Fusion AI

Version 4.0

Lucidworks | Make Search a Superpower™

Table of Contents

About Legacy Docs

This site includes legacy documentation for product versions that are no longer supported, according to our Lucidworks Fusion Product Lifecycle *(https://doc.lucidworks.com/policies/7shln5/lucidworks-version-support-lifecycle)*.

Legacy documentation is not updated or maintained. As a result, you may discover mistakes or inaccuracies within the documentation.

Recent migrations

Product	End of support	Migrated to legacy
Fusion 5.6.x	February 10, 2023	October 21, 2024
Fusion 5.7.x	June 13, 2023	October 21, 2024
Fusion 5.8.x	September 22, 2023	October 21, 2024
Managed Fusion 5.6.x	February 10, 2023	Managed Fusion documentation is not migrated.
Managed Fusion 5.7.x	June 13, 2023	Managed Fusion documentation is not migrated.
Managed Fusion 5.8.x	September 22, 2023	Managed Fusion documentation is not migrated.

Frequently asked questions

Why did you migrate legacy docs to a new site?

A new site is the best way to provide a high quality user experience. The legacy documentation site is based on this site and will continue to receive feature updates, as appropriate.

What about my saved pages?

We understand that you may have saved pages in bookmarks, shared documents, and other places. To help minimize the impact of this migration, we added page redirects from the current documentation site. However, we recommend that you update your saved pages as soon as possible.

Will more products be affected in the future?

Yes, we plan to migrate documentation for product versions that reach their end of life support to the legacy documentation site in the future. This will take place on or after the date stated in the Lucidworks Fusion Product Lifecycle *(https://doc.lucidworks.com/policies/7shln5/lucidworks-version-support-lifecycle)* policy.

Fusion AI Concepts

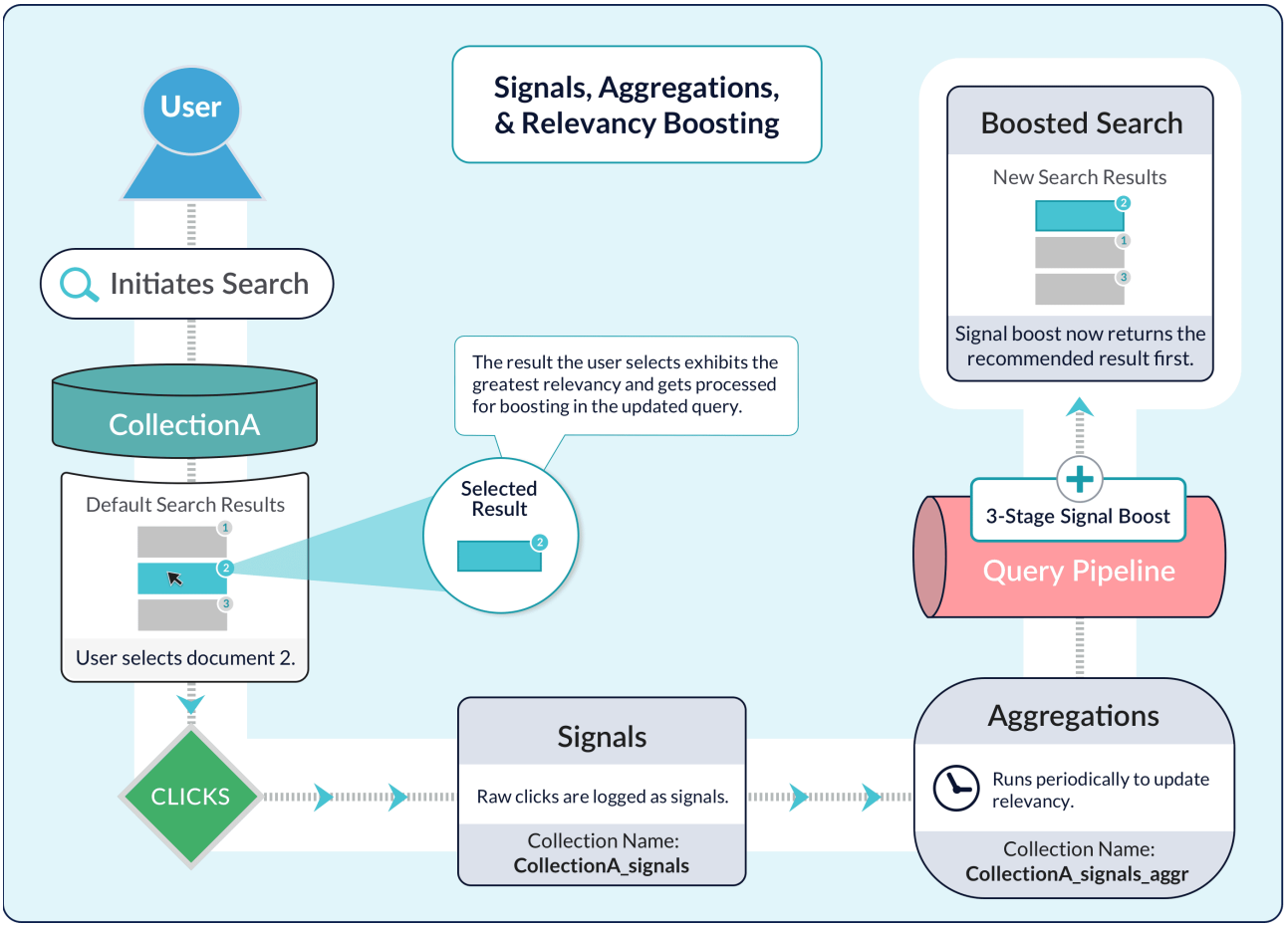
These topics explains some of the concepts behind Fusion AI's powerful features:

