# Table of Contents

Signals and Aggregations .......................................................... 1
- Signals ........................................................................ 2
- Aggregations .................................................................. 3
- The cold start problem ....................................................... 4
Signals .................................................................................. 5
- Enabling and disabling signals ............................................. 6
  - Using the UI .................................................................. 6
  - Using the Collection Features API .................................... 7
- Signals data flow ................................................................. 9
Signals types and structure ...................................................... 11
- Implicit signals vs explicit signals ........................................ 11
- Built-in signal types .......................................................... 11
  - Request signals ............................................................ 11
  - Response signals ......................................................... 12
  - Click signals .................................................................. 14
- Custom signal types ........................................................... 15
- Important fields for signals ................................................ 16
  - Signal field count analysis ............................................ 16
  - The query_id field ........................................................ 17
- Signal type ranking ............................................................ 18
- The default signals index pipeline ....................................... 20
Dealing old signals ................................................................. 22
Aggregations ....................................................................... 24
- Creating Aggregation Jobs ................................................ 24
  - Creating an aggregation job using the Fusion UI ............ 24
  - Aggregation configuration parameters ......................... 25
Query Rewriting .................................................................. 28
Architectural overview .......................................................... 29
Query rewriting strategies ...................................................... 30
  - Business rules ............................................................. 30
- Underperforming query rewriting .................................... 30
- Misspelling detection ....................................................... 31
- Phrase detection .............................................................. 31
- Synonym detection .......................................................... 31
The Query Rewriting UI ......................................................... 32
Query rewrite collections ...................................................... 34
Rules Simulator query profile ................................................ 35
Query pipeline stages for query rewriting ............................. 36
Spark jobs for query rewriting ............................................. 37
  - Daily query rewriting jobs ............................................. 37
  - Additional query rewriting jobs ...................................... 37
  - "rules" role for query rewriting users ............................... 38
Redirecting...                                                                                           150
Follow Us                                                                                               150
© 2019 Lucidworks                                                                                       150
Products                                                                                                 152
Overview                                                                                                 152
User Guide (PDF)                                                                                         153
Reference Guides (PDF)                                                                                   153
Release Notes                                                                                           153
Redirecting...                                                                                           154
Follow Us                                                                                               154
© 2019 Lucidworks                                                                                       154
Products                                                                                                 156
Overview                                                                                                 156
User Guide (PDF)                                                                                         157
Reference Guides (PDF)                                                                                   157
Release Notes                                                                                           157
Redirecting...                                                                                           158
Follow Us                                                                                               158
© 2019 Lucidworks                                                                                       158
Products                                                                                                 160
Overview                                                                                                 160
User Guide (PDF)                                                                                         161
Reference Guides (PDF)                                                                                   161
Release Notes                                                                                           161
Redirecting...                                                                                           162
Follow Us                                                                                               162
© 2019 Lucidworks                                                                                       162
Spark and Machine Learning                                                                               163
Further Reading                                                                                          164
Spark Components                                                                                         165
Spark components in Fusion                                                                               166
Spark Getting Started                                                                                     167
  Starting the Spark Master and Spark Worker services                                                    167
  Viewing the Spark Master                                                                               168
  Running a job in the Spark shell                                                                       169
  The Spark master web UI                                                                               171
Spark job tuning                                                                                         171
Spark Driver Processes                                                                                   174
  Custom jobs                                                                                             174
  Drivers                                                                                                 174
    Default driver                                                                                       175
    Script-job driver                                                                                     176
  Spark drivers in a multinode cluster                                                                   176
Spark Configuration                                                                                      178
Plan an experiment

6. Choose an implementation approach

5. Calculate sample sizes

4. Choose traffic weights

2. Plan what you want to measure

1. Plan what you want to vary

Information flow

Example

Spark Troubleshooting

Scaling Spark Aggregations

Spark-related directories and files in Fusion

Spark components

Spark logs

Connection configurations for an SSL-enabled Solr cluster

Scaling Spark Aggregations

Speed up tasks and avoid timeouts.

Increase rows read per page

Improve job performance.

Spark Troubleshooting

Log API endpoints for Spark jobs

Specific issues

Job hung in waiting status

Lost executor due to heartbeat timeout

Spark Master won't start on EC2

Machine Learning Models in Fusion.

Training Models

Model Prediction

Model Checking

Metadata file spark-mllib.json

Experiments.

A/B/n experiments

Example

High-level workflow

Information flow

Metrics generation.

Plan an experiment

1. Plan what you want to vary

2. Plan what you want to measure

3. Design the experiment

4. Choose traffic weights

   4.1. Automatic traffic weights (multi-armed bandit)

   4.2. Manually specifying traffic weights

5. Calculate sample sizes

6. Choose an implementation approach

Next step

Experiment Metrics

Click-Through Rate

Conversion Rate.

Mean Reciprocal Rank (MRR).
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Time</td>
<td>207</td>
</tr>
<tr>
<td>Supported functions</td>
<td>208</td>
</tr>
<tr>
<td>Custom SQL</td>
<td>209</td>
</tr>
<tr>
<td>Query Relevance</td>
<td>210</td>
</tr>
<tr>
<td>Ground Truth Queries</td>
<td>211</td>
</tr>
<tr>
<td>Calculating Performance vs. Ground Truth</td>
<td>212</td>
</tr>
<tr>
<td>App Insights Dashboards</td>
<td>213</td>
</tr>
<tr>
<td>Analytics</td>
<td>213</td>
</tr>
<tr>
<td>Events</td>
<td>214</td>
</tr>
<tr>
<td>Custom reports</td>
<td>215</td>
</tr>
<tr>
<td>Dashboard filtering</td>
<td>215</td>
</tr>
<tr>
<td>Reports</td>
<td>216</td>
</tr>
<tr>
<td>App Insights Dashboards</td>
<td>217</td>
</tr>
<tr>
<td>Reports</td>
<td>218</td>
</tr>
<tr>
<td>App Insights pages</td>
<td>219</td>
</tr>
<tr>
<td>Reports</td>
<td>220</td>
</tr>
<tr>
<td>App Insights dashboards</td>
<td>221</td>
</tr>
<tr>
<td>Reports</td>
<td>222</td>
</tr>
<tr>
<td>Use an Experiment stage</td>
<td>222</td>
</tr>
<tr>
<td>How an Experiment stage works</td>
<td>222</td>
</tr>
<tr>
<td>Before you begin</td>
<td>222</td>
</tr>
<tr>
<td>1. Create an experiment</td>
<td>223</td>
</tr>
<tr>
<td>2. Set up an Experiment stage</td>
<td>225</td>
</tr>
<tr>
<td>Next step</td>
<td>225</td>
</tr>
<tr>
<td>Run an experiment</td>
<td>227</td>
</tr>
<tr>
<td>Activate an experiment</td>
<td>227</td>
</tr>
<tr>
<td>Verify that data is coming into the experiment</td>
<td>227</td>
</tr>
<tr>
<td>Deactivate an experiment</td>
<td>228</td>
</tr>
<tr>
<td>Next step</td>
<td>228</td>
</tr>
<tr>
<td>Analyze experiment results</td>
<td>229</td>
</tr>
<tr>
<td>Default schedules for metrics jobs</td>
<td>229</td>
</tr>
<tr>
<td>Modify metrics jobs schedules (optional)</td>
<td>229</td>
</tr>
<tr>
<td>Check the last time metrics jobs ran</td>
<td>229</td>
</tr>
<tr>
<td>Consider the metrics produced</td>
<td>230</td>
</tr>
<tr>
<td>Statistical significance</td>
<td>230</td>
</tr>
<tr>
<td>Confidence index</td>
<td>230</td>
</tr>
<tr>
<td>Percent chance of beating</td>
<td>231</td>
</tr>
<tr>
<td>Best practices</td>
<td>231</td>
</tr>
<tr>
<td>App Insights</td>
<td>232</td>
</tr>
<tr>
<td>App Insights pages</td>
<td>233</td>
</tr>
<tr>
<td>Reports</td>
<td>234</td>
</tr>
<tr>
<td>App Insights Dashboards</td>
<td>235</td>
</tr>
<tr>
<td>Dashboard filtering</td>
<td>236</td>
</tr>
<tr>
<td>Custom reports</td>
<td>237</td>
</tr>
<tr>
<td>Events</td>
<td>238</td>
</tr>
<tr>
<td>Custom reports</td>
<td>238</td>
</tr>
<tr>
<td>Sessions</td>
<td>240</td>
</tr>
<tr>
<td>Analytics</td>
<td>242</td>
</tr>
<tr>
<td>Standard reports</td>
<td>242</td>
</tr>
<tr>
<td>Custom reports</td>
<td>243</td>
</tr>
</tbody>
</table>
Signals and Aggregations

In addition to the basic search experience enabled through query pipelines, Fusion provides ways to develop an enhanced search experience for your end users and provide useful data for your analytics team. The primary mechanisms for doing this are signals and aggregations.

By collecting signals and aggregating them, you compile a body of data that allows you to develop a sophisticated search experience, with rich search results for your end users, based on past user behavior.

Signals and aggregated signals are stored each in their own collection. These collections are associated with a primary collection, so that a collection named "products" will have two related collections: "products_signals" and "products_signals_aggr". By default, when using the UI to create a collection, a "signals" and "aggregated signals" collection are also created.
Signals

Signals are events that are collected for analysis or to enhance the search experience for end users. Common types of signal events include clicks, purchases, downloads, ratings, and so on.

You can use App Insights to get visualizations and reports with which to analyze your signals data. App Insights mainly uses raw signals, but also uses some aggregated signals.
Aggregations

Aggregations are processed signals. An aggregator reads the raw signals and returns interesting summaries, ranging from simple sums to sophisticated statistical functions.

Crucially, it must be possible to relate the documents in an aggregated signals collection to documents in the primary collection, in order to use the aggregated signals for recommendations and/or boosting of searches over the primary collection.
The cold start problem

The "cold start" problem means it is hard to personalize the search experience when insufficient signals have been aggregated. For example, it is hard to offer recommendations to users who have never visited before, or for queries that have never been issued before, or for items that have been recently introduced into the system.

Fusion provides solutions for this problem using its query pipelines. A query pipeline that includes stages for blocking, boosting, or recommending based on signals can also include stages that provide fallbacks. In the case where there is not enough data to provide specialized blocking, boosting, or recommendations, the pipeline can return a simpler set of search results using Solr's normal relevancy calculation.

A common solution to the cold start problem is to sort or boost on a certain field to provide pseudo-recommendations when more specific recommendations are not available. For example, you can sort on the sales_rank field to recommend the most popular products, or boost on the date_added field to recommend the newest items.
Signals

A signal is a recorded event related to one or more documents in a collection. Signals can record any kind of event that is useful to your organization. Click signals are the most common type of signals as this is the most common action a user takes with an item. In addition, other signal types can be defined, such as "addToCart", "purchase", and so on.

Using a sufficiently large collection of signals, Fusion can automatically generate recommendations such as these:

- Based on the user's search query, which items are most likely to interest them?
- Based on the user's similarity to other users, which additional items are likely to interest them?

Signals are indexed in a secondary collection which is linked to the primary collection by the naming convention `<primarycollectionname>_signals`. So, if your main collection is named `products`, the associated signals collection is named `products_signals`. The signals collection is created automatically when signals are enabled for the primary collection. Signals are enabled by default whenever a new collection is created.

Signals are indexed just like ordinary documents. The signals collection can be searched like any other collection, for example by using the Query Workbench with the signals collection selected.

App Insights provides visualizations and reports with which to analyze your signals. App Insights mainly uses raw signals, but also uses some aggregated signals. Currently only the signal types Request, Response and Click are supported within the App Insights dashboards.

| Note | The signals schema changed in Fusion 4.0. See the descriptions of signals types and structure below. |
Enabling and disabling signals

You can enable and disable signals using the Fusion UI or the REST API.

Tip

When you disable signals, the aggregation jobs are deleted, but the _signals and _signals_aggr collections are not; your legacy signal data remains intact.

Using the UI

When you create a collection using the Fusion UI, signals are enabled and a signals collection created by default. You can also enable and disable signals for existing collections using the Collections Manager.

Enable signals for a collection

1. In the Fusion workspace, navigate to Collections > Collections Manager.
2. Hover over the primary collection for which you want to enable signals.
3. Click Configure to open the drop-down menu.
4. Click Enable Signals.

The Enable Signals window appears, with a list of collections and jobs that are created when you enable signals.
5. Click **Enable Signals**.

Disable signals for a collection

1. In the Fusion workspace, navigate to **Collections > Collections Manager**.
2. Hover over the primary collection for which you want to disable signals.
3. Click **Configure** to open the drop-down menu.
4. Click **Disable Signals**.

The **Disable Signals** window appears, with a list of jobs that are created when you enable signals.

5. Click **Disable Signals**.

Your `_signals` and `_signals_aggr` collections remain intact so that you can access your legacy signals data.

**Using the Collection Features API**

Using the API, the `/collections/{collection}/features/{feature}` endpoint enables or disables signals for any collection:

Check whether signals are enabled for a collection

```
curl -u user:pass http://localhost:8764/api/collections/<collection-name>/features/signals
```

Enable signals for a collection

```
```

Disable signals for a collection

```
curl -u user:pass -X DELETE http://localhost:8764/api/collections/<collection-name>/features/signals
```
curl -u user:pass -X PUT -H "Content-type: application/json" -d '{"enabled": false}'
http://localhost:8764/api/collections/<collection-name>/features/signals
Signals data flow

This diagram shows the flow of signals data from the search app through Fusion AI. The numbered steps are explained below.

1. The search app sends a query to a Fusion query pipeline.

   The query request should include a user ID and session query parameter to identify the user.

2. Optionally, the Fusion query pipeline queries the _signals_aggr collection to get boosts for the main query based on aggregated click data.

3. The search app also sends a request signal to the Fusion /signals endpoint.

   The primary intent of a request signal is to capture the raw user query and contextual information about the user's current activity in the app, such as the user agent and the page where they generated the query. The request signal does not contain any information about the results sent to Solr; it is created before a query is processed.

4. Once Solr returns the response to Fusion, the SearchLogger component indexes the complete request/response data into the _signals collection as a response signal using the _signals_ingest pipeline. Therefore, the response signal captures all results from Fusion as it related to the original query.

   Note

   This is a departure from pre-4.0 versions of Fusion where query impressions were logged in a separate _logs collection. Query activity is no longer indexed into the _logs collection. All response signals use the fusion_query_id (see below) as the unique document ID in Solr.

5. When the user clicks a link in the search results, the search app sends a click event to the Fusion signals endpoint (which invokes the _signals_ingest pipeline behind the scenes).
The click signal must include a field named fusion_query_id in the params object of the raw click signal. The fusion_query_id field is returned in the query response (from step 1) in a response header named x-fusion-query-id. This allows Fusion to associate a click signal with the response signal generated in step 4. The fusion_query_id is also used by Fusion to associate click signals with experiments. For experiments to work, each click signal must contain the corresponding fusion_query_id that produced the document/item that was clicked.

6. The _signals_ingest pipeline enriches signals before indexing into the _signals collection.

   This enrichment includes field mapping, geolocation resolution, and updating the has_clicks flag to “true” on request signals when the first click signal is encountered for a given request using the Update Related Document index stage.

7. Fusion’s App Insights queries the _signals collection through a Fusion query pipeline to generate query analytics reports from raw signals.

   Note that App Insights app uses Fusion security for authentication.

8. Behind the scenes, the SQL aggregation framework aggregates click signals to compute a weight for each query + doc_id + filters group.

   The resulting metrics are saved to the _signals_aggr collection to generate boosts on queries to the main collection (step 2 above).

9. Recommendations also use aggregated documents in the _signals_aggr collection to build a collaborative filtering-based recommender model.
Signals types and structure

Signals can be broadly categorized as implicit or explicit. When signals are enabled, Fusion produces several built-in signal types by default, all of which are implicit signals. You can also create custom signal types, including explicit signals. Be sure to verify that your signals include all of the important fields for best results. It's also useful to rank your signal types in terms of how strongly each type indicates a user's interest in an item.

Implicit signals vs explicit signals

Signals can reveal a user's level of interest in an item in two main ways:

- Implicit

  The user shows interest by engaging with the item/document through clicks, searches, and so on. Since this type of interaction requires no additional effort on the user's part, these types of signals tend to be plentiful. They can be used to infer a measurable value of interest in order to build an accurate recommender system.

- Explicit

  An explicit signal is created when a user intentionally assigns a clear, measurable value to an item, such as by giving it a rating. This value can be used to rank items, for example. Since this requires the user to invest extra time to provide the information, the number of ratings tends to be small compared to the total number of users interacting with the item.

You can create recommendations based on implicit signals out of the box. For recommenders based on explicit signals, contact your Lucidworks Professional Services representative.

Built-in signal types

There are three built-in signal types:

- request
- response
- click

Request signals

A request signal is generated by a front-end search app and captures the raw user query and other contextual information about a user and their journey through the search app. A request signal should have the following fields:
Additional optional fields are used by App Insights. In the raw signal, optional fields should be inside the `params` object. Optional fields are as follows:

```
"page_title":"Fusion Search",
"path": "/search",
"browser_type": "Browser",
"browser_version": "64.0.3282.140",
"browser_name": "Chrome",
"referrer": "http://localhost:8080/",
"ctx_prev_uri": "/",
"ctx_prev_query": ":",
"ctx_prev_path": "/",
"os_manufacturer": "Apple Inc.",
"os_name": "Mac OS X",
"os_id": "778",
"os_device": "Computer",
"os_group": "Mac OS X"
```

**Response signals**

Response signals are automatically generated by a query pipeline when the signals feature is enabled for a collection.

| Note | Front-end search applications should not send response signals to Fusion directly, as those would conflict with the auto-generated signals. |

A response signal has the following explicit fields, plus any additional query parameters sent by the search application for a query:
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The x-fusion-query-id generated by the query-pipeline used for associating click signals with queries in experiments and aggregation jobs.</td>
<td>TwWCn3Dz</td>
</tr>
<tr>
<td>type</td>
<td>Signal type</td>
<td>response</td>
</tr>
<tr>
<td>response_type</td>
<td>Used by Insights to determine if this query had results or was empty</td>
<td><code>results</code></td>
</tr>
<tr>
<td>empty`</td>
<td>User session ID; the search app should pass the session ID in the query params for a query</td>
<td></td>
</tr>
<tr>
<td>UUID</td>
<td>The actual query string sent to Solr from Fusion</td>
<td></td>
</tr>
<tr>
<td>ipad</td>
<td>The incoming query from the search app before it is enriched by the query pipeline</td>
<td></td>
</tr>
<tr>
<td>ipad</td>
<td>A hash generated from the session, query, and filters fields; used as a rollup key in Insights to group activity by a specific</td>
<td></td>
</tr>
<tr>
<td>SHA1 hash</td>
<td>Filter queries sent to Solr; the Fusion SearchLogger component combines multiple fq parameters into a single value delimited by &quot; $ &quot;</td>
<td></td>
</tr>
<tr>
<td>{!tag=format:(vhs) $ {!tag?type}(movie)</td>
<td>Reformatted filter queries for use by App Insights</td>
<td></td>
</tr>
<tr>
<td>field1/value</td>
<td>User ID; the search app should pass the user_id in the query params</td>
<td></td>
</tr>
<tr>
<td>admin</td>
<td>A comma-delimited list of document IDs returned for the page of results; this field is used by Fusion Spark jobs, such as the ground truth job, to perform click/skip analysis</td>
<td></td>
</tr>
<tr>
<td>Field Name</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>123,456,789</td>
<td>pipeline_id</td>
<td>Fusion query pipeline that processed this query</td>
</tr>
<tr>
<td>_system</td>
<td>collection</td>
<td>Fusion collection</td>
</tr>
<tr>
<td>my_collection</td>
<td>qtime</td>
<td>Query time from Solr, in milliseconds</td>
</tr>
<tr>
<td>10</td>
<td>rows</td>
<td>Number of rows requested for this query</td>
</tr>
<tr>
<td>10</td>
<td>hits</td>
<td>Total number of documents matching the query</td>
</tr>
<tr>
<td>10000</td>
<td>totaltime</td>
<td>Total processing time of this query in milliseconds, includes Solr qtime and Fusion query processing time</td>
</tr>
<tr>
<td>15</td>
<td>timestamp_tdt</td>
<td>Timestamp when the query request was received by Fusion</td>
</tr>
<tr>
<td>2018-02-15T18:42:560Z</td>
<td>res_offset</td>
<td>Offset of results; this field is used by experiment metrics to calculate MRR</td>
</tr>
<tr>
<td>0</td>
<td>params.*</td>
<td>Any other query param sent from the search app to Fusion that was not already mapped to a declared field</td>
</tr>
</tbody>
</table>

Fusion’s experiment framework relies heavily on response signals and the linking between response and clicks signals using the `fusion_query_id`.

**Click signals**

Click signals are sent from the search app to Fusion. All click signals should include a `fusion_query_id` field pulled from the query response header `x-fusion-query-id`. In addition, click signals should include the following fields:
Additional optional fields are used by App Insights. In the raw signal, optional fields should be inside the `params` object. Optional fields are as follows:

```
"browser_type":"Browser",
"browser_version":"64.0.3282.140",
"browser_name":"Chrome",
"referrer":"http://localhost:8080/",
"ctx_prev_uri": "/",
"ctx_prev_query": "/",
"ctx_prev_path": "/",
"os_manufacturer": "Apple Inc.",
"os_name": "Mac OS X",
"os_id": "778",
"os_device": "Computer",
"os_group": "Mac OS X"
```

Custom signal types

The signal `type` parameter can also take arbitrary values for custom signal types. For example, you can create special signals for purchase events, cart addition/subtraction events, “favorite” or “like” events, customer service events, and so on.

To collect custom signals, configure your front-end search application to send signals to Fusion using a custom value for the `type` field. Custom signals should also include the fields described below in order to get the best results from
aggregation and recommendation jobs.

To use custom signals in recommendations, you must add them to the value of the `signalTypeWeights` parameter in the configuration for the `_user_item_preferences_aggregation` job and the `_user_query_history_aggregation` job.

Custom signals can be analyzed in App Insights just like pre-defined signal types.

**Important fields for signals**

The jobs that aggregate signals and generate recommendations work best when all of the following fields are present in your signals:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Example Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count_i</code></td>
<td>1</td>
<td>Number of times an interaction event occurred with this item</td>
</tr>
<tr>
<td><code>doc_id</code></td>
<td>NMDDV</td>
<td>Product ID or Item ID</td>
</tr>
<tr>
<td><code>id</code></td>
<td>68f66808-6bfc-4d73-95f7-8a58529160b</td>
<td>The signal ID. If no ID is supplied, one will be automatically generated.</td>
</tr>
<tr>
<td><code>query</code></td>
<td>xwearabletech</td>
<td>A query string from the user</td>
</tr>
<tr>
<td><code>session_id</code></td>
<td>91aa66d11af44b6c90ccef44d055cf9a</td>
<td>Id for session in which user generated the signal</td>
</tr>
<tr>
<td><code>type</code></td>
<td>quick_view_click</td>
<td>Type of session the user used to interact with the platform</td>
</tr>
<tr>
<td><code>user_id</code></td>
<td>11506893</td>
<td>ID of user during the session</td>
</tr>
</tbody>
</table>

Some signal types, including custom signal types, may include additional fields.

**Signal field count analysis**

Lucidworks recommends performing signal field count analysis to determine whether any of the fields above are missing from some of your signals.

The table below shows how to query for specific fields using the Query Workbench in order to compare the number of results for each field with the total number of documents in the signals collection. In the examples in the third column, some fields appear in all 33,477,919 signals documents, while others appear in fewer documents.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Query</th>
<th>Example number of documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>:</td>
<td>33,477,919</td>
</tr>
<tr>
<td>count_i</td>
<td>count_i:[* TO *]</td>
<td>11,101,165</td>
</tr>
<tr>
<td>doc_id</td>
<td>doc_id:[* TO *]</td>
<td>23,216,297</td>
</tr>
<tr>
<td>id</td>
<td>id:[* TO *]</td>
<td>33,477,919</td>
</tr>
<tr>
<td>query</td>
<td>query:[* TO *]</td>
<td>19,724,598</td>
</tr>
<tr>
<td>session_id</td>
<td>session_id:[* TO *]</td>
<td>11,101,165</td>
</tr>
<tr>
<td>type</td>
<td>type:[* TO *]</td>
<td>33,477,919</td>
</tr>
<tr>
<td>user_id</td>
<td>user_id:[* TO *]</td>
<td>26,117,399</td>
</tr>
<tr>
<td>timestamp_tdt</td>
<td>timestamp_tdt:[* TO *]</td>
<td>26,117,399</td>
</tr>
</tbody>
</table>

You can also get the number of signals documents that contain all of the required fields by using the following query:

```
count_i:[* TO *] doc_id:[* TO *] id:[* TO *] query:[* TO *] type:[* TO *] user_id:[* TO *] timestamp_tdt:[* TO *] session_id:[* TO *]
```

**The query_id field**

For each incoming signal, Fusion calculates a value for the query_id field, which App Insights uses to create group-by-query reports like the one shown below:
To calculate the value, Fusion creates a hash based on session, query, and filter fields, then saves it into the query_id field.

The filter field can either be passed in by the search app, or computed by the SignalFormatterStage (the first stage in the signals_ingest pipeline) using the raw filter queries. For instance, on a response signal that is generated by a query pipeline, the following fq query params get translated into the multi-valued filter field:

- Raw query parameters:
  
  \[\text{fq}={!\text{tag}=\text{format}}\text{format}:\text{VHS}\&\text{fq}={!\text{tag}=\text{type}}\text{type}:\text{Movie}\]

- \text{filters_s} field (created by the SearchLogger component):

  \[{!\text{tag}=\text{format}}\text{format}:\text{vhs}\ $ {!\text{tag}=\text{type}}\text{type}:\text{movie}\]

- filter field:

  "filter":["format/VHS", "type/Movie"]

App Insights uses the filter field to generate various reports.

**Signal type ranking**

When you have defined some custom fields, it is useful to rank them according to how strongly they indicate a user's interest in an item. While it's not necessary to exclude certain signal types from the main signals collection, some can be excluded from signal aggregations in order to focus on the most important fields when generating recommendations.

**How to get the list of signal types**

1. In the Fusion UI, select your signals collection.
2. Open the Query Workbench by navigating to Query > Query Workbench.
3. Click Add a field facet.
4. Select the type field.

The list of signal types appears in the facet panel:
Add a field facet

▼ type

product_click_plp (13894353)
search (10261620)
add_to_cart (6031138)
order_submission (2479417)
quick_view_click (806126)

View next 10
The default signals index pipeline

When indexing signals, Fusion uses a default index pipeline named `_signals_ingest` unless you specify a different index pipeline.

The `_signals_ingest` index pipeline has five stages:

1. Format Signals stage
2. Field Mapping stage
3. GeoIP Lookup stage
4. Solr Indexer stage
5. Update `has_clicks` flag stage

The Update `has_clicks` flag stage is an instance of the Update Related Document stage that updates the `has_clicks` flag to "true" on an existing request signal after the first click signal is processed for the request.

The update stage works as follows:

6. When a click signal is encountered (type == click)
7. Look at the incoming click signal for a field named `request_id_s`, which gets set by the Format Signals stage using a distributed cache of recently processed request signals.

If the `request_id_s` field is set, then send a real-time `GET` query to Solr to find a request signal with ID equal to the value of the `request_id_s` field on the click signal. To avoid re-updating request signals, the RTG query also filters on...
has_clicks==false, which avoids duplicate atomic updates on the same document in Solr. Real-time GET is used to avoid timing issues between a request signal being sent to Solr and when it gets committed. This prevents missing updates when clicks occur soon after the initial request signal is sent by the search app.

8. If the click signal does not have the request_id_s field set, then do a normal Solr lookup for the request signal using: +query_id:"${query_id}" +type:request +has_clicks:false. A click signal may not have a request_id_s if there is a cache miss in the distributed cache used by the Format Signals stage.

9. If the stage performs a normal query, there may be multiple request signals that have the same query_id. This is because the query_id is based on session + query + filter, so if a user sends the same query + filter during the same session, there will be multiple request signals with the same query_id value. Thus, the stage sorts to get the latest request signal to update.

10. If a related document is found (in this case a request signal), then the stage updates the has_clicks field to true and performs an atomic update in Solr.

This stage performs its work in a background thread, so it does not impact the indexing performance of the click signal.
Deleting old signals

Signals are not automatically deleted by default, and over time they occupy an increasing amount of storage space.

To avoid running out of storage space as a result of your growing collection of signals, you must decide on a signals retention policy, then configure and schedule a REST Call job that regularly deletes old signals.

The duration for which signals should be kept depends on your use case and your organization’s policies. For example, in some cases signals could be deleted after 30 days, while in other cases they may remain useful for a year, or even forever.

How to configure a REST Call job to delete old signals

1. Navigate to Collections > Jobs.
2. Click Add and select REST Call.
   
   The REST Call job configuration panel appears.
3. Enter an arbitrary ID for this job, such as “Delete-old-signals”.
4. Enter the following endpoint URI, substituting the name of your signals collection for signalsCollectionName:
   
   solr://signalsCollectionName/update
5. In the Call Method field, select “post”.
6. Under Query Parameters, enter the property name "wt" with the property value "json".
7. In the Request entity (as string) field, enter the following:
   
   <root><delete><query>timestamp_tdt:[* TO NOW-2WEEKS]</query></delete><commit /></root>

See Working with Dates for details about date formatting.

Your job configuration should look similar to this:
**Delete-old-signals**
Run arbitrary REST/HTTP/Solr command

**ID**
Delete-old-signals

**ENDPOINT URI**
solr://Movie_Search_signals/update

**CALL METHOD**
post

**QUERY PARAMETERS**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>wt</td>
<td>json</td>
</tr>
</tbody>
</table>

**REQUEST PROTOCOL HEADERS**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
</table>

**REQUEST ENTITY (AS STRING)**

```xml
<root><delete><query>timestamp_tdt:[* TO NOW-6MONTHS]</query></delete><commit /></root>
```

**Tip**
You can configure a schedule for this job at **System > Scheduler**.
Aggregations

Aggregations compile Signals into a set of summaries that you can use to enrich the search experience through recommendations and boosting.

You can create two kinds of aggregations:

- **SQL aggregations** (strongly recommended) – SQL is a familiar query language that is well suited to data aggregation. Fusion’s new SQL Aggregation Engine has more power and flexibility than Fusion’s legacy aggregation engine.

- **Legacy aggregations** (deprecated) – This aggregation approach available in prior Fusion releases is still available, though it is deprecated and will be removed in a future release. Aggregator functions apply solely to legacy aggregations.

Aggregations are created automatically whenever you enable signals or recommendations. This topic explains how to create or modify aggregations individually. You can do this using the Fusion UI or the Jobs API. For more information, see Creating Aggregations.

**Creating Aggregation Jobs**

Aggregations are created automatically whenever you enable signals or recommendations. This topic explains how to create or modify aggregations individually. You can do this using the Fusion UI or the Jobs API.

**Creating an aggregation job using the Fusion UI**

An aggregation is a type of job. Aggregation jobs can be created or modified at **Search > Jobs** in the Fusion UI.

1. Navigate to **Search > Jobs**.
2. Click **Add**.
3. Select **Aggregation**.

The New Job Configuration panel appears.
4. Enter an arbitrary Spark job ID.

5. Enter the name of the signals collection to be aggregated.

Note
Be sure to specify the signals collection (usually `<primarycollectionname>_signals`), not the primary `<primarycollectionname>` collection.

6. Under Aggregation Settings, click **include**.

7. Configure the aggregation parameters as needed.

See Aggregation configuration parameters below for descriptions.

8. Click **Save**.

The new aggregation job appears in the jobs list. Now you can run it or schedule it.

