>string</code></td>
</tr>
<tr>
<td></td>
<td>default value: ‘weight’</td>
</tr>
</tbody>
</table>

Toggle navigation

[Fusion Platform](https://lucidworks.com/products/fusion-server/)

Highly scalable search engine and NoSQL datastore.

Learn more ›

[https://lucidworks.com/products/fusion-ai/](https://lucidworks.com/products/fusion-ai/)
Superior relevancy with machine learning, and artificial intelligence.

Learn more ›

https://lucidworks.com/products/fusion-app-studio/

Quickly create bespoke data applications for web and mobile.

Learn more › * AI-Powered Search

* Augmented Intelligence  Machine Learning  Clustering  Query Analysis

• Signals
• Faceted Search
• Indexing
• Hyper-Personalization
  ◦ Enterprise Solutions
• Digital Workplace
• Digital Commerce
• Search Engine Replacement
Products

Fusion AI 4.2

Overview

User Guide (PDF)

Reference Guides (PDF)

Release Notes
Jobs Configuration Reference

These reference topics provide complete information about configuration properties for the Spark jobs that are enabled with a Fusion AI license.

For conceptual information and instructions for configuring and scheduling jobs, see Jobs and Schedules.

Additional jobs are available as part of the basic Fusion Server feature set.

• ALS Recommender
  
  Use this job when you want to compute user recommendations or item similarities using a collaborative filtering recommender. You can also implement a user-to-item recommender in the advanced section of this job's configuration UI.

• Cluster Labeling
  
  Use this job when you already have clusters or well-defined document categories, and you want to discover and attach keywords to see representative words within those existing clusters. (If you want to create new clusters, use the Document Clustering job.)

• Collection Analysis
  
  Use this job when you want to compute basic metrics about your collection, like average word length, phrase percentages, and outlier documents (with very many or very few documents).

• Document Clustering
  
  Cluster a set of documents and attach cluster labels.

• Ground Truth
  
  Estimate ground truth queries using click signals and query signals, with document relevance per query determined using a click/skip formula.

• Head/Tail Analysis
  
  Perform head/tail analysis of queries from collections of raw or aggregated signals, to identify underperforming queries and the reasons. This information is valuable for improving overall conversions, Solr configurations, auto-suggest, product catalogs, and SEO/SEM strategies, in order to improve conversion rates.

• Legacy Item Recommender
  
  Compute user recommendations based on a pre-computed item similarity model.

• Legacy Item Similarity
  
  Use this job when you only want to compute item-to-item similarities. This method is more lightweight than the generic Recommendations job.

• Logistic Regression Classifier Training
  
  Train a regularized logistic regression model for text classification.
• Outlier Detection

Use this job when you want to find outliers from a set of documents and attach labels for each outlier group.

• Parallel Bulk Loader

The Parallel Bulk Loader (PBL) job enables bulk ingestion of structured and semi-structured data from big data systems, NoSQL databases, and common file formats like Parquet and Avro.

• Parameterized SQL Aggregation

A SQL aggregation job where user-defined parameters are injected into a built-in SQL template at runtime.

• Phrase Extraction

Identify multi-word phrases in signals.

• Query-to-Query Similarity

Train a collaborative filtering matrix decomposition recommender using SparkML's Alternating Least Squares (ALS) to batch-compute query-query similarities. This can be used for items-for-query recommendations as well as queries-for-query recommendations.

• Random Forest Classifier Training

Train a random forest classifier for text classification.

• Ranking Metrics

Calculate relevance metrics (nDCG and so on) by replaying ground truth queries against catalog data using variants from an experiment.

• SQL-Based Experiment Metric (deprecated)

This job is created by an experiment in order to calculate an objective.

<table>
<thead>
<tr>
<th>Note</th>
<th>This job is deprecated as of Fusion AI 4.0.2.</th>
</tr>
</thead>
</table>

• Synonym and Similar Queries Detection Jobs

Use this job to generate pairs of synonyms and pairs of similar queries. Two words are considered potential synonyms when they are used in a similar context in similar queries.

• Token and Phrase Spell Correction

Detect misspellings in queries or documents using the numbers of occurrences of words and phrases.

• Word2Vec Model Training

Train a shallow neural model, and project each document onto this vector embedding space.
ALS Recommender Jobs

Use this job when you want to compute user recommendations or item similarities using a collaborative filtering recommender. You can also implement a user-to-item recommender in the advanced section of this job’s configuration UI.

| Note | The `<collection>_user_item_preferences_aggregation` job provides input data for this job and must run before it. See SQL Aggregations for details. |

This job assumes that your signals collection contains the preferences of many users. It uses this collection of preferences to predict another user’s preference about an item that the user has not yet seen. A preference which can be viewed as a triple: user, item, and interaction-value.

When you enable recommendations for a collection, Fusion automatically creates an ALS Recommender job called `<collection>_item_recommendations`. This job generates both items-for-user recommendations and items-for-item recommendations, then stores the results in the `<collection>_items_for_user_recommendations` and `<collection>_items_for_item_recommendations` collections.
Basic job configuration

For items-for-item and items-for-user recommendations, the basic fields for configuring the `<collection>_item_recommendations` job are described below. To refine this job further, see Advanced job configuration.

• **numRecs/Number of User Recommendations to Compute**

  This is the number of recommendations that you want to return *per item* (for items-for-item recommendations) or *per user* (for items-for-user recommendations) in your dataset.

  Increasing this number up to 1000 will not cost too much computationally because the intensive work of computing the matrix decomposition (involving optimization) is already done by the time these recommendations are generated.

  Think of this as generating a matrix where the rows are the users and the columns are the recommendations. If we choose 1000 items to recommend, the size of the matrix will be \((\text{number of users}) \times (\text{number of items to recommend})\). For instance, if there are 10,000 users and 1000 recommendations, then the size of the matrix will be \(10,000 \times 1000\).

Input/output parameters

• **trainingCollection/Recommender Training Collection**

  Usually this should point to the `<collection>_signals_aggr` collection. If you are using another aggregated signals collection, verify that this field points to the correct collection name.