**Aggregation configuration parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupingFields</td>
<td>An array of strings specifying the fields to group on.</td>
</tr>
<tr>
<td>signalTypes</td>
<td>The signal types. If not set then any signal type is selected.</td>
</tr>
<tr>
<td>selectQuery</td>
<td>The query to select the desired signals. If not set then `: will be used, or equivalent.</td>
</tr>
<tr>
<td>sort</td>
<td>The criteria to sort on within a group. If not set then sort order is by ID, ascending.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>timeRange</strong></td>
<td>The time range to select signals on.</td>
</tr>
<tr>
<td><strong>outputPipeline</strong></td>
<td>What pipeline to use to process the output. If not set then _system pipeline will be used.</td>
</tr>
<tr>
<td><strong>rollupPipeline</strong></td>
<td>Pipeline to use for processing results of roll-up. This is by default the same index pipeline used for processing the aggregation results.</td>
</tr>
<tr>
<td><strong>rollupAggregator</strong></td>
<td>The aggregator to use when rolling up. If not set then the same aggregator will be used for roll-up.</td>
</tr>
<tr>
<td><strong>outputCollection</strong></td>
<td>The collection to write the aggregates to on output. This property is required if the selected output/rollup pipeline requires it (the default pipeline does). A special value of - disables the output.</td>
</tr>
<tr>
<td><strong>aggregator</strong></td>
<td>Aggregator implementation to use. This is either one of the symbolic names (simple, click, em) or a fully-qualified class name of a class extending EventAggregator. If not set then 'simple' is used.</td>
</tr>
<tr>
<td><strong>sourceRemove</strong></td>
<td>If true, the processed source signals will be removed after aggregation. Default is false.</td>
</tr>
<tr>
<td><strong>sourceCatchup</strong></td>
<td>If true, only aggregate the signals since the last time the job was successfully run. If there is a record of such previous run then this overrides the starting time of time range set in timeRange property.</td>
</tr>
<tr>
<td><strong>outputRollup</strong></td>
<td>Roll-up current results with all previous results for this aggregation id, which are available in outputCollection.</td>
</tr>
<tr>
<td>aggregates</td>
<td>List of functions defining how to aggregate events with results. Aggregation functions have these properties:</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>• <strong>type</strong></td>
</tr>
<tr>
<td></td>
<td>The function type defining how to aggregate events with results.</td>
</tr>
<tr>
<td></td>
<td>• <strong>sourceFields</strong></td>
</tr>
<tr>
<td></td>
<td>The fields that the function will read from.</td>
</tr>
<tr>
<td></td>
<td>• <strong>targetField</strong></td>
</tr>
<tr>
<td></td>
<td>The field that the function will write to.</td>
</tr>
<tr>
<td></td>
<td>• <strong>mapper</strong></td>
</tr>
<tr>
<td></td>
<td>When true the function will be used in map phase only.</td>
</tr>
<tr>
<td></td>
<td>• <strong>parameters</strong></td>
</tr>
<tr>
<td></td>
<td>Other parameters specific to individual functions.</td>
</tr>
</tbody>
</table>

| statsFields | List of numeric fields in results for which to compute overall statistics. |

| parameters | Other aggregation parameters (such as start / aggregate / finish scripts, cache size, and so on). |
Query Rewriting

Query rewriting is a strategy for improving relevancy using AI-generated data. Many of Fusion AI’s features can be used to rewrite incoming queries prior to submitting them to Fusion’s Solr core. These rewrites produce more relevant search results with higher conversion rates.

For example, when spelling corrections are used for query rewriting, a misspelled query can return the same search results as a correctly-spelled query, instead of returning irrelevant results or no results. Spelling corrections are one of several available query rewriting strategies. Apply all available strategies for best results.

See also the Query Rewriting API.

Fusion can also rewrite Solr’s responses before returning them to the search application; see Response Rewriting.

If you have apps created in Fusion 4.1 and earlier, see these instructions for enabling business rules in those apps.
Architectural overview

The following diagram depicts the main components involved in query rewriting:
Query rewriting strategies

Fusion AI provides a variety of query rewriting strategies to improve relevancy:

- Business rules
- Underperforming query rewriting
- Misspelling detection
- Phrase detection
- Synonym detection

With the exception of business rules, which are always manually created, these strategies correspond to certain Spark jobs. Lucidworks recommends configuring and scheduling all of these jobs for best results. You can also train the jobs by manually adding documents to their output. Manually-added documents are used for machine learning and are never overwritten by new job output.

Query rewriting strategies are applied in the following order:

1. Business rules
   If a query triggers a business rule, then the business rule overrides any query rewriting strategies that conflict with it.

2. Underperforming query rewriting
   If a query triggers an underperforming query rewrite, then this strategy overrides all subsequent query rewriting strategies.

3. Synonym detection
4. Misspelling detection and phrase detection

The query rewriting results from both of these strategies are applied together. To use only the strategy with the longest surface form, you can configure the Text Tagger query stage with Overlapping Tag Policy set to "LONGEST_DOMINANT_RIGHT".

Business rules

Business rules are manually-created formulas for rewriting queries. This is the most versatile strategy for creating custom query rewrites. It supports a variety of conditions and actions to address a wide range of use cases. When you need a very specific query rewrite, this is the best strategy.

Business rules are applied in the Apply Rules stage of the query pipeline.

See Business Rules to learn how to create, edit, and publish business rules.

Underperforming query rewriting

The Underperforming Query Rewriting feature uses your signals data to identify underperforming queries and suggest improved queries that could produce better conversion rates. When underperforming query rewriting is enabled and
an incoming query contains a matching underperforming query term, the original term is replaced by the improved
term.

Query improvements are applied in the Text Tagger stage of the query pipeline.

See Underperforming Query Rewriting to learn how to review, edit, create, and publish query improvements.

**Misspelling detection**

The Misspelling Detection feature maps misspellings to their corrected spellings. When Fusion receives a query
containing a known misspelling, it rewrites the query using the corrected spelling in order to return relevant results
instead of an empty or irrelevant results set.

Spelling corrections are applied in the Text Tagger stage of the query pipeline.

See Misspelling Detection to learn how to review, edit, create, and publish spelling corrections.

| Tip | 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. |

**Phrase detection**

Phrase detection identifies phrases in your signals so that results with matching phrases can be boosted. This helps
compensate for queries where phrases are not distinguished with quotation marks. For example, the query `ipad case` is
rewritten as `"ipad case" ~10^2`, meaning that if `ipad` and `case` appear within 10 characters of each other then boost the
result by a factor of two.

Phrases are applied in the Text Tagger stage of the query pipeline.

See Phrase Detection to learn how to review, edit, create, and publish phrases.

**Synonym detection**

The Synonym Detection feature generates 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. When synonym detection is enabled, a
query that contains a matching term is expanded to include all of its synonyms, with the original term boosted by a
factor of two.

Synonyms are applied in the Text Tagger stage of the query pipeline.

See Synonym Detection to learn how to review, edit, create, and publish pairs of synonyms and similar queries.
The Query Rewriting UI

To open the query rewriting interface, navigate to Relevance > Query Rewriting.

The query rewriting dashboard appears:

This page gives you access to the Simulator and the query rewriting strategies:

- Business Rules
- Underperforming Query Rewriting
- Misspelling Detection
- Phrase Detection
- Synonym Detection

All of these components are enabled by default. You can click "Enabled" to toggle it to "Disabled".

| Note | Enabling and disabling strategies in the Query Rewriting UI does not enable or disable their corresponding Spark jobs. |
Query rewrite collections

For each app, two auxiliary collections are dedicated to documents used for query rewriting:

- **_query_rewrite_staging**
  
  As of Fusion 4.2, certain Spark jobs send their output to this collection. Rules are also written to this collection initially.

  Some of the content in this collection requires manual review before it can be migrated to the _query_rewrite, where query pipelines can read it. See below for details.

- **_query_rewrite**
  
  This collection is optimized for high-volume traffic. Query pipelines can read from this collection to find rules, synonyms, spelling corrections, and more with which to rewrite queries and responses.

Each app contains exactly one of each of these collections, associated with the app's default collection. They are not created again for additional collections created within the same app.

Documents move from query_rewrite_staging to the _query_rewrite collection only when they are approved (either automatically on the basis of their confidence scores or manually by a human reviewer) _and a Fusion user clicks Publish. The review field value indicates whether a document will be published when the user clicks Publish:

<table>
<thead>
<tr>
<th>review</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>review=auto</td>
<td>A job-generated document has a sufficiently high confidence score and is automatically approved for publication.</td>
</tr>
<tr>
<td>review=pending</td>
<td>A job-generated document has an ambiguous confidence score and must be reviewed by a Fusion user.</td>
</tr>
<tr>
<td>review=approved</td>
<td>A Fusion user has reviewed the document and approved it for publication.</td>
</tr>
<tr>
<td>review=denied</td>
<td>A job-generated document has a low confidence score, or a Fusion user has reviewed and denied it for publication.</td>
</tr>
</tbody>
</table>

Tip

In the query rewriting UI, the value of the review field appears in the Status column.

You can review and approve or deny documents using the query rewriting UI. You can also change a document's status to "pending" to save it for later review.
Rules Simulator query profile

The Rules Simulator allows product owners to experiment with rules and other query rewrites in the `_query_rewrite_staging` collection before deploying them to the `_query_rewrite` collection.

Each app has a `_rules_simulator` query profile, configured to use the `_query_rewrite_staging` collection for query rewrites instead of the `_query_rewrite` collection. This profile is created automatically whenever a new app is created.
Query pipeline stages for query rewriting

These query rewriting stages are part of any default query pipeline:

• Apply Rules query stage

This stage looks up rules that have been deployed to the `_query_rewrite` collection and matches them against the query. Matching rules that perform query rewriting are applied at this stage, while matching rules that perform response rewriting are applied by the Modify Response with Rules stage later in the pipeline.

• Text Tagger query pipeline stage

This stage uses the SolrTextTagger handler to identify known entities in the query by searching the `_query_rewrite` collection (or the `_query_rewrite_staging` collection in the case of the Fusion AI query rewriting Simulator) to find matching spelling corrections, phrase boosts, underperforming query improvements, and synonym expansions in order to perform query rewriting.
Spark jobs for query rewriting

This section describes how Spark jobs support query rewriting. These jobs read from the signals collection and write their output to the _query_rewrite_staging collection. High-confidence results are automatically migrated from there to the _query_rewrite collection, while ambiguous results remain in the staging collection until they are reviewed and approved. You can review job results in the Query Rewriting UI.

- Daily query rewriting jobs are created and scheduled automatically when you create a new app.
- Additional query rewriting jobs can be created manually.

Tip

For best relevancy, enable all of these jobs.

Daily query rewriting jobs

When a new app is created, the jobs below are also created and scheduled to run daily, beginning 15 minutes after app creation, in the following order:

1. Token and Phrase Spell Correction job
   
   Detect misspellings in queries or documents using the numbers of occurrences of words and phrases.

2. Phrase Extraction job
   
   Identify multi-word phrases in signals.

3. Synonym and Similar Queries Detection job
   
   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.

   The first and second jobs can provide input to improve the Synonym job's output:

   - Token and Phrase Spell Correction job results can be used to avoid finding mainly misspellings, or mixing synonyms with misspellings.
   - Phrase Extraction job results can be used to find pairs of synonyms with multiple tokens, such as "lithium ion"/"ion battery".

   The second and third jobs are triggered by the success of the previous job, that is, the phrase detection job runs only if the spell correction job succeeds, and the synonym job runs only if the phrase detection job succeeds.

Additional query rewriting jobs

These jobs also produce results that are used for query rewriting, but must be created manually:

- Head/Tail Analysis job

  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.
• Ground Truth job

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

"rules" role for query rewriting users

The "rules" role provides permissions to access query rewriting features for all Fusion apps. A Fusion admin can create a user account with this role to give a business user access to the Query Rewriting UI. :leveloffset: +1
Business Rules

Business rules are manually-created formulas for rewriting queries. This is the most versatile strategy for creating custom query rewrites. It supports a variety of conditions and actions to address a wide range of use cases. When you need a very specific query rewrite, this is the best strategy.

Business rules are applied in the Apply Rules stage of the query pipeline.

| Tip | When a query triggers a business rule, then the business rule overrides any query rewriting strategies that conflict with it. |
Create a new business rule

1. Navigate to Relevance > Query Rewriting.

2. Under Business Rules, click View.

3. Click the + icon.

The New Rule window appears:
In the **General** column, only the **Name** field is required. Other fields are optional:

- **Description** is an arbitrary string you can use to describe this rule.
- **Rule Group** can be a user-defined group that you use to organize your rules.
- **Tags** are another way to organize your rules. Tags appear as facets in the Business Rules interface, so you can filter the set of visible rules by tag.
- **Priority** can be used to determine which rule should apply first if two rules are activated for the same request. If two rules have the same priority, one will be chosen at random to be applied before the other. The value range is 1 to infinity.
- **Enabled** means that this rule is applied (but not necessarily published). Disabling a rule helps ensure that it is not accidentally published.

<table>
<thead>
<tr>
<th>General</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the <strong>General</strong> column, only the <strong>Name</strong> field is required. Other fields are optional:</td>
<td>Conditions are triggers that activate the rule when they match the current date and time, query, or field values. See Rule conditions below.</td>
<td>A rule can take different types of actions when the specified conditions are met. See Action types below. You can also create custom actions.</td>
</tr>
</tbody>
</table>

4. Click **Save**.
Edit, enable, or disable an existing rule

1. Navigate to Relevance > Query Rewriting.
2. Under Business Rules, click View.
3. Hover over the rule you want to edit.
4. Click the icon next to the rule.
Publishing your changes

How to publish updated business rules

1. In the Business Rules screen, click the PUBLISH button.

   Fusion prompts you to confirm that you want to publish your changes.

2. Click PUBLISH.

Tip

You can un-publish a query rewrite by changing its status to "pending" or "denied", then clicking PUBLISH.
**Rule conditions**

A rule can have one or more conditions which define the criteria that trigger the rule’s actions. The + Add Condition button allows you to add a condition type to a rule, and you can add multiple conditions to the same rule. When multiple conditions are defined, they are joined by Boolean OR, that is, one or more conditions must be met in order to trigger the rule.

Tips:

- Since the Query condition can be configured with multiple criteria, you only need one condition of that type per rule.
- To configure a rule that always fires with every query, select no conditions.
- Multiple conditions of different types are joined by Boolean OR, that is, the rule is triggered when one or more conditions of different types are met.
- By default, multiple field value conditions are joined by Boolean OR. To use AND so that all field value conditions must be met, go to the Apply Rules query pipeline stage and de-select Partially Matched Filter Queries Will Trigger the Rule.

Condition types are described below:

- **Dates**
  The rule is applied only during the specified date and time range, which can be open-ended.
  
- **Query**
  The rule is applied when the query terms match using one of the following methods:
  
  - **Keywords**
    The query term exactly matches one or more specified query terms. Keyword matching is case-insensitive. Multiple keywords can be specified with a comma delimiter.
    
    | Tip | This is the fastest type of query matching. |
    |-----|----------------------------------------|
    |     | Tip using | This is the fastest type of query matching. |
    |
    - **Text**
      The query term is matched using text tokenization. For example, "customer" matches "customer service" and "customer help".
      
      Enter one or more query terms to match using the selected method. When you enter multiple terms, any match against one of them will trigger the action (logical OR).

- **Field Value**
  This condition type identifies a predefined field name and value that may be present in the Solr filter query parameter (fq), such as an "on_sale" field whose value is "true". In that example, your rule could block results whose "on_sale" field value is "false".

Field Value conditions require whole-field exact matches and are generally called programmatically, by a link to a
category or a click on a facet.
Rule actions

A rule can have one or more actions that are triggered when the rule’s conditions are met by the incoming query.

**Built-in rule actions**

- **Boost List**

  Boost particular items to the top of the results.

  For example, boost gift items on sale to the top of search results during the first weeks of December.

  To use the Query Elevation Component, you must first configure your Solr cluster using the instructions below, then select **Use Query Elevation Component**. Note that query elevation does not boost scores.

- **Banner**

  Display a user-defined banner message when the rule fires.

  For example, a search for “office dvd” could return a results page that includes a special banner that advertises “The Office DVD boxed set: Now 50% off”.

- **Bury List**

  When the condition is met, the rule down-ranks all the documents with the specified field values for the given field name. Use this action when you want to minimize certain results without blocking them.

- **Block List**

  Suppress one or more items from the list of results.

- **Set Facets**

- **Set Params**

  This rule type corresponds to the Additional Query Parameters stage and permits the complex modeling of a rule.

- **Filter List**

  Change the results so only a specified set of content is shown.

  For example, a search for “kids movies” could return only the titles whose MPAA_rating field matches “G” or “PG”.

  To use the Query Elevation Component, you must first configure your Solr cluster using the instructions below, then select **Use Query Elevation Component**. Note that query elevation does not boost scores.

- **Redirect**

  Send users to a different URL instead of the search results.

  For example, a search for “black friday” could redirect users to a special sale page instead of a list of products that match “black” and “friday”.


• Boost Attributes

This action boosts documents with specific attributes by adding the `bq` or `boost` parameter to the incoming Solr request and executing a boost query.

When the condition is met, the boost query is executed and the docs returned from the boost query are boosted in the results. For example, this action can be used to boost items where `color="red"` when the incoming query contains "red".

• My Custom Rule

Select an alternate query pipeline that implements special processing for this rule. See below for more details.

**Query Elevation Component for Rule Actions**

Fusion AI currently supports the usage of the Query Elevation Component (QEC) with boost lists and filter lists. You must configure Solr in order to enable the QEC option.

1. Create an XML file named `elevate.xml` in the same directory as the `solrconfig.xml` file. Use the following as the contents of the file:

   ```xml
   <?xml version="1.0" encoding="UTF-8" ?>
   <elevate/>
   </elevate>
   ```

   **Note**

   Although it will not be accessed for the elevation process, Fusion requires this independent `elevate.xml` file for the use of built-in rule actions with QEC.

2. Add the following in the `elevator` searchComponent:

   ```xml
   <str name="config-file">elevate.xml</str>
   ```

3. Add `elevate` as a component in the `solrconfig.xml` file:

   ```xml
   <arr name="last-components">
   <str>elevator</str>
   </arr>
   ```

4. When you create a new business rule, check the **USE QUERY ELEVATION COMPONENT** checkbox to use elevation.
Important

The Query Elevation Component only elevates documents within the query rewriting rules engine by the document id field. Ensure id is entered in the “FIELD NAME” option.

Custom rule actions

You can create custom actions for rules by creating a special query pipeline that implements the desired functionality, then creating a custom rule using the UI or the API.

Using the UI to create custom rules

How to create a custom rule using the UI

1. Navigate to Relevance > Query Rewriting.
2. Under Business Rules, click View.
3. Click click the + icon.

The New Rule window appears.

4. Enter the general parameters and conditions for this custom rule.
6. Select the query pipeline to use when this rule is triggered.
7. Enter a name for this custom rule type.
8. Optionally, you can click Add under Return Parameter Policy Override to override the value of a parameter returned from the specified pipeline:
9. Enter the name of the parameter.
10. Enter the desired value.
11. Select one of the following policies:
   ○ Replace the returned value with the specified value.
   ○ Append the specified value to the returned value.
   ○ Remove the specified parameter from the set of returned parameters.
   ○ Default
12. Enter a value for From Custom Rule.

Using the API to create custom rules

You can create a custom rule type by POSTing to the /query-rewrite/schema endpoint. A custom rule type has:

• an id field, which must be unique across all apps in Fusion (global namespace)
• a pipeline_id that is invoked during rule processing
• a display_type that gives a human-friendly name to the custom rule type
• a JSON schema that defines a set of parameters to pass to the pipeline during rule processing

For example, the following custom rule type with ID some_custom_rule calls the custom_pipeline during rule processing
and passes the values for `custom_param1` and `custom_param2`:

```bash
curl -XPOST -H "Content-type:application/json" \
  "id": "some_custom_rule",
  "pipeline_id": "custom_pipeline",
  "response_pipeline_id": "do_more_custom_stuff_here",
  "display_type": "Custom Rule",
  "schema": {
    "type": "object",
    "properties": {
      "custom_param1": {
        "type": "string",
        "title": "Custom Param One",
        "description": "Some param from the rule creator"
      },
      "custom_param2": {
        "type": "string",
        "title": "Custom Param 2",
        "description": "Some other param the rule creator needs to provide"
      }
    }
  }
}'
```

The rules editor UI will render this custom rule type with input fields for `custom_param1` and `custom_param2`.

Here's an example of how to create an instance of the custom rule type:

```bash
curl -XPOST -H "Content-type:application/json" \
  "id": "some_custom_rule1",
  "type": "some_custom_rule",
  "custom_type": "some_custom_rule",
  "name": "My Custom Rule 1",
  "description": "Call another pipeline to do custom things",
  "search_terms": ["fusion"],
  "custom_param1": "This is the user-supplied value of custom_param1",
  "custom_param2": "This is the user-supplied value of custom_param2",
  "pipeline_id": "custom_pipeline",
  "display_type": "Custom Rule"
}'
```

When the user queries for “fusion”, the custom rule will match (using `search_terms`) and the rules processing stage will call `custom_pipeline`, passing parameters `custom_param1` and `custom_param2` in the request.

Presumably, the `custom_pipeline` uses these parameters to perform some custom logic. When the invoked pipeline (such as `custom_pipeline`) returns, the rules processing stage running in the main pipeline extracts the parameters returned from the invoked pipeline and adds them to the main request. If the invoked pipeline has a rules processing stage, it will not re-invoke itself if any matching rules in the invoked pipeline would result in an infinite loop.
Query pipeline stages for rules

These stages are part of the default query pipeline:

- **Apply Rules**

  This stage looks up rules that have been deployed to the `_query_rewrite` collection and matches them against the query. Matching rules that perform query rewriting are applied at this stage, while matching rules that perform response rewriting are applied by the `width=600%` stage later in the pipeline.

- **Modify Response with Rules**

  Most rules operate on the request, but some rule types, such as banner rules or redirect rules, do their work when the response comes back. The Modify Response with Rules stage applies those rules to the response. For example, a banner rule can add a banner URL to the response before returning it to the client.

Generally, if you are using rules then you need both of these stages enabled in your query pipeline. The Apply Rules stage must come before the Solr Query stage, while the Modify Response with Rules stage must come after the Solr Query stage. Disabling or removing the Apply Rules stage will disable rules entirely, while disabling or removing the Modify Response with Rules stage will disable only the rules that perform response rewriting, if any.
Rules on response signals

Response signals capture a list of rule IDs that match the query, in a multi-valued field named `rule_ss`. This allows for downstream analysis on rule activity using SQL, App Insights, or a custom Spark job.
Rules and experiments

The easiest way to integrate rules and experiments is to create one query pipeline with rules enabled and one without, with all other settings constant between the two pipelines. The pipeline without rules enabled must be configured manually.

To create an experimental rule, use tags on the rule and set the tag in one of the variant pipelines.
Simulator

The Simulator provides an interactive preview of how your staged rules affect relevancy, using your search data and a simple search interface. When you enter a query, the Simulator shows you the triggered rules, in the order in which they were triggered, along with any triggered facets.

You can edit any of the triggered rules, then re-run the query to see new results. You can also return to the query rewriting dashboard to enable or disable query rewriting strategies.

The Simulator sends requests to the _rules_simulator query profile, which you can configure to point to any pipeline and collection in your app.

How to open the Simulator

1. In the Fusion UI, navigate to Relevancy > Query Rewriting.

   The query rewriting dashboard appears.

2. Click the Simulator button.

   The Simulator appears:
From here, you can:

- Enter search terms to see the results, using query rewriting data from the `_query_rewrite_staging` collection
- Edit rules that are triggered by the current query
- Click **Query Rewriting Dashboard** to go to the dashboard, where you can enable or disable query rewriting strategies, then return to the Simulator to see how relevancy is affected

**Editing rules in the Simulator**

You can edit any rule in the list of triggered rules, including disabling it, by clicking the icon next to the rule.

De-select "Enabled" in the Edit Rule window to turn off this rule. See below for details about the conditions and actions that you can modify when editing a rule.
Tip: Only rules triggered by the current query are available to edit in the Simulator. To edit other rules, see Business Rules.

**Rule conditions**

- **Dates**
  The rule is applied only during the specified date and time range, which can be open-ended.

- **Query**
  The rule is applied when the query terms match using one of the following methods:
  - **Keywords**
    The query term exactly matches one or more specified query terms. Keyword matching is case-insensitive. Multiple keywords can be specified with a comma delimiter.
    
    Tip: This is the fastest type of query matching.
  - **Text**
    The query term is matched using text tokenization. For example, "customer" matches "customer service" and "customer help".
    
    Enter one or more query terms to match using the selected method. When you enter multiple terms, any match against one of them will trigger the action (logical OR).

- **Field Value**
  This condition type identifies a predefined field name and value that may be present in the Solr filter query parameter (fq), such as an "on_sale" field whose value is "true". In that example, your rule could block results whose "on_sale" field value is "false".
  
  Field Value conditions require whole-field exact matches and are generally called programmatically, by a link to a category or a click on a facet.

**Rule actions**

- **Boost List**
  Boost particular items to the top of the results.
  
  For example, boost gift items on sale to the top of search results during the first weeks of December.
  
  To use the Query Elevation Component, you must first configure your Solr cluster using the instructions below, then select **Use Query Elevation Component**. Note that query elevation does not boost scores.

- **Banner**
  Display a user-defined banner message when the rule fires.
For example, a search for "office dvd" could return a results page that includes a special banner that advertises "The Office DVD boxed set: Now 50% off".

- **Bury List**

  When the condition is met, the rule down-ranks all the documents with the specified field values for the given field name. Use this action when you want to minimize certain results without blocking them.

- **Block List**

  Suppress one or more items from the list of results.

- **Set Facets**

- **Set Params**

  This rule type corresponds to the Additional Query Parameters stage and permits the complex modeling of a rule.

- **Filter List**

  Change the results so only a specified set of content is shown.

  For example, a search for "kids movies" could return only the titles whose MPAA_rating field matches "G" or "PG".

  To use the Query Elevation Component, you must first configure your Solr cluster using the instructions below, then select **Use Query Elevation Component**. Note that query elevation does not boost scores.

- **Redirect**

  Send users to a different URL instead of the search results.

  For example, a search for "black friday" could redirect users to a special sale page instead of a list of products that match "black" and "friday".

- **Boost Attributes**

  This action boosts documents with specific attributes by adding the bq or boost parameter to the incoming Solr request and executing a boost query.

  When the condition is met, the boost query is executed and the docs returned from the boost query are boosted in the results. For example, this action can be used to boost items where color="red" when the incoming query contains "red".

- **My Custom Rule**

  Select an alternate query pipeline that implements special processing for this rule. See below for more details.

### The _rules_simulator query profile

Each app has a _rules_simulator query profile, configured to use the _query_rewrite_staging collection for query rewrites instead of the _query_rewrite collection. This profile is created automatically whenever a new app is created.

By default, this query profile points to your default query pipeline and collection. You can configure it to point to any pipeline or collection, for example when testing a new pipeline before it has been deployed.
To change the query pipeline, collection, and query parameters used by the `_rules_simulator` query profile

1. Open the Fusion UI.
2. Navigate to **Querying > Query Profiles**.
3. Select the `_rules_simulator` query profile for your app.

   For example, if your app is called "Demo" then the name of the query profile is **Demo_rules_simulator**.

4. Modify the configuration as desired.
5. Click **Save**.

**Enabling and disabling query rewriting strategies**

By default, all query rewriting strategies are enabled, and all of the data in the `_query_rewrite_staging` collection is applied in the Simulator. To see how relevancy is affected by individual strategies, you can selectively enable or disable strategies in the query rewriting dashboard.

**How to enable or disable query rewriting strategies**

1. From the Simulator, click **Query Rewriting Dashboard**.

   From the Simulator, click **Query Rewriting Dashboard**.
The query rewriting dashboard appears:

2. Click **Enabled** to disable an active strategy, or click **Disabled** to enable an inactive strategy.

3. Click **Simulator** to return to the Simulator and see how your changes affect search results.
Underperforming Queries

The Underperforming Query Rewriting feature uses your signals data to identify underperforming queries and suggest improved queries that could produce better conversion rates. When underperforming query rewriting is enabled and an incoming query contains a matching underperforming query term, the original term is replaced by the improved term.

Query improvements are applied in the Text Tagger stage of the query pipeline.

The Head/Tail Analysis job automatically creates query improvements based on your AI-generated data. When you navigate to **Relevance > Query Rewriting > Underperforming Query Rewriting**, you can review or edit the output from the job and manually add new query improvements. Your changes remain in the `{pass-qrs}` collection until you publish them.

**Tip**

When you manually add new query improvements, subsequent job runs use those documents as input for machine learning to improve the job’s output. Unlike job-generated documents, manually-added query rewriting documents are never overwritten by new job output.

**Note**

Job-generated query improvements are always assigned an initial status of "Pending", never "Auto". Query improvements must be explicitly approved and published in order to be copied to the `_query_rewrite` collection.

---

### Reviewing auto-generated query improvements

Query improvements that are automatically generated by the Head/Tail Analysis job are assigned the following Status value:

- **Auto**
All results will be automatically deployed to the {pass-qr} collection.

No action is required on these results, though you can edit them if you wish.

Adding new query improvements

In addition to the query improvements generated by the Head/Tail Analysis job, you can manually add your own.

How to add a query improvement

1. Navigate to Relevance > Query Rewriting > Underperforming Query Rewriting.
2. At the bottom of the rules list, click the icon.

A new query improvement appears at the top of the list:

3. Enter the underperforming query.
4. Enter one or more query improvements.

Tip  It's not necessary to set a confidence value.

5. Select the query improvement's status, depending on whether you want to deploy it the next time you publish your changes ("Approved") or save it for further review ("Pending").
6. Click the check mark to save the new query improvement:
Publishing your changes

How to publish updated query improvements

1. In the Underperforming Query Rewriting screen, click the PUBLISH button.

   Fusion prompts you to confirm that you want to publish your changes.

2. Click PUBLISH.

Tip

You can un-publish a query rewrite by changing its status to "pending" or "denied", then clicking PUBLISH.
Misspelling Detection

The Misspelling Detection feature maps misspellings to their corrected spellings. When Fusion receives a query containing a known misspelling, it rewrites the query using the corrected spelling in order to return relevant results instead of an empty or irrelevant results set.

Spelling corrections are applied in the Text Tagger stage of the query pipeline.

The Token and Phrase Spell Correction job automatically creates spelling corrections based on your AI-generated data. When you navigate to Relevance > Query Rewriting > Misspelling Detection, you can review or edit the output from the job and manually add new spelling corrections. Your changes remain in the _query_rewrite_staging collection until you publish them.

Tip

When you manually add new spelling corrections, subsequent job runs use those documents as input for machine learning to improve the job’s output. Unlike job-generated documents, manually-added query rewriting documents are never overwritten by new job output.

Tip

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.

Reviewing auto-generated spelling corrections

Spelling corrections that are automatically generated by the Token and Phrase Spell Correction job are assigned one of these Status values:

- Auto

There are three values for confidence level:
<table>
<thead>
<tr>
<th>Value</th>
<th>Confidence</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>low confidence</td>
<td>Pending</td>
</tr>
<tr>
<td>0.5</td>
<td>median confidence</td>
<td>Auto</td>
</tr>
<tr>
<td>1</td>
<td>high confidence</td>
<td>Auto</td>
</tr>
</tbody>
</table>

No action is required on these results, though you can edit them if you wish.

**Pending**

The confidence level is ambiguous, and the result must be reviewed by a user before it can be deployed. It will only be moved from the `query_rewrite_staging` collection to the `_query_rewrite` collection when its status has changed to "Approved" and it has been published.

See below for instructions.

How to review a pending spelling correction result

1. Navigate to **Relevance > Query Rewriting > Misspelling Detection**.

   **Tip**  
   Notice the **Status** facet on the left. Click **Pending** to view only the items that need review.

2. Click the pencil icon next to the spelling correction.

3. In the **Status** column, select either "Approved" or "Denied".

   Optionally, you can also edit the spelling correction itself.

   **Tip**  
   Although the Confidence field is also editable, changing its value makes no difference.

4. Click the Close icon next to the updated spelling correction:
Note

Adding new spelling corrections

In addition to the spelling corrections generated by the Token and Phrase Spell Correction job, you can manually add your own.

How to add a spelling correction

1. Navigate to Relevance > Query Rewriting > Misspelling Detection.
2. At the bottom of the rules list, click the icon.

A new spelling correction appears at the top of the list:

Note

Approving a spelling correction does not automatically deploy it to the _query_rewrite collection. When you have finished your review, you must click Publish to deploy your changes.
3. Enter the misspelled word or phrase.

4. Enter one or more spelling corrections.

Tip
It’s not necessary to set a confidence value.

5. Select the spelling correction’s status, depending on whether you want to deploy it the next time you publish your changes (“Approved”) or save it for further review (“Pending”).

6. Click the check mark to save the new spelling correction:

---

**Publishing your changes**

How to publish updated spelling corrections

1. In the **Misspelling Detection** screen, click the **PUBLISH** button.

   Fusion prompts you to confirm that you want to publish your changes.

2. Click **PUBLISH**.
Tip | You can un-publish a query rewrite by changing its status to "pending" or "denied", then clicking PUBLISH.

### Tuning the misspelling detection job

The default configuration for the Token and Phrase Spell Correction job is designed for high accuracy and works well with most signal datasets, depending on the volume and quality of the signals. If you are seeing too few results, or too many inaccurate results, then you can try tuning the job to achieve better results.

Tip | To modify job configurations, you must be a Fusion user with the admin or developer role, or custom permissions that include access to job configurations.
Phrase Detection

Phrase detection identifies phrases in your signals so that results with matching phrases can be boosted. This helps compensate for queries where phrases are not distinguished with quotation marks. For example, the query `ipad case` is rewritten as "ipad case"~10^2, meaning that if `ipad` and `case` appear within 10 characters of each other then boost the result by a factor of two.

Phrases are applied in the Text Tagger stage of the query pipeline.

The Phrase Extraction job automatically creates phrases based on your AI-generated data. When you navigate to Relevance > Query Rewriting > Phrase Detection, you can review or edit the output from the job and manually add new phrases. Your changes remain in the `{pass-qrs}` collection until you publish them.

**Tip**

When you manually add new phrases, subsequent job runs use those documents as input for machine learning to improve the job’s output. Unlike job-generated documents, manually-added query rewriting documents are never overwritten by new job output.

**Reviewing auto-generated phrases**

Phrases that are automatically generated by the Phrase Extraction job are assigned one of these Status values:

- **Auto**

  These results have a confidence level as a threshold to automatically deploy them to the `{pass-qr}` collection. This threshold can be specified in the configuration parameter **Minimum Likelihood Score** (default value 0.1).

  No action is required on these results, though you can edit them if you wish.

- **Pending**

  The confidence level is ambiguous, and the result must be reviewed by a user before it can be deployed. It will only
be moved from the \textit{pass-qrs} collection to the \textit{pass-qr} collection when its status has changed to "Approved" \textit{and} it has been published.

See below for instructions.

How to review a pending phrase result

1. Navigate to \textbf{Relevance > Query Rewriting > Phrase Detection}.

   | Tip | Notice the \textbf{Status} facet on the left. Click \textbf{Pending} to view only the items that need review. |

2. Click the \textbullet{} icon next to the phrase.

3. In the \textbf{Status} column, select either "Approved" or "Denied".

   Optionally, you can also edit the phrase itself.

   | Tip | Although the Confidence field is also editable, changing its value makes no difference. |

4. Click the \textbf{Close} icon next to the updated phrase:
Adding new phrases

In addition to the phrases generated by the Phrase Extraction job, you can manually add your own.

How to add a phrase

1. Navigate to Relevance > Query Rewriting > Phrase Detection.
2. At the bottom of the rules list, click the ☰️ icon.

   A new phrase appears at the top of the list:

   Approving a phrase does not automatically deploy it to the \{pass-qr\} collection. When you have finished your review, you must click Publish to deploy your changes.
3. Enter the phrase.

Tip

It's not necessary to set a confidence value.

4. Select the phrase's status, depending on whether you want to deploy it the next time you publish your changes ("Approved") or save it for further review ("Pending").

5. Click the check mark to save the new phrase:

Publishing your changes

How to publish updated phrases

1. In the Phrase Detection screen, click the PUBLISH button.
Fusion prompts you to confirm that you want to publish your changes.

2. Click **PUBLISH**.

| Tip | You can un-publish a query rewrite by changing its status to "pending" or "denied", then clicking **PUBLISH**. |
Synonym Detection

The Synonym Detection feature generates 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. When synonym detection is enabled, a query that contains a matching term is expanded to include all of its synonyms, with the original term boosted by a factor of two.

Synonyms are applied in the Text Tagger stage of the query pipeline.

The Synonym and Similar Queries Detection job automatically creates synonym pairs based on your AI-generated data. When you navigate to Relevance > Query Rewriting > Synonym Detection, you can review or edit the output from the job and manually add new synonym pairs. Your changes remain in the {pass-qrs} collection until you publish them.

Tip

When you manually add new synonym pairs, subsequent job runs use those documents as input for machine learning to improve the job’s output. Unlike job-generated documents, manually-added query rewriting documents are never overwritten by new job output.

Synonym directionality

When you edit or create a synonym pair, you can configure its directionality, either uni-directional or bi-directional.

To toggle between uni-directional and bi-directional, click the icon next to the synonym pair, then click the directionality icon:
Uni-directional synonyms

Uni-directional synonyms produce query substitutions. That is, when a query term matches a known uni-directional synonym, the original query term is replaced with the synonym.

In the uni-directional example below, "iphone" is a synonym of "phone" but "phone" is not a synonym of "iphone". When "phone" is found in a query, it is replaced with "iphone".

Tip
When you manually create a new synonym pair, the default is uni-directional.

Bi-directional synonyms

Bi-directional synonyms produce expanded queries. That is, when a query contains a known bi-directional synonym, the query is rewritten to include the original term plus all of its known bi-directional synonyms, resulting in a greater number of potentially relevant results. The original term is also boosted to help preserve the user's intent.

In the bi-directional example below, "tv" is a synonym of "television" and "television" is a synonym of "tv". If a query contains either "tv" or "television", the query is expanded to include both terms.

Tip
Job-generated synonyms are always bi-directional unless a reviewer edits them.

Reviewing auto-generated synonym pairs

Synonyms that are automatically generated by the Synonym and Similar Queries Detection job are assigned the following Status value:

- Pending

The confidence level is ambiguous, and the result must be reviewed by a user before it can be deployed. It will only be moved from the {pass-qrs} collection to the {pass-qr} collection when its status has changed to "Approved" and it has been published.
By default, all results from synonym job are set to "Pending", since there are usually limited number of synonyms and synonym expansion can have high impact on relevancy.

See below for instructions.

How to review a pending synonym pair result

1. Navigate to Relevance > Query Rewriting > Synonym Detection.

   Tip | Notice the Status facet on the left. Click Pending to view only the items that need review.

2. Click the icon next to the synonym pair.

3. In the Status column, select either "Approved" or "Denied".

   Optionally, you can also edit the synonym pair itself.

   Where alternative synonyms were detected, you can click Suggestions to view and select them as replacements for the displayed synonym pair:

   Tip | Although the Confidence field is also editable, changing its value makes no difference.

4. Click the Close icon next to the updated synonym pair:
Approving a synonym pair does not automatically deploy it to the \{pass-qr\} collection. When you have finished your review, you must click **Publish** to deploy your changes.

### Adding new synonym pairs

In addition to the synonym pairs generated by the Synonym and Similar Queries Detection job, you can manually add your own.

**How to add a synonym pair**

1. Navigate to **Relevance** > **Query Rewriting** > **Synonym Detection**.
2. At the bottom of the rules list, click the \(+)\) icon.

A new synonym pair appears at the top of the list:
3. Enter the query term.

4. Enter one or more synonym pairs.

Tip

It’s not necessary to set a confidence value.

5. Select the synonym pair’s status, depending on whether you want to deploy it the next time you publish your changes (“Approved”) or save it for further review (“Pending”).

6. Click the check mark to save the new synonym pair:

---

**Publishing your changes**

How to publish updated synonym pairs

1. In the Synonym Detection screen, click the PUBLISH button.
Fusion prompts you to confirm that you want to publish your changes.

2. Click **PUBLISH**.

| Tip | You can un-publish a query rewrite by changing its status to “pending” or “denied”, then clicking **PUBLISH**. |
Response Rewriting

Response rewriting is a strategy for improving relevancy using AI-generated data by modifying Solr’s response before passing it to the search application. There are two approaches to configuring response rewriting:
Rules

Any rule type can add content to the response by modifying the `responseValues` field. These two rule types can replace the whole response:

- Banner
  Display a user-defined banner message when the rule fires.

- Redirect
  Send users to a different URL instead of the search results.

See the main Rules topic for details.
Query pipeline stages

Query pipeline stages that perform response rewriting must appear after the Solr Query stage. These stages fall into two categories:

• **Stages that act on the whole set of results**
  ◦ Response Shuffle stage
    "De-bias" results by shuffling the top N results randomly.
  ◦ Response Pairwise Swap stage
    "De-bias" results by swapping the search results at any two positions, such as positions 1 and 2, positions 3 and 4, and so on.

• **Stages that act on individual documents**
  ◦ Machine Learning (Responses) stage
    Apply a machine learning model to the response.
  ◦ Response Document Exclusion stage
    Drop all documents that match all of the specified rules.
  ◦ Response Document Field Redaction stage
    Remove fields that match a regular expression from a document.
  ◦ Modify Response with Rules stage
    Apply rules to the response.

= Natural Language Processing

This topic describes Fusion AI’s Natural Language Processing (NLP) features, available in the legacy OpenNLP NER Extraction index pipeline stage and the newer NLP Annotator index and query pipeline stages.
OpenNLP NER Extraction pipeline stage

The OpenNLP NER Extraction index pipeline stage performs only Named Entity Recognition (NER). This stage is available in all versions of Fusion AI.

For additional NLP functionality, use the NLP Annotator pipeline stages, available in Fusion AI versions 4.2 and up. See below for details.
NLP Annotator pipeline stages (4.2.0 and above)

Fusion AI 4.2 introduced the NLP Annotator as both an index pipeline stage and a query pipeline stage. The NLP Annotator performs a variety of fundamental NLP tasks:

- Sentence detection
- Named Entity Extraction (NER)
- Part-of-Speech (POS) Tagging

If configured in an index pipeline, the NLP annotator performs selected NLP tasks on raw document content during the indexing process (see more details here). If configured in a query pipeline, the NLP annotator performs selected NLP tasks on the query text content (see more details here).
NLP features

Fusion's NLP Annotator pipeline stages include the NLP features described below.

Sentence detection

Sentence detection is the process of analyzing text to determine sentence boundaries. It is typically the first step taken when performing any kind of natural language processing on a document. Commonly, a sentence is indexed as a multi-value field that can be used for various purposes, as in these examples:

- Relevancy: Boost documents whose first sentence matches the query terms.
- Snippets: When presenting the search results, display the first few sentences of each document.

Named Entity Recognition (NER)

Named Entity Recognition is a popular technique used in information extraction to identify and segment the named entities and classify or categorize them under these predefined classes:

- person
- organization
- location

For example:

Will Hayes is CEO of Lucidworks, based in San Francisco.

Name entity recognition is widely leveraged by today’s text mining projects. When organizations store large volumes of business documents in Fusion AI, the natural next step is to turn the large volume of text-centric data into some kind of knowledge base.

Take entity linking projects, for example: The client may want to link all relevant documents with an existing list of entities of interest. One way of doing this is to extract entities from all raw text documents, then perform fuzzy matching or another kind of text pattern matching to link relevant documents with a specific entity from the given list. This is more efficient than scanning the whole document and trying to search for the entity name. In this scenario, NER extraction is an ideal tool.

Fusion AI has integrated NER capability into its indexing and query pipelines to enable customers to perform knowledge discovery easily.

Part-of-Speech (POS) tagging

One of the most important roles of POS tagging is "word sense disambiguation". For instance, when searching for the word "present", if the intent is to look for the concept of gift, then having the word "present" tagged as a "noun" will help filter out content with "present" as a verb, representing an action of bringing before the public.
Signals and Aggregations

In addition to the basic search experience enabled through query pipelines, Fusion provides ways to develop an enhanced search experience for your end users and provide useful data for your analytics team. The primary mechanisms for doing this are signals and aggregations.

By collecting signals and aggregating them, you compile a body of data that allows you to develop a sophisticated search experience, with rich search results for your end users, based on past user behavior.

Signals and aggregated signals are stored each in their own collection. These collections are associated with a primary collection, so that a collection named "products" will have two related collections: "products_signals" and "products_signals_aggr". By default, when using the UI to create a collection, a "signals" and "aggregated signals" collection are also created.
Signals

*Signals* are events that are collected for analysis or to enhance the search experience for end users. Common types of signal events include clicks, purchases, downloads, ratings, and so on.

You can use App Insights to get visualizations and reports with which to analyze your signals data. App Insights mainly uses raw signals, but also uses some aggregated signals.
Aggregations

*Aggregations* are processed signals. An *aggregator* reads the raw signals and returns interesting summaries, ranging from simple sums to sophisticated statistical functions.

Crucially, it must be possible to relate the documents in an aggregated signals collection to documents in the primary collection, in order to use the aggregated signals for recommendations and/or boosting of searches over the primary collection.
The cold start problem

The "cold start" problem means it is hard to personalize the search experience when insufficient signals have been aggregated. For example, it is hard to offer recommendations to users who have never visited before, or for queries that have never been issued before, or for items that have been recently introduced into the system.

Fusion provides solutions for this problem using its query pipelines. A query pipeline that includes stages for blocking, boosting, or recommending based on signals can also include stages that provide fallbacks. In the case where there is not enough data to provide specialized blocking, boosting, or recommendations, the pipeline can return a simpler set of search results using Solr's normal relevancy calculation.

A common solution to the cold start problem is to sort or boost on a certain field to provide pseudo-recommendations when more specific recommendations are not available. For example, you can sort on the sales_rank field to recommend the most popular products, or boost on the date_added field to recommend the newest items.
Signals

A signal is a recorded event related to one or more documents in a collection. Signals can record any kind of event that is useful to your organization. Click signals are the most common type of signals as this is the most common action a user takes with an item. In addition, other signal types can be defined, such as "addToCart", "purchase", and so on.

Using a sufficiently large collection of signals, Fusion can automatically generate recommendations such as these:

- Based on the user's search query, which items are most likely to interest them?
- Based on the user's similarity to other users, which additional items are likely to interest them?

Signals are indexed in a secondary collection which is linked to the primary collection by the naming convention `<primarycollectionname>_signals`. So, if your main collection is named products, the associated signals collection is named products_signals. The signals collection is created automatically when signals are enabled for the primary collection. Signals are enabled by default whenever a new collection is created.

Signals are indexed just like ordinary documents. The signals collection can be searched like any other collection, for example by using the Query Workbench with the signals collection selected.

App Insights provides visualizations and reports with which to analyze your signals. App Insights mainly uses raw signals, but also uses some aggregated signals. Currently only the signal types Request, Response and Click are supported within the App Insights dashboards.

| Note                                      | The signals schema changed in Fusion 4.0. See the descriptions of signals types and structure below. |
Enabling and disabling signals

You can enable and disable signals using the Fusion UI or the REST API.

**Tip**

When you disable signals, the aggregation jobs are deleted, but the `_signals` and `_signals_aggr` collections are not; your legacy signal data remains intact.

Using the UI

When you create a collection using the Fusion UI, signals are enabled and a signals collection created by default. You can also enable and disable signals for existing collections using the Collections Manager.

Enable signals for a collection

1. In the Fusion workspace, navigate to **Collections > Collections Manager**.
2. Hover over the primary collection for which you want to enable signals.
3. Click **Configure** to open the drop-down menu.
4. Click **Enable Signals**.

The **Enable Signals** window appears, with a list of collections and jobs that are created when you enable signals.
5. Click **Enable Signals**.

Disable signals for a collection

1. In the Fusion workspace, navigate to **Collections** > **Collections Manager**.
2. Hover over the primary collection for which you want to disable signals.
3. Click **Configure** to open the drop-down menu.
4. Click **Disable Signals**.

   The **Disable Signals** window appears, with a list of jobs that are created when you enable signals.

5. Click **Disable Signals**.

   Your `_signals` and `_signals_aggr` collections remain intact so that you can access your legacy signals data.

**Using the Collection Features API**

Using the API, the `/collections/{collection}/features/{feature}` endpoint enables or disables signals for any collection:

**Check whether signals are enabled for a collection**

```bash
curl -u user:pass http://localhost:8764/api/collections/<collection-name>/features/signals
```

**Enable signals for a collection**

```bash
```

**Disable signals for a collection**

```bash
curl -u user:pass -X DELETE http://localhost:8764/api/collections/<collection-name>/features/signals
```
curl -u user:pass -X PUT -H "Content-type: application/json" -d '{"enabled" : false}'
http://localhost:8764/api/collections/<collection-name>/features/signals
Signals data flow

This diagram shows the flow of signals data from the search app through Fusion AI. The numbered steps are explained below.

1. The search app sends a query to a Fusion query pipeline.

   The query request should include a user ID and session query parameter to identify the user.

2. Optionally, the Fusion query pipeline queries the `_signals_aggr` collection to get boosts for the main query based on aggregated click data.

3. The search app also sends a request signal to the Fusion `/signals` endpoint.

   The primary intent of a request signal is to capture the raw user query and contextual information about the user’s current activity in the app, such as the user agent and the page where they generated the query. The request signal does not contain any information about the results sent to Solr; it is created before a query is processed.

4. Once Solr returns the response to Fusion, the SearchLogger component indexes the complete request/response data into the `_signals` collection as a response signal using the `_signals_ingest` pipeline. Therefore, the response signal captures all results from Fusion as it related to the original query.

   **Note**
   
   This is a departure from pre-4.0 versions of Fusion where query impressions were logged in a separate `_logs` collection. Query activity is no longer indexed into the `_logs` collection. All response signals use the `fusion_query_id` (see below) as the unique document ID in Solr.

5. When the user clicks a link in the search results, the search app sends a click event to the Fusion signals endpoint (which invokes the `_signals_ingest` pipeline behind the scenes).
The click signal must include a field named `fusion_query_id` in the `params` object of the raw click signal. The `fusion_query_id` field is returned in the query response (from step 1) in a response header named `x-fusion-query-id`. This allows Fusion to associate a click signal with the response signal generated in step 4. The `fusion_query_id` is also used by Fusion to associate click signals with experiments. For experiments to work, each click signal must contain the corresponding `fusion_query_id` that produced the document/item that was clicked.

6. The `_signals_ingest` pipeline enriches signals before indexing into the `_signals` collection.

   This enrichment includes field mapping, geolocation resolution, and updating the `has_clicks` flag to "true" on request signals when the first click signal is encountered for a given request using the Update Related Document index stage.

7. Fusion's App Insights queries the `_signals` collection through a Fusion query pipeline to generate query analytics reports from raw signals.

   Note that App Insights app uses Fusion security for authentication.

8. Behind the scenes, the SQL aggregation framework aggregates click signals to compute a weight for each query + `doc_id` + filters group.

   The resulting metrics are saved to the `_signals_aggr` collection to generate boosts on queries to the main collection (step 2 above).

9. Recommendations also use aggregated documents in the `_signals_aggr` collection to build a collaborative filtering-based recommender model.
Signals types and structure

Signals can be broadly categorized as implicit or explicit. When signals are enabled, Fusion produces several built-in signal types by default, all of which are implicit signals. You can also create custom signal types, including explicit signals. Be sure to verify that your signals include all of the important fields for best results. It’s also useful to rank your signal types in terms of how strongly each type indicates a user’s interest in an item.

Implicit signals vs explicit signals

Signals can reveal a user’s level of interest in an item in two main ways:

• Implicit

  The user shows interest by engaging with the item/document through clicks, searches, and so on. Since this type of interaction requires no additional effort on the user’s part, these types of signals tend to be plentiful. They can be used to infer a measurable value of interest in order to build an accurate recommender system.

• Explicit

  An explicit signal is created when a user intentionally assigns a clear, measurable value to an item, such as by giving it a rating. This value can be used to rank items, for example. Since this requires the user to invest extra time to provide the information, the number of ratings tends to be small compared to the total number of users interacting with the item.

You can create recommendations based on implicit signals out of the box. For recommenders based on explicit signals, contact your Lucidworks Professional Services representative.

Built-in signal types

There are three built-in signal types:

• request
• response
• click

Request signals

A request signal is generated by a front-end search app and captures the raw user query and other contextual information about a user and their journey through the search app. A request signal should have the following fields:
Additional optional fields are used by App Insights. In the raw signal, optional fields should be inside the `params` object. Optional fields are as follows:

```
"page_title":"Fusion Search",
"path":"/search",
"browser_type": "Browser",
"browser_version": "64.0.3282.140",
"browser_name": "Chrome",
"referrer": "http://localhost:8080/",
"ctx_prev_uri": "/",
"ctx_prev_query": "",
"ctx_prev_path": "/",
"os_manufacturer": "Apple Inc.",
"os_name": "Mac OS X",
"os_id": "778",
"os_device": "Computer",
"os_group": "Mac OS X"
```

**Response signals**

Response signals are automatically generated by a query pipeline when the signals feature is enabled for a collection.

| Note | Front-end search applications should not send response signals to Fusion directly, as those would conflict with the auto-generated signals. |

A response signal has the following explicit fields, plus any additional query parameters sent by the search application for a query:
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The x-fusion-query-id generated by the query-pipeline used for associating click signals with queries in experiments and aggregation jobs.</td>
<td>TwWCn3Dz</td>
</tr>
<tr>
<td>type</td>
<td>Signal type</td>
<td>response</td>
</tr>
<tr>
<td>response_type</td>
<td>Used by Insights to determine if this query had results or was empty</td>
<td>'results</td>
</tr>
<tr>
<td>empty`</td>
<td>session</td>
<td>User session ID; the search app should pass the session ID in the query params for a query</td>
</tr>
<tr>
<td>UUID</td>
<td>query</td>
<td>The actual query string sent to Solr from Fusion</td>
</tr>
<tr>
<td>ipad</td>
<td>query_orig_s</td>
<td>The incoming query from the search app before it is enriched by the query pipeline</td>
</tr>
<tr>
<td>ipad</td>
<td>query_id</td>
<td>A hash generated from the session, query, and filters fields; used as a rollup key in Insights to group activity by a specific</td>
</tr>
<tr>
<td>SHA1 hash</td>
<td>filters_s</td>
<td>Filter queries sent to Solr; the Fusion SearchLogger component combines multiple fq parameters into a single value delimited by &quot;$&quot;</td>
</tr>
<tr>
<td><code>format:(vhs) $</code></td>
<td>filter</td>
<td>Reformatted filter queries for use by App Insights</td>
</tr>
<tr>
<td><code>type:(movie)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>field1/value</td>
<td>user_id</td>
<td>User ID; the search app should pass the user_id in the query params</td>
</tr>
<tr>
<td>admin</td>
<td>doc_ids_s</td>
<td>A comma-delimited list of document IDs returned for the page of results; this field is used by Fusion Spark jobs, such as the ground truth job, to perform click/skip analysis</td>
</tr>
<tr>
<td>Field Name</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>123,456,789</td>
<td>pipeline_id</td>
<td>Fusion query pipeline that processed this query</td>
</tr>
<tr>
<td>_system</td>
<td>collection</td>
<td>Fusion collection</td>
</tr>
<tr>
<td>my_collection</td>
<td>qtime</td>
<td>Query time from Solr, in milliseconds</td>
</tr>
<tr>
<td>10</td>
<td>rows</td>
<td>Number of rows requested for this query</td>
</tr>
<tr>
<td>10</td>
<td>hits</td>
<td>Total number of documents matching the query</td>
</tr>
<tr>
<td>10000</td>
<td>totaltime</td>
<td>Total processing time of this query in milliseconds, includes Solr qtime and Fusion query processing time</td>
</tr>
<tr>
<td>15</td>
<td>timestamp_tdt</td>
<td>Timestamp when the query request was received by Fusion</td>
</tr>
<tr>
<td>2018-02-15T18:17:42.560Z</td>
<td>res_offset</td>
<td>Offset of results; this field is used by experiment metrics to calculate MRR</td>
</tr>
<tr>
<td>0</td>
<td>params.*</td>
<td>Any other query param sent from the search app to Fusion that was not already mapped to a declared field</td>
</tr>
</tbody>
</table>

Fusion’s experiment framework relies heavily on response signals and the linking between response and clicks signals using the fusion_query_id.

**Click signals**

Click signals are sent from the search app to Fusion. All click signals should include a fusion_query_id field pulled from the query response header x-fusion-query-id. In addition, click signals should include the following fields:
Additional optional fields are used by App Insights. In the raw signal, optional fields should be inside the `params` object. Optional fields are as follows:

```json
"browser_type":"Browser",
"browser_version":"64.0.3282.140",
"browser_name":"Chrome",
"referrer":"http://localhost:8080/",
"ctx_prev_uri":"/",
"ctx_prev_query":"

To collect custom signals, configure your front-end search application to send signals to Fusion using a custom value for the `type` field. Custom signals should also include the fields described below in order to get the best results from Fusion.

**Custom signal types**

The signal `type` parameter can also take arbitrary values for custom signal types. For example, you can create special signals for purchase events, cart addition/subtraction events, “favorite” or “like” events, customer service events, and so on.
aggregation and recommendation jobs.

To use custom signals in recommendations, you must add them to the value of the signalTypeWeights parameter in the configuration for the _user_item_preferences_aggregation job and the _user_query_history_aggregation job.

Custom signals can be analyzed in App Insights just like pre-defined signal types.

**Important fields for signals**

The jobs that aggregate signals and generate recommendations work best when all of the following fields are present in your signals:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Example Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count_i</td>
<td>1</td>
<td>Number of times an interaction event occurred with this item</td>
</tr>
<tr>
<td>doc_id</td>
<td>NMDDV</td>
<td>Product ID or Item ID</td>
</tr>
<tr>
<td>id</td>
<td>68f66808-6bfc-4d73-95f7-8a558529160b</td>
<td>The signal ID. If no ID is supplied, one will be automatically generated.</td>
</tr>
<tr>
<td>query</td>
<td>xwearabletech</td>
<td>A query string from the user</td>
</tr>
<tr>
<td>session_id</td>
<td>91aa66d11af44b6c90cccf44d055cf9a</td>
<td>Id for session in which user generated the signal</td>
</tr>
<tr>
<td>type</td>
<td>quick_view_click</td>
<td>Type of session the user used to interact with the platform</td>
</tr>
<tr>
<td>user_id</td>
<td>11506893</td>
<td>ID of user during the session</td>
</tr>
</tbody>
</table>

Some signal types, including custom signal types, may include additional fields.

**Signal field count analysis**

Lucidworks recommends performing signal field count analysis to determine whether any of the fields above are missing from some of your signals.

The table below shows how to query for specific fields using the Query Workbench in order to compare the number of results for each field with the total number of documents in the signals collection. In the examples in the third column, some fields appear in all 33,477,919 signals documents, while others appear in fewer documents.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Query</th>
<th>Example number of documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>:</td>
<td>33,477,919</td>
</tr>
<tr>
<td>count_i</td>
<td>count_i:[* TO *]</td>
<td>11,101,165</td>
</tr>
<tr>
<td>doc_id</td>
<td>doc_id:[* TO *]</td>
<td>23,216,297</td>
</tr>
<tr>
<td>id</td>
<td>id:[* TO *]</td>
<td>33,477,919</td>
</tr>
<tr>
<td>query</td>
<td>query:[* TO *]</td>
<td>19,724,598</td>
</tr>
<tr>
<td>session_id</td>
<td>session_id:[* TO *]</td>
<td>11,101,165</td>
</tr>
<tr>
<td>type</td>
<td>type:[* TO *]</td>
<td>33,477,919</td>
</tr>
<tr>
<td>user_id</td>
<td>user_id:[* TO *]</td>
<td>26,117,399</td>
</tr>
<tr>
<td>timestamp_tdt</td>
<td>timestamp_tdt:[* TO *]</td>
<td>26,117,399</td>
</tr>
</tbody>
</table>

You can also get the number of signals documents that contain all of the required fields by using the following query:

```
count_i:[* TO *] doc_id:[* TO *] id:[* TO *] query:[* TO *] type:[* TO *] user_id:[* TO *] timestamp_tdt:[* TO *] session_id:[* TO *]
```

**The query_id field**

For each incoming signal, Fusion calculates a value for the `query_id` field, which App Insights uses to create group-by-query reports like the one shown below:
The `query_id` field should not be confused with the `fusion_query_id`, which is a unique ID for each query processed by a Fusion query pipeline, or with `query_s` which is the query string.

To calculate the value, Fusion creates a hash based on `session`, `query`, and `filter` fields, then saves it into the `query_id` field.

The `filter` field can either be passed in by the search app, or computed by the SignalFormatterStage (the first stage in the `_signals_ingest` pipeline) using the raw filter queries. For instance, on a response signal that is generated by a query pipeline, the following `fq` query params get translated into the multi-valued `filter` field:

- **Raw query parameters:**
  
  ```
  fq={!tag=format}format:(VHS)&fq={!tag=type}type:(Movie)
  ```

- **`filters_s` field (created by the SearchLogger component):**
  
  ```
  {!tag=format}format:(vhs) $ {!tag=type}type:(movie)
  ```

- **`filter` field:**
  
  ```
  "filter": ["format/VHS", "type/Movie"]
  ```

App Insights uses the `filter` field to generate various reports.

**Signal type ranking**

When you have defined some custom fields, it is useful to rank them according to how strongly they indicate a user's interest in an item. While it's not necessary to exclude certain signal types from the main signals collection, some can be excluded from signal aggregations in order to focus on the most important fields when generating recommendations.

How to get the list of signal types

1. In the Fusion UI, select your signals collection.
2. Open the Query Workbench by navigating to **Query > Query Workbench**.
3. Click **Add a field facet**.
4. Select the `type` field.

   The list of signal types appears in the facet panel:
Add a field facet

▼ type

product_click_plp (13894353)
search (10261620)
add_to_cart (6031138)
order_submission (2479417)
quick_view_click (806126)

View next 10
The default signals index pipeline

When indexing signals, Fusion uses a default index pipeline named `_signals_ingest` unless you specify a different index pipeline.

The `_signals_ingest` index pipeline has five stages:

1. Format Signals stage
2. Field Mapping stage
3. GeoIP Lookup stage
4. Solr Indexer stage
5. Update `has_clicks` flag stage

The Update `has_clicks` flag stage is an instance of the Update Related Document stage that updates the `has_clicks` flag to "true" on an existing request signal after the first click signal is processed for the request.