• **outputItemSimCollection/Item-to-item Similarity Collection**

  Usually this should point to the `<collection>_items_for_item_recommendations` collection. This collection will store the \(N\) most similar items for every item in the collection, where \(N\) is determined by the **numSims/Number of Item Similarities to Compute** field described below. Fusion can query this collection after the job to determine the most similar items to recommend based on an item choice.

  | Note | You can only specify a secondary collection of the collection with which this job is associated. For example, if you have a Movies collection and a Films collection and this job is associated with the Movies collection, then you cannot specify the Films__items_for_item_recommendations collection here. |

Model tuning parameters

• **numSims/Number of Item Similarities to Compute**

  This is similar to **numRecs/Number of User Recommendations to Compute** in the sense that this number of similar items are found for each item in the collection. Think of it as a matrix of size: \((\text{number of items}) \times (\text{number of item similarities to compute})\).

  This is not computationally expensive because it is just a similarity calculation (which involves no optimization). A reasonable value would be 30–250. It will also depend on the number of items displayed in your search application.
• **Implicit Preferences**

The concept of Implicit preferences is explained in Implicit vs explicit signals.

In this tutorial it is assumed that we submit no information about the items and the users (think of user and item features) but simply rely on the user-item interaction as a means to recommend similar products. That is the power of using implicit signals: we don’t need to know information about the user or the item, just how much they interact with each other.

If explicit ratings values are used (such as ratings from the user) then this box can be unchecked.

• **Delete Old Recommendations**

If you have reasons not to draw on old recommendations, then check this box. If this box is unchecked, then old recommendations will not be deleted but new recommendations will be appended with a different job ID. Both sets of recommendations will be contained within the same collection.
Advanced job configuration

You can achieve higher accuracy, and often reduce the training time too, by tuning the `<collection>_item_recommendations` job using the advanced configuration keys described here. In the job configuration panel, click Advanced to display these additional fields.

- **excludeFromDeleteFilter/Exclude from Delete Filter**

  If you have selected deleteOldRecs/Delete Old Recommendations but you do not want to completely delete all old recommendations, this field allows you to input a query that captures the data you want keep and removes the rest.

- **numUserRecsPerItem/Number of Users to Recommend to each Item**

  This setting indicates which users (from the known user group) are most likely to be interested in a particular item. The setting allows you to choose how many of the most interested users you would like to precompute and store.

  If one thinks of an estimated user-item matrix (after optimization), an item is a single column from the matrix, so if we wanted the top 100 users per item, we would sort the interest values in that column in descending order and take the top 100 row indices which would correspond to individual users.

- **maxTrainingIterations/Maximum Training Iterations**

  The Alternating Least Squares algorithm involves optimization to find the two matrices (user x latent factor and latent factor x item) that best approximate the original user-item matrix (formed from the signals aggregation).

  The optimization occurs at the matrix entry level (every non-zero element) and it is iterative. Therefore, the more iterations that are allowed during optimization, the lower the cost function value, meaning more accurate hyperparameters which lead to better recommendations.

  However, the bigger the data, the longer the job takes to run because the number of constraints to satisfy have increased. A value of 10 iterations usually leads to effective results. Above a value of 15, the job will begin to slow dramatically for above 25 million signals.

Training data settings

- **trainingDataFilterQuery/Training Data Filter Query**

  This query setting is useful when the main signals collection does not have the recommended fields. The two most important fields are doc_id and user_id because the job must have a user-item pairing. Note that depending on how the signals are collected the names doc_id and user_id can be different, but the concept remains the same.

  There are times when not all the signals have these fields. In this case we can add a query to select a subset of data that does have a user-item pairing. It is done with the following query:

  \[+\text{doc\_id}:[* \text{TO} *] +\text{user\_id}:[* \text{TO} *]\]

  This query returns all signals documents that have a user_id and doc_id field. Each query is separated by a space. The plus (+) sign is a positive request for the field of interest, meaning return signals with doc_id instead of signals without doc_id (negated or opposite queries are returned by prefixing with a negative (-) sign).
• **popularItemMin/Training Data Filter By Popular Items**

   The underlying assumption of this parameter is that the more users that view an item, the more popular that item is. Therefore, this value signifies the minimum number of interactions that must occur with the item for it to be considered a training data point.

   The higher the number, the smaller amount of data available for training because it is unlikely that many users interacted with all of the items. However, the quality of the data will be higher.

   One way to speed up training is to increase this number along with the training data sampling fraction. A reasonable number is between 10 and 20 depending on the application and user base. For instance, a song may be played much more than a movie and both may have more interaction than purchasing an item.

• **trainingSampleFraction/Training Data Sampling Fraction**

   This value is the percentage of the signal data or training data that you want to use for training the recommender job. It is advised to set this value to 1 and reduce the training data size (while increasing quality) by increasing the Training Data Filter By Popular Items as well as increasing the weight threshold in the Training Data Filter Query.

• **userIdField/Training Collection User Id Field**

   The ALS algorithm needs users, items, and a score of their interaction. The user ID field is the field name within the signal data that represents a user ID.

• **itemIdField/Training Collection Item Id Field**

   The item ID field is the field name within the aggregated signal data that represents the item or documents of interest.

• **weightField/Training Collection Weight Field**

   The weight field contains the score representing the interest of the user in an item.

• **initialBlocks/Training Block Size**

   In Spark, the training data is split amongst the executors in unchangeable blocks. This parameter sets the size of these blocks for training, but it requires advanced knowledge of Spark internals. We recommend leaving this setting at -1.

### Model settings

• **modelId/Recommender Model ID**

   The **Recommender Model ID** is assigned the field `modelId` in the `_items_for_item_recommendations` and `_items_for_user_recommendations` recommendations collections. This allows you to filter the recommendations by the recommender model ID. When the recommender job runs, a job ID is also assigned; therefore, you can see the results from different runs of the same job parameters. If you want to experiment with different parameters, it is advised to change the recommender model ID to reflect the parameters so that you can find the best parameters.