```
Find Related Document (RTG)
② id:$[request_id_s] AND has_clicks:false

Find Related Document (Query)
③ +query_id:"$[query_id]"+type:request +has_clicks:false

Sort Order
④ timestamp_tdt desc

Only lookup related docs when
① type == click

Fields to Update
* One or more fields to set on the document; the value can be pulled from the main document or simply a constant provided by this config

+ Parameter Name | Parameter Value
--- | ---
⑤ has_clicks | true
```

The update stage works as follows:

6. When a click signal is encountered (type == click)
7. Look at the incoming click signal for a field named `request_id_s`, which gets set by the Format Signals stage using a distributed cache of recently processed request signals.

If the `request_id_s` field is set, then send a real-time GET query to Solr to find a request signal with ID equal to the value of the `request_id_s` field on the click signal. To avoid re-updating request signals, the RTG query also filters on
has_clicks==false, which avoids duplicate atomic updates on the same document in Solr. Real-time GET is used to avoid timing issues between a request signal being sent to Solr and when it gets committed. This prevents missing updates when clicks occur soon after the initial request signal is sent by the search app.

8. If the click signal does not have the request_id_s field set, then do a normal Solr lookup for the request signal using: +query_id:"${query_id}" +type:request +has_clicks:false. A click signal may not have a request_id_s if there is a cache miss in the distributed cache used by the Format Signals stage.

9. If the stage performs a normal query, there may be multiple request signals that have the same query_id. This is because the query_id is based on session + query + filter, so if a user sends the same query + filter during the same session, there will be multiple request signals with the same query_id value. Thus, the stage sorts to get the latest request signal to update.

10. If a related document is found (in this case a request signal), then the stage updates the has_clicks field to true and performs an atomic update in Solr.

This stage performs its work in a background thread, so it does not impact the indexing performance of the click signal.
Deleting old signals

Signals are not automatically deleted by default, and over time they occupy an increasing amount of storage space.

To avoid running out of storage space as a result of your growing collection of signals, you must decide on a signals retention policy, then configure and schedule a REST Call job that regularly deletes old signals.

The duration for which signals should be kept depends on your use case and your organization's policies. For example, in some cases signals could be deleted after 30 days, while in other cases they may remain useful for a year, or even forever.

How to configure a REST Call job to delete old signals

1. Navigate to Collections > Jobs.
2. Click Add and select REST Call.

   The REST Call job configuration panel appears.

3. Enter an arbitrary ID for this job, such as "Delete-old-signals".
4. Enter the following endpoint URI, substituting the name of your signals collection for signalsCollectionName:

   solr://signalsCollectionName/update

5. In the Call Method field, select "post".
6. Under Query Parameters, enter the property name "wt" with the property value "json".
7. In the Request entity (as string) field, enter the following:

   `<root><delete><query>timestamp_tdt:[* TO NOW-2WEEKS]</query></delete><commit /></root>`

See Working with Dates for details about date formatting.

Your job configuration should look similar to this:
Delete-old-signals
Run arbitrary REST/HTTP/Solr command

ID: 
Delete-old-signals

ENDPOINT_URI: 
solr://Movie_Search_signals/update

CALL_METHOD: 
post

QUERY_PARAMETERS

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>wt</td>
<td>json</td>
</tr>
</tbody>
</table>

REQUEST_PROTOCOL_HEADERS

REQUEST_ENTITY (AS STRING)

<root><delete><query>timestamp_tdt:[* TO NOW-6MONTHS]*/query></delete><commit /></root>

Tip
You can configure a schedule for this job at System > Scheduler.
Aggregations

Aggregations compile Signals into a set of summaries that you can use to enrich the search experience through recommendations and boosting.

You can create two kinds of aggregations:

- **SQL aggregations** (strongly recommended) – SQL is a familiar query language that is well suited to data aggregation. Fusion’s new SQL Aggregation Engine has more power and flexibility than Fusion’s legacy aggregation engine.

- **Legacy aggregations** (deprecated) – This aggregation approach available in prior Fusion releases is still available, though it is deprecated and will be removed in a future release. Aggregator functions apply solely to legacy aggregations.

Aggregations are created automatically whenever you enable signals or recommendations. This topic explains how to create or modify aggregations individually. You can do this using the Fusion UI or the Jobs API. For more information, see Creating Aggregations.

Creating Aggregation Jobs

Aggregations are created automatically whenever you enable signals or recommendations. This topic explains how to create or modify aggregations individually. You can do this using the Fusion UI or the Jobs API.

Creating an aggregation job using the Fusion UI

An aggregation is a type of job. Aggregation jobs can be created or modified at Search > Jobs in the Fusion UI.

1. Navigate to **Search > Jobs**.
2. Click **Add**.
3. Select **Aggregation**.

The New Job Configuration panel appears.
4. Enter an arbitrary Spark job ID.

5. Enter the name of the signals collection to be aggregated.

   Note: Be sure to specify the signals collection (usually `<primarycollectionname>_signals`), not the primary `<primarycollectionname>` collection.

6. Under Aggregation Settings, click *include*.

7. Configure the aggregation parameters as needed.

   See Aggregation configuration parameters below for descriptions.

8. Click *Save*.

   The new aggregation job appears in the jobs list. Now you can run it or schedule it.

**Aggregation configuration parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupingFields</td>
<td>An array of strings specifying the fields to group on.</td>
</tr>
<tr>
<td>signalTypes</td>
<td>The signal types. If not set then any signal type is selected.</td>
</tr>
<tr>
<td>selectQuery</td>
<td>The query to select the desired signals. If not set then <code>:</code> will be used, or equivalent.</td>
</tr>
<tr>
<td>sort</td>
<td>The criteria to sort on within a group. If not set then sort order is by ID, ascending.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>timeRange</td>
<td>The time range to select signals on.</td>
</tr>
<tr>
<td>outputPipeline</td>
<td>What pipeline to use to process the output. If not set then _system pipeline will be used.</td>
</tr>
<tr>
<td>rollupPipeline</td>
<td>Pipeline to use for processing results of roll-up. This is by default the same index pipeline used for processing the aggregation results.</td>
</tr>
<tr>
<td>rollupAggregator</td>
<td>The aggregator to use when rolling up. If not set then the same aggregator will be used for roll-up.</td>
</tr>
<tr>
<td>outputCollection</td>
<td>The collection to write the aggregates to on output. This property is required if the selected output/rollup pipeline requires it (the default pipeline does). A special value of - disables the output.</td>
</tr>
<tr>
<td>aggregator</td>
<td>Aggregator implementation to use. This is either one of the symbolic names (simple, click, em) or a fully-qualified class name of a class extending EventAggregator. If not set then 'simple' is used.</td>
</tr>
<tr>
<td>sourceRemove</td>
<td>If true, the processed source signals will be removed after aggregation. Default is false.</td>
</tr>
<tr>
<td>sourceCatchup</td>
<td>If true, only aggregate the signals since the last time the job was successfully run. If there is a record of such previous run then this overrides the starting time of time range set in timeRange property.</td>
</tr>
<tr>
<td>outputRollup</td>
<td>Roll-up current results with all previous results for this aggregation id, which are available in outputCollection.</td>
</tr>
</tbody>
</table>
Aggregates

List of functions defining how to aggregate events with results. Aggregation functions have these properties:

- **type**
  - The function type defining how to aggregate events with results.
- **sourceFields**
  - The fields that the function will read from.
- **targetField**
  - The field that the function will write to.
- **mapper**
  - When true the function will be used in map phase only.
- **parameters**
  - Other parameters specific to individual functions.

StatsFields

List of numeric fields in results for which to compute overall statistics.

Parameters

Other aggregation parameters (such as start / aggregate / finish scripts, cache size, and so on).

Aggregator Functions

Aggregator Functions provide many ways to customize signals aggregations. These functions execute a specified operation on data coming from source event fields and accumulate the new value in a target field of the aggregated result.

Functions are implemented in an aggregator job definition, as a list within the `aggregates` property. Each function definition includes the function type, source fields, target fields, and additional parameters as needed for the function type. Specifically, each function takes the following properties (unless otherwise noted); additional parameters are noted in the function descriptions below.

- **type**: the function type.
- **sourceFields**: the list of fields from source events. Data will be retrieved from these fields as inputs to the function.
- **targetField**: the name of the target field where the aggregated result will be stored.
- **params**: any additional parameters for the specific function type, as described below.

The "sourceFields" and "targetField" field names in function specifications can be optionally prefixed with "event:" or "result:". If there are no prefixes the sourceFields take values from the current event being aggregated, and the
targetField takes (or updates) the value in the current partial aggregated result. With these prefixes values can be processed and e.g. the original event can be updated, or event fields can be considered taking into account the accumulated values in the result.

Examples:

Override default input field source:

```
"sourceField": "result:tweet_split_ss"
```

Override default target field source:

```
"targetField": "event:tweet_split_ss"
```

**Arithmetic Functions**

Arithmetic functions operate on all valid numeric values (including string fields that are parseable into double numbers) from source fields and compute a single result to the target field.

**sum**

A sum of numeric values, as a double number.

Example:

```
{
   "type": "sum",
   "sourceFields": [ "count_i" ],
   "targetField": "sum_count_d"
}
```

**sumOfLogs**

A sum of natural logs of numeric value, as a double number.

Example:

```
{
   "type": "sumOfLogs",
   "sourceFields": [ "script_d" ],
   "targetField": "script_sum_logs_d"
}
```

**sumOfSquares**

A sum of squares of numeric value, as a double number.

Example:
count

A count of source values, as a long number.

Example:

```json
{
  "type": "count",
  "sourceFields": [ "id" ],
  "targetField": "count_d"
}
```

goMean

A geometric mean of numeric values, as a double number.

Example:

```json
{
  "type": "geoMean",
  "sourceFields": [ "params.position_s" ],
  "targetField": "geoMean_position_d"
}
```

mean

An arithmetic mean of numeric values, as a double number.

Example:

```json
{
  "type": "mean",
  "sourceFields": [ "params.position_s" ],
  "targetField": "mean_position_d"
}
```

min

The minimum numeric value.

Example:
max

The maximum numeric value.

{  
  "type" : "max",  
  "sourceFields" : [ "params.position_s" ],  
  "targetField" : "max_position_d" 
}

decay_sum

A sum of time-based exponentially decayed numeric values. The difference between the aggregationTime and the event time will be decayed using an exponential function with a half-life equaling 30 days.

This function has some additional properties:

- halfLife: the number of seconds for the half-life decay function.
- timestampField: the name of the field that contains the source event's timestamp. By default, this is timestamp_dt.
- defaultWeight: the weight of an event if all values from source fields are missing. The default is 0.1f, and this is expressed as a float.

Example:

{  
  "type" : "decay_sum",  
  "sourceFields" : [ "weight_d" ],  
  "targetField" : "decay_sum_weight_d",  
  "params" : { } 
}

String Functions

String functions operate all values from source fields treated as strings.

cat

A concatenation of string values.

This function has some additional properties:

- separator: the character to use as a delimiter between values. The default is a single space.
- maxStringLength: the maximum length of the concatenated values (including separators). When this limit is exceeded, additional values are discarded. The default value is 10485760 characters (10 * 1024 * 1024).
• `maxValueCount`: the maximum number of values to concatenate. Any values collected after this limit are discarded. The default is 100.

Example:

```json
{
    "type": "cat",
    "sourceFields": [ "user_id_s" ],
    "targetField": "cat_user_id_txt",
    "params": { }
}
```

**ucat**

A concatenation of unique string values.

This function has some additional properties:

• `separator`: the character to use as a delimiter between values. The default is a single space.

• `maxStringLength`: the maximum length of the concatenated values (including separators). When this limit is exceeded, additional values are discarded. The default value is 10485760 characters (10 * 1024 * 1024).

• `maxValueCount`: the maximum number of values to concatenate. Any values collected after this limit are discarded. The default is 100.

Example:

```json
{
    "type": "ucat",
    "sourceFields": [ "user_id_s" ],
    "targetField": "ucat_user_id_txt",
    "params": { }
}
```

**split**

A simple regex-based string splitting function.

The following function params are supported:

• `regex` - (string, required) a regular expression used for splitting.

• `lower` - (boolean, optional, false by default) after the regex has been applied the resulting parts are optionally lower-cased (using US locale).

Example:
In the example above, the raw signal event field "query_s" is first split on whitespace and then lower-cased, and the result is put back into the raw signal event field "query_split".

**replace**

A simple regex-based string replace. The java.util.regex.Pattern syntax is supported for the regex matching and replacement.

The following function params are supported:

- **regex** - (string, required) a pattern to match.
- **replace** - (string, required) replacement.

Example:

```json
{
    "type": "replace",
    "sourcefields": [ "query_split" ],
    "targetfield": "event:query_split_clean",
    "params": {
        "regex": "\P{Alpha}+",
        "replace": "_"
    }
}
```

In the example above, this function takes the "query_split" values and replaces all non-alphabetic characters with underscores, and stores the result in the event's field "query_split_clean". As an extended example, this function would follow after the example split function and would add the field "query_split_clean" to the raw signal event. The "query_split_clean" field could be aggregated via other aggregation functions.

**Collection Functions**

Collection functions simply collect values from the source fields and add them as multiple values to the target field.

**discard**

This function discards all values from source fields and the target field. This modifies the source event and any in-progress aggregation result. This creates side-effects for subsequent functions, so should be used with care.

Example:
collect

Collect values from source fields.

This function has one additional property, 'maxValueCount', which defines the number of fields to collect from source fields. Any fields collected after this limit are discarded. The default is 100.

Example:

```json
{
  "type": "collect",
  "sourceFields": [ "user_id_s" ],
  "targetField": "collect_user_id_ss",
  "params": { }
}
```

ucollect Collect unique values from source fields.

This function has one additional property, 'maxValueCount', which defines the number of fields to collect from source fields. Any fields collected after this limit are discarded. The default is 100.

Example:

```json
{
  "type": "ucollect",
  "sourceFields": [ "user_id_s" ],
  "targetField": "unique_user_id_ss",
  "params": { }
}
```

Statistical Functions

Statistical functions compute scalar and matrix statistics. When the function has multiple results, such as for matrix or vector results, the data is stored in multiple fields.

variance The square of standard deviation of numeric values, as a double number.

Example:
```
{
    "type": "covariance",
    "sourceFields": [ "params.position_s", "position_rnd_1", "position_rnd_2" ],
    "targetField": "cov_position_d",
    "params": { }
}
```

**stddev**

The standard deviation of numeric values, as a double number.

Example:

```
{
    "type": "stddev",
    "sourceFields": [ "params.position_s" ],
    "targetField": "stddev_position_d",
    "params": { }
}
```

**cardinality**

An estimate of the number of unique elements in the set of values from source fields (which are treated as strings). This uses the HyperLogLog implementation.

This function has one additional property, 'error', which defines the acceptable probability of error from real value, specifically the standard deviation from real results. Smaller values cause exponentially higher RAM consumption during processing. For example, the default, 0.1, uses ~8Kb of RAM, while tests have shown 0.0001 uses ~64Mb.

Example:

```
{
    "type": "cardinality",
    "sourceFields": [ "params.position_s" ],
    "targetField": "cardinality_position_l",
    "params": { }
}
```

**skewness**

The measure of asymmetry of the distribution around its mean. This function is performed on numeric values and is expressed as a double number.

Example:

```
{
    "type": "skewness",
    "sourceFields": [ "params.position_s" ],
    "targetField": "skewness_position_d",
    "params": { }
}
```
**kurtosis**

The adjusted Pearson’s kurtosis of numeric values, expressed as a double. This provides a comparison of the shape of the distribution to that of the normal distribution.

Example:

```json
{
  "type": "kurtosis",
  "sourceFields": [ "params.position_s" ],
  "targetField": "kurtosis_position_d",
  "params": { }
}
```

**quantiles**

The quantiles of numeric values, stored as a double number in 0-N.targetField, or as a list of values in the target field (depending on the ‘multivalued’ property, described below). This implementation uses the T-Digest structure.

This function has the following additional properties:

- quantiles: the number of quantiles. The default is 10.
- multiValued: when true, all quantiles will be stored as multiple values in the target field. If false, then multiple values will be created in the format ‘0.targetField’ to ‘N.targetField’.

Example:

```json
{
  "type": "quantiles",
  "sourceFields": [ "params.position_s" ],
  "targetField": "quantiles_position_ss",
  "params": {
    "multiValued": true
  }
}
```

**topk**

An estimate of the top-K elements in the source fields and their frequency. The result is stored in three multi-valued fields, each with the same number of values. The three fields are:

- counts.targetField: integer counts (frequencies) of elements.
- values.targetField: elements.
- errors.targetField: estimation errors.

This function has one additional property, 'k', which is the number of elements to report. The default is 10.

Example:
covariance

A covariance matrix of numeric values from \( N > 1 \) source fields, with no smoothing. Missing or invalid values are treated as 0.0. A row of missing values is ignored. The resulting covariance matrix is stored in \( N \times (N - 1) \) fields following the naming pattern 'sourceField1.sourceField2.targetField'.

If source fields contain multiple values, only the first value from each source field will be used.

This implementation runs in a constant and small memory budget.

Example:

```json
{
    "type": "covariance",
    "sourceFields": [ "params.position_s", "position_rnd_1", "position_rnd_2" ],
    "targetField": "cov_position_d",
    "params": { }
}
```

correlation

A correlation matrix of numeric values from \( N > 1 \) source fields. This implementation is based on the covariance function. The resulting correlation matrix is stored in \( N \times (N - 1) \) fields following the naming pattern 'sourceField1.sourceField2.targetField'.

Example:

```json
{
    "type": "correlation",
    "sourceFields": [ "params.position_s", "position_rnd_1", "position_rnd_2" ],
    "targetField": "corr_position_d",
    "params": { }
}
```

histogram

An approximate histogram of values and their counts in source fields, using the T-Digest algorithm. Results are stored as corresponding multiple values in 'means.targetField' (for double values) and 'counts.targetField' (for integer values).

Example:

```json
{
    "type": "histogram",
    "sourceFields": [ "position_s", "position_rnd_1", "position_rnd_2" ],
    "targetField": "hist_position_d",
    "params": { }
}
```
sigmoid

This function uses hyperbolic tangent (tanh) to limit the impact of source values according to an s-shaped curve. The following parameters control the shape of the curve:

- **weight** - controls the range of values. Default weight is 1.0, which means that the sigmoid function values will range between (-1, 1). E.g. weight = 2.0 means that values will range between (-2, 2).

- **intercept** - sets the constant shift of function values. Default is 0, which means that sigmoid(0) = 0 and sigmoid(Inf) = 1. E.g. intercept = 2.0 means that sigmoid(0) = 2.0 and sigmoid(Inf) = 3.0.

- **slope** - this parameter controls the slope of the function, i.e. how quickly it reaches saturation. Default value is 1.0. E.g. slope = 2 will cause the function to saturate quickly, slope = 0.1 will cause the function to saturate for larger values of source.

- **final** - boolean, default is true. This controls how the sigmoid is applied to the source value. First, all numeric values from source fields are summed. Then, if final = false the current sum is passed to the sigmoid function and added to the previous total. If final = true then the current sum is added to the total and the sigmoid function is applied only at the end of the aggregation.

Example:

```json
{
    "type": "sigmoid",
    "sourceFields": [ "params.position_s" ],
    "targetField": "sigmoid_position_ss",
    "params": {
        "weight": 2.0,
        "intercept": 10.0,
        "slope": 0.5,
        "final": true
    }
}
```

Logical Functions

**when**

A logical function where processing will continue only if this function evaluates to true.

This function takes one additional property, `expr`, which is a JavaScript expression that must evaluate to a Boolean true/false. This property takes the same objects as the `expr` function, described above. If this property is missing, the function will evaluate to true when any sourceField or targetField is present.

Example:
unless

A logical function where processing will continue only if this function evaluates to false.

This function takes one additional property, 'expr', which is a JavaScript expression that must evaluate to a Boolean true/false. This property takes the same objects as the 'expr' function, described above. If this property is missing, the function will evaluate to false when any sourceField or targetField is present.

Example:

```json
{
  "type": "unless",
  "sourceFields": [ "params.position_s" ],
  "params": {
    "expr": "parseFloat(params_position_s) > 1"
  }
}
```

Scripting Functions

script

A scripted function. Scripts are evaluated as snippets, not as a function, and are expected to operate directly on the source event and the result. Their final values are discarded, since snippets in JavaScript are treated as expressions that evaluate to a specific value.

This function ignores the sourceFields and targetFields properties. Instead, the snippets are passed the following properties:

- startScript: the script defined is executed when the aggregation for the next unique tuple starts.
- aggregateScript: the script defined is executed for each source event.
- finishScript: the script is defined when all events for the current tuple have been processed and the result is about to be returned.

Example:
expr

A script expression. The script is evaluated as a snippet, and its final value is assigned to the targetField.

This function has only one additional property, 'expr', which contains the script expression.

Example:

```json
{  "type": "expr",  "sourceFields": [ "query_s", "filters_s" ],  "targetField": "expr_s",  "params": {    "expr": "v = ''; if (value != null) v = value + ' '; v + query_s + '_' + filters_s"  }}
```

Special Functions

noop

A function that does nothing (is non-operational). This is a fallback function when invalid function parameters or execution errors are encountered.

Example:

```json
{  "type": "noop"  }
```
SQL Aggregations

Fusion 4.0 introduces SQL aggregation for aggregating signals or other data. SQL is a familiar query language that is well suited to data aggregation. Fusion's new SQL Aggregation Engine has more power and flexibility than Fusion's legacy aggregation engine.

| Note | The aggregation approach available in prior Fusion releases is still available, though it is deprecated and will be removed in a future release. We now refer to the prior aggregation approach as "legacy aggregations." |

Advantages of SQL aggregation

These are advantages of SQL aggregation relative to legacy aggregation:

- **It's SQL!** – You can write SQL queries to aggregate your data.
- **Built-in aggregation functions** – A SQL query can use any of the functions provided by Spark SQL. Use these functions to perform complex aggregations or to enrich aggregation results.
- **A customizable time-decay function** – You can now customize the exponential time-decay function that Fusion uses for aggregations. Prior to release 4.0, Fusion used a fixed time-decay function to determine aggregation weights. In Fusion 4.0, the time-decay function is implemented as a UDAF, so you can easily implement your own time-decay function.
- **Aggregate data from many types of data sources** – You can use any asset in the Fusion Catalog as a data source. This lets you aggregate data from any data source supported by Spark.
- **Performance** – Although performance results can vary, Fusion SQL aggregations are roughly 5 times faster than legacy Fusion aggregations (using the default aggregation as the comparison).

Key features

Rollup SQL

Most aggregation jobs run with the catch-up flag set to true, which means that Fusion only computes aggregations for new signals that have arrived since the last time the job was run, and up to and including ref_time, which is usually the run time of the current job. Fusion must "roll up" the newly aggregated rows into any existing aggregated rows in the _aggr collection.

Fusion generates a basic rollup SQL script automatically, by consulting the schema of the aggregated documents. If your rollup logic is complex, you can provide a custom rollup SQL script.

This is an example of a rollup query:

```
SELECT query_s, doc_id_s, time_decay(1, timestamp_tdt, "30 days", ref_time, weight_d) AS weight_d, SUM(aggr_count_i) AS aggr_count_i
FROM `commerce_signals_aggr` GROUP BY query_s, doc_id_s
```

Time-range filtering

When Fusion rolls up new data into an aggregation, time-range filtering lets you ensure that Fusion doesn't aggregate the same data over and over again.
Fusion applies a time-range filter when loading rows from Solr, before executing the aggregation SQL statement. In other words, the SQL executes over rows that are already filtered by the appropriate time range for the aggregation job.

Notice that the examples Perform the Default SQL Aggregation and Use Different Weights Based on Signal Types don’t include a time-range filter. Fusion computes the time-range filter automatically as follows:

- If the catch-up flag is set to `true`, Fusion uses the last time the job was run and `ref_time` (which you typically set to the current time). This is equivalent to the WHERE clause `WHERE time > last_run_time AND time < ref_time`.
- If the catch-up flag isn’t set to `true`, Fusion uses a filter with `ref_time` (and no start time). This is equivalent to the WHERE clause `WHERE time < ref_time`.

The built-in time logic should suffice for most use cases. You can set the time range filter to `TO` and specify a WHERE clause filter to achieve more complex time based filtering.

**SQL functions**

A Spark SQL aggregation query can use any of the functions provided by Spark SQL. Use these functions to perform complex aggregations or to enrich aggregation results.

**Weight aggregated values using a time-decay function**

Fusion automatically uses a default `time_decay` function to compute and apply appropriate weights to aggregation groups during aggregation. Larger weights are assigned to more recent events. This reduces the impact of less-recent signals. Intuitively, older signals (and the user behavior they represent) should count less than newer signals.

If the default `time_decay` function doesn’t meet your needs, you can modify it. The `time_decay` function is implemented as a `UserDefinedAggregateFunction` (UDAF).

**Full function signature**

This is the UDAF signature of the default `time_decay` function:

```python
time_decay(count: Long,
          timestamp: Timestamp,
          halfLife: String (calendar interval),
          ref_time: Timestamp,
          weight_d: Double)
```

One small difference between the prior and current behavior is worth mentioning in passing:

- Prior to Release 4.0, the `decay_sum` aggregator function used the difference between the `aggregationTime` (the time at which the aggregation job is run) and the event time to calculate exponentially decayed numerical values.
- In Release 4.0, `time_decay` is a similar function. In `time_decay`, `ref_time` is used instead of `aggregationTime`. You can set `aggregationTime` to some other time than the run time of the aggregation job.

In practice, you will probably want to use `aggregationTime` as `ref_time`.

**Abbreviated function signature and default values**

Your function call can also use this abbreviated UDAF signature, that omits `halfLife, ref_time, and weight_d`. 
In this case, Fusion fills in these values for the omitted parameters: \( \text{halfLife} = 30 \text{ days}, \text{ref_time} = \text{NOW}, \) and \( \text{weight} = 0.1. \)

Matching legacy aggregation

To match the results of legacy aggregation, either use the abbreviated function signature or supply these values for the mentioned parameters: \( \text{halfLife} = 30 \text{ days}, \text{ref_time} = \text{NOW}, \) and \( \text{weight} = 0.1. \)

Parameters

Parameters for \( \text{time_decay} \) are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{count} )</td>
<td>Number of occurrences of the event. Typically, the increment is 1, though there is no reason it couldn't be some other number. In most cases, you simply pass ( \text{count}_i ), which is the event count field used by Fusion signals, as shown in the SQL aggregation examples.</td>
</tr>
<tr>
<td>( \text{timestamp} )</td>
<td>The date-and-time for the event. This time is the beginning of the interval used to calculate the time-based decay factor.</td>
</tr>
<tr>
<td>( \text{halfLife} )</td>
<td>Half life for the exponential decay that Fusion calculates. It is some interval of time, for example, ( 30 \text{ days} ) or ( 10 \text{ minutes} ). The interval prefix is optional. Fusion treats ( 30 \text{ days} ) as equivalent to ( \text{interval} 30 \text{ days} ).</td>
</tr>
<tr>
<td>( \text{ref_time} )</td>
<td>Reference time used to compute the age of an event for the time-decay computation. It is usually the time when the aggregation job runs (NOW). The reference time is not present in the data; Fusion determines the reference time at runtime. Fusion automatically attaches a ( \text{ref_time} ) column to every row before executing the SQL.</td>
</tr>
<tr>
<td>( \text{weight} )</td>
<td>Initial weight for an event, prior to the decay calculation. This value is typically not present in the signal data. You can use SQL to compute ( \text{weight} ); see Use Different Weights Based on Signal Types for an example.</td>
</tr>
</tbody>
</table>

Sample calculation of the age of a signal

This is an example of how Fusion calculates the age of a signal:
Imagine a SQL aggregation job that runs at Tuesday, July 11, 2017 1:00:00 AM (1499734800). For a signal with the timestamp Tuesday, July 11, 2017 12:00:00 AM (1499731200), the age of the signal in relation to the reference time is 1 hour.

**Built-in SQL aggregation jobs**

Enabling signals automatically creates the necessary _signals and _signals_aggr collections, plus several Parameterized SQL Aggregation jobs for signal processing and aggregation:

<table>
<thead>
<tr>
<th>Job</th>
<th>Default input collection</th>
<th>Default output collection</th>
<th>Default schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;collection&gt;_click_signals_aggregation</code></td>
<td><code>&lt;collection&gt;_signals</code></td>
<td><code>&lt;collection&gt;_signals_aggr</code></td>
<td>Every 15 minutes</td>
</tr>
<tr>
<td><code>&lt;collection&gt;_session_rollup</code></td>
<td><code>&lt;collection&gt;_signals</code></td>
<td><code>&lt;collection&gt;_signals</code></td>
<td>Every 15 minutes</td>
</tr>
<tr>
<td><code>&lt;collection&gt;_user_item_preferences_aggregation</code></td>
<td><code>&lt;collection&gt;_signals</code></td>
<td><code>&lt;collection&gt;_signals_aggr</code></td>
<td>Once per day</td>
</tr>
<tr>
<td><code>&lt;collection&gt;_user_query_history_aggregation</code></td>
<td><code>&lt;collection&gt;_signals</code></td>
<td><code>&lt;collection&gt;_signals_aggr</code></td>
<td>Once per day</td>
</tr>
</tbody>
</table>

When signals are enabled, you can view these jobs at Collections > Jobs. Each one is explained in more detail below.

**<collection>_click_signals_aggregation**

The `<collection>_click_signals_aggregation` job computes a time-decayed weight for each document, query, and filters group in the signals collection. Fusion computes the weight for each group using an exponential time-decay on signal count (30 day half-life) and a weighted sum based on the signal type. This approach gives more weight to a signal that represents a user purchasing an item than to a user just clicking on an item.

You can customize the the signal types and weights for this job by changing the `signalTypeWeights` SQL parameter in the Fusion Admin UI.

**SQL Parameters**

Parameters bound on the SQL template at runtime.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>signalTypeWeights</td>
<td>click:1.0,cart:10.0,purchase:25.0</td>
</tr>
</tbody>
</table>

When the SQL aggregation job runs, Fusion translates the `signalTypeWeights` parameter into a `WHERE IN` clause to filter signals by the specified types (click, cart, purchase), and also passes the parameter into the `weighted_sum` SQL function. Notice that Fusion only displays the SQL parameters and not the actual SQL for this job. This is to simplify the configuration because, in most cases, you only need to change the parameters and not worry about the actual SQL. However, if you need to change the SQL for this job, you can edit it under the **Advanced** toggle on the form.
Tip

A user can configure the `<collection>_click_signals_aggregation` job to use a parquet file as the source of raw signals instead of a signal Fusion collection.

1. Use catalog api to set up a “catalog project” in Fusion:

```bash
sample code:
curl -u <username>:<pw> -X POST -H "Content-type:application/json" --data-binary '{
  "name": "fusion_test",
  "assetType": "project",
  "description": "test",
  "cacheOnLoad": false
}' http://localhost:8764/api/catalog
```

2. Create an assets table in the project created in previous step:

```bash
sample code:
curl -u <username>:<pw> -X POST -H "Content-type:application/json" --data-binary '{
  "name": "doc_test",
  "assetType": "table",
  "projectId": "fusion_test",
  "description": "for documentation",
  "tags": ["fusion"],
  "format": "parquet",
  "cacheOnLoad": false,
  "options": ["path -> <path to your .parquet file>"]
}' http://localhost:8764/api/catalog/fusion_test/assets
```

| Note | The parquet file listed above needs to have all the fields in the SQL script which the `<collection>_click_signals_aggregation` job is selecting/using. |

3. In the `<collection>_click_signals_aggregation` job, change “source” from `${collection}_signals` to `catalog:${project_name}.${asset_name}` (e.g. `catalog:fusion_test.doc_test` per the sample code).
4. Start the job.

<collection>_session_rollup

The <collection>_session_rollup job aggregates related user activity into a session signal that contains activity count, duration, and keywords (based on user search terms). The Fusion App Insights application uses this job to show reports about user sessions. Use the elapsedSecsSinceLastActivity and elapsedSecsSinceSessionStart parameters to determine when a user session is considered to be complete. You can edit the SQL using the Advanced toggle.

The <collection>_session_rollup job uses signals as the input collection and output collection. Unlike other aggregation jobs that write aggregated documents to the <collection>_signals_aggr collection, the <collection>_session_rollup job creates session signals and saves them to the <collection>_signals collection.

<collection>_user_item_preferences_aggregation

The <collection>_user_item_preferences_aggregation job computes an aggregated weight for each user/item combination found in the signals collection. The weight for each group is computed using an exponential time-decay on signal count (30 day half-life) and a weighted sum based on the signal type.

Note

This job is a prerequisite for the ALS recommender job.

Job configuration tips:

- In the job configuration panel, click Advanced to see all of the available options.
- When aggregating signals for the first time, uncheck the Aggregate and Merge with Existing checkbox. In production, once the jobs are running automatically then this box can be checked. Note that if you want to discard older signals then by unchecking this box those old signals will essentially be replaced completely by the new ones.
- If the original signal data has missing fields, edit the SQL query to fill in missing values for fields such as “count_i” (the number of times a user interacted with an item in a session).
• Sometimes the aggregation job can run faster by unchecking the **Job Skip Check Enabled** box. Do this when first loading the signals.

• Use the *signalTypeWeights* SQL parameter to set the correct signal types and weights for your dataset. Its value is a comma-delimited list of signal types and their stakeholder-defined level of importance. Think of this numeric value as a weight that tells which type of signal is most important for determining a user’s interest in an item. An example of how to weight the signal types is shown below:

```
signal_type_1:1.0, signal_type_2: 3.0, signal_type_3: 20.0
```

Rank your signal types to determine which types should be added. Add only the signal types that are significant. Signal types that are not added to the list will not be included in the aggregation job, and for some signal types this is fine.

The weights should be within orders of magnitude of each other. The spread of values should not be wide. For instance, click:1.0, cart:100000.0 is too wide of a spread. The values of click:1.0 and cart:50.0 would be a reasonable setting, indicating that the signal type of cart is 50 times more important for measuring a user’s interest in an item.

• The **Time Range** field value is used in a weight decay function that reduces the importance of signals the older they are. This time range is in days and the default is 30 days. If you want to increase this time because the time duration of your signals is greater than 30 days, edit the SQL query to reflect the desired number of days. The SQL query is visible when you click **Advanced** in the job configuration panel. Modify the following line in the SQL query, changing "30 days" to your desired timeframe:

```
time_decay(count_i, timestamp_tdt, "30 days", ref_time, weight_d) AS typed_weight_d
```

If recommendations are enabled for your collection, then the ALS recommender job is automatically created with the name `<collection>_item_recommendations` and scheduled to run after this job completes. Consequently, you should only run this aggregation once or twice a day, because training a recommender model is a complex, long-running job that requires significant resources from your Fusion cluster.

**<collection>_user_query_history_aggregation**

The `<collection>_user_query_history_aggregation` job computes an aggregated weight for each user/query combination found in the signals collection. The weight for each group is computed using an exponential time-decay on signal count (30 day half-life) and a weighted sum based on the signal type. Use the *signalTypeWeights* parameter to set the correct signal types and weights for your dataset. You can use the results of this job to boost queries for a user based on their past query activity.

**Join signals with item metadata**

Fusion’s basic aggregation jobs aggregate using the document ID. You can also aggregate at a more coarse-grained level using other fields available for documents (item metadata), such as manufacturer or brand for products. Aggregating with item metadata is useful for building personalization boosts into your search application.

The following PUT request creates additional aggregation jobs that join signals with the primary `products` collection to compute an aggregated weight for a `manufacturer` field:
After performing the PUT request shown above, you will have two additional aggregation jobs in Fusion.

**<collection>_user_<metadata>_preferences_aggregation**

This job computes an aggregated weight for each user/item metadata combination, e.g. user/manufacturer, found in the signals collection. Fusion computes the weight for each group using an exponential time-decay on signal count (30 day half-life) and a weighted sum based on the signal type. Use the `signalTypeWeights` parameter to set the correct signal types and weights for your dataset. Use the `primaryCollectionMetadataField` parameter to set the name of a field from the primary collection to join into the results, e.g. `manufacturer`. You can use the results of this job to boost queries based on user preferences regarding item-specific metadata such as manufacturer (e.g. Ford vs. BMW) or brand (e.g. Ralph Lauren vs. Abercrombie & Fitch).

**<collection>_query_<metadata>_preferences_aggregation**

This job computes an aggregated weight for each query/item metadata combination, e.g. query/manufacturer, found in the signals collection. Fusion computes the weight for each group using an exponential time-decay on signal count (30 day half-life) and a weighted sum based on the signal type. Use the `signalTypeWeights` parameter to set the correct signal types and weights for your dataset. Use the `primaryCollectionMetadataField` parameter to set the name of a field from the primary collection to join into the results.

| Tip | These additional item/item metadata aggregation jobs also serve as examples of how to join between the signals and primary collections to perform aggregations on fields other than the document ID. You can re-execute the same PUT request shown above using a different metadata field name in the `metadata_column` parameter. |

**Write SQL aggregations**

In this section, we provide guidance to help you make the most of the Fusion SQL aggregation engine.

**Project fields into the signals_aggr collection**

For legacy reasons, the `<collection>_signals_aggr` collection relies on dynamic field names, such as `doc_id_s` and `query_s` instead of `doc_id` and `query`. Consequently, when you project fields to be written to the `<collection>_signals_aggr` collection, you should use dynamic field suffixes as shown in the SQL snippet below:

```sql
SELECT SUM(typed_aggr_count_i) AS aggr_count_i,
quad AS query_s,
query AS query_t,
quad AS doc_id_s,
filters AS filters_s,
SPLIT(filters, ' \$ ') AS filters_ss,
weighted_sum(...) AS weight_d
FROM signal_type_groups
GROUP BY query, doc_id, filters_s
```

You're not required to use this approach, but if you don't use dynamic field suffixes as shown above, you'll need to...
change the boosting stages in Fusion to work with different field names.

**Use WITH to organize complex queries**

A common pattern in SQL aggregation queries is the use of subqueries to break up the logic into comprehensible units. For more information about the WITH clause, see https://modern-sql.com/feature/with. Let's work through an example to illustrate the key points:

```sql
1:  WITH signal_type_groups AS (
2:      SELECT SUM(count_i) AS typed_aggr_count_i,
3:             doc_id,
4:             user_id,
5:             type,
6:             time_decay(count_i, timestamp_tdt) AS typed_weight_d
7:        FROM product_signals
8:       WHERE type IN ('click','cart','purchase')
9:    GROUP BY user_id, doc_id, type
10: ) SELECT SUM(typed_aggr_count_i) AS aggr_count_i,
11:          doc_id AS doc_id_s,
12:          user_id AS user_id_s,
13:      weighted_sum(...) AS weight_d
14:     FROM signal_type_groups
15: GROUP BY doc_id, user_id
```

- At line 1, we declare a statement scoped view named `signal_type_groups` using the WITH keyword.
- Lines 2-9 define the subquery for the `signal_type_groups` view.
- At line 7, we read from the `product_signals` collection in Fusion.
- Line 8 filters the input to only include `click`, `cart`, and `purchase` signals. Behind the scenes, Fusion translates this WHERE IN clause to a Solr filter query, e.g. `fq=type:(click OR cart OR purchase)`. The `signal_type_groups` view produces rows grouped by `user_id`, `doc_id`, and `type` (line 9).
- Starting at line 10, we define a subquery that performs a rollup over the rows in the `signal_type_groups` view by grouping on `doc_id` and `user_id` (line 15). Notice how the WITH statement helps break this complex query up into two units that help make aggregation queries easier to comprehend. You are encouraged to adopt this pattern in your own SQL aggregation queries.

**Built-in SQL Functions**

In addition to the SQL functions provided by Spark, Fusion provides several additional functions to simplify common aggregation tasks. To recap, a UDAF aggregates multiple rows for the same group by key and a UDF performs some operation on a single row.

`weighted_sum`

The `weighted_sum` UDAF takes a weight, type, and type-weight mapping to produce an aggregated weight. For example, consider the following SQL snippet:
SELECT query, 
doc_id, 
filters, 
weighted_sum(typed_weight_d, type, 'click:1.0,cart:10.0') AS weight_d 
FROM signal_type_groups 
GROUP BY query, doc_id, filters

When applied to the rows in the table below, the `weighted_sum` function produces a final `weight_d` of 12.0 (2*1.0 + 1*10.0). The UDAF is passed rows grouped by `query`, `doc_id`, and `filters`.

<table>
<thead>
<tr>
<th>query</th>
<th>type</th>
<th>doc_id</th>
<th>filters</th>
<th>typed_weight_d</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPad</td>
<td>click</td>
<td>1</td>
<td>gear</td>
<td>2</td>
</tr>
<tr>
<td>iPad</td>
<td>cart</td>
<td>1</td>
<td>gear</td>
<td>1</td>
</tr>
</tbody>
</table>

timediff

The `timediff` UDF computes the difference, in milliseconds, between to timestamps in the same row. From the `session_rollup` job, the `timediff` function computes the difference between the current time and the last activity in a session.

click_pos

The `click_pos` UDF computes either a reciprocal rank or a raw click position (using a 0-based index) of a document in a page of results. This UDF is used to compute the mean reciprocal rank (MRR) for experiments. For example, given the following list of documents and a doc ID, the `click_pos` UDF will return 2:

```
docs: a,b,c,d
doc ID: c
```

concat_text

The `concat_text` UDF combines multivalued text fields coming from Solr into a field with a single value delimited by spaces. This UDF is useful when a field returned from Solr uses the `_txt` suffix, which indicates a multivalued text field.

Create and run a SQL aggregation job

You can perform a SQL aggregation on a signals collection for a datasource (or on some other collection), through the Fusion UI or using the Fusion API.

Preliminaries

Before you can create and run a SQL aggregation job, you must create an app, create a collection (or use the default collection for the app), and add a datasource. Before you run a SQL aggregation job, you need signal data. Otherwise, there is nothing to aggregate.

Use the Fusion UI

You can use the Fusion UI to perform a SQL aggregation.
Set up a SQL aggregation job

1. With the app open, navigate to Collections > Jobs.
2. Click Add and select Aggregation from the dropdown list.
3. Specify an arbitrary Spark Job ID.
4. For the Source Collection, select the collection that contains the data to aggregate.

| Tip | This is not the base collection. For example, to aggregate the signals in experiment_signals, you would select experiment_signals, not experiment. |

5. Under Aggregation Settings, expand SQL Aggregation.
6. Enter or paste SQL in the SQL text box. Optionally, click Open Editor to open a dialog box with a larger editor. Click Close to close the dialog box.
7. Click Save to save the job.

Run a SQL aggregation job

1. With the app open, navigate to Collections > Jobs.
2. In the list of jobs, click the job you want to run, and then click Run.
3. Click Start.
4. Click Close ✗ to close the job management part of the Fusion UI.

Use the Fusion API

You can use the Fusion API to perform a SQL aggregation. For example:

Configure a SQL aggregation job:
Run a SQL aggregation job:

curl -u user:pass -X POST -H "Content-Type: application/json" 
http://localhost:8764/api/jobs/spark:experiment_click_signals_aggregation/actions -d '{"action": "start"}"

SQL aggregation examples

To acquaint you with how to use Spark SQL in SQL aggregations, here are several examples:

Perform the default SQL aggregation

This is the default SQL aggregation of signals for a base collection named "products". It produces the same results as legacy aggregation:

```
SELECT SUM(count_i) AS aggr_count_i,
       query AS query_s,
       doc_id AS doc_id_s,
       time_decay(count_i, date) AS weight_d
FROM products_signals
GROUP BY query, doc_id
```

Notice the following about this SQL:

- `SELECT SUM(count_i) AS aggr_count_i` - count_i is summed as aggr_count_i.
- `time_decay(count_i, date) AS weight_d` - The `time_decay` function computes the aggregated weight_d field. This function is a Spark `UserDefinedAggregateFunction` (UDAF) that is built into Fusion. The function computes a weight for each aggregation group, using the count and an exponential decay on the signal timestamp, with a 30-day half life.
**GROUP BY query, doc_id** – The GROUP BY clause defines the fields used to compute aggregate metrics, which are typically the query, doc_id, and any filters. With SQL, you have more options to compute aggregated metrics without having to write custom JavaScript functions (which would be needed to supplement legacy aggregations). You can also use standard WHERE clause semantics, for example, \texttt{WHERE type_s = 'add'} , to provide fine-grained filters.

- The \texttt{time_decay} function uses an abbreviated function signature, \texttt{time_decay(count_i, timestamp_tdt)}, instead of the full function signature shown in Use Different Weights Based on Signal Types.

An example of how SQL aggregation works

This is an example of how this aggregation works. Consider the following four input signals for a fictitious query \texttt{q1} and document \texttt{1}:

```json
[{  "type_s": "add",  "doc_id_s": "1",  "query_s": "q1",  "count_i": 1,  "timestamp_tdt": "2017-07-11T00:00:00Z"},
 {  "type_s": "view",  "doc_id_s": "1",  "query_s": "q1",  "count_i": 1,  "timestamp_tdt": "2017-07-11T00:00:00Z"},
 {  "type_s": "add",  "doc_id_s": "1",  "query_s": "q1",  "count_i": 1,  "timestamp_tdt": "2017-07-11T00:00:00Z"},
 {  "type_s": "view",  "doc_id_s": "1",  "query_s": "q1",  "count_i": 1,  "timestamp_tdt": "2017-07-11T00:00:00Z"}]
```

Fusion generates the following aggregated document for \texttt{q1}:

```json
{  "aggr_count_i": 4,  "query_s": "q1",  "doc_id_s": "1",  "weight_d": 0.36644220285922535,  "aggr_id_s": "products_sql_agg",  "aggr_job_id_s": "15d4279d1287155e5137",  "flag_s": "aggr",  "query_t": "q1",  "aggr_type_s": "sql",  "timestamp_tdt": "2017-07-14T19:01:05.950Z"}
```
Use different weights based on signal types

This is a slightly more complex example that uses a subquery to compute a custom weight for each signal based on the signal type (add vs. click):

```
SELECT SUM(count_i) AS aggr_count_i,
       query_s,
       doc_id_s,
       time_decay(count_i, timestamp_tdt, "5 days", ref_time, signal_weight) AS weight_d
FROM (SELECT count_i,
       query_s,
       doc_id_s,
       timestamp_tdt,
       ref_time,
       CASE WHEN type_s='add' THEN 0.25 ELSE 0.1 END AS signal_weight
       FROM products_signals)
GROUP BY query_s, doc_id_s
```

Compute metrics for sessions

This aggregation query uses a number of Spark SQL functions to compute some metrics for sessions:

```
SELECT concat_ws('||', clientip, session_id) as id,
       first(clientip) as clientip,
       min(ts) as session_start,
       max(ts) as session_end,
       (unix_timestamp(max(ts)) - unix_timestamp(min(ts))) as session_len_secs_l,
       sum(asInt(bytes)) as total_bytes_l,
       count(*) as total_requests_l
FROM sessions
GROUP BY clientip, session_id
```

Legacy Aggregations

<table>
<thead>
<tr>
<th>Note</th>
<th>This aggregation approach is still available, though it is deprecated and will be removed in a future release. We now refer to this aggregation approach as &quot;legacy aggregations.&quot;</th>
</tr>
</thead>
</table>

Signals are most useful when they are aggregated into a set of summaries that can be used to enrich the search experience through recommendations and boosting.

Aggregation jobs are a subtype of Spark jobs.

When signals are enabled for a "primary" collection, a `<primarycollectionname>_signals` collection and a `<primarycollectionname>_signals_aggr` collection are created automatically.

Aggregation Pipelines

Aggregated events are indexed, and use a default pipeline named "aggr_rollup". This pipeline contains one stage, a Solr Indexer stage to index the aggregated events.
You can create your own custom index pipeline to process aggregated events differently if you choose.

**Aggregation Functions**

The section Aggregator Functions documents the available set of aggregation functions.

Custom aggregation functions can be defined via a JavaScript stage.

**Aggregation properties**

The aggregation process is specified by an aggregation type consisting of the following list of properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Aggregation ID</td>
</tr>
<tr>
<td>groupingFields</td>
<td>List of signal field names</td>
</tr>
<tr>
<td>signalTypes</td>
<td>List of signal types</td>
</tr>
<tr>
<td>aggregator</td>
<td>Symbolic name of the aggregator implementation</td>
</tr>
<tr>
<td>selectQuery</td>
<td>Query string, default :</td>
</tr>
<tr>
<td>sort</td>
<td>Ordering of aggregated signals</td>
</tr>
<tr>
<td>timeRange</td>
<td>String specifying time range, e.g., [* TO NOW]</td>
</tr>
<tr>
<td>outputPipeline</td>
<td>Pipeline ID for processing aggregated events</td>
</tr>
<tr>
<td>outputCollection</td>
<td>Output collection name</td>
</tr>
<tr>
<td>rollupPipeline</td>
<td>Rollup pipeline ID</td>
</tr>
<tr>
<td>rollupAggregator</td>
<td>Name of the aggregator implementation used for rollups</td>
</tr>
<tr>
<td>sourceRemove</td>
<td>Boolean, default is false</td>
</tr>
<tr>
<td>sourceCatchup</td>
<td>Boolean, default is true</td>
</tr>
<tr>
<td>outputRollup</td>
<td>Boolean, default is true</td>
</tr>
<tr>
<td>aggregates</td>
<td>List of aggregation functions</td>
</tr>
<tr>
<td>params</td>
<td>Arbitrary parameters to be used by specific aggregator implementations</td>
</tr>
</tbody>
</table>

**Aggregation job configuration**

The groupingFields should use just `user_id_s`, and optionally the "sort" parameter should be set to `timestamp_tdt asc` - this way the sessionization process will work most efficiently. On the other hand, sorting by timestamp requires more work on the Solr-side, so it may be omitted, with the possible side-effect that there will be additional partial documents created.
Recommendations and Boosting

Signals contain data about how users interact with search results. Once your Fusion app has accumulated a sufficient number of aggregated signals, they become useful for automatically producing recommendations and boosts for better relevance and higher conversion rates.

The same data used to produce recommendations can also be used for automatic boosting. So although recommendations and boosts are different (as explained below), these topics use “recommender data” to refer to the data that can be used for boosting as well as for recommendations.
# Recommendations vs boosts

Recommendations and boosts are two ways of presenting AI-powered estimations about which items are most likely to interest a user:

<table>
<thead>
<tr>
<th>Recommendations</th>
<th>Boosts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personalized search results from a special, automatic query, <em>whether or not the user has performed a query.</em></td>
<td>Modifications to the scores of the items in a query's search results.</td>
</tr>
<tr>
<td>Recommendations can be based on a variety of criteria, as in the examples below:</td>
<td>The set of search results does not change, but their ranking does. For example:</td>
</tr>
<tr>
<td>• &quot;Based on your purchase history, you might be interested in these items.&quot;</td>
<td>• Boost the most popular items</td>
</tr>
<tr>
<td>• &quot;People who viewed this item also viewed these items.&quot;</td>
<td>• Boost items this user has clicked before</td>
</tr>
<tr>
<td>• &quot;People who searched for 'ipad' also searched for these items.&quot;</td>
<td>• Boost seasonal items</td>
</tr>
</tbody>
</table>

Boosts can also be configured manually, using the Query Workbench or the Query Rewriting UI. You can also perform boosts within a set of recommendations.

These topics explain how to configure and use recommendations and boosts:

- **Getting Started** shows you how to quickly enable the basic feature set and see some results.
- **Recommendation Methods** explains the available approaches to recommendations and boosting, including methods that do not rely on signals.
Recommendations data flow

The diagram below shows the flow of data between the default objects created when you enable recommendations. You can create additional objects for different recommendation methods.

1. **Signals data** is aggregated into a format that can be consumed by recommender jobs.

   Your search application should be sending well-formed signals data to Fusion whenever there is user activity on your site.

   By default, the `_click_signals_aggregation` job runs every 15 minutes while the `_user_item_preferences_aggregation` and `_user_query_history_aggregation` jobs run once per day.

   You can use signals data for boosting, without enabling recommendations, using the Boost with Signals query pipeline stage. The default query pipeline includes this stage.

2. **Recommender data** is produced when recommender jobs analyze aggregated signals.

   The default `_item_recommendations` job must be manually scheduled or run on demand. Make sure that the `_user_item_preferences_aggregation` job runs first.

3. Query pipelines retrieve recommender data to produce recommendations and boosts.

   When recommendations are enabled, Fusion creates two query pipelines for recommendations: `_items_for_user_recommendations` _items_for_item_recommendations_
Recommendations storage requirements

Collaborative recommendations are derived from data generated by Fusion AI jobs and stored in special collections. You can estimate the required storage for this data if you know the following:

- the number of unique users in the signals collection (num_users)

To find the number of unique values of the user_id field, navigate to Analytics > App Insights > Analytics > Users. The Number of Distinct Data Values panel displays the number of unique users:

![Number of Distinct Data Values](image)

If no value is displayed here or the value is not accurate, see Important fields for signals to verify that your signals data is well-formed.

- the number of unique items in the main collection (num_items)

Navigate to Collections > Collections Manager to see the total number of documents in your main collection.

- the number of recommendations to be computed per user (num_recs_per_user)

This is a configuration parameter in the recommender job, with a default value of 10.

- the number of item similarities to be computed per item (num_item_sims)

This is a configuration parameter in the recommender job, with a default value of 10.

Every time the recommender job runs, this many new recommender documents are created:

\[ \text{num_users} \times \text{num_recs_per_user} + \text{num_items} \times \text{num_item_sims} \]

Each of these documents is about 300 bytes. This is the minimum size; they can be larger if you configure the recommender job to pull item metadata fields from the main collection for inclusion in recommender documents. These fields are useful when displaying recommendations in your front-end search application.
| Tip | In your recommender jobs, verify that **Delete Old Recommendations** is selected so that only the latest recommender data is stored. |

Tip

In your recommender jobs, verify that **Delete Old Recommendations** is selected so that only the latest recommender data is stored.

Toggle navigation

![Lucidworks Logo](https://lucidworks.com/)

- **Fusion Platform**

  [https://lucidworks.com/products/fusion-server/](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.
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
- Government
  - Resources
  - Blog
  - Try It Now
  - Search

CONTACT US

Products

Fusion AI 4.2
• Fusion Platform

https://lucidworks.com/products/fusion-server/

Highly scalable search engine and NoSQL datastore.

Learn more ›

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
- Government
  - Resources
  - Blog
  - Try It Now
  - Search

CONTACT US

Products

Fusion AI 4.2

Overview
Redirecting...

Click here if you are not redirected.
• Fusion Platform

https://lucidworks.com/products/fusion-server/

Highly scalable search engine and NoSQL datastore.

Learn more ›

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
- Government
  - Resources
  - Blog
  - Try It Now
  - Search

CONTACT US

**Products**

Fusion AI 4.2

<a href="https://www.activate-conf.com/?utm_source=lw_docs&amp;utm_medium=website&amp;utm_campaign=activate19">Overview</a>
• Fusion Platform
  
  https://lucidworks.com/products/fusion-server/

Highly scalable search engine and NoSQL datastore.

Learn more ›

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
  - Government
    - Resources
    - Blog
    - Try It Now
    - Search

CONTACT US

**Products**

Fusion AI 4.2

<ahref="https://www.activate-conf.com/?utm_source=lw_docs&amp;utm_medium=website&amp;utm_campaign=activate19">img src="/home/jenkins/jenkins-slave/workspace/Fusion/Fusion-docs/Fusion-docs-publish-docker/pdfs/assets/images/site/Activate_Ads_Early_Reg.jpg" format="jpg" alt="[image]" tmp="false">

**Overview**
- **Fusion Platform**
  
  [https://lucidworks.com/products/fusion-server/](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 ›

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
- Government
  - Resources
  - Blog
  - Try It Now
  - Search

CONTACT US

**Products**

Fusion AI 4.2

<a href="https://www.activate-conf.com/?utm_source=lw_docs&amp;utm_medium=website&amp;utm_campaign=activate19"><img src="/home/jenkins/jenkins-slave/workspace/Fusion/Fusion-docs/Fusion-docs-publish-docker/pdfs/assets/images/site/Activate_Ads_Early_Reg.jpg" alt="[image]" tmp="false"></a>

**Overview**
Spark and Machine Learning

Apache Spark is an open-source cluster-computing framework that serves as a fast and general execution engine for large-scale data processing jobs that can be decomposed into stepwise tasks, which are distributed across a cluster of networked computers.

Spark improves on previous MapReduce implementations by using resilient distributed datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

Fusion manages a Spark cluster that is used for all signal aggregation processes.

With a Fusion AI license, you can also use the Spark cluster to train and compile machine learning models, as well as to run experiments via the Fusion UI or the Spark Jobs API.

These topics provide information about Spark administration and machine learning:

- Spark Components – Spark integration in Fusion, including a diagram
- Spark Getting Started – Starting Spark processes and working with the shell and the Spark UI
- Spark Driver Processes – Fusion jobs run on Spark use a driver process started by the API service
- Spark Configuration – How to configure Spark for maximum performance. The article also provides information about ports, directories, and configuring connections for an SSL-enabled Solr cluster.
- Scaling Spark Aggregations – How to configure Spark so that aggregations scale
- Spark Troubleshooting – How to troubleshoot Spark
- Machine Learning Models in Fusion - Example of using a Spark cluster to train a sentiment classifier for tweets
Further Reading

- Machine Learning in Lucidworks Fusion
- Apache Spark Key Terms, Explained
- Apache Spark on Wikipedia
Spark Components

This diagram shows the Spark components available from Fusion:
Spark components in Fusion

- **Application**: An active SparkContext in the Spark Master web UI, which consists of a classpath and a configuration. Jobs submitted to the cluster always run as classes in a specific application, that is, using the application’s classpath and configuration.

- **SparkDriver**: The Spark driver program, a JVM process launched by the Fusion API service to execute Fusion jobs in Spark. SparkDriver creates and manages SparkContext for the Fusion application, and stops SparkContext when it’s no longer needed.

- **Spark master** (*spark-master*): Agent-managed Fusion service that coordinates worker processes and applications in a Spark cluster. You should run at least 2 spark-master processes per cluster to achieve high-availability. ZooKeeper determines which spark-master process is the leader and handles fail-over.

- **Spark worker** (*spark-worker*): Agent-managed Fusion service that launches executors for Spark applications. Spark-workers communicate with the master to launch executors for an application.

- **SQL service** (*sql*): Agent-managed Fusion service that runs Spark’s thrift-based SQL engine. It provides JDBC access to a Spark cluster.

- **Spark shell** (*spark-shell*): Wrapper script provided with Fusion to launch the Spark Scala REPL shell with the correct master URL (pulled from Fusion’s API) and a shaded Fusion JAR added.

- **Custom script job**: A Fusion job that executes a custom Scala script using the Spark shell.

- **Spark Job Workbench**: A toolkit provided by Lucidworks to help build custom Spark jobs using Scala, Java, or Python. See Spark Job Workbench.

- **CoarseGrainedExecutorBackend**: Executor process(es) launched by a spark-worker to execute the tasks for a specific application, such as the spark-shell.

- **Shaded JAR**: The Fusion API service creates an assembly jar (also call an uber jar) that contains all of the dependencies needed to use spark-solr and Fusion classes within a Spark job.

Classes that conflict with classes on Spark’s classpath are shaded to ensure that Fusion classes use the correct version.
Spark Getting Started

The public GitHub repository Fusion Spark Bootcamp contains examples and labs for learning how to use Fusion’s Spark features.

In this section, you’ll walk through some basic concepts of using Spark in Fusion. For more exposure, you should work through the labs in the Fusion Spark Bootcamp.

Starting the Spark Master and Spark Worker services

The Fusion run script /opt/fusion/4.2.x/bin/fusion (on Unix) or C:\lucidworks\fusion\4.2.x\bin\fusion.cmd (on Windows) doesn’t start the spark-master and spark-worker processes. This reduces the number of Java processes needed to run Fusion and therefore reduces memory and CPU consumption.

Jobs that depend on Spark, for example, aggregations, will still execute in what Spark calls local mode. When in local mode, Spark executes tasks in-process in the driver application JVM. Local mode is intended for jobs that consume/produce small datasets.

One caveat about using local mode is that a persistent Spark UI is not available. But you can access the driver/job application UI at port :4040 while the local SparkContext is running.

To scale Spark in Fusion to support larger data sets and to speed up processing, you should start the spark-master and spark-worker services.

On Unix:

```
./spark-master start
./spark-worker start
```

On Windows:

```
spark-master.cmd start
spark-worker.cmd start
```

Give these commands from the bin directory below the Fusion home directory, for example, /opt/fusion/4.2.x (on Unix) or C:\lucidworks\fusion\4.2.x (on Windows).

<table>
<thead>
<tr>
<th>Tip</th>
<th>To have the spark-master and spark-worker processes start and stop with bin/fusion start and bin/fusion stop (on Unix) or bin\fusion.cmd start and bin\fusion.cmd stop (on Windows), add them to the group.default definition in fusion.properties. For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>group.default = zookeeper, solr, api, connectors-classic, connectors-rpc, proxy, webapps, admin-ui, log-shipper, spark-master, spark-worker</td>
</tr>
</tbody>
</table>

Viewing the Spark Master

After starting the master and worker services, direct your browser to http://localhost:8767 to view the Spark master web UI, which should resemble this:

![Spark Master at spark://192.168.1.9:8766](image)

If you don’t see the master UI and at least one worker in the ALIVE state, check these logs.

**On Unix:**

```
/opt/fusion/4.2.x/var/log/spark-master/spark-master.log
/opt/fusion/4.2.x/var/log/spark-worker/spark-worker.log
```

**On Windows:**

```
C:\lucidworks\fusion\4.2.x\var\log\spark-master\spark-master.log
C:\lucidworks\fusion\4.2.x\var\log\spark-worker\spark-worker.log
```

Use this Fusion API request to get the status of the Spark master:

```
curl http://localhost:8764/api/spark/master/status
```

This request should return a response of the form:

```
[ {
  "url" : "spark://192.168.1.9:8766",
  "status" : "ALIVE",
  "workers" : [ {
    "id" : "worker-20161005175058-192.168.1.9-8769",
    "host" : "192.168.1.9",
    "port" : 8769,
    "webuiaddress" : "http://192.168.1.9:8770",
    "cores" : 8,
    "coresused" : 0,
    "coresfree" : 8,
    "memoryused" : 0,
    "memoryfree" : 2048,
    "state" : "ALIVE",
    "lastheartbeat" : 1475711489460
  } ], ...
```
If you have multiple Spark masters running in a Fusion cluster, each will be shown in the status but only one will be ALIVE; the other masters will be in STANDBY mode.

| Tip | If you are operating a multi-node Spark cluster, we recommend running multiple Spark master processes to achieve high-availability. If the active one fails, the standby will take over. |

**Running a job in the Spark shell**

After you have started the Spark master and Spark worker, run the Fusion Spark shell.

**On Unix:**

```
./spark-shell
```

**On Windows:**

```
spark-shell.cmd
```

Give these commands from the bin directory below the Fusion home directory, for example, `/opt/fusion/4.2.x` (on Unix) or `C:\lucidworks\fusion\4.2.x` (on Windows).

The shell can take a few minutes to load the first time because the script needs to download the shaded Fusion JAR file from the API service.

If ports are locked down between Fusion nodes, specify the Spark driver and BlockManager ports, for example:

**On Unix:**

```
./spark-shell --conf spark.driver.port=8772 --conf spark.blockManager.port=8788
```

**On Windows:**

```
spark-shell.cmd --conf spark.driver.port=8772 --conf spark.blockManager.port=8788
```

When the Spark shell is initialized, you'll see the prompt:

```
scala>
```

Type `:paste` to activate paste mode in the shell and paste in the following Scala code:
val readFromSolrOpts = Map(
  "collection" -> "system_logs",
  "fields" -> "host_s,level_s,type_s,message_txt,thread_s,timestamp_tdt",
  "query" -> "level_s:[* TO *]"
)
val logsDF = spark.read.format("solr").options(readFromSolrOpts).load
logsDF.registerTempTable("fusion_logs")
var sqlDF = spark.sql(""
  | SELECT COUNT(*) as num_values, level_s as level
  | FROM fusion_logs
  | GROUP BY level_s
  | ORDER BY num_values desc
  | LIMIT 10"").stripMargin
sqlDF.show(10,false)

Press CTRL+D to execute the script. Your results should resemble these results:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
val readFromSolrOpts = Map(
  "collection" -> "system_logs",
  "fields" -> "host_s,level_s,type_s,message_txt,thread_s,timestamp_tdt",
  "query" -> "level_s:[* TO *]"
)
val logsDF = spark.read.format("solr").options(readFromSolrOpts).load
logsDF.registerTempTable("fusion_logs")
var sqlDF = spark.sql(""
  | SELECT COUNT(*) as num_values, level_s as level
  | FROM fusion_logs
  | GROUP BY level_s
  | ORDER BY num_values desc
  | LIMIT 10"").stripMargin
sqlDF.show(10,false)

// Exiting paste mode, now interpreting.

warning: there was one deprecation warning; re-run with -deprecation for details
+----------+-----+
|num_values|level|
+----------+-----+
|3960      |INFO |
|257       |WARN |
+----------+-----+

readFromSolrOpts: scala.collection.immutable.Map[String,String] = Map(collection -> system_logs, fields -> host_s,level_s,type_s,message_txt,thread_s,timestamp_tdt, query -> level_s:[* TO *])
logsDF: org.apache.spark.sql.DataFrame = [host_s: string, level_s: string ... 4 more fields]
```

Don’t worry about WARN log messages when running this script. They are benign messages from Spark SQL.

Congratulations, you just ran your first Fusion Spark job that reads data from Solr and performs a simple aggregation!
The Spark master web UI

The Spark master web UI lets you dig into the details of the Spark job. This handy Mastering Apache Spark guide helps you understand the Spark web UI.

In your browser (http://localhost:8767), there should be a job named "Spark shell" under running applications (the application ID will be different than the following screenshot):

![Running Applications](image)

Click the application ID, and then click the Application Detail UI link. You'll see this information about the completed job:

![Completed Jobs](image)

Notice the tabs at the top of the UI that let you dig into details about the running application. Take a moment to explore the UI. It can answer these questions about your application:

- How many tasks were needed to execute this job?
- Which JARs were added to the classpath for this job? (Look under the Environment tab.)
- How many executor processes were used to run this job? Why? (Look at the Spark configuration properties under the Environment tab.)
- How many rows were read from Solr for this job? (Look under the SQL tab.)

For the above run, the answers are:

- 205 tasks were needed to execute this job.
- The Environment tab shows that one of the JAR files is named spark-shaded-*.jar and was "Added By User".
- It took 2 executor processes to run this job. Each executor has 2 CPUs allocated to it and the bin/spark-shell script asked for 4 total CPUs for the shell application.
- This particular job read about 21K rows from Solr, but this number will differ based on how long Fusion has been running.

The key take-away is that you can see how Spark interacts with Solr using the UI.

Spark job tuning

Returning to the first question, why were 202 tasks needed to execute this job?
Details for Query 2

Submitted Time: 2016/10/05 16:17:41
Duration: 0.2 s
Succeeded Jobs: 2

```
com.lucidworks.spark.SolrRelation@1f0630a9
```

**Filter**

- number of input rows: 21486
- number of output rows: 21486

**Project**

- number of rows: 21486

**TungstenAggregate**

- number of input rows: 21486
- number of output rows: 108
- data size total (min, med, max): 8.5 MB (4.2 MB, 4.2 MB, 4.2 MB)
- spill size total (min, med, max): 0.0 B (0.0 B, 0.0 B, 0.0 B)
The reason is that SparkSQL defaults to using 200 partitions when performing distributed group by operations; see the property `spark.sql.shuffle.partitions`.

Because our data set is so small, let's adjust Spark so that it only uses 4 tasks. In the Spark shell, execute the following Scala:

```scala
spark.conf.set("spark.sql.shuffle.partitions", "4")
```

You just need to re-execute the final query and `show` command:

```scala
val readFromSolrOpts = Map(
  "collection" -> "logs",
  "fields" -> "host_s,port_s,level_s,message_t,thread_s,timestamp_tdt"
)
val logsDF = spark.read.format("solr").options(readFromSolrOpts).load
logsDF.registerTempTable("fusion_logs")
var sqlDF = spark.sql(""
  | SELECT COUNT(*) as num_values, level_s as level
  | FROM fusion_logs
  | GROUP BY level_s
  | ORDER BY num_values desc
  | LIMIT 10"").stripMargin
sqlDF.show(10,false)
```

Now if you look at the Job UI, you'll see a new job that executed with only 6 executors! You've just had your first experience with tuning Spark jobs.
Spark Driver Processes

A Spark “driver” is an application that creates a SparkContext for executing one or more jobs in the Spark cluster. The following diagram depicts the driver’s role in a Spark cluster:

In the diagram above, the spark-master service in Fusion is the Cluster Manager.

If your Spark job performs any collect operations, then the result of the collect (or collectAsMap) is sent back to the driver from all the executors. Consequently, if the result of the collect is too big too fit into memory, you will encounter OOM issues (or other memory related problems) when running your job.