• **saveModel/Save Model in Solr**
Saving the model in Solr adds the parameters to the `_recommender_models` collection as a document. Using this method allows you to track all the recommender configurations.

- **modelCollection/Model Collection**

  This is the collection to store the experiment configurations (_recommender_models by default).

- **alwaysTrain/Force model re-training**

  When the job runs, it checks to see whether the model ID for the job already exists in the model collection. If the model does exist, it uses the pre-existing model to get the recommendations. Otherwise, if the box is checked it will re-run the recommender job and redo the optimization from scratch. Unless you need to maintain this ID name, it is advisable to create a separate model ID for each new combination of parameters.

**Grid search settings**

- **initialRank/Recommender Rank**

  The recommender rank is the number of latent factors into which to decompose the original user-item matrix. A reasonable range is 50–200. Above 200, the performance of the optimization can degrade dramatically depending on computing resources.

- **gridSearchWidth/Grid Search Width**

  Grid search is an automatic way to determine the best parameters for the recommender model. It tries different combinations of parameters of equally spaced units within a parameter domain and takes the model that has the lowest cost function value. This is a long process because a single run can take several hours depending on the computing resources, so trying multiple combinations can take some time. Depending on the size of your training data, it is better to do a manual grid search to reduce the number of runs needed to find a suitable recommender model.

- **initialAlpha/Implicit Preference Confidence**

  The implicit preference confidence is an approximation of how confident you are that the implicit data does indeed represent an accurate level of interest of a user in an item. Typical values are 1–100, with 100 being more confident in the training data representing the interest of the user. This parameter is used as a regularizer for optimization. The higher the confidence value, the more the optimization is penalized for a wrong approximation of the interest value.

- **initialLambda/Initial Lambda**

  Lambda is another optimization parameter that prevents overfitting. Remember we are decomposing the user-item matrix by estimating two matrices. The values in these matrices can be any number, large or small, and have a wide spread in the values themselves. To keep the scale of the value consistent or reduce the spread of the values, we use a regularizer. The higher the lambda, the smaller the values in the two estimated matrices. A smaller lambda gives the algorithm more freedom to estimate an answer which can result in overfitting. Typical values are between 0.01 and 0.3.

- **randomSeed/Random Seed**

  When the two matrices are first being estimated, their values are set randomly as an initialization. As the optimization proceeds the values are changed according to the error in the optimization. When training it is
important to keep the initialization the same in order to determine the effect of different values of parameters in the model. Keep this value the same across all experiments.

**Item metadata settings**

- **Item Metadata Collection**

  The main collection has very detailed information about each item, much of which is not necessary for training the recommender system. All that is important to train the recommender are the document IDs and the known users. If you have this metadata in a different collection than the main collection, enter that collection’s name here. Once the training is complete, the document ID of the relevant documents can be used to retrieve detailed information from the item catalog. The point is to train on small data per item and retrieve the detailed information for only relevant documents.

- **Item Metadata Join Field**

  This is the field that is common to the aggregated signal data and the original data. It is used to join each document from the recommender collection to the original item in the main collection. Usually this is the id field.

- **Item Metadata Fields**

  These are fields from the main collection that should be returned with each recommendation. You can add fields here by clicking the Add icon. To ensure that this works correctly, verify that Item Metadata Join Field has the correct value.
Configuration Properties
Cluster Labeling Jobs

Use this job when you already have clusters or well-defined document categories, and you want to discover and attach keywords to see representative words within those existing clusters. (If you want to create new clusters, use the Document Clustering job.)

Configuration Properties
Co-occurrence Similarity Jobs

Use this job when you only want to compute item-to-item similarities. This method is more lightweight than the generic Recommendations job.
Collection Analysis Jobs

Use this job when you want to compute basic metrics about your collection, like average word length, phrase percentages, and outlier documents (with very many or very few documents).

Configuration Properties
Document Clustering Jobs

Cluster a set of documents and attach cluster labels.

Configuration Properties
Ground Truth Jobs

Estimate ground truth queries using click signals and query signals, with document relevance per query determined using a click/skip formula.

| Tip | Pair this job with the Ranking Metrics job to calculate relevance metrics, such as nDCG. |

Configuration Properties
Head/Tail Analysis Jobs

Perform head/tail analysis of queries from collections of raw or aggregated signals, to identify underperforming queries and the reasons. This information is valuable for improving overall conversions, Solr configurations, auto-suggest, product catalogs, and SEO/SEM strategies, in order to improve conversion rates.

| Note | A minimum of 10,000 signals is required to successfully run this job. |

You can review the output from this job using the Query Rewriting UI.

Head/tail analysis configuration

The job configuration must specify the following:

- The signals collection (the Input Collection parameter)
  Signals can be raw (the _signals collection) or aggregated (the _signals_aggr collection).

- The query string field (the Query Field Name parameter)

- The event count field

  For example, if signal data follows the default Fusion setup, then count_i is the field that records the count of raw signals and aggr_count_i is the field that records the count after aggregation.

The job allows you to analyze query performance based on two different events:

- The main event (the mainType/Main Event Type parameter)

- The filtering/secondary event (the filterType/Filtering Event Type parameter)

  If you only have one event type, leave this parameter empty.

For example, if you specify the main event to be clicks with minimum count of 0 and the filtering event to be queries with minimum count of 20, then the job will filter on the queries that get searched at least 20 times and check among those popular searched queries to see which ones didn’t get clicked at all or only a few times.

An example configuration is shown below:
Head-n-Tail Analysis
Perform head tail analysis of queries from raw or aggregated signals collection.

* Spark Job ID
  ecommerce-head-tail

* Input Collection
  ecommerce_signals

* Query Field Name
  query_s

Signals data filter query
  type_s:click OR type_s:query

* Event Count Field Name
  count_i

* Main Event Type
  click

Filtering Event Type
  query

Minimum Main Event Count
  1

Minimum Filtering Event Count
  20

Minimum Query Length
  2

Keywords blob name
  ecommerce_keyword.csv
The suggested schedule for this head-n-tail analysis job is to run bi-weekly or monthly. You can change schedule under the run panel.