All Fusion jobs run on Spark using a driver process started by the API service.

Custom jobs

Fusion supports custom Spark jobs that are written in Scala, Java, or Python jobs and that are built using the Spark Job Workbench, a toolkit provided by Lucidworks. See the examples in the repository for details.

To troubleshoot problems with a custom job, start by looking for errors in the script-job driver log, /opt/fusion/4.2.x/var/log/api/spark-driver-launcher.log (on Unix) or C:\lucidworks\var\fusion\4.2.x\var\log\api\spark-driver-launcher.log (on Windows).

Drivers

Fusion has four types of job drivers:

- **Default driver** – Executes built-in Fusion jobs, such as a signal aggregation job or a metrics rollup job.

- **Script-job driver** – Executes custom script jobs; a separate driver is needed to isolate the classpath for custom Scala scripts.

- **Spark-shell driver** – Wrapper script provided with Fusion to launch the Spark Scala REPL shell with the correct master URL (pulled from Fusion’s API) and a shaded Fusion JAR added. Launched using
**Custom-job driver** – Executes custom jobs built using the Spark Job Workbench, a toolkit provided by Lucidworks to help build custom Spark jobs using Scala, Java, or Python.

**Default driver**

Navigate to **Collections > Collections Manager** and select the `system_monitor` collection. Then navigate to **Collections > Jobs** and select one of the built-in aggregation jobs, such as `session_rollup`. In the diagram above, the spark-master service in Fusion is the Cluster Manager.

You must delete any existing driver applications before launching the job. Even if you haven’t started any jobs by hand, Fusion’s API service might have started one automatically, because Fusion ships with built-in jobs that run in the background which perform rollups of metrics in the `system_monitor` collection. Therefore, before you try to launch a job, you should run the following command:

```
curl -X DELETE http://localhost:8764/api/spark/driver
```

Wait a few seconds and use the Spark UI to verify that no Fusion-Spark application (for example, `Fusion-20161005224611`) is running.

In a terminal window or windows, set up a `tail -f` job (on Unix, or the equivalent on Windows) on the `api` and `spark-driver-default` logs:

```
tail -f var/log/api/api.log var/log/api/spark-driver-default.log
```

Give this command from the `bin` directory below the Fusion home directory, for example, `/opt/fusion/4.2.x` (on Unix) or `C:\lucidworks\fusion\4.2.x` (on Windows).

Now, start any aggregation job from the UI. It doesn’t matter whether or not this job performs any work; the goal of this exercise is to show what happens in Fusion and Spark when you run an aggregation job. You should see activity in both logs related to starting the driver application and running the selected job. The Spark UI will now show a Fusion-Spark app:

```
+----+-------+--------+---------------+----------+----------+--------+--------+
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Cores</th>
<th>Memory per Node</th>
<th>Submitted Time</th>
<th>User</th>
<th>State</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>app-20161005164611-0003</td>
<td>Fusion-20161005224611</td>
<td>8</td>
<td>2.0 GB</td>
<td>2016/10/05 16:46:11</td>
<td>timpoter</td>
<td>RUNNING</td>
<td>32 s</td>
</tr>
</tbody>
</table>
```

Use the `ps` command to get more details on this process:

```
ps waux | grep SparkDriver
```

The output should show that the Fusion SparkDriver is a JVM process started by the API service; it is not managed by the Fusion agent. Within a few minutes, the Spark UI will update itself:
Notice that the application no longer has any cores allocated and that all of the memory available is not being used (0.0B Used of 2.0 GB Total). This is because we launch our driver applications with `spark.dynamicAllocation.enabled=true`. This setting allows the Spark master to reclaim CPU & memory from an application if it is not actively using the resources allocated to it.

Both driver processes (default and scripted) manage a SparkContext. For the default driver, the SparkContext will be shut down after waiting a configurable (`fusion.spark.idleTime`: default 5 mins) idle time. The scripted driver shuts down the SparkContext after every scripted job is run to avoid classpath pollution between jobs.

**Script-job driver**

Fusion supports custom script jobs.

Script jobs require a separate driver to isolate the classpath for custom Scala scripts, as well as to isolate the classpath between the jobs, so that classes compiled from scripts don’t pollute the classpath for subsequent scripted jobs.

For this reason, the SparkContext that each scripted job uses is immediately shut down after the job is finished and is recreated for new jobs. This adds some startup overhead for scripted jobs.

Refer to the apachelogs lab in the Fusion Spark Bootcamp project for a complete example of a custom script job.

To troubleshoot problems with a script job, start by looking for errors in the script-job driver log `spark-driver-scripted.log` in `/opt/fusion/4.2.x/var/log/api/` (on Unix) or `C:\lucidworks\var\fusion\4.2.x\var\log\api\` (on Windows).

**Spark drivers in a multinode cluster**

To find out which node is running the Spark driver which node is running the driver when running a multi-node Fusion deployment which has several nodes running Fusion’s API services, you can query the driver status via the following call:

```
curl http://localhost:8764/api/spark/driver/status
```

This returns a status report:
{"/spark-drivers/15797426d56T537184c2": {
  "id": "15797426d56T537184c2",
  "hostname": "192.168.1.9",
  "port": 8601,
  "scripted": false
}}
Spark Configuration

Spark has a number of configuration properties. In this section, we’ll cover some of the key settings you’ll need to use Fusion’s Spark integration.

For the full set of Fusion’s spark-related configuration properties, see the Spark Jobs API.

Spark master/worker resource allocation

<table>
<thead>
<tr>
<th>Note</th>
<th>If you co-locate Spark workers and Solr nodes on the same server, then be sure to reserve some CPU for Solr to avoid a compute intensive Spark job from starving Solr of CPU resources.</th>
</tr>
</thead>
</table>

Number of cores allocated

To change the CPU usage per worker, you need to use the Fusion configuration API to update this setting, as in the following example.

```
curl -u user:password -H 'Content-type:application/json' -X PUT -d '6' \  
http://localhost:8764/api/configurations/fusion.spark.worker.cores
```

You can also over-allocate cores to a spark-worker, which usually is recommended for hyper-threaded cores by setting the property `spark-worker.envVars` to `SPARK_WORKER_CORES=<number of cores>` in the `fusion.properties` file on all nodes hosting a spark-worker. For example, a r4.2xlarge instance in EC2 has 8 CPU cores, but the following configuration will improve utilization and performance:

```
spark-worker.envVars=SPARK_WORKER_CORES=16,SPARK_SCALA_VERSION=2.11,SPARK_PUBLIC_DNS=${default.address},SPARK_LOCAL_IP=${default.address}
```

You can obtain the IP address that the Spark master web UI binds to with this API command:

```
curl http://localhost:8765/api/v1/spark/master
```

<table>
<thead>
<tr>
<th>Tip</th>
<th>We encourage you to set the <code>default.address</code> property in <code>fusion.properties</code> to ensure that all Spark processes have a consistent address to bind to.</th>
</tr>
</thead>
</table>

After making this change to your Spark worker nodes, you must restart the spark-worker process on each.

**On Unix:**

```
./spark-worker restart
```

Give this command from the `bin` directory below the Fusion home directory, for example, `/opt/fusion/4.2.x`.  

178
On Windows:

```bash
spark-worker.cmd restart
```

Give this command from the `bin` directory below the Fusion home directory, for example, `C:\lucidworks\fusion\4.2.x`.

### Memory allocation

The amount of memory allocated to each worker process is controlled by Fusion property `fusion.spark.worker.memory` which specifies the total amount of memory available for all executors spun up by that Spark Worker process. This is the quantity seen in the memory column against a worker entry in the Workers table.

The JVM memory setting (`-Xmx`) for the spark-worker process configured in the `fusion.properties` file controls how much memory the spark-worker needs to manage executors (and not how much memory should be made available to your job(s)). When modifying the `-Xmx` value, use `curl` as follows:

```bash
curl -u user:password -H 'Content-type:application/json' -X PUT -d '8g' \\ http://localhost:8764/api/configurations/fusion.spark.worker.memory
```

**Tip**

Typically, 512m to 1g is sufficient for the spark-worker JVM process.

The Spark worker process manages executors for multiple jobs running concurrently. For certain types of aggregation jobs you can also configure the per executor memory, but this can impact how many jobs you can run concurrently in your cluster. Unless explicitly specified using the parameter `spark.executor.memory`, Fusion calculates the amount of memory that can be allocated to the executor.

Aggregation Spark jobs always get half the memory of the amount assigned to the workers. This is controlled by the `fusion.spark.executor.memory.fraction` property, which is set to `0.5` by default.

For example, Spark workers have 4 Gb of memory by default and the executors for aggregator Spark jobs are assigned 2 Gb for each executor.

To let Fusion aggregation jobs use more of the memory of the workers, increase `fusion.spark.executor.memory.fraction` property to `1`. Use this property instead of the Spark executor memory property.

```bash
curl -u user:password -H 'Content-type:application/json' -X PUT -d '1' \\ http://localhost:8764/api/configurations/fusion.spark.executor.memory.fraction
```

After making these changes and restarting the workers, when we run a Fusion job, we get the following:
Cores per driver allocation

The configuration property `fusion.spark.cores.fraction` lets you limit the number of cores used by the Fusion driver applications (default and scripted). For example, in the screenshot above, we see 18 total CPUs available.

We set the cores fraction property to 0.5 via the following command:

```
curl -u user:password -H 'Content-type:application/json' -X PUT -d '0.5' http://localhost:8764/api/configurations/fusion.spark.cores.fraction
```

This cuts the number of available cores in half, as shown in the following screenshot:

Ports used by Spark in Fusion

This table lists the default port numbers used by Spark processes in Fusion.

<table>
<thead>
<tr>
<th>Port number</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>4040</td>
<td>SparkContext web UI</td>
</tr>
<tr>
<td>7337</td>
<td>Shuffle port for Apache Spark worker</td>
</tr>
<tr>
<td>8767</td>
<td>Spark master web UI</td>
</tr>
<tr>
<td>8770</td>
<td>Spark worker web UI</td>
</tr>
<tr>
<td>Port number</td>
<td>Process</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>8766</td>
<td>Spark master listening port</td>
</tr>
<tr>
<td>8769</td>
<td>Spark worker listening port</td>
</tr>
<tr>
<td>8772 (spark.driver.port)</td>
<td>Spark driver listening port</td>
</tr>
<tr>
<td>8788 (spark.blockManager.port)</td>
<td>Spark BlockManager port</td>
</tr>
</tbody>
</table>

If a port is not available, Spark uses the next available port by adding 1 to the assigned port number. For example, if 4040 is not available, Spark uses 4041 (if available, or 4042, and so forth).

Ensure that the ports in the above table are accessible, as well as a range of up to 16 subsequent ports. For example, open ports 8772 through 8787, and 8788 through 8804, because a single node can have more than one Spark driver and Spark BlockManager.

**Spark-related directories and files in Fusion**

The following directories and files are for Spark components and logs in Fusion.

**Spark components**

These directories and files are for Spark components:

<table>
<thead>
<tr>
<th>Path (relative to Fusion home)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin/spark-master</td>
<td>Script to manage (start, stop, status, etc.) the Spark Master service in Fusion</td>
</tr>
<tr>
<td>bin/spark-worker</td>
<td>Script to manage (start, stop, status, etc.) the Spark Worker service in Fusion</td>
</tr>
<tr>
<td>bin/sql</td>
<td>Script to manage (start, stop, status, etc.) the SQL service in Fusion</td>
</tr>
<tr>
<td>bin/spark-shell</td>
<td>Wrapper script to launch the interactive Spark shell with the Spark Master URL and shaded JAR</td>
</tr>
<tr>
<td>apps/spark-dist</td>
<td>Apache Spark distribution; contains all JAR files needed to run Spark in Fusion</td>
</tr>
<tr>
<td>apps/spark/hadoop</td>
<td>Hadoop home directory used by Spark jobs running in Fusion</td>
</tr>
<tr>
<td>apps/spark/driver/lib</td>
<td>Add custom JAR files to this directory to include in all Spark jobs</td>
</tr>
<tr>
<td>Path (relative to Fusion home)</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>apps/spark/lib</td>
<td>JAR files used to construct the classpath for the spark-worker, spark-master, and sql services in Fusion</td>
</tr>
<tr>
<td>var/spark-master</td>
<td>Working directory for the spark-master service</td>
</tr>
<tr>
<td>var/spark-worker</td>
<td>Working directory for the spark-worker service; keep an eye on the disk usage under this directory as temporary application data for running Spark jobs is saved here</td>
</tr>
<tr>
<td>var/spark-workDir-*</td>
<td>Temporary work directories are created in when an application is running. They are removed after the driver is shut down or closed.</td>
</tr>
<tr>
<td>var/sql</td>
<td>Working directory for the SQL service</td>
</tr>
<tr>
<td>var/api/work/spark-shaded-*.jar</td>
<td>The shaded JAR built by the API service; contains all classes needed to run Fusion Spark jobs. If one of the jars in the Fusion API has changed, then a new shaded jar is created with an updated name.</td>
</tr>
</tbody>
</table>

**Spark logs**

These directories and files are for configuring and storing Spark logs:

**Path (relative to Fusion home)**

**Notes**

**Log configuration**

conf/spark-master-log4j2.xml

Log configuration file for the spark-master service

conf/spark-worker-log4j2.xml

Log configuration file for the spark-worker service

conf/spark-driver-log4j2.xml

Log configuration file for the Spark Driver application launched by Fusion; this file controls the log settings for most Spark jobs run by Fusion

conf/spark-driver-scripted-log4j.xml

Log configuration file for custom script jobs and Parallel Bulk Loader (PBL) based jobs

conf/spark-driver-launcher-log4j2.xml
Log configuration file for jobs built using the Spark Job Workbench

conf/spark-executor-log4j2.xml

Log configuration file for Spark executors; log messages are sent to STDOUT and can be viewed from the Spark UI

conf/sql-log4j2.xml

Log configuration file for the Fusion SQL service

Logs

var/log/spark-master/*

Logs for the spark-master service

var/log/spark-worker/*

Logs for the spark-worker service

var/log/sql/*

Logs for the sql service

var/log/api/spark-driver-default.log

Main log file for built-in Fusion Spark jobs

var/log/api/spark-driver-scripted.log

Main log file for custom script jobs

var/log/api/spark-driver-launcher.log

Main log file for custom jobs built using the Spark Job Workbench

Connection configurations for an SSL-enabled Solr cluster

You'll need to set these Java system properties used by SolrJ:

- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.net.ssl.trustStoreType

For the following Spark configuration properties:

- spark.executor.extraJavaOptions
- fusion.spark.driver.jvmArgs
- spark.driver.extraJavaOptions
> curl -H 'Content-type:application/json' -X PUT \\n-\d 'Djavax.net.ssl.trustStore=/opt/app/jobs/ssl/solrtrust.jks -Djavax.net.ssl.trustStorePassword=changeit \\
-Djavax.net.ssl.trustStoreType=jks' \\
"http://localhost:8764/api/configurations/spark.executor.extraJavaOptions"

> curl -H 'Content-type:application/json' -X PUT \\n-\d 'Djavax.net.ssl.trustStore=/opt/app/jobs/ssl/solrtrust.jks -Djavax.net.ssl.trustStorePassword=changeit \\
-Djavax.net.ssl.trustStoreType=jks' \\
"http://localhost:8764/api/configurations/fusion.spark.driver.jvmArgs"

> curl -H 'Content-type:application/json' -X PUT \\n-\d 'Djavax.net.ssl.trustStore=/opt/app/jobs/ssl/solrtrust.jks -Djavax.net.ssl.trustStorePassword=changeit \\
-Djavax.net.ssl.trustStoreType=jks' \\
"http://localhost:8764/api/configurations/spark.driver.extraJavaOptions"
Scaling Spark Aggregations

Consider the process of running a simple aggregation on 130M signals. For an aggregation of this size, it helps to tune your Spark configuration.

Speed up tasks and avoid timeouts

One of the most common issues encountered when running an aggregation job over a large signals data set is task timeout issues in Stage 2 (foreachPartition). This is typically due to slowness indexing aggregated jobs back into Solr or due to JavaScript functions.

The solution is to increase the number of partitions of the aggregated RDD (the input to Stage 2). By default, Fusion uses 25 partitions. Here, we increase the number of partitions to 72. Set these configuration properties:

- `spark.default.parallelism` – Default number of partitions in RDDs returned by transformations like `join`, `reduceByKey`, and `parallelize` when not specified by the user:

  curl -u user:password -H 'Content-type:application/json' -X PUT -d '72' "${FUSION_API}/configurations/spark.default.parallelism"

- `spark.sql.shuffle.partitions` – Number of partitions to use when shuffling data for joins or aggregations.

  curl -u user:password -H 'Content-type:application/json' -X PUT -d '72' "${FUSION_API}/configurations/spark.sql.shuffle.partitions"

After making these changes, the `foreachPartition` stage of the job will use 72 partitions:

Increase rows read per page

You can increase the number of rows read per page (the default is 10000) by passing the rows parameter when starting your aggregation job; for example:
For example, we were able to read 130M signals from Solr in 18 minutes at ~120K rows/sec using rows=20000 vs. 21 minutes using the default 10000.

**Improve job performance**

You can increase performance when reading input data from Solr using the `splits_per_shard` read option, which defaults to 4. This configuration setting governs how many Spark tasks can read from Solr concurrently. Increasing this value can improve job performance but also adds load on Solr.
Spark Troubleshooting

This article contains tips and techniques for troubleshooting Spark.

Log API endpoints for Spark jobs

Log endpoints are useful for debugging Spark jobs on multiple nodes. In a distributed environment, the log endpoints parse the last N log lines from different Spark log files on multiple nodes and output the responses from all nodes as text/plain (which renders nicely in browsers) sorted by the timestamp.

The REST API Reference documents log endpoints for Spark jobs. The URIs for the endpoints contain /api/spark/log.

The most useful log API endpoint is the spark/log/job/ endpoint, which goes through all Fusion REST API and Spark logs, filters the logs by the jobId (using MDC, the mapped diagnostic context), and merges the output from different files.

For example, to obtain log content for the job jobId:

```
curl -u user:password "$FUSION_API/spark/log/job/jobId"
```

Note - Log endpoints will only output data from log files on nodes on which the API service is running.

Specific issues

These are some specific issues you might encounter.

Job hung in waiting status

Check the logs for a message that looks like:

```
2016-10-07T11:51:44,800 - WARN  [Timer-0:Logging$class@70] - Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources
```

If you see this, then it means your job has requested more CPU or memory than is available. For instance, if you ask for 4g but there is only 2g available, then the job will just hang in WAITING status.

Lost executor due to heartbeat timeout

If you see errors like the following:
This is most likely due to an OOM in the executor JVM (preventing it from maintaining the heartbeat with the application driver). However, we've seen cases where tasks fail, but the job still completes, so you'll need to wait it out to see if the job recovers.

Another situation when this can occur is when a shuffle size (incoming data for a particular task) exceeds 2GB. This is hard to predict in advance because it depends on job parallelism and the number of records produced by earlier stages. The solution is to re-submit the job with increased job parallelism.

**Spark Master won't start on EC2**

See this article for a solution.
Machine Learning Models in Fusion

Fusion provides the following tools required for the model training process:

- Solr can easily store all your training data.
- Spark jobs perform the iterative machine learning training tasks.
- Fusion's blob store makes the final model available for processing new data.

Training Models

Note

The approach for training models explained in this section still works in Fusion 4.2. An alternative approach introduced in Fusion 3.1 lets you create model-training jobs in the Fusion UI. See Machine Learning in Lucidworks Fusion for more information.

An example Scala script to train an SVM-based sentiment classifier for tweets is provided in the spark-solr repository.

The following diagram depicts this process:

Supervised Machine Learning Model Training Workflow

Model Prediction

Fusion's blob store requires all stored objects have a unique ID. Once the model is stored in the Fusion blobstore, it is available to Fusion's index and query Machine Learning pipeline stages, which use the model to make predictions for new data in pipeline documents and queries. The following diagram shows how this process works:
Model Checking

To test the goodness of your model in Fusion, first create either a document index pipeline or a query processing pipeline which contains a Machine Learning stage that uses your model to make predictions on your data, and then send a document or query through that pipeline pipeline which contains data for which you know what the predicted value should be. For example, given a trained sentiment classifier and an index stage configured to use it, the following document should be classified as a highly positive tweet, with a value of (close to) 1.0 in the "sentiment_d" field:

```json
{ "id":"tweets-2",
 "fields": [
 { "name": "tweet_txt",
   "value": "I am super excited that spring is finally here, yay! #happy" }
 ]
}
```

Metadata file spark-mllib.json

The file `spark-mllib.json` contains metadata about the model implementation. In particular, how the model derives feature vectors from a document or query.

The JSON object has the following attributes:

- **id** - A string label that is used as a unique ID for the Fusion blobstore, for example, `tweets_sentiment_svm`.
- **modelClassName** - The name of the `spark-mllib` class or the custom Java class that implements the `com.lucidworks.spark.ml.MLModel` interface.
- **featureFields** - A list of one or more field names.
- **vectorizer** - Specifies the processing required to derive a vector of features from the contents of the document fields listed in the `featureFields` entry.

The following example shows the `spark-mllib.json` file for the model with id `tweets_sentiment_svm`:
The vectorizer consists of two steps: a lucene-analyzer step followed by a hashingTF step. The lucene-analyzer step can use any Lucene analyzer to perform text analysis. For more information about using the Lucene analyzer, see: https://lucidworks.com/blog/2016/04/13/spark-solr-lucenetextanalyzer/.

Other available vectorizer operations include the MLlib normalizer, the standard scaler, and the ChiSq selector. To see how to use the standard scaler, see the examples in the spark-solr repository.
Experiments

When making changes to a query pipeline or query parameters that will affect users' search experience, it is often a good idea to run an experiment in order to verify that the results are what you intended. Fusion AI lets you create and run experiments that take care of dividing traffic between variants and calculating the results of each variant with respect to configurable objectives such as purchases, click-through rate or search relevance.

There are two ways that a search application might interact with an experiment:

- using a query profile
- using an Experiment query pipeline stage

If a query profile is configured to use an experiment, then a search app sends queries and signals to the query profile endpoint. If the experiment is active, then Fusion routes each query through one of the experiment variants. The search app will also send subsequent signal data relating to that query — clicks, purchases, “likes”, or whatever is relevant to the application — to that same query profile, and Fusion will record it along with information about the experiment variant that the user was exposed to. Fusion generates and stores the data that metrics calculations use. Metrics jobs periodically calculate the metrics. After metrics have been calculated, they are available in App Insights.

This topic explains the experiment workflow and basic concepts. These additional topics provide details about how to implement experiments to improve the user experience:

- Plan an Experiment
- Set Up an Experiment
- Run an Experiment
- Analyze Experiment Results
- Manage Experiments
A/B/n experiments

Fusion AI's experiments feature set implements A/B/n experiments, also called A/B experiments or A/B tests, where A and B are experiment groups with one or more variants.

Fusion AI's implementation of an A/B experiment uses consistent hashing on a unique ID field (typically `userId`), concatenated with the experiment's name, to assign each request to one of the experiment groups. Any future requests with that hash are assigned to the same group, guaranteeing user "stickiness".

| Tip | If you prefer "stickiness" only at the session level, you can send a session ID instead of a user ID. |

If you send no ID at all, the request is not assigned to a variant since there is no way to consistently assign it to the same one. In that case, the request uses the "default" configuration of the query profile or experiment stage.
Example

The following experiment is an example of an A/B/n experiment with three variants:

- **Variant 1 (control)** – Use the default query pipeline with no modifications. Each experiment should have a "control" variant as the first variant; the other variants will be compared against this one.

- **Variant 2 (content-based filtering with a Solr MoreLikeThis stage)** – Content-based filtering uses data about a user's search results, browsing history, and/or purchase history to determine which content to serve to the user. The filtering is non-collaborative.

- **Variant 3 (collaborative filtering with a Recommend Items for User stage)** – Collaborative filtering takes advantage of knowledge about the behavior of many individuals. It makes serendipitous discovery possible—a user is presented with items that other users deem relevant, for example, socks when buying shoes.
### High-level workflow

In an experiment:

1. A Fusion administrator defines the experiment. An experiment has variants with differences in query pipelines, query pipeline stages, collections, and/or query parameters.
2. The Fusion administrator assigns the experiment to a query profile.
3. A user searches using that query profile.
4. If the experiment is running, Fusion assigns the user to one of the experiment variants, in accordance with traffic weights. Assignment to a variant is persistent. The next time the user searches, Fusion assigns the same variant.
5. Different experiment variants return different search results to users.
6. Users interact with the search results, for example, viewing them, possibly clicking on specific results, possibly buying things, and so forth.
7. Based on the interactions, the search app backend sends signals to the signals endpoint of the query profile for the experiment.
8. Using signal data, a Metrics Spark job periodically computes metrics for each experiment variant and writes the metrics to the `_signals_aggr` collection.
9. In the Fusion UI, an administrator can use App Insights to view reports about the experiment.
10. Once the results of the experiment are conclusive, the Fusion administrator can stop the experiment and change the query profile to use the winning variant, or start a new experiment.
Information flow

This diagram illustrates information flow through an experiment. Numbers correspond to explanations below the diagram.

1. A user searches in a search app. For example, the user might search for shirt.

2. The search app backend appends a userId or other unique ID that identifies the user, for example, userId=123, to the query and sends the query to the query profile endpoint for the experiment.

3. Using information in the query profile and the value of the unique ID, Fusion routes the query through one of the experiment's variants. In this example, Fusion routes the query through query pipeline 1.

4. A query pipeline adds a x-fusion-query-id to the response header, for example, x-fusion-query-id=abc.

5. Based on the query, Fusion obtains a search result from the index, which is stored in the primary collection. Fusion sends the search result back to the search app.

6. Fusion sends a response signal to the signals collection.

7. A different user might be routed through the other experiment variant shown here, and through query pipeline 2. This query pipeline has an enabled Boost with Signals stage, unlike query pipeline 1.

8. The search user interacts with the search results, viewing them, possibly clicking on specific results, possibly buying things, and so forth. For example, the user might click the document with docId=757.

9. Based on the interactions, the search app backend sends click signals to the signals endpoint for the query profile. Signals include the same query ID so Fusion can associate the signals with the experiment.

10. Using information in the query profile, Fusion routes the signals to the _signals_ingest pipeline.

11. The _signals_ingest pipeline stores signals in the _signals collection. Signals include the collection ID of the primary collection and experiment tracking information.
Metrics generation

This diagram illustrates metrics generation:

1. A Fusion administrator can configure which metrics are relevant for a given experiment and the frequency with which experiment metrics are generated. They can also generate metrics on demand.

2. Using signal data, a Metrics Spark job periodically runs in the background. It obtains signal data from the _signals collection, computes metrics for each experiment variant, and writes the metrics to the _signals_aggr collection.

3. In the Fusion UI, a Fusion administrator can view experiment metrics.

4. App Insights uses these calculated metrics and displays reports about the experiment. :leveloffset: +1
Plan an experiment

From a planning standpoint, an experiment has these parts:

- **A baseline control** – One of the experiment variants will be the control. This is "how we are doing things today." If you are experimenting from the start, choose the simplest variant as the control.

- **Experiment variants** – Experiment variants other than the control are attempts to improve the user’s extended search experience. Which relevancy strategy works best for your search app and your users?

- **Metrics** – This is how you know whether the search variants produce differences in user interactions, and whether the differences are statistically significant.

In the remainder of this topic, you’ll make decisions about these broad areas, as well as about experiment details.
1. Plan what you want to vary

Identify different relevancy strategies, where each represents a hypothesis about which user experience will drive more click-throughs, purchases, and so on. Use the Query Workbench to explore how to produce different search results and recommendations using different query pipelines, and evaluate which ones might engage your users most effectively.
2. Plan what you want to measure

Metrics compare the control against other variants pairwise. For example, if the variants are experiment, B, C, and D, and you choose experiment as the control, then the comparisons for which metrics are generated will be experiment/B, experiment/C, and experiment/D.

You can learn more about metrics.
3. Design the experiment

When designing an experiment, you must make these decisions:

- How users are identified
- Percentage of total traffic to send through the experiment
- Number of variants and how they differ
- Metrics to generate

In many cases identifying users is straightforward, using an existing user ID or session ID if the application has one. In other cases, you may need to generate an identifier of some sort to send in on queries. It is important to send in some kind of identifier with each query so that the experiment can route the query to a variant, and to send that same identifier with any subsequent signals that resulted from that query. Queries without a user ID will not be routed through the experiment.

The percentage of total traffic to send through the experiment is the one variable that can change over the course of the experiment. It is often a good practice to start out sending only a small percentage of search traffic through a new experiment, in order to verify that each of the variants are functioning properly. Then, once you have established that the behavior is as intended, you can increase the percentage of traffic through the experiment to the desired level.

With modest usage and for a possibly small effect, or when testing multiple variants at the same time, you might want to send 100% of users through the experiment and let it run longer. For high usage and an effect that is expected to be larger, and with only two variants, you might not need to send all users through the experiment and the experiment won’t take as long.
4. Choose traffic weights

Fusion AI uses traffic weights to apportion search traffic among the variants. This allows you to send a different percentage of traffic through each variant if desired.

4.1. Automatic traffic weights (multi-armed bandit)

The Automatically Adjust Weights Between Variants configuration option enables multi-armed bandits and eliminates the need to specify a traffic weight for each variant.

In multi-arm bandit mode, metrics jobs are created and scheduled automatically once the experiment starts. The weights between variants only change after the metrics jobs run.

Fusion’s multi-arm bandit implementation uses a variation of Thompson Sampling (sometimes called Bayesian Bandits). This algorithm uses the current count of successes versus failures to build a beta distribution that represents the level of confidence in the primary metric value for each variant. It then samples a random number from each variant’s distribution, and picks the highest number.

This type of implementation has three effects:

• It weights better-performing variants higher.

  Since the beta distribution of each variant is centered around the primary metric value for that variant, a random number selected from a higher-performing variant is likely to be higher than a random number picked from a lower-performing variant.

• Lower-performing variants remain in play

  Picking a random number from each distribution preserves the chance that Fusion will try a lower-performing variant, as long as there is still a chance that it is better.

• The more confident the measurements, the narrower the beta distributions become.

  The more uncertain the measurements, the wider the distributions will be, and thus the more likely that Fusion will choose variants that appear to be performing more poorly.

Since Fusion adjusts the weights between variants each time the metrics jobs run, users can still get different results on subsequent visits. For example, if variant A is getting 80% of traffic, but after recalculating metrics it is only getting 50% of traffic, then some users who were previously assigned to variant A will be assigned to variant B. However, only the bare minimum of users will be reassigned to a new variant. Most users will see no changes. Once the experiment has been running for some time, the changes between the variants should be fairly small, so relatively few users should be affected.

4.2. Manually specifying traffic weights

The formula for variant A is:

\[
\text{ProportionA} = \frac{\text{Traffic weightA}}{\text{Sum of traffic weights for all variants}}
\]
For example:

<table>
<thead>
<tr>
<th>Variant traffic weights</th>
<th>Sum of traffic weights</th>
<th>Variant proportions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 1.0</td>
<td>2</td>
<td>0.5 0.5</td>
</tr>
<tr>
<td>1.0 1.0 2.0</td>
<td>4</td>
<td>0.25 0.25 0.5</td>
</tr>
<tr>
<td>0.5 1.0 1.0 2.5</td>
<td>5</td>
<td>0.1 0.2 0.2 0.5</td>
</tr>
</tbody>
</table>
5. Calculate sample sizes

Fusion will calculate the required sample size to detect a statistically significant result based on the results at runtime. The "confidence level" metric that is displayed in App Insights has this minimum sample size factored in, so that confidence is always low for experiments that have not yet reached their required sample size.

However, if you would like to use different power or significance level in evaluating your experiment (Fusion will use 0.08 and 0.05), or if you would like to establish your own sample size based on a desired minimum detectable effect, you may do so.
6. Choose an implementation approach

You can construct an experiment in either of two ways:

- **Experiment and query profile** (recommended) – For most cases, you’ll want to create additional query pipelines that return different search results. A query profile directs traffic through the query pipelines in accordance with the traffic weights of experiment variants.

- **Experiment stage in a query pipeline** – If you want to use parts of a single query pipeline in all experiment variants, you can add an Experiment stage to that pipeline (the pipeline that receives search queries). The app can direct queries to the endpoint of a query profile that references the pipeline (recommended) or to the endpoint of the query pipeline. If used, the query profile doesn’t reference an experiment.
Next step

You've planned the experiment. Next, you will set it up using either a query profile or an Experiment stage.
Experiment Metrics

This section describes metrics available for experiments.

Click-Through Rate

The Click-Through Rate (CTR) metric provides the rate of clicks per query for a variant. The CTR is a number between 0 and 1, that is, what proportion of queries lead to clicks. Variants with a CTR closer to 1 perform better than variants with a lower rate.