Job output

By default, the output collection is the `<input-collection>_job_reports` collection. The head/tail job adds a set of analytics results tables to the collection. You can find these table names in the `doc_type_s` field of each document:

- `overall_distribution`
- `summary_stat`
- `queries_ordered`
- `tokens_ordered`
- `queryLength`
- `tail_reasons`
- `tail_rewriting`

You can use App Insights to visualize each of these tables:

1. In the Fusion workspace, navigate to Analytics > App Insights.
   
   The App Insights dashboard appears.

2. On the left, click Analytics 📊.

3. Under Standard Reports, click Head Tail analysis.

   The Head/Tail Analysis job output tables appear. These are described in more detail below.

Head/Tail Plot (`overall_distribution`)

This head/tail distribution plot provides an overview of the query traffic distribution. In order to provide better visualization, the unique queries are in descending order based on traffic and put into bins of 100 queries on the x axis, with the sum of traffic coming from each bin on the y axis.

For example, the head/tail distribution plot below shows a long tail, indicating that the majority of queries produce very little traffic. The goal of analyzing this data is to shorten that tail, so that a higher proportion of your queries produce traffic.
• Green = head
• Yellow = torso
• Red = tail

**Summary Stats (summary_stat)**

This user-configurable summary statistics table shows how much traffic is produced by various query groups, to help understand the head/tail distribution.
You can configure this table before running the job. Click Advanced in the Head/Tail Analysis job configuration panel, then tune these parameters:

- Top X% Head Query Event Count (topQ)
- Number of Queries that Constitute X% of Total Events (trafficPerc)
- Bottom X% Tail Query Event Count (lastTraffic)
- Event Count Computation Threshold (trafficCount)

**Query Details (queries_ordered)**

The Query Details table helps you discover which queries are the best performers and which are worst. You can filter results by issuing a search in the search bar. For example, search "segment_s:tail" to get tail queries or search "num_events_l:0" to get zero results queries. (Note: field names are listed in the "what is this" toolkit).
Top Tokens (tokens_ordered)

The "Top Tokens" table lists the number of times each token shown in the queries.

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv</td>
<td>675</td>
</tr>
<tr>
<td>the</td>
<td>613</td>
</tr>
<tr>
<td>samsung</td>
<td>609</td>
</tr>
<tr>
<td>sony</td>
<td>543</td>
</tr>
<tr>
<td>case</td>
<td>514</td>
</tr>
<tr>
<td>laptop</td>
<td>476</td>
</tr>
<tr>
<td>iphone</td>
<td>442</td>
</tr>
<tr>
<td>dvd</td>
<td>396</td>
</tr>
<tr>
<td>ipod</td>
<td>395</td>
</tr>
<tr>
<td>hp</td>
<td>376</td>
</tr>
<tr>
<td>car</td>
<td>359</td>
</tr>
<tr>
<td>wireless</td>
<td>356</td>
</tr>
</tbody>
</table>

Query Length (queryLength)

This table shows how users are querying your database. Are most people searching very long strings or very short strings? These distributions will give you insight into how to tune your search engine to be performant on the majority of queries.

<table>
<thead>
<tr>
<th>Avg String Length</th>
<th>One Word Perc</th>
<th>Two Words Perc</th>
<th>Three Words Perc</th>
<th>Four Words Perc</th>
<th>Five Words Perc</th>
<th>Six+ Words Perc</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>17.89</td>
<td>40.06</td>
<td>27.01</td>
<td>10.07</td>
<td>3.36</td>
<td>1.61</td>
</tr>
</tbody>
</table>
Tail Reasons table and pie chart (*tail_reasons*)

Based on the difference between the tail and head queries, the Head/Tail Analysis job assigns probable reasons for why any given query is a tail query. Tail reasons are displayed as both a table and a pie chart:

<table>
<thead>
<tr>
<th>Count</th>
<th>Tail Reasons</th>
<th>Proportions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1249</td>
<td>number</td>
<td>6.61%</td>
</tr>
<tr>
<td>1129</td>
<td>other-specific</td>
<td>5.98%</td>
</tr>
<tr>
<td>930</td>
<td>brand</td>
<td>4.92%</td>
</tr>
<tr>
<td>565</td>
<td>spelling</td>
<td>2.99%</td>
</tr>
<tr>
<td>205</td>
<td>other-extra</td>
<td>1.08%</td>
</tr>
<tr>
<td>156</td>
<td>color</td>
<td>0.82%</td>
</tr>
<tr>
<td>83</td>
<td>stopwords</td>
<td>0.43%</td>
</tr>
<tr>
<td>75</td>
<td>re-wording</td>
<td>0.39%</td>
</tr>
<tr>
<td>68</td>
<td>rare-term</td>
<td>0.36%</td>
</tr>
<tr>
<td>42</td>
<td>others</td>
<td>0.22%</td>
</tr>
</tbody>
</table>

### Pre-defined tail reasons

Based on Lucidworks' observations on different signal datasets, we summarize tail reasons into several pre-defined categories:

<p>| spelling | The query contains one or more misspellings; we can apply spelling suggestions based on the matching head. |</p>
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>The query contains an attribute search on a specific dimension. To normalize these queries we can parse the number to deal with different formatting, and/or pay attention to unit synonyms or enrich the product catalog. For example, &quot;3x5&quot; should be converted to &quot;3' X 5'&quot; to match the dimension field.</td>
</tr>
<tr>
<td>other-specific</td>
<td>The query contains specific descriptive words plus a head query, which means the user is searching for a very specific product or has a specific requirement. We can boost on the specific part for better relevancy.</td>
</tr>
<tr>
<td>other-extra</td>
<td>This is similar to ‘other-specific’ but the descriptive part may lead to ambiguity, so it requires boosting the head query portion of the query instead of the specific or descriptive words.</td>
</tr>
<tr>
<td>rare-term</td>
<td>The user is searching for a rare item; use caution when boosting.</td>
</tr>
<tr>
<td>re-wording</td>
<td>The query contains a sequence of terms in a less-common order. Flipping the word order to a more common one can change a tail query to a head query, and allows for consistent boosting on the last term in many cases.</td>
</tr>
<tr>
<td>stopwords</td>
<td>Query contains stopwords plus head query. We would need to drop stopwords.</td>
</tr>
</tbody>
</table>

**Custom dictionary**

You can also specify your own attributes through a keywords file in CSV format. The header of the CSV file must be called "keyword" and "type", and stopwords must be called "stopword" for the program to recognize them.

Below is an example dictionary that defines "color" and "brand" reason types. The job will parse the tail query, assign reasons such as "color" or "brand", and perform filtering or focused search on these fields. (Note: color and brand are also the field names in your catalog.)

<table>
<thead>
<tr>
<th>keyword, type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a,stopword</td>
<td>a,stopword</td>
</tr>
<tr>
<td>an,stopword</td>
<td>an,stopword</td>
</tr>
<tr>
<td>and,stopword</td>
<td>and,stopword</td>
</tr>
<tr>
<td>blue,color</td>
<td>blue,color</td>
</tr>
<tr>
<td>white,color</td>
<td>white,color</td>
</tr>
<tr>
<td>black,color</td>
<td>black,color</td>
</tr>
<tr>
<td>hp,brand</td>
<td>hp,brand</td>
</tr>
<tr>
<td>samsung,brand</td>
<td>samsung,brand</td>
</tr>
<tr>
<td>sony,brand</td>
<td>sony,brand</td>
</tr>
</tbody>
</table>
How to install a custom dictionary

1. Construct the CSV file as described above.
2. Upload the CSV file to the blob store.
   Note the blob ID.
3. In the Head/Tail Analysis job configuration, enter the blob ID in the **Keywords blob name** (keywordsBlobName) field.

**Head Tail Similarity (tail_rewriting)**

For each tail query (the **tailQuery_orig** field), Fusion tries to find its closest matching head queries (the **headQuery_orig** field), then suggests a query rewrite (the **suggested_query** field) which would improve the query. The rewrite suggestions in this table can be implemented in a variety of ways, including utilizing rules editor or configuring a query parser that rewrites tail queries.

<table>
<thead>
<tr>
<th>Tail Query</th>
<th>Suggested Rewriting</th>
<th>Matched Head</th>
<th>Reason Code</th>
<th>Matched Head (No Num)</th>
<th>Number</th>
<th>Other Specific</th>
<th>Number Unit</th>
<th>Tail Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>car amplifier</td>
<td>amplifier (car)*1.5</td>
<td>amplifier</td>
<td>other-specific</td>
<td>amplifier</td>
<td>car</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>headphone bluetooth</td>
<td>bluetooth headphone*2</td>
<td>bluetooth headphone</td>
<td>re-wording</td>
<td>bluetooth headphone</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>subwoofer</td>
<td>subwoofer</td>
<td>subwoofer</td>
<td>spelling</td>
<td>subwoofer</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>lord of the ring blu ray</td>
<td>lord of the ring*2 blu ray</td>
<td>lord of the ring</td>
<td>other-extra</td>
<td>lord of the ring</td>
<td>blu ray</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>wireless n router</td>
<td>wireless router</td>
<td>wireless router</td>
<td>stopwords</td>
<td>wireless router</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>mw2</td>
<td>mw 2 mw2</td>
<td>mw</td>
<td>number</td>
<td>mw</td>
<td>2</td>
<td>mw</td>
<td>mw2</td>
<td>5</td>
</tr>
<tr>
<td>hosa cable</td>
<td>cable hosa</td>
<td>cable</td>
<td>rare-term</td>
<td>cable</td>
<td>hosa</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>the lord of the ring blu ray</td>
<td>lord of the ring*2 the blu ray</td>
<td>lord of the ring</td>
<td>other-extra</td>
<td>lord of the ring</td>
<td>the blu ray</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**Configuration Properties**
Item Similarity Recommender Jobs

Use this job when you want to compute user recommendations based on pre-computed item similarities.
Levenshtein Spell Checking Jobs

Compute the edit distance between all values in a field.

| Note | This job is deprecated as of Fusion AI 4.1.0. Use the Token and Phrase Spell Correction job instead. |
Logistic Regression Classifier Training Jobs

Train a regularized logistic regression model for text classification.

Configuration Properties
Matrix Decomposition-Based Query-Query Similarity Jobs

Train a collaborative filtering matrix decomposition recommender using SparkML's Alternating Least Squares (ALS) to batch-compute query-query similarities.
Outlier Detection Jobs

Use this job when you want to find outliers from a set of documents and attach labels for each outlier group.

Configuration Properties
Parameterized SQL Aggregation Jobs

A SQL aggregation job where user-defined parameters are injected into a built-in SQL template at runtime.

Configuration Properties
Random Forest Classifier Training Jobs

Train a random forest classifier for text classification.

Configuration Properties
Ranking Metrics Jobs

Calculate relevance metrics (nDCG and so on) by replaying ground truth queries against catalog data using variants from an experiment.

Configuration Properties
SQL-Based Experiment Metric Jobs

This job is created by an experiment in order to calculate an objective.

<table>
<thead>
<tr>
<th>Note</th>
<th>This job is deprecated as of Fusion AI 4.0.2.</th>
</tr>
</thead>
</table>

Configuration Properties
Statistically Interesting Phrases Jobs

Use this job when you want to identify phrases in your content.
Token and Phrase Spell Correction Job

Detect misspellings in queries or documents using the numbers of occurrences of words and phrases.

This job extracts tail tokens (one word) and phrases (two words) and finds similarly-spelled head tokens and phrases. For example, if two queries are spelled similarly, but one leads to a lot of traffic (head) and the other leads to a little or zero traffic (tail), then it's likely that the tail query is misspelled and the head query is its correction.