CTR is *cumulative*, that is, each time it is calculated, it is calculated from the beginning of the experiment. After each variant has reached a stable level, you shouldn't see large day-to-day fluctuations in the CTR.

The job that generates the Click-Through Rate metrics is named `<experiment-name>-<metric-name>`, for example, `Experiment-CTR`.

Conversion Rate

The Conversion Rate metric provides the rate of some type of signal per variant, that is, what proportion of queries lead to some type of signal, such as cart, purchase or like signals. (These signal types aren't predefined.)

For example, if you're interested in how many queries convert into cart signals, specify the cart signal type in the conversion rate metric.

The Click-Through Rate metric is a conversion rate for click signals.

The job that generates the Conversion Rate metrics is named `<experiment-name>-<metric-name>`, for example, `Experiment-Conversion`.

Mean Reciprocal Rank (MRR)

The Mean Reciprocal Rank (MRR) metric measures the position of documents that were clicked on in ranked results. It ranges from 0 (at the very bottom) to 1 (at the very top). MRR penalizes clicks that occur further down in the results, which indicate a ranking issue where relevant documents are not ranked high enough. Variants with an MRR closer to 1 indicate that users are clicking on documents that have higher ranks.

The job that generates the Mean Reciprocal Rank metrics is named `<experiment-name>-<metric-name>`, for example, `Experiment-MRR`.

Response Time

The Response Time metric computes the named statistic (for example, mean, variance or max) from response-time data. The default statistic is avg (average, the same as mean).

You can use the Response Time metric to evaluate the impact of adding additional stages to a query pipeline, for example, a recommendation or machine learning stage.

The response time is the end-to-end processing time from when a query pipeline receives a query to when the pipeline supplies a response:
• **No Experiment stage** – If a query pipeline doesn’t have an Experiment stage, then there is no experiment-processing overhead in the response times.

• **Experiment stage** – If a query pipeline includes an Experiment stage, then processing by that stage is included in the response times.

The job that generates the Response Time metrics is named `<experiment-name><metric-name>`, for example, `Experiment-Response_time`.

### Supported functions

When adding the Response Time metric to an experiment, specify one of these Spark SQL function names or aliases for the Statistic.

<table>
<thead>
<tr>
<th>Function name or alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg</td>
<td>Mean response time</td>
</tr>
<tr>
<td>kurtosis</td>
<td>Kurtosis of the response times</td>
</tr>
<tr>
<td>max</td>
<td>Maximum response time</td>
</tr>
<tr>
<td>mean</td>
<td>Mean response time</td>
</tr>
</tbody>
</table>
| median                | Median response time. This is an alias for `percentile(query_time,0.5)`.
| min                   | Minimum response time |
| percentile_N          | Nth percentile of the response times, that is, the value at or closest to the percentile. N is an integer between 1 and 100. This is an alias for the function `percentile(query_time,N/100)`.
| skewness              | Skewness of the response times |
| sum                   | Sum of the response times |
| stddev                | Standard deviation of the response times |
| variance              | Variance of the response times |

For more information about these functions, see the documentation for [Spark SQL Built-in Functions](#).
**Custom SQL**

Under the covers, Fusion AI computes all experiment metrics using Fusion’s SQL aggregation engine.

The Custom SQL metric lets you define your own SQL to compute a metric per variant. The SQL must project these three columns in the final output, and perform a GROUP BY on `variant_id`:

- **value** – A double field that represents the metric provided by this custom SQL
- **count** – The number of rows used to compute the value for a variant, that is, how many signals contributed to this value
- **variant_id** – The unique identifier of the variant

An internal view named `variant_queries` is built into the experiment job framework. This view is transient and is not defined in the table catalog; it only exists for the duration of the metrics job. The `variant_queries` view exposes all response signals for a given variant ID. The `variant_queries` view exposes the following fields pulled from response signals:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Response signal ID set by a query pipeline and returned to the client application using the <code>x-fusion-query-id</code> response header</td>
</tr>
<tr>
<td>variant_id</td>
<td>Experiment variant this response signal is associated with</td>
</tr>
<tr>
<td>query_doc_ids</td>
<td>Comma-delimited list of document IDs returned in the response, in ranked order</td>
</tr>
<tr>
<td>query_timestamp</td>
<td>ISO-8601 timestamp for the time when Fusion executed the query</td>
</tr>
<tr>
<td>query_user_id</td>
<td>User associated with the query. The front-end application must supply this.</td>
</tr>
<tr>
<td>query_rows</td>
<td>Number of rows returned for this query, that is, the page size</td>
</tr>
<tr>
<td>query_hits</td>
<td>Total number of documents that match this query, that is, the number of documents that were found</td>
</tr>
<tr>
<td>query_offset</td>
<td>Page offset</td>
</tr>
<tr>
<td>query_time</td>
<td>Total time to execute the query (in milliseconds)</td>
</tr>
</tbody>
</table>

You can use the `fusion_query_id` field to join the `variant_signals` view with other signal types such as `click`. For example,
if you want to get a count of clicks per variant, you would use:

```sql
1: SELECT COUNT(1) AS value, COUNT(1) AS count, vq.variant_id as variant_id
2: FROM ${inputCollection} c
3: INNER JOIN variant_queries vq ON c.fusion_query_id = vq.id
4: WHERE c.type = 'click'
5: GROUP BY variant_id
```

In this SQL:

- At line 1, we project the required `value`, `count`, and `variant_id` columns as the output for our custom SQL; this is required for all custom SQL metrics.
- At line 2, we use a built-in macro that represents the input collection for our metrics job. The SQL engine replaces the `${inputCollection}` variable with the correct collection name at runtime, which is typically a signals collection.
- At line 3, we use the `fusion_query_id` column to join click signals with the `id` column of the `variant_queries` view. This illustrates how the `variant_queries` view helps simplify the SQL you have to write to build a custom metric.
- At line 4, we filter signals to only include `click` signals. Behind the scenes, Fusion will send a query to Solr with `fq=type:click`.
- At line 5, we group by `variant_id` to compute the aggregated metrics for each variant; all Custom SQL must perform a group by `variant_id`.

To illustrate the power of Custom SQL metrics for experiments, let's build the SQL to compute the average page depth of clicks for each variant, to indicate if users are having to navigate beyond the first page to find results. The intuition behind this metric is that variants having a higher average page depth might indicate a ranking problem. Users aren't finding relevant documents on the first page of results.

Specifically, to build our query, we need the `query_offset` and `query_rows` columns associated with each click in a variant:

```sql
SELECT AVG((vq.query_offset/vq.query_rows)+1) as value,
      COUNT(1) as count,
      vq.variant_id as variant_id
FROM ${inputCollection} c
INNER JOIN variant_queries vq ON c.fusion_query_id = vq.id
WHERE c.type = 'click'
GROUP BY variant_id
```

In practice, MRR is a better metric for determining the ranked position of clicks, but this SQL gives a basic illustration of how to build Custom SQL metrics.

Lastly, when building Custom SQL metrics, you have the full power of Spark SQL functions, see: https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$.

The job that generates the Custom SQL metrics is named `<experiment-name>-<metric-name>`, for example, `Experiment-SQL`.

**Query Relevance**

The Query Relevance metric calculates the performance of queries against a "gold standard" or "ground truth" dataset that lists which documents should be returned for each query. You can either predetermine the queries that will be used and the documents that should be returned, and place them in a Solr collection in the correct format, or let the
groundTruth job use historical click signals to generate the ground truth data automatically.

Note that the Query Relevance metric doesn't calculate metrics based on live traffic. Instead, it issues the queries specified in the ground truth collection against each variant, and calculates the performance of the queries.

The jobs that generate the Query Relevance metrics are named <experiment-name>-groundTruth-<metric-name> and <experiment-name>-rankingMetrics-<metric-name>, for example, Experiment-groundTruth-QR and Experiment-rankingMetrics-QR.

| Important | You must run the groundTruth job by hand the first time. Query Relevance rankingMetrics jobs that run before the groundTruth job runs don't produce metrics. Subsequently, the groundTruth job runs once a month. |

**Ground Truth Queries**

Query relevance metrics rely on having a set of queries and a list of documents that should be returned for those queries in ranked order. Specifically, a ground truth dataset contains tuples of query + document ID + weight, such as the following data for a fictitious Home Improvement search application:

<table>
<thead>
<tr>
<th>Query</th>
<th>Document ID</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>hammer</td>
<td>123</td>
<td>0.9</td>
</tr>
<tr>
<td>hammer</td>
<td>456</td>
<td>0.8</td>
</tr>
<tr>
<td>hammer</td>
<td>789</td>
<td>0.7</td>
</tr>
<tr>
<td>masking tape</td>
<td>234</td>
<td>0.85</td>
</tr>
<tr>
<td>masking tape</td>
<td>567</td>
<td>0.82</td>
</tr>
<tr>
<td>masking tape</td>
<td>890</td>
<td>0.76</td>
</tr>
</tbody>
</table>

Typically, the queries included in the ground truth set represent important queries for a given search application. The weight assigned to each document is used to determine the expected ranking order for the query. Ideally, your ground truth dataset should specify the same number of documents per query, for example, 10. But this isn’t required technically for computing query relevance metrics. In other words, one query can have 10 documents specified and another query can only specify 5.

In Fusion, you can either load a curated ground truth dataset into a Fusion collection or use Fusion’s ground truth job to build a ground truth dataset using signals. If you use the ground truth job, Fusion looks at click/skip behavior for documents by analyzing response and click signals. It follows that you need a sufficient number of signals to generate an accurate ground truth dataset.

The basic intuition behind the ground truth job is that for queries that occur frequently in your search application, whether a user clicks or skips over a document serves as a relevance judgement of a document for a given query. With a sufficient sample size per query, Fusion can decide which documents are relevant and which are not for any given
query. It is important to note, however, that, because the ground truth dataset is generated from your click signals, if you have relevant documents that are never clicked (maybe because they are on the second page of results), then they will never appear in your ground truth set.

Calculating Performance vs. Ground Truth

After you have a ground truth dataset loaded into Fusion, the Query Relevance metric will calculate all of the following metrics:

**Precision**

Precision is the fraction of returned documents that are relevant to the query (that is, how many of the documents returned by this variant exist in the ground truth dataset).

**Recall**

Recall is the fraction of total relevant docs that are returned by this query (that is, how many of the documents in the ground truth set appear in the result set for this variant).

**Normalized Discounted Cumulative Gain (nDGC)**

The Normalized Discounted Cumulative Gain (nDCG) indicates whether a variant is returning highly relevant documents near the top of results.

The nDCG has a value between 0 and 1. Larger values indicate that more highly relevant documents occur earlier in the results for a query. Conversely, if a variant returns highly relevant documents lower in the results, then its nDCG score will be lower, penalizing the ranking strategy used by the variant for returning highly relevant documents lower in the results. For more details on nDCG, see https://en.wikipedia.org/wiki/Discounted_cumulative_gain.

**F1**

The F1 score is the harmonic mean between precision and recall at a given depth (10 by default). The F1 score ranges between 0 and 1, with larger values indicating that a variant is achieving a better balance of precision and recall than variants with lower F1 scores. For more details, see https://en.wikipedia.org/wiki/F1_score.

**Mean Average Precision (MAP)**

The Mean Average Precision (MAP) metric indicates how many documents returned for a query, down to a specific depth, are considered relevant to a query averaged over all queries in the ground truth dataset. MAP is a value between 0 and 1. Larger values mean that the variant returns more relevant than non-relevant documents. For example, if the relevance judgement for a result set containing 3 documents is: 1, 0, 1, then the average precision for that query will be 1/1, 0, ⅔~ 0.834 (1.667/2).
Set up an experiment

Perform the tasks needed to set up an experiment. This topic describes how to set up an experiment using a query profile that references the experiment. For the alternative approach of using an Experiment stage in the primary query pipeline, see Use an Experiment stage.

Before you begin

Before you set up an experiment, you must already have:

- **A search app** – The key aspect of the search app for experiments is that the search app identifies users in some way. A user ID might be associated with users persistently (this is best) or only for the current session. Searches and subsequent actions by anonymous users aren’t sent through an experiment.
- **A Fusion app** – The Fusion app provides the search functionality you want to provide. Below, you’ll modify the app to include an experiment.
- **Data for users to search** – The app should have data that users can search and search results that users can interact with. Typically, users will search a single dataset in the sense that different users aren’t given search results from different datasets. But in an experiment, different experiment variants can use data in different collections.
- **Results for users to interact with** – Experiment metrics depend on users interacting with search results, for example, clicking on them. A search app uses signals to report the interactions to Fusion.
- **A plan for the experiment** – This plan includes which control and variants to compare, the projected traffic, sample sizes, experiment duration, metrics, and decision criteria.

Basically, you need a working system in some environment on which you want to perform experiments, and a plan for experiment variants and for deciding which results are best.

1. Create query pipelines

The primary approach for experiment variants is to have multiple query pipelines. You must create additional query pipelines as needed before creating the experiment.

<table>
<thead>
<tr>
<th>Tip</th>
<th>You can also vary the collection that is searched and and query parameters, which you do when creating the experiment. In fact, if that is all you are varying, you can define an experiment that uses a single query pipeline. (And you can skip this section.)</th>
</tr>
</thead>
</table>

You’ll need the following pipelines:

- **Pipeline for the control variant** – The first variant in an experiment is the control. The A/B/n experiment metrics compare the control variant pairwise with the other variants. The control variant should probably use the default query pipeline. Here, we assume that is the case.
| Note | You could modify the configuration of the default query pipeline (which the control variant will use) before the experiment. However, doing so has two disadvantages. First, pairwise comparisons in the experiment won’t be against the status quo. Second, Fusion AI won’t have a baseline of historical signal data to compare against (that comparison lets an experiment reach higher confidence numbers more quickly). |

- **Pipelines for other variants** – Other variants in an experiment can use other query pipelines, each with different stages and/or that are configured differently. These are "how you might do things differently."

None of these pipelines will include Experiment stages. For the alternative approach of using an Experiment stage in the primary query pipeline, see Use an Experiment stage.

To create query pipelines for non-control variants

Create and configure the additional query pipelines you need for non-control variants. Repeat these steps for each pipeline.

1. Navigate to Querying > Query Pipelines.
2. Click Add.
3. Enter a **Pipeline ID** (arbitrary name) for the pipeline, and then click Save.
4. Modify the pipeline as desired, for example, by adding, configuring, and reordering stages.
5. Click Save.

### 2. Create the query profile and the experiment

In the Fusion UI, you can use either of these equivalent approaches to set up an experiment:

- **Create the experiment in the Query Profiles UI** – Choose an existing query profile or create a new one. Then
create the experiment in the **Querying > Query Profiles UI**.

- **Create the experiment in the Experiments UI** – Create an experiment in the **Analytics > Experiments UI**, and then reference the experiment in the **Querying > Query Profiles UI**.

**Option A: Create the experiment in the Query Profiles UI**

1. **Choose or create a query profile** – Choose an existing query profile (for example, the default query profile) or create a new query profile for the experiment.

To choose an existing query profile:

   a. Navigate to **Querying > Query Profiles**.
   b. In the left pane, click the query profile you want to use.
   c. Verify that the pipeline, search handler, collection ID, and parameters are correct.
   d. (Optional) Click **New params** and specify URL parameters to add to all queries for this query profile.

To create a new query profile:

   e. Navigate to **Querying > Query Profiles**.
   f. Click **New**.
   g. Enter a **Query Profile ID** (an arbitrary name for the query profile).
   h. Choose a pipeline and a collection.
   i. (Optional) Click **New params** and specify URL parameters to add to all queries for this query profile.
2. Enable experimentation and specify experiment details in the query profile:
   
a. Click the checkbox in front of **Enable experimentation**.
   
b. Specify the percent of traffic to include in the experiment.
   
c. Click the \(V\), and then click **Add Experiment**.
   
d. Enter an arbitrary **ID (name)** for the experiment.
   
e. Verify that the **unique ID parameter** is correct. It’s the parameter that uniquely identifies each user. The default is **userId**. Correct the parameter if necessary, for example by specifying the session ID field instead.
   
f. Choose the base collection for signals. Signals resulting from requests that flow through the experiment are stored in the **_signals** collection associated with this collection.
   
g. (Optional) Enter a description for the experiment.
h. (Optional) To use a multi-armed bandit, select **Automatically Adjust Weights Between Variants**.

i. Add variants. Click **Add Variant** to add each non-control variant in your experiment.

j. For each variant:

k. Enter an arbitrary name. For the first variant, which is the control, Fusion uses the name **control**. You can change that name if you wish.

l. Click **Specify what varies** and specify what varies. Items you select are visible in the variant UI and have a green check mark in the dropdown menu. You can vary the query pipeline, query parameters (URL parameters to add to the query), and/or the collection.

m. (For query parameters) Click **New params**. In the dialog box, specify the **Parameter Name**, **Parameter Value**, and **Update Policy** for each parameter (append, default, remove, or replace).

n. Add metrics. For each metric:

o. Click **Add Metric** and select the type of metric.

p. Fill in information for the metric.
q. Click **Save** to save the experiment.

**Option B: Create the experiment in the Experiments UI**

You can create the experiment first and reference it from a query profile.

1. **Create an experiment** – The experiment defines variants and metrics, as well as the user ID parameter and the base collection for signals:
   a. Navigate to Analytics > Experiments.
   b. Click **New**.
   c. Enter an arbitrary **ID** (name) for the experiment.
   d. Verify that the **unique ID parameter** is correct. It’s the parameter that uniquely identifies each user. The default is `userId`. Correct the parameter if necessary, for example by specifying the session ID field instead.
   e. Choose the base collection for signals. Signals resulting from requests that flow through the experiment are stored in the `_signals` collection associated with this collection.
   f. (Optional) Enter a description for the experiment.
g. (Optional) To use a multi-armed bandit, select **Automatically Adjust Weights Between Variants**.

h. Add variants. Click **Add Variant** to add each non-control variant in your experiment.

i. For each variant:

j. Enter an arbitrary name. For the first variant, which is the control, Fusion uses the name **control**. You can change that name if you wish.

k. Click **Specify what varies** and specify what varies. Items you select are visible in the variant UI and have a green check mark in the dropdown menu. You can vary the query pipeline, query parameters (URL parameters to add to the query), and/or the collection.
1. (For query parameters) Click New params. In the dialog box, specify the Parameter Name, Parameter Value, and Update Policy for each parameter (append, default, remove, or replace).

m. Add metrics. For each metric:

n. Click Add Metric and select the type of metric.

o. Fill in information for the metric.

p. Click Save to save the experiment.

2. Reference the experiment from the query profile – Open the query profile you want to use for the experiment.

a. Navigate to Querying > Query Profiles.

b. Click the query profile you want to use for the experiment.

c. (Optional) If necessary, modify the query profile ID, default pipeline, and/or the search handler. These modifications aren’t related to experiments and are probably not required.

d. (Optional) Click New params and specify URL parameters to add to all queries for this query profile.

e. Click the checkbox in front of Enable experimentation.
f. Specify the percent of traffic to include in the experiment.

g. Click the `Save` button, and then click the experiment name.

h. Click **Save** to save the query profile.

**Next step**

You've set up the experiment. Next, you will run it.
Use an Experiment stage

Here we describe how to set up an experiment that uses an Experiment query pipeline stage. This approach is an alternative to setting up an experiment and a query profile that references it. Using an Experiment query pipeline stage is a bit more complicated.

How an Experiment stage works

An Experiment stage applies the experiment actions in-line in the query pipeline (wherever it is located in the pipeline), instead of performing the actions before passing queries to a query pipeline or pipelines.

An Experiment stage that apportions traffic among query pipelines is similar to a Run Query Pipeline stage, but the processing is conditional. Queries that lack a user ID parameter aren’t sent processed by the stage and aren’t sent to other pipelines (if that is what the stage does).

For the primary pipeline to process queries that don’t include a user identifier, it must contain a Solr Query stage as the last stage. If the Experiment query stage references other pipelines, then there are two options:

- **Solr Query stage as the last stage in the variant pipelines** – The variant pipelines send queries to Solr. Control doesn’t return to the primary pipeline. In the primary pipeline, the Experiment stage must be the second-to-last stage. The last stage must be the Solr Query stage.

- **No Solr Query stage in the variant pipelines** – The variant pipelines don’t send queries to Solr. Control returns from the variant pipelines to the primary pipeline. In the primary pipeline, the Experiment stage can be in any position except the last.

Before you begin

Before you set up an experiment, you must already have:

- **A search app** – The key aspect of the search app for experiments is that the search app identifies users in some way. A user ID might be associated with users persistently (this is best) or only for the current session. Searches and subsequent actions by anonymous users aren’t sent through an experiment.

- **A Fusion app** – The Fusion app provides the search functionality you want to provide. Below, you’ll modify the app to include an experiment.

- **Data for users to search** – The app should have data that users can search and search results that users can interact with. Typically, users will search a single dataset in the sense that different users aren’t given search results from different datasets. But in an experiment, different experiment variants can use data in different collections.

- **Results for users to interact with** – Experiment metrics depend on users interacting with search results, for example, clicking on them. A search app uses signals to report the interactions to Fusion.

- **A plan for the experiment** – This plan includes which control and variants to compare, the projected traffic, sample sizes, experiment duration, metrics, and decision criteria.

Basically, you need a working system in some environment on which you want to perform experiments, and a plan for experiment variants and for deciding which results are best.
1. Create an experiment

Create an experiment. The experiment defines variants and metrics, as well as the user ID parameter and the base collection for signals:

1. Navigate to Analytics > Experiments.
2. Click New.
3. Enter an arbitrary ID (name) for the experiment.
4. Verify that the unique ID parameter is correct. It’s the parameter that uniquely identifies each user. The default is userId. Correct the parameter if necessary, for example by specifying the session ID field instead.
5. Choose the base collection for signals. Signals resulting from requests that flow through the experiment are stored in the _signals collection associated with this collection.
6. (Optional) Enter a description for the experiment.
7. (Optional) To use a multi-armed bandit, select Automatically Adjust Weights Between Variants.
8. Add variants. Click Add Variant to add each non-control variant in your experiment.
9. For each variant:
   a. Enter an arbitrary name. For the first variant, which is the control, Fusion uses the name control. You can change that name if you wish.
   b. Click Specify what varies and specify what varies. Items you select are visible in the variant UI and have a green check mark in the dropdown menu. You can vary the query pipeline, query parameters (URL parameters to add to the query), and/or the collection.
   c. (For query parameters) Click New params. In the dialog box, specify the Parameter Name, Parameter Value, and Update Policy for each parameter (append, default, remove, or replace).

10. Add metrics. For each metric:
   a. Click Add Metric and select the type of metric.
   b. Fill in information for the metric.
11. Click **Save** to save the experiment.

### 2. Set up an Experiment stage

If part or all of what you will vary in the experiment is encompassed by differences in query pipelines, create additional pipelines for experiment variants. You *can’t* use the default query pipeline (the pipeline to which you are adding the Experiment stage) as one of the variants. That pipeline will be a part of all variants. Fusion directs traffic that doesn’t identify users through the default pipeline but not through the experiment.

To set up an experiment stage

1. Navigate to Querying > **Query Pipelines**.
2. Click the name of the pipeline to which you want to add the Experiment stage.
3. Click **Add a new pipeline stage** and select **Experiment stage**.
4. (Optional) Specify a label for the stage.

5. (Optional) Specify a condition that must be satisfied for queries to pass through the experiment.

6. Under **Experiment ID**, choose the experiment.

7. (Optional) Specify the percent of traffic to include in the experiment.

8. Click **Save**.

9. Drag the stage to where you want it in the pipeline.

10. Click **Save**.

You control the experiment in **Analytics** > **Experiments**.
Run an experiment

Now that you've set up an experiment, you can run it. While an experiment is running:

- Fusion receives queries from the search app.
- For the queries that identify a user, Fusion routes the specified percentage of traffic through the experiment. Fusion apportions the queries from different users among the variants in accordance with traffic weights.
- Fusion records what users do after receiving search results, for example, what links they click.

We recommend that you only run one experiment at a time.

Running an experiment involves three steps:

1. **Activate the experiment** – Activating an experiment turns on the logic that sends queries through the experiment.
2. **Users submit queries and interact with results** – Queries and signals about interactions with results come from a search app. In this topic, we explain how to verify that data is flowing through the experiment correctly.
3. **Deactivate the experiment** – When enough data have been collected for metrics to be significant, you can deactivate the experiment. To run the experiment again, just activate it again. With each activation, you get new metrics.

| Tip | Metrics will calculate periodically while an experiment is running. You can also generate metrics for a running experiment manually. Just run the metrics job(s) for the metrics you want to generate. |

Activate an experiment

You can activate an experiment in either of these ways. The outcome is identical.

From the experiment

1. With the app open, navigate to Analytics > Experiments.
2. In the left pane, click the experiment you want to activate. In the upper right, click **Activate**.

From the query profile

1. With the app open, navigate to Querying > Experiments. In the left pane, click the experiment you want to activate. In the upper right, click **Activate Experiment**.

| Note | If you stop Fusion while an experiment is running, then Fusion restarts the experiment automatically the next time you start Fusion. |

Verify that data is coming into the experiment

Signals resulting from requests that flow through the experiment are stored in the `_signals` collection associated with the primary collection. You can use the Query Workbench or App Insights to examine this collection to verify that requests are being distributed among your experiment's query pipelines.
Tip

Don’t modify a running experiment. If you need to make a change, stop the experiment, make the modifications, and then start a new experiment that uses the modified object.

Deactivate an experiment

You can deactivate an experiment in either of these ways. The outcome is identical.

From the experiment

1. With the app open, navigate to Analytics > Experiments.
2. In the left pane, click the experiment you want to deactivate. In the upper right, click Deactivate.

From the query profile

1. With the app open, navigate to Querying > Experiments. In the left pane, click the experiment you want to deactivate. In the upper right, click Deactivate Experiment.

Next step

You’ve run the experiment. Next, you will analyze the experimental results.
Analyze experiment results

After you have run an experiment, you can analyze the results. When you stop an experiment, Fusion runs jobs that calculate metrics for the data that were collected. All jobs associated with an experiment are prefixed with the name of the experiment, that is, `<experiment-name>-<metric-name>`. For the Query Relevance metric, there are two jobs: `<experiment-name>-groundTruth-<metric-name>` and `<experiment-name>-rankingMetrics-<metric-name>`.

You can also have Fusion generate metrics while an experiment is still running, by running metrics jobs by hand.

Default schedules for metrics jobs

When you activate an experiment, Fusion AI schedules metrics jobs for the experiment. These are the default schedules for metrics jobs:

Ground Truth (used for the Query Relevance metric):

- **First run** – *Not scheduled*. The first time, you must run the Ground Truth job by hand.
- **Subsequent runs** – Every month until the experiment is stopped (by default; you can specify a different schedule)

All other metrics jobs:

- **First run** – 20 minutes after the experiment starts
- **Subsequent runs** – Every 24 hours until the experiment is stopped (by default; you can specify a different schedule)
- **Last run** – Immediately after the experiment is stopped

Modify metrics jobs schedules (optional)

You can modify the default schedule as follows:

To modify the schedules of periodic metrics jobs

1. Navigate to the experiment: Analytics ➔ Experiments.
2. Next to each metric, find the **Processing Schedule** link. This link is active even if the experiment is running.
3. Edit the schedule as desired.
4. Click **Save**.

Periodic runs of metrics jobs are intended to give you up-to-date metrics. The metrics are always calculated from the beginning of the experiment.

| Tip | Even with periodically updated metrics, we recommend that you let an experiment run its course before drawing conclusions and taking action. |

Check the last time metrics jobs ran

When you view experiment metrics and statistics in App Insights, that information reflects the experiment’s state the last time the metrics jobs ran. When you stop an experiment, it is especially important that you verify that the end-of-experiment metrics jobs have run.
To check the last time metrics jobs ran

1. Navigate to Collections > Jobs.
2. In the Filter field, enter the experiment name.
   This displays only the experiment jobs.
3. Examine the Last run value below each job name.

Consider the metrics produced

After metrics jobs run, you can view the metrics that they have produced in App Insights. For more information about the metrics, read this topic.

Statistical significance

Statistical significance calculations inform you whether differences among experiment variants are likely to result from random chance, as opposed to real causal effects.

Fusion AI provides two measures of statistical significance:

- **Confidence index** – The confidence index expresses the confidence that the experiment results are statistically significant. It takes into account the current sample size of the experiment, the required sample size to accurately establish statistical significance, as well as the calculated p-value.
- **Percent chance of beating** – The percent chance of beating uses a Bayesian algorithm to calculate the percent chance that another variant performs better than the control.

Confidence index

The confidence index expresses the confidence that the experiment results are statistically significant. It gives you a gauge for whether the differences between variants are due to a causal effect (as opposed to random chance). The confidence index combines two concepts: the minimum sample size, and the p-value. If the number of samples is lower than the minimum sample size, then the confidence index is based entirely on the percentage of sample size. If the number of samples is above the minimum sample size, then the confidence index directly related to the p-value generated using Welch’s t-test, which is a variation of the Student's t-test. Welch’s t-test is better than the Student's t-test when samples have unequal variances and/or sample sizes.

The test is a pairwise test, with each comparison being two-tailed (there is no a priori assumption that the difference will be in a specific direction). Fusion AI compares each variant against the first variant (the control), and generates a p-value for the comparison. The confidence index score is based on the lowest p-value amongst the variants.

The confidence index is this, rounded to the nearest whole number:

\[
CI = 100 \times (1-p)
\]

You can recover two digits of the p-value from the confidence index as follows:

\[
p = 1 - CI/100
\]
Percent chance of beating

The percent chance of beating uses a Bayesian algorithm to calculate the percent chance that another variant than the control does better than the control.

When calculating the percent chance of beating, Fusion AI uses up to 30 days of historical signal data to establish a baseline to compare against. The baseline is useful but not required. If the historical data is available, an experiment can reach higher confidence numbers more quickly.

Fusion AI calculates historical metrics one time and stores them, so subsequent runs of the metrics calculation jobs won't need to recalculate them.

<table>
<thead>
<tr>
<th>Note</th>
<th>At the moment, percent chance of beating is only accessible through the Fusion AI API, not through App Insights. Use the metrics endpoint hostname:8764/api/experiments/experiment-name/metrics.</th>
</tr>
</thead>
</table>

Best practices

Note the following best practices regarding statistical significance:

- **If you peek, don't act** – P-values only reach significant levels when there is enough data. This leads to the problem of peeking (when people look at experiment results too early and make incorrect decisions). Wait until an experiment is over before making decisions based on the experiment. The confidence index is intended to encourage this practice.

- **Don't modify running experiments** – To modify an experiment, you have to stop it, and data collection for the experiment stops. This is nice and clean and as it should be. You could, however, modify some object that the experiment uses (for example, you could modify a query pipeline) while the experiment is running. But this makes it unclear what you have been testing. We recommend against this practice. Instead, stop the first experiment, make the modifications, and then activate (start) an experiment that uses the modified object.
App Insights

App Insights provides detailed, real-time, searchable reports and visualizations derived from your signals data. It also provides alerts and triggers to notify you when specific events occur.

| Note | Signals data from Fusion 3.x or earlier will not produce useful visualizations in App Insights. Signals generated with Fusion 4.0 will produce the best results. |

- To open App Insights from the Fusion workspace, navigate to **Analytics > Insights**.
- To switch apps while in App Insights, hover over and then click a different app.
- To exit App Insights, hover over and then click **Return to Fusion**:
App Insights pages

When App Insights is open, you can hover over the left navigation panel to reach these pages:

- **Dashboards**
  View graphs and tables about:
  - requests
  - queries
  - results
  - clicks
  - users
  - sessions

  You can filter by timeframe or by the content of the data, and create custom reports.

- **Events**
  View histograms and timelines about events, filtered by:
  - event type
  - users
  - request/response information

  You can also create custom reports about events here.

- **Sessions**
  View charts and timelines about search events, filtered by:
  - user
  - subject
  - event duration
  - events per session

- **Analytics**
  A variety of standard reports are available here. You can also define custom reports to suit your needs.

- **Experiment results**
  If you have configured an experiment, you can see visualizations about the results here.
Reports

App Insights provides a standard set of reports, plus the ability to create custom reports. Standard reports are located on the Analytics page, while custom reports can be defined on the Dashboard, Events, or Analytics pages. Report data can be filtered by time or by free text search, and reports can be exported by clicking Export table.
App Insights Dashboards

When you open the Dashboards page, it displays graphs and tables for attributes that are common to all signals. For more specific charts, you can open the All Dashboard Categories menu at the top and select a category:
Dashboard filtering

You can filter by timeframe using the All Time menu:

To filter by the content of the data, enter a string in the Filter this dashboard... text box:
Custom reports

There are several ways to create a custom report:

On the Events or Analytics pages

1. Apply one or more filters.
2. Click New custom report.
3. Define the report's parameters.
4. Click Create and launch report.