If several matching head tokens are found for each tail token, the job can pick the best correction using multiple configurable criteria.

For additional background, see the blog post Advanced Spell Check with Fusion 4.

You can review, edit, deploy, or delete output from this job using the Query Rewriting UI.

<table>
<thead>
<tr>
<th>Tip</th>
<th>Misspelled terms are completely replaced by their corrected terms. To instead expand the query to include all alternative terms, see the Synonym Detection feature and set your synonyms to be bi-directional.</th>
</tr>
</thead>
</table>

This job's output, and output from the Phrase Extraction job, can be used as input for the Synonym and Similar Queries Detection job.

<table>
<thead>
<tr>
<th>Tip</th>
<th>Solr treats spelling corrections as synonyms. See the blog post Multi-Word Synonyms: Solr Adds Query-Time Support for more details.</th>
</tr>
</thead>
</table>

1. Create a job

Create a Token and Phrase Spell Correction job in the Jobs Manager.

To create a new job

1. In the Fusion workspace, navigate to Jobs.
2. Click Add and select the job type Token and phrase spell correction.
   The New Job Configuration panel appears.
3. Configure the new job as needed. Configuration is explained in the next section.

2. Configure the job

Configure the Token and Phrase Spell Correction job.

Required configuration

The configuration must specify:

- The input collection (the Input Collection/trainingCollection parameter)

   The input collection can contain signal data or non-signal data. If it is signal data, then select Input is Signal Data
(signalDataIndicator). Signals can be raw (from the _signals collection) or aggregated (from the _signals_aggr collection).

- The input field (the **Input Field**/fieldToVectorize parameter)
- The event count field

For example, if signal data follows the default Fusion setup, then count_i is the field that records the count of raw signals and aggr_count_i is the field that records the count after aggregation.

**Event types**

The spell correction job lets you analyze query performance based on two different events:

- The main event (the **Main Event Type**/mainType parameter)
- The filtering/secondary event (the **Filtering Event Type**/filterType parameter)

If you only have one event type, leave this parameter empty.

For example, if you specify the main event type to be **click** with a minimum count of 0 and the filtering event type to be **query** with a minimum count of 20, then the job will filter on the queries that get searched at least 20 times and check among those popular queries to see which ones didn't get clicked at all, or were only clicked a few times.

**Spell check documents**

If you unselect the **Input is Signal Data** checkbox to indicate finding misspellings from content documents rather than signals, then you don’t need to specify the following parameters: Count Field, Main Event Field, Filtering Event Type, Field Name of Signal Type, Minimum Main Event Count and Minimum Filtering Event Count.

**Use a custom dictionary**

You can upload a custom dictionary of terms that are specific to your data, and specify it using the **Dictionary Collection** (dictionaryCollection) and **Dictionary Field** (dictionaryField) parameters. For example, in an e-commerce use case, you can use the catalog terms as the custom dictionary by specifying the product catalog collection as the dictionary collection and the product description field as the dictionary field.

**Example configuration**

This is an example configuration:
Token and phrase spell correction
Use this job to compute token and phrase level spell correction which you can use in your synonym list.

Advanced

* Spark Job ID
  spell-correction
  The ID for the Spark job. Used in the API to reference the job. Allowed characters: a-z, A-Z, dash (-), and underscore (_)

** INPUT/OUTPUT PARAMETERS **

* Input Collection
  E-commerce_signals
  Collection containing search strings and event counts. Should ideally be the signals collection if an aggregation call...

Output Collection
  E-commerce_signals_aggr
  Collection to store mispelling and correction pairs. Defaults to the agg collection.

Data filter query
type:click OR type:search
Sort query to additionally filter the search strings. Please let it be empty if type field is unavailable in the input call...

Input is Signal Data [ ]

** FIELD PARAMETERS **

* Input Field
  query
  Field containing search strings.

Main Event Type
click
The main signal event type (e.g. click) that the job is based on if input is signal data. E.g., if main type is click, then...

Filtering Event Type
  search
  The secondary event type (e.g. impression) that can be used for filtering out rare searches. Note: In order to use this ‘...

Count Field
count
Sort field containing query count

** MODEL TUNING PARAMETERS **

Minimum Main Event Count
  1
  Minimum number of main events (e.g., clicks) after aggregation necessary for the query to be considered. The job wi...

Minimum Filtering Event Count
  10
  Minimum number of filtering events (e.g., searches) after aggregation necessary for the query to be considered. The...

Maximum Edit Distance
  2
  The maximum number of matches on starting characters. Note: Setting it to 0 may largely increase running time.

** FEATURIZATION PARAMETERS **

Lucene Analyzer Schema for Processing Queries
  

** MISC. PARAMETERS **

Dictionary Collection
  E-commerce-catalog
  Sort Collection containing dictionary with correct spellings. E.g., product catalog.

Dictionary Field
  product-description
  Sort field containing dictionary text. Multiple fields can be specified using the format field:field;

Lucene Analyzer Schema for Processing Dictionary
When you have configured the job, click **Save** to save the configuration.

## 3. Run the job

Run the Token and Phrase Spell Correction job.

<table>
<thead>
<tr>
<th>Tip</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you are finding spelling corrections in aggregated data, you need to run an aggregation job before running the Token and Phrase Spell Correction job. You <em>don't</em> need to run a Head/Tail Analysis job. The Token and Phrase Spell Correction job does the head/tail processing it requires.</td>
</tr>
</tbody>
</table>

To run the job

1. In the Fusion workspace, navigate to **Jobs**.
2. Select the job from the job list.
3. Click **Run**.
4. Click **Start**.

## 4. Analyze job output

After the job finishes, misspellings and corrections are output into the `query_rewrite_staging` collection by default; you can change this by setting the `outputCollection`.

An example record is as follows:

```plaintext
correction_s          laptop battery
mis_string_len_i       14
misspelling_s          laptop baytery
aggr_job_id_s          162cf94b20T3704c333
score                   1
collation_check_s      token correction included
corCount_misCount_ratio_d  2095
sound_match_b          true
id                     bf79c43b-fc6d-43a7-931e-185fdac5b624
aggr_type_s            tokenPhraseSpellCorrection
aggr_id_s              ecom_spell_check
correction_types_s     phrase => phrase
cor_count_i            68648960
suggested_correction_s baytery=>battery
cor_string_len_i       14
token_wise_correction_s baytery=>battery
cor_token_size_i       2
edit_dist_i             1
mis_count_i             32768
lastChar_match_b        true
mis_token_size_i        2
token_corr_for_phrase_cnt_i  1
```

For easy evaluation, you can export the result output to a CSV file.
5. Use spell correction results

You can use the resulting corrections in various ways. For example:

- Put misspellings into the synonym list to perform auto-correction.
- Help evaluate and guide the Solr spellcheck configuration.
- Put misspellings into typeahead or autosuggest lists.
- Perform document cleansing (for example, clean a product catalog or medical records) by mapping misspellings to corrections.

**Useful output fields**

In the job output, you generally only need to analyze the `suggested_corrections` field, which provides suggestions about using token correction or whole-phrase correction. If the confidence of the correction is not high, then the job labels the pair as "review" in this field. Pay special attention to the output records with the "review" labels.

With the output in a CSV file, you can sort by `mis_string_len` (descending) and `edit_dist` (ascending) to position more probable corrections at the top. You can also sort by the ratio of correction traffic over misspelling traffic (the `corCount_misCount_ratio` field) to only keep high-traffic boosting corrections.

For phrase misspellings, the misspelled tokens are separated out and put in the `token_wise_correction` field. If the associated token correction is already included in the one-word correction list, then the `collation_check` field is labeled as "token correction include." You can choose to drop those phrase misspellings to reduce duplications.

Fusion counts how many phrase corrections can be solved by the same token correction and puts the number into the `token_corr_for_phrase_cnt` field. For example, if both "outdoor servailance" and "servailance camera" can be solved by correcting "servailance" to "surveillance", then this number is 2, which provides some confidence for dropping such phrase corrections and further confirms that correcting "servailance" to "surveillance" is legitimate.

You might also see cases where the token-wise correction is not included in the list. For example, "xbow" to "xbox" is not
included in the list because it can be dangerous to allow an edit distance of 1 in a word of length 4. But if multiple phrase corrections can be made by changing this token, then you can add this token correction to the list.

| Tip | Phrase corrections with a value of 1 for token_corr_for_phrase_cnt and with collation_check labeled as "token correction not included" could be potentially-problematic corrections. |

Fusion labels misspellings due to misplaced whitespaces with "combine/break words" in the correction_types field. If there is a user-provided dictionary to check against, and both spellings are in the dictionary with and without whitespace in the middle, we can treat these pairs as bi-directional synonyms ("combine/break words (bi-direction)" in the correction_types field).

The sound_match and lastChar_match fields also provide useful information.

**Job tuning**

The job’s default configuration is a conservative, designed for higher accuracy and lower output. To produce a higher volume of output, you can consider giving more permissive values to the parameters below. Likewise, give them more restrictive values if you are getting too many results with low accuracy.

| Note | When tuning these values, always test the new configuration in a non-production environment before deploying it in production. |

- **trainingDataFilterQuery/Data filter query**
  - See Event types above, then adjust this value to reflect the secondary event for your search application. To query all data, set this to :.

- **minCountFilter/Minimum Filtering Event Count**
  - Lower this value to include less-frequent misspellings based on the data filter query.

- **maxDistance/Maximum Edit Distance**
  - Raise this value to increase the number of potentially-related tokens and phrases detected.

- **minMispellingLen/Minimum Length of Misspelling**
  - Lower this value to include shorter misspellings (which are harder to correct accurately).

**Configuration Properties**
Word2Vec Model Training Jobs

Train a shallow neural model, and project each document onto this vector embedding space.

Configuration Properties
REST API Reference

Fusion API services are designed to be accessed via Fusion’s authentication proxy module which is part of the Fusion UI service (default port 8764). All applications should use this method to access the API service. However, for development purposes it is possible to access the API service directly:

<table>
<thead>
<tr>
<th>Development</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;code&gt;&lt;a href=&quot;http://&amp;lt;fusion-host&amp;gt;:8764/api/&amp;lt;endpoint&amp;gt&quot; class=&quot;bare&quot;&gt;http://&amp;lt;fusion-host&amp;gt;:8764/api/&amp;lt;endpoint&amp;gt&lt;/a&gt;;&lt;/code&gt;</code></td>
<td><code>&lt;code&gt;&lt;a href=&quot;http://&amp;lt;fusion-host&amp;gt;:8764/api/&amp;lt;endpoint&amp;gt&quot; class=&quot;bare&quot;&gt;http://&amp;lt;fusion-host&amp;gt;:8764/api/&amp;lt;endpoint&amp;gt&lt;/a&gt;;&lt;/code&gt;</code></td>
</tr>
</tbody>
</table>

Direct access to the API service, bypassing the authentication proxy. *Access to this port should be restricted.*

Access via the authentication proxy, using either a username/password pair or a session cookie with the user ID.
Listing all Fusion component services

The Fusion introspect endpoint lists basic information about endpoints and parameters for all Fusion endpoints, including the Connectors services endpoints:

```
curl -u user:pass http://localhost:8764/api/introspect
```
REST API Reference Pages

- Experiments API
- Recommendations API (Deprecated)
- Signals API :leveloffset: +1
Experiments API

Use the Experiments API to compare different configuration variants and determine which ones are most successful. For example, configure different variants that use different query pipelines, and then analyze and compare search activity to see which variant best meets your goals.

Experiments let you evaluate multiple variants which can differ from each other by pipeline, collection, search handler, request parameters, or some combination of those. An experiment uses one or more metrics to measure the performance of each variant, so they can be compared quantitatively.

Experiments are also available through the Experiments UI in the Fusion UI.

| Note | This API requires a Fusion AI license. |
Experiment types

The Experiments API supports A/B (or A/B/n) tests.