On the Dashboard page

1. Filter any dashboard.
2. Scroll to the bottom.
3. Click <n> events match your criteria. View Now.

   20,380 events match your criteria. View Now.

This takes you to the Events page, where you can click New Custom Report. Define additional parameters for the report, then click Create and launch report.
Events

Click Events to view histograms and timelines about events, filtered by:

- event type
- users
- request/response information

Custom reports

There are several ways to create a custom report:

On the Events or Analytics pages

1. Apply one or more filters.
2. Click New custom report.
3. Define the report’s parameters.
4. Click Create and launch report.

On the Dashboard page

1. Filter any dashboard.
2. Scroll to the bottom.
3. Click <n> events match your criteria. View Now.

20,380 events match your criteria. View Now.
This takes you to the Events page, where you can click **New Custom Report**. Define additional parameters for the report, then click **Create and launch report**.
Sessions

Click **Sessions** to view charts and timelines about search events, filtered by:

- user
- subject
- event duration
- events per session

Charts are located on the left, below the filters. Scroll down or minimize the filters to view the charts:

Hover over the bars on any chart to filter the timeline by event duration or events per session:
NUMBER OF EVENTS DURING A SESSION

6 252 sessions with 2 events
Analytics

On the Analytics page, App Insights provides a standard set of reports, plus the ability to create custom reports. Custom reports can also be defined on the Dashboards and Events pages. Report data can be filtered by time or by free text search, and reports for tabular data can be exported by clicking Export table.

Standard reports

To view the standard reports, click Analytics, then select one of the standard reports:

- Facets used
- Facet filters applied
- Application servers
- Applications
- Response times
- URLs clicked
- Type of query
- Search pages
- Search platforms
- Types of response
- Visitor countries
- Visitor cities
- Browsers
- Operating System
- Device types
- Websites people are coming from
Custom reports

There are several ways to create a custom report:

On the Events or Analytics pages

1. Apply one or more filters.
2. Click New custom report.
3. Define the report’s parameters.
4. Click Create and launch report.

On the Dashboard page

1. Filter any dashboard.
2. Scroll to the bottom.
3. Click <n> events match your criteria. View Now.

20,380 events match your criteria. View Now.

This takes you to the Events page, where you can click New Custom Report. Define additional parameters for the report, then click Create and launch report.
Experiment Results

If you have configured an experiment, you can see visualizations about the results here.

This example shows a 92% confidence index for the difference in click-through rate (CTR). Conversions use the same signal type (click), so the result is identical.
Import Data with the Parallel Bulk Loader (PBL)

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

The Parallel Bulk Loader leverages the popularity of Spark as a prominent distributed computing platform for big data. A number of companies invest heavily in building production-ready Spark SQL data source implementations for big data and NoSQL systems, much as Lucidworks has done for Solr. The Parallel Bulk Loader uses connectors provided by the experts who develop these complex systems.
Available data sources

The Parallel Bulk Loader can load documents from any system that implements the Data Sources API for Spark SQL 2.2.1 or later.

These are data sources that the Parallel Bulk Loader can use. For data sources that use Spark SQL connectors, the source of the connector is indicated.

- Solr databases
  - Connector (Lucidworks): spark-solr
- Files in these common formats: JSON, CSV, XML, Apache Avro, and Apache Parquet
- JDBC-compliant databases
- Apache HBase databases
  - Connector (Hortonworks): Apache Spark - Apache HBase Connector
    - Spark-on-HBase
- Datasets accessible through Apache Hive
- Apache Cassandra NoSQL databases
  - Connector (DataStax): Spark-Cassandra connector
- Elastic databases
  - Connector: Elasticsearch-Hadoop connector
    - Use the package: org.elasticsearch:elasticsearch-spark-20_2.11:6.2.2
- MongoDB databases
- Riak databases
- Couchbase NoSQL databases
- Redis in-memory data structure stores
- Google data sources, including Google Analytics, Sheets, and BigQuery
  - Connectors:
    - Analytics
    - Sheets
    - Big Query
- Microsoft Azure DataLake, Cosmos DB, and SQL Database
Key features

Key features of the Parallel Bulk Loader are:

- **Load distribution** – To distribute load and maximize performance, the Parallel Bulk Loader parallelizes operations and distributes them across the Fusion Spark cluster.

- **No parsing** – No parsing is needed. The Spark Data Sources API returns a DataFrame (RDD + schema) that has an easy-to-use tabular format.

- **Dynamic resolution of dependencies** – There is nothing to download or install. Users just provide the Maven coordinates of dependencies during configuration, and Spark distributes the necessary JAR files to worker nodes.

- **Leverage integration libraries** – Similar to JDBC, the Parallel Bulk Loader leverages integration libraries built by the experts of the underlying systems, for example, Databricks, DataStax, Hortonworks, Elastic, Lucidworks, Microsoft, and so forth.

- **Direct write operations** – The Parallel Bulk Loader writes directly to Solr (for maximum performance) or to Fusion Server index pipelines (for maximum flexibility).

- **Solr atomic updates** – The Parallel Bulk Loader uses atomic updates to update existing documents in Solr.

- **Incremental queries** – To obtain the latest changes to data sources, macros built into the Parallel Bulk Loader use timestamps to filter queries.

- **Seamless integration with Spark-NLP** Do natural language processing, including part-of-speech tagging, stemming or lemmatization, sentiment analysis, named-entity recognition, and other NLP tasks.

- **SQL joins** – Use SQL to join data from multiple Solr collections.

- **Load Fusion ML models** – To classify incoming documents, load Fusion Machine Learning models stored in the Fusion blob store.

- **SQL transformations** – Leverage the full power of the Spark Scala’s DataFrame APIs and SQL to filter and transform data.

- **UDF and UDAF functions** – Select from hundreds of user-defined functions and user-defined aggregate functions.
Differences between the Parallel Bulk Loader and Fusion classic connectors

The primary difference between the Bulk Loader and Fusion classic connectors is that the Bulk Loader uses Spark SQL and Spark/Solr integration to perform distributed reads from data sources.

Here are some examples of how the Parallel Bulk Loader performs distributed reads:

- **HBase table** – To support high-volume data ingestion into Solr, the Parallel Bulk Loader can distribute queries sent to HBase tables across multiple region servers.
- **Parquet files** – The Parallel Bulk Loader processes a directory of Parquet files in HDFS in parallel using the built-in support for computing splits in HDFS files.
- **Spark/Solr integration** – With Spark/Solr integration, the Parallel Bulk Loader uses a Spark-Solr data source to send queries to all replicas of a collection, so it can read from Solr in parallel across the Spark cluster.

This diagram depicts how the Spark-Solr data source partitions queries across all shards/replicas of a Solr collection to better utilize cluster resources and to improve read performance. Most Spark-SQL data sources do something similar for their respective databases, which is one of the main benefits of using the Parallel Bulk Loader job.

In contrast, most classic connectors have only a single-fetcher mode. To scale the fetching with classic connectors, you must distribute the connector itself, which differs from relying on the built-in parallelization of Spark.

Also, most classic connectors rely on Fusion parsers and index pipelines to prepare data for indexing, whereas no parsing is needed for the Parallel Bulk Loader, which can achieve maximum indexing performance by writing directly to Solr.
Create and run Parallel Bulk Loader jobs

Use the Jobs manager to create and run Parallel Bulk Loader jobs. You can also use the Scheduler to schedule jobs.

In the procedures, select Parallel Bulk Loader as the job type and configure the job as needed.
## Configuration settings for the Parallel Bulk Loader job

This section provides configuration settings for the Parallel Bulk Loader job. Also see configuration properties in the Jobs Configuration Reference.

### Read settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>format</strong></td>
<td>Unique identifier of the data source provider. Spark scans the job’s <code>classpath</code> for a class named <code>DefaultSource</code> in the <code>&lt;format&gt;</code> package. For example, for the <code>solr</code> format, we provide the <code>solr.DefaultSource</code> class in spark-solr: <a href="https://github.com/lucidworks/spark-solr/blob/master/src/main/scala/solr/DefaultSource.scala">https://github.com/lucidworks/spark-solr/blob/master/src/main/scala/solr/DefaultSource.scala</a></td>
</tr>
<tr>
<td><strong>path (optional)</strong></td>
<td>Comma-delimited list of paths to load. Some data sources, such as parquet, require a path. Others, such as Solr, don’t. Refer to the documentation for your data source to determine if you need to provide a path.</td>
</tr>
<tr>
<td><strong>readOptions</strong></td>
<td>Options passed to the Spark SQL data source to configure the read operation. Options differ for every data source. Refer to the specific data source documentation for more information.</td>
</tr>
<tr>
<td><strong>sparkConfig (optional)</strong></td>
<td>List of Spark configuration settings needed to run the Parallel Bulk Loader.</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>shellOptions</td>
<td>Behind the scenes, the Parallel Bulk Loader job submits a Scala script to the Fusion Spark shell. The shellOptions setting lets you pass any additional options needed by the Spark shell. The two most common options are --packages and --repositories:</td>
</tr>
<tr>
<td></td>
<td><strong>--packages</strong></td>
</tr>
<tr>
<td></td>
<td>Comma-separated list of Maven coordinates of JAR files to include on the driver and executor classpaths. Spark searches the local Maven repository, and then Maven central and any additional remote repositories given in the config. The format for the coordinates should be groupId:artifactId:version. The HBase example below demonstrates the use of the packages option for loading the com.hortonworks:shc-core:1.1.1-2.1-s_2.11 package.</td>
</tr>
<tr>
<td></td>
<td><strong>--repositories</strong></td>
</tr>
<tr>
<td></td>
<td>Comma-separated list of additional remote Maven repositories to search for the Maven coordinates given in the packages config setting. The Index HBase tables example below demonstrates the use of the repositories option for loading the com.hortonworks:shc-core:1.1.1-2.1-s_2.11 package from the <a href="http://repo.hortonworks.com/content/repositories/releases">http://repo.hortonworks.com/content/repositories/releases</a> repository.</td>
</tr>
</tbody>
</table>

Use the [https://spark-packages.org/](https://spark-packages.org/) site to find interesting packages to add to your Parallel Bulk Loader jobs.
For datasources that support time-based filters, the Parallel Bulk Loader computes the timestamp of the last document written to Solr and the current timestamp of the Parallel Bulk Loader job. For example, the HBase data source lets you filter the read between a `MIN_STAMP` and `MAX_STAMP`, for example:

```scala
val timeRangeOpts = Map(HBaseRelation.MIN_STAMP -> minStamp.toString, HBaseRelation.MAX_STAMP -> maxStamp.toString)
```

This lets Parallel Bulk Loader jobs run on schedules, and pull only the newest rows from the underlying datasources.

To support timestamp based filtering, the Parallel Bulk Loader provides two simple macros:

- `$lastTimestamp(format)`
- `$nowTimestamp(format)`

The `format` argument is optional. If not supplied, then an ISO-8601 date/time string is used. The `timestampFieldName` setting is used to determine the value of `lastTimestamp`, using a Top 1 query to Solr to get the max timestamp. You can also pass `$lastTimestamp(EPOCH)` or `$lastTimestamp(EPOCH_MS)` to get the timestamp in seconds or milliseconds.

See the Index HBase tables example below for an example of using this configuration property.

### Transformation settings
Sometimes, you can write a small script to transform input data into the correct form for indexing. But at other times, you might need the full power of the Spark API to transform data into an indexable form.

The `transformScala` option lets you filter and/or transform the input DataFrame any way you’d like. You can even define UDFs to use during your transformation. For an example of using Scala to transform the input DataFrame before indexing in Solr, see the Read from Parquet example.

Another powerful use of the `transformScala` option is that you can pull in advanced libraries, such as Spark NLP (from John Snow Labs) to do NLP work on your content before indexing. See the Use NLP during indexing example.

Your Scala script can do other things but, at a minimum, it must define the following function that the Parallel Bulk Loader invokes:

```scala
def transform(inputDF: Dataset[Row]) : Dataset[Row] = {
    // do transformations and/or filter the inputDF here
}
```

Your script can rely on the following vals:
- `spark`: SparkSession
- `sc`: SparkContext
- `fusionZKConn`: ZKConnection // needed to access Fusion API
- `solrCollection`: SolrCollection // output collection
- `jobId`: Loader job config ID

Also, the following classes have already been imported:
- `import org.apache.spark.SparkContext._`
- `import spark.implicits._`
- `import spark.sql`
- `import org.apache.spark.sql.functions._`
- `import com.lucidworks.spark.util.{SparkShellSupport => _lw}`
- `import com.lucidworks.spark.job.sql.SparkSQLExceptionLoader`
- `import com.lucidworks.spark.ml.recommenders.SolrCollection`
- `import com.lucidworks.spark.ZKConnection`
- `import org.apache.spark.sql.{Dataset, Row}`
### Setting

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transformSql</td>
<td>The <code>transformSql</code> option lets you write a SQL query to transform the input DataFrame. The SQL is executed after the <code>transformScala</code> script (if both are defined). The input DataFrame is exposed to your SQL as the <code>_input</code> view. See the Clean up data with SQL transformations example below for an example of using SQL to transform the input before indexing in Solr. This option also lets you leverage the UDF/UDAF functions provided by Spark SQL.</td>
</tr>
</tbody>
</table>
| mlModelId | If you have a Spark ML PipelineModel loaded into the blob store, you can supply the blob ID to the Parallel Bulk Loader and it will:  
  1. Load the model from the blob store.  
  2. Transform the input DataFrame (after the Scala transform but before the SQL transform).  
  3. Add the predicted output field (specified in the model metadata stored in the blob store) to the projected fields list.  

This lets you use Spark ML models to make predictions in a more scalable, performant manner than what can be achieved with a Machine Learning index stage. |

### Output settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputCollection</td>
<td>Name of the Fusion collection to write to. The Parallel Bulk Loader uses the Collections API to resolve the underlying Solr collection at runtime.</td>
</tr>
<tr>
<td>outputIndexPipeline</td>
<td>Name of a Fusion index pipeline to which to send documents, instead of directly indexing to Solr. This option lets you perform additional ETL (extract, transform, and load) processing on the documents before they are indexed in Solr. If you need to write to time-partitioned indexes, then you must use an index pipeline, because writing directly to Solr is not partition aware.</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>defineFieldsUsingInputSchema</td>
<td>Flag to indicate if the Parallel Bulk Loader should use the input schema to create fields in Solr, after applying the Scala and/or SQL transformations. If false, then the Parallel Bulk Loader relies on the Fusion index pipeline and/or Solr field guessing to create the fields. If true, only fields that don’t exist already in Solr are created. Consequently, if there is a type mismatch between an existing field in Solr and the input schema, you’ll need to use a transformation to rename the field in the input schema.</td>
</tr>
<tr>
<td>clearDatasource</td>
<td>If checked, the Parallel Bulk Loader deletes any existing documents in the output collection that match the query _lw_loader_id_s:&lt;JOB&gt;. Consequently, the Parallel Bulk Loader adds two metadata fields to each row: _lw_loader_id_s and _lw_loader_job_id_s.</td>
</tr>
<tr>
<td>atomicUpdates</td>
<td>Flag to send documents directly to Solr as atomic updates instead of as new documents. This option is not supported when using an index profile. Also note that the Parallel Bulk Loader tracking fields _lw_loader_id_s and _lw_loader_job_id_s are not sent when using atomic updates, so the clear datasource option doesn’t work with documents created using atomic updates.</td>
</tr>
<tr>
<td>outputOptions</td>
<td>Options used when writing directly to Solr. See Spark-Solr: <a href="https://github.com/lucidworks/spark-solr#index-parameters">https://github.com/lucidworks/spark-solr#index-parameters</a></td>
</tr>
<tr>
<td></td>
<td>For example, if your docs are relatively small, you might want to increase the batch_size (2000 default) as shown below:</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Write Options" /></td>
</tr>
<tr>
<td>outputPartitions</td>
<td>Coalesce the DataFrame into N partitions before writing to Solr. This can help spread the indexing work out across more executors that are available in Spark, or limit the parallelism when writing to Solr.</td>
</tr>
</tbody>
</table>
Tune performance

As the name of the Parallel Bulk Loader job implies, it's designed to ingest large amounts of data into Fusion by parallelizing the work across your Spark cluster. To achieve scalability, you might need to increase the amount of memory and/or CPU resources allocated to the job.

By default, Fusion’s Spark configuration settings control the resources allocated to Parallel Bulk Loader jobs.

You can pass these properties in the job configuration to override the default Spark shell options:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description and Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>--driver-cores</td>
<td>Cores for the driver</td>
</tr>
<tr>
<td></td>
<td>Default: 1</td>
</tr>
<tr>
<td>--driver-memory</td>
<td>Memory for the driver (for example, 1000M or 2G)</td>
</tr>
<tr>
<td></td>
<td>Default: 1024M</td>
</tr>
<tr>
<td>--executor-cores</td>
<td>Cores per executor</td>
</tr>
<tr>
<td></td>
<td>Default: 1 in YARN mode, or all available cores on the worker in standalone mode</td>
</tr>
<tr>
<td>--executor-memory</td>
<td>Memory per executor (for example, 1000M or 2G)</td>
</tr>
<tr>
<td></td>
<td>Default: 16</td>
</tr>
<tr>
<td>--total-executor-cores</td>
<td>Total cores for all executors</td>
</tr>
<tr>
<td></td>
<td>Default: Without setting this parameter, the total cores for all executors is the number of executors in YARN mode, or all available cores on all workers in standalone mode.</td>
</tr>
</tbody>
</table>

Spark Shell Options

Additional options to pass to the Spark shell when running this job.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>--total-executor-cores</td>
<td>8</td>
</tr>
<tr>
<td>--executor-memory</td>
<td>2g</td>
</tr>
<tr>
<td>--executor-cores</td>
<td>4</td>
</tr>
</tbody>
</table>
Examples

Here we provide screenshots and example JSON job definitions to illustrate key points about how to load from different data sources.

Use NLP during indexing

In this example, we leverage the John Snow labs NLP library during indexing. This is just quick-and-dirty to show the concept.

Also see:

https://github.com/JohnSnowLabs/spark-nlp

https://databricks.com/blog/2017/10/19/introducing-natural-language-processing-library-apache-spark.html
Use this transform Scala script:

```scala
val pipeline = new Pipeline().setStages(Array(documentAssembler, sentenceDetector, finisher)).fit(inputDF).transform(inputDF).drop("document").drop("sent")
```

Spark Shell Options

Additional options to pass to the Spark shell when running this job.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>--packages</td>
<td>JohnSnowLabs:spark-nlp:1.4.2</td>
</tr>
</tbody>
</table>
Be sure to add the `JohnSnowLabs:spark-nlp:1.4.2` package using Spark Shell Options.

**Clean up data with SQL transformations**

Fusion has a Local Filesystem connector that can handle files such as CSV and JSON files. Using the Parallel Bulk Loader lets you leverage features that are not in the Local Filesystem connector, such as using SQL to clean up the input data.
Use the following SQL to clean up the input data before indexing:

```sql
/* Spark Job ID */
-------------
CSV

/* Format */
--------
CSV

Path
-----
/Users/tip/dev/lw/projects/fusion-spark-bootcamp/labs/movie lens/ml-100k/u.user

** Read Options **

Options passed to the SparkSQL data source to configure the read operation; options differ for every data source so refer to the documentation for more information.

+ Parameter Name  Parameter Value
| delimiter       | |  
| header          | false |

** Output Collection **

users

-----
SELECT _c0 as user_id,
       CAST(_c1 as INT) as age,
       _c2 as gender,
       _c3 as occupation,
       _c4 as zip_code
FROM _input

Job JSON:
{
  "type": "parallel-bulk-loader",
  "id": "csv",
  "format": "csv",
  "path": "/Users/tjp/dev/lw/projects/fusion-spark-bootcamp/labs/movielens/ml-100k/u.user",
  "readOptions": [
    {
      "key": "delimiter",
      "value": "|"
    },
    {
      "key": "header",
      "value": "false"
    }
  ],
  "outputCollection": "users",
  "clearDatasource": false,
  "defineFieldsUsingInputSchema": true,
  "atomicUpdates": false,
  "transformSql": "SELECT _c0 as user_id, 
                   CAST(_c1 as INT) as age, 
                   _c2 as gender,
                   _c3 as occupation, 
                   _c4 as zip_code 
FROM _input"
}

Read from S3

It's easy to read from an S3 bucket without pulling data down to your local workstation first. To avoid exposing your AWS credentials, add them to a file named core-site.xml in the apps/spark-dist/conf directory, such as:

```xml
<configuration>
  <property>
    <name>fs.s3a.access.key</name>
    <value>???</value>
  </property>
  <property>
    <name>fs.s3a.secret.key</name>
    <value>???</value>
  </property>
</configuration>
```

Then you can load files using the S3a protocol, such as: s3a://sstk-dev/data/u.user. If you are running a Fusion cluster then each instance of Fusion will need a core-site.xml file. S3a is the preferred protocol for reading data into Spark because it uses Amazon’s libraries to read from S3 instead of the legacy Hadoop libraries. If you need other S3 protocols (for example, s3 or s3n) you’ll need to add the equivalent properties to core-site.xml.
You’ll need to add the `org.apache.hadoop:hadoop-aws:2.7.3` package to the job using the `--packages` Spark option. Also, you’ll need to exclude the `com.fasterxml.jackson.core:jackson-core,joda-time:joda-time` packages using the `--exclude-packages` option.

You can also read from Google Cloud Storage (GCS), but you’ll need a few more properties in your `core-site.xml`; see Installing the Cloud Storage connector.

### Spark Shell Options

Additional options to pass to the Spark shell when running this job.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>--packages</td>
<td><code>org.apache.hadoop:hadoop-aws:2.7.3</code></td>
</tr>
<tr>
<td>--exclude-packages</td>
<td><code>com.fasterxml.jackson.core:jackson-core,joda-time:joda-time</code></td>
</tr>
</tbody>
</table>

You can also read from Google Cloud Storage (GCS), but you’ll need a few more properties in your `core-site.xml`; see Installing the Cloud Storage connector.

### Read from Parquet

Reading from parquet files is built into Spark using the "parquet" format. For additional read options, see Configuration.
of Parquet.

ecomm demo parquet signals
Loads data from a SparkSQL datasource and sends to Solr directly or to an index profile for additional ETL processing.

* Spark Job ID
  ecomm demo parquet signals

* Format
  parquet

Path
/Users/tjp/dev/lw/projects/fusion-ecommerce-demo/datasets/bestbuy/bestbuy-signals-

Read Options
Options passed to the SparkSQL data source to configure the read operation.

+ Parameter Name Parameter Value

   Click the green plus icon above to add a row

* Output Collection
  best-buy_signals

Job JSON:

```json
{
  "type": "parallel-bulk-loader",
  "id": "ecomm demo parquet signals",
  "format": "parquet",
  "path": "/Users/tjp/dev/lw/projects/fusion-ecommerce-demo/datasets/bestbuy/bestbuy-signals-",
  "outputCollection": "best-buy_signals",
  "clearDatasource": false,
  "defineFieldsUsingInputSchema": true,
  "atomicUpdates": false
}
```

This example also uses the `transformScala` option to filter and transform the input DataFrame into a better form for indexing using the following Scala script:
import java.util.Calendar
import java.util.Locale
import java.util.TimeZone

def transform(inputDF: Dataset[Row]) : Dataset[Row] = {
  // do transformations and/or filter the inputDF here
  val signalsDF = inputDF.filter((unix_timestamp("timestamp_tdt", "MM/dd/yyyy HH:mm:ss.SSS") < 1325376000))
  val now = System.currentTimeMillis()
  val maxDate = signalsDF.agg(max("timestamp_tdt")).take(1)(0).getAs[java.sql.Timestamp](0).getTime
  val diff = now - maxDate
  val add_time = udf((t: java.sql.Timestamp, diff : Long) => new java.sql.Timestamp(t.getTime + diff))
  val day_of_week = udf((t: java.sql.Timestamp) => {
    val calendar = Calendar.getInstance(TimeZone.getTimeZone("UTC"))
    calendar.setTimeInMillis(t.getTime)
    calendar.getDisplayName(Calendar.DAY_OF_WEEK, Calendar.LONG, Locale.getDefault)
  })

  //Remap some columns to bring the timestamps current
  signalsDF
    .withColumnRenamed("timestamp_tdt", "orig_timestamp_tdt")
    .withColumn("timestamp_tdt", add_time("orig_timestamp_tdt", lit(diff)))
    .withColumn("date", "timestamp_tdt")
    .withColumn("tx_timestamp_txt", date_format("timestamp_tdt", "E YYYY-MM-d HH:mm:ss.SSS 'Z'"))
    .withColumn("param.query_time_dt", "timestamp_tdt")
    .withColumn("date_day", date_format(date_sub("date", 0), "YYYY-MM-d'T'HH:mm:ss.SSS'Z'"))
    .withColumn("date_month", date_format(trunc("date", "mm"), "YYYY-MM-d'T'HH:mm:ss.SSS'Z'"))
    .withColumn("date_year", date_format(trunc("date", "yyyy"), "YYYY-MM-d'T'HH:mm:ss.SSS'Z'"))
    .withColumn("day_of_week", day_of_week("date"))
}
Read from a table

**load dbtable**
Loads data from a SparkSQL datasource and sends to Solr directly or to an index profile for additional ETL processing.

**Spark Job ID**
load dbtable

**Format**
jdbc

**Path**

**Read Options**
Options passed to the SparkSQL data source to configure the read operation.

<table>
<thead>
<tr>
<th>* Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>jdbc:mysql://localhost/employees?us</td>
</tr>
<tr>
<td>dbtable</td>
<td>employees</td>
</tr>
<tr>
<td>partitionColumn</td>
<td>emp_no</td>
</tr>
<tr>
<td>numPartitions</td>
<td>4</td>
</tr>
<tr>
<td>driver</td>
<td>com.mysql.jdbc.Driver</td>
</tr>
<tr>
<td>lowerBound</td>
<td>$MIN(emp_no)</td>
</tr>
<tr>
<td>upperBound</td>
<td>$MAX(emp_no)</td>
</tr>
</tbody>
</table>

**Output Collection**
employees

For more information on reading from JDBC-compliant databases, see:

http://spark.apache.org/docs/latest/sql-programming-guide.html#jdbc-to-other-databases
Notice the use of the $\text{MIN}(\text{emp}\_\text{no})$ and $\text{MAX}(\text{emp}\_\text{no})$ macros in the read options. These are macros offered by the Parallel Bulk Loader to help configure parallel reads of JDBC tables. Behind the scenes, the macros are translated into SQL queries to get the MAX and MIN values of the specified field, which Spark uses to compute splits for partitioned queries. As mentioned above, the field must be numeric and must have a relatively balanced distribution of values between MAX and MIN; otherwise, you're unlikely to see much performance benefit to partitioning.

**Index HBase tables**

To index an HBase table, use the Hortonworks connector found at [https://github.com/hortonworks-spark/shc](https://github.com/hortonworks-spark/shc).

<table>
<thead>
<tr>
<th>Note</th>
<th>The Parallel Bulk Loader lets us replace the HBase Indexer.</th>
</tr>
</thead>
</table>

You'll need to add an hbase-site.xml (and possibly core-site.xml) to apps/spark-dist/conf in Fusion, for example:
For this example, we'll create a test table in HBase. If you already have a table in HBase, feel free to use that table instead.

1. Launch the HBase shell and create a table named `fusion_nums` with a single column family named `lw`:

   ```
   create 'fusion_nums', 'lw'
   ```

2. Do a list command to see your table:

   ```
   hbase(main):002:0> list
   TABLE
   fusion_nums
   1 row(s) in 0.0250 seconds
   => ["fusion_nums"]
   ```

3. Fill the table with some data:

   ```
   for i in '1'..'100' do for j in '1'..'2' do put 'fusion_nums', "row#{i}", "lw:c#{j}", "#{i}#{j}" end end
   ```

4. Scan the `fusion_nums` table to see your data:

   ```
   scan 'fusion_nums'
   ```

The HBase connector requires a catalog read option that defines the columns you want to read and how to map them into a Spark DataFrame. For our sample table, the following suffices:
hbase

Loads data from a SparkSQL datasource and sends to Solr directly or to an index profile for additional ETL processing.

* Spark Job ID

hbase

* Format

org.apache.spark.sql.execution.datasources.hbase

Path

---

Read Options

Options passed to the SparkSQL data source to configure the read operation; options differ for every data source so refer to the documentation for more information.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>catalog</td>
<td><code>{ &quot;table&quot;: {&quot;namespace&quot;: &quot;default&quot;, &quot;name&quot;: &quot;fusion_nums&quot;}, &quot;rowkey&quot;: &quot;key&quot;, &quot;columns&quot;: { &quot;id&quot;: {&quot;cf&quot;: &quot;rowkey&quot;, &quot;col&quot;: &quot;key&quot;, &quot;type&quot;: &quot;string&quot;}, &quot;lw_c1_s&quot;: {&quot;cf&quot;: &quot;lw&quot;, &quot;col&quot;: &quot;c1&quot;, &quot;type&quot;: &quot;string&quot;}, &quot;lw_c2_s&quot;: {&quot;cf&quot;: &quot;lw&quot;, &quot;col&quot;: &quot;c2&quot;, &quot;type&quot;: &quot;string&quot;} } }</code></td>
</tr>
<tr>
<td>minStamp</td>
<td><code>$lastTimestamp(EPOCH_MS)</code></td>
</tr>
</tbody>
</table>

* Output Collection

hbase

Notice the use of the `$lastTimestamp` macro in the read options. This lets us filter rows read from HBase using the timestamp of the last document the Parallel Bulk Loader wrote to Solr, that is, to get the newest updates from HBase only (incremental updates). Most Spark data sources provide a way to filter results based on timestamp.

Job JSON:
Index Elastic data

With Elasticsearch 6.2.2 using the org.elasticsearch:elasticsearch-spark-2.0.2_2.11:6.2.1 package, here's a Scala script to run in bin/spark-shell to index some test data:

```scala
import spark.implicits._

case class SimpsonCharacter(name: String, actor: String, episodeDebut: String)

val simpsonsDF = sc.parallelize(
  SimpsonCharacter("Homer", "Dan Castellaneta", "Good Night") ::
  SimpsonCharacter("Marge", "Julie Kavner", "Good Night") ::
  SimpsonCharacter("Bart", "Nancy Cartwright", "Good Night") ::
  SimpsonCharacter("Lisa", "Yeardley Smith", "Good Night") ::
  SimpsonCharacter("Maggie", "Liz Georges and more", "Good Night") ::
  SimpsonCharacter("Sideshow Bob", "Kelsey Grammer", "The Telltale Head") :: Nil).toDF()

val writeOpts = Map("es.nodes" -> "127.0.0.1",
  "es.port" -> "9200",
  "es.index.auto.create" -> "true",
  "es.resource.auto.create" -> "shows/simpsons")
simpsonsDF.write.format("org.elasticsearch.spark.sql").mode("Overwrite").save("shows/simpsons")
```
elastic
Loads data from a SparkSQL datasource and sends to Solr directly or to an index profile for additional ETL processing.

* Spark Job ID
  elastic

* Format
  org.elasticsearch.spark.sql

Path

Read Options
Options passed to the SparkSQL data source to configure the read operation; options differ for every data source so refer to the documentation for more information.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>es.nodes</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>es.port</td>
<td>9200</td>
</tr>
<tr>
<td>es.resource</td>
<td>shows/simpsons</td>
</tr>
</tbody>
</table>

Job JSON:
{
  "type": "parallel-bulk-loader",
  "id": "elastic",
  "format": "org.elasticsearch.spark.sql",
  "readOptions": [
    {
      "key": "es.nodes",
      "value": "127.0.0.1"
    },
    {
      "key": "es.port",
      "value": "9200"
    },
    {
      "key": "es.resource",
      "value": "shows/simpsons"
    }
  ],
  "outputCollection": "hbase_signals_aggr",
  "clearDatasource": false,
  "defineFieldsUsingInputSchema": true,
  "atomicUpdates": false,
  "shellOptions": [
    {
      "key": "--packages",
      "value": "org.elasticsearch:elasticsearch-spark-2.11:6.2.2"
    }
  ]
}
Licensing

Fusion Server requires a valid license. Depending on the details of your contract, your license may also enable optional connectors or Fusion AI.

When you download Fusion Server, it comes with a 30-day trial license. Contact Lucidworks to obtain a permanent license.

Fusion Server provides a license management UI and a license API for installing and managing licenses. When you upload a license, Fusion Server stores it in ZooKeeper, so you only need to upload it to one node per cluster.

The Fusion UI notifies you when your trial license is about to expire. When your license has expired, Fusion Server accepts no configuration changes until you upload a valid license.
Uploading a license using the UI

1. Log in to the Fusion UI.
2. In the upper right, open the profile menu and select **License Details**:

The License Details window appears:
3. Click **Choose License** and select your license file.

4. Click **Upload**.