A/B testing

The Experiments API in Fusion lets you set up straightforward A/B (or A/B/n) tests. An A/B test is a two-sample hypothesis test with two variants. In A/B/n testing, variant A (the first variant that is defined) is compared with B, and with each of the other variants. For example, with an A/B/C/D test, the comparisons are A/B, A/C, and A/D.

The simplest setup is to create just two variants (A and B). Variant A is customarily considered the “control” configuration, and variant B is the test configuration or “treatment” to compare with the control configuration.

It is also common to perform A/A testing, that is, to create a separate variant with exactly the same configuration as the control. A/A testing is useful for detecting any systemic errors.

You can perform both A/B and A/A testing as a part of the same experiment. To do this, create three variants: one variant that is the control (A1), a second variant with the same configuration as the control (A2), and a third variant with the treatment configuration (B). The comparisons are A1/A2 and A1/B.

For example, the following experiment definition creates an experiment with 3 variants. The first two are identical, while the third uses a different query pipeline, called with-recommendations:

```json
{
   "id": "sample-experiment",
   "uniqueIdParameter": "userId",
   "baseSignalsCollection": "bestbuy",
   "variants": [
      {
         "name": "control-a1",
      },
      {
         "name": "control-a2",
      },
      {
         "name": "b",
         "queryPipeline": "with-recommendations",
      }
   ],
   "enabled": true,
   "metrics": [
      {
         "type": "ctr",
         "name": "CTR",
         "primary": true
      },
      {
         "type": "conversion-rate",
         "name": "purchase rate",
         "signalType": "purchase"
      }
   ]
}
```
As users interact with this experiment, we expect the results of metrics for the first two variants to be quite similar, since they are configured identically. With a small sample size, metrics could vary somewhat based on random chance but, as more users interact with the experiment, we expect the metrics to converge on identical results. The third variant, however, will likely perform differently, and its performance will tell us whether we should be using the with-recommendations pipeline for all search traffic.
Recommendations API

The Recommendations REST API uses signals and aggregated signals to return a list of items that can be used for recommendations.

| Note | This API requires a Fusion AI license. |

To use the REST API Recommendations service to get recommendations for items in some collection, that collection must have associated signals and aggregated-signals system collections. How good the recommendations are depends on how well the information in the signals and aggregated signals collections, which is derived from observed user behavior, matches user behavior going forward.

In addition to these endpoints, is also possible to get recommendations as part of a query request.

See Recommendations and Boosting for a discussion of when to use the API and when to use recommender query stages.

Recommendation types

The API includes separate endpoints for retrieving different types of recommendations:

| itemsForQuery | Get items for a defined query. The response is a list of document IDs and their weights. |
| queriesForItem | Get queries for a defined item (a document ID). This finds the top queries that led users to the defined item. The response is a list of query terms and their weights. |

Output

The output includes the following sections:

| header | The query parameters (in a section named queryParams) and the total time it took to process the query. |
## Examples

Below are examples for each recommendation type.

### itemsForQuery

Get the top items for the query 'ipod':

**INPUT**

```
curl -u user:pass
```

**OUTPUT**
queriesForItem

INPUT

```
curl -u user:pass http://localhost:8764/api/recommend/lucidworks102/queriesForItem?docId=9225439
```

OUTPUT
"header" : {
  "queryParams" : {
    "aggrType" : "*",
    "rows" : 10,
    "collection" : "lucidworks102",
    "docId" : "9225439"
  },
  "totalTime" : 8
},
"items" : [
  {
    "query" : "ipod",
    "weight" : 3.865899E-12
  },
  {
    "query" : "columbusday ipod mp3 20111009",
    "weight" : 3.5141304E-12
  },
  {
    "query" : "apple itouch",
    "weight" : 2.3619889E-12
  },
  {
    "query" : "ipod 4th generation",
    "weight" : 1.6436526E-12
  },
  {
    "query" : "ipod touch 4th generation",
    "weight" : 9.674966E-13
  },
  {
    "query" : "onlinemidnightsale ipod mp3players",
    "weight" : 9.568035E-13
  },
  {
    "query" : "ipod touch",
    "weight" : 7.774231E-13
  },
  {
    "query" : "itouch",
    "weight" : 7.707221E-13
  }
]
Signals API

The Signals API accepts a set of signals, encoded as JSON objects, for indexing into a signals collection.

<table>
<thead>
<tr>
<th>Note</th>
<th>This API requires a Fusion AI license.</th>
</tr>
</thead>
</table>

Normally, signals are indexed just like ordinary documents, through a configured datasource and index pipeline. This API is provided for cases where it is more convenient to index signals directly.

To aggregate signals, use a SQL aggregation job, which is a kind of Spark job.

<table>
<thead>
<tr>
<th>Tip</th>
<th>You can manage aggregation jobs in the Jobs manager and Scheduler in the Admin UI, or with the Spark jobs API. We recommend using the Admin UI.</th>
</tr>
</thead>
</table>

See Signal types and structure to learn how to send well-formed signals to this API.

Examples

Send two signal events to record user clicks:

REQUEST

```bash
curl \
-u user:pass -X POST -H 'Content-type:application/json' -d @- \nhttp://localhost:8764/api/signals/docs?commit=true \
<<EOF
[
  {"params": {
    "query": "Televisiones Panasonic 50 pulgadas",
    "filterQueries": ["cat00000","abcat0100000","abcat0101000","abcat0101001"],
    "doc_id_s": "2125233" },
  "type":"click",
  "timestamp": "2011-09-01T23:44:52.53Z"
},
  {"params": {
    "query": "Sharp",
    "filterQueries": ["cat00000","abcat0100000","abcat0101000","abcat0101001"],
    "doc_id_s": "2009324" },
  "type":"click",
  "timestamp": "2011-09-05T12:25:37.42Z"
  }
]
EOF
```

A successful request results in events being added to the signals collection. For the above example, the events will be represented as follows:

```json
{
  "responseHeader":{
    "status":0,
```