- *Boosting*
- *Experiments*
- *Insights*
- *Machine Learning*
- *Query Rewriting*
- *Signals and Aggregations*
- *Parallel Bulk Loader (PBL) Overview*
- *Natural Language Processing*

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.

See also these subtopics:

- *Aggregations*
- *SQL Aggregations*
- *Signals*
- *Signals Data Flow*
- *Default Signals Index Pipeline*
- *Deleting Old Signals*
- *Signals Types and Structures*

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.

See the descriptions of *signals types and structure* for more information.

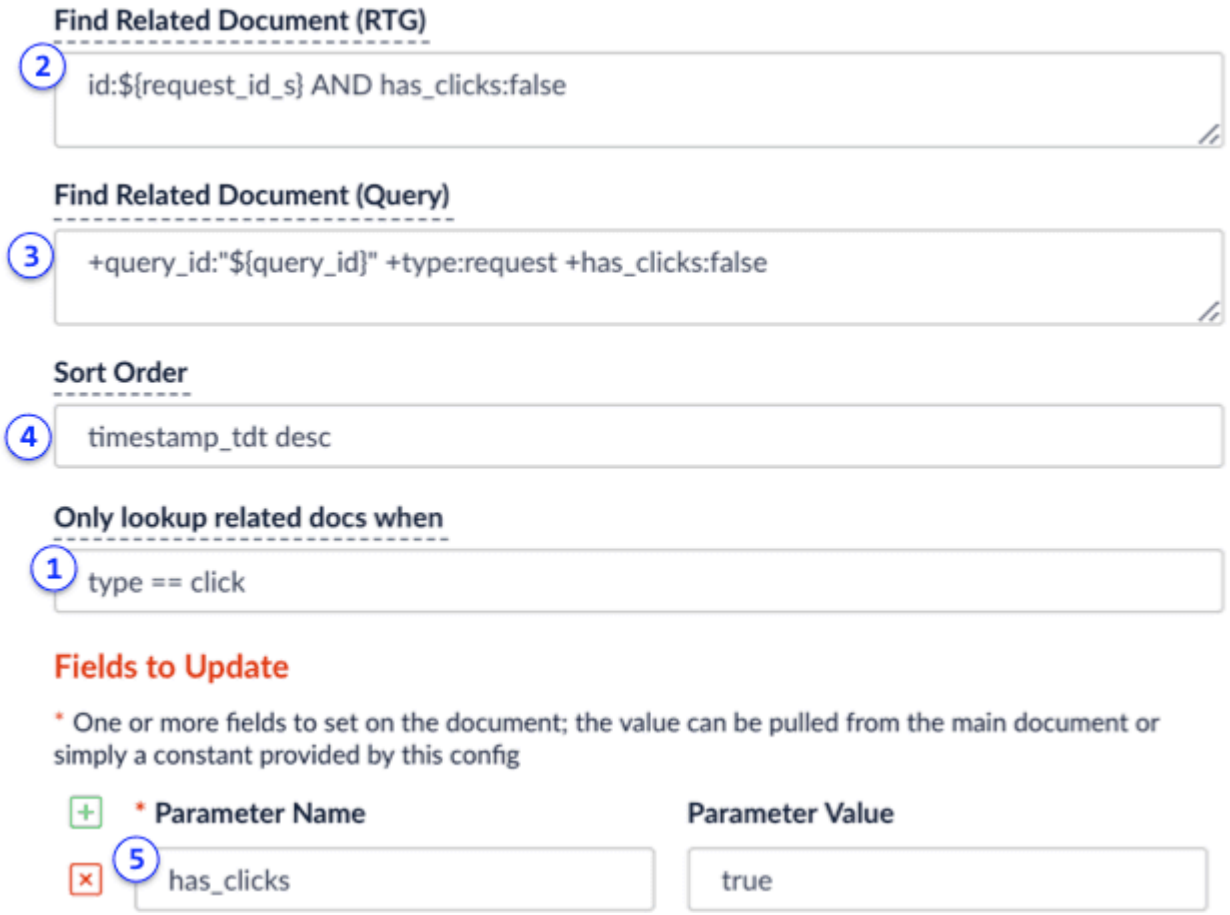
Default Signals Index Pipeline

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:

   1. When a click signal is encountered (type == click)
   2. 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.
   3. 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.
   4. 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.
   5. 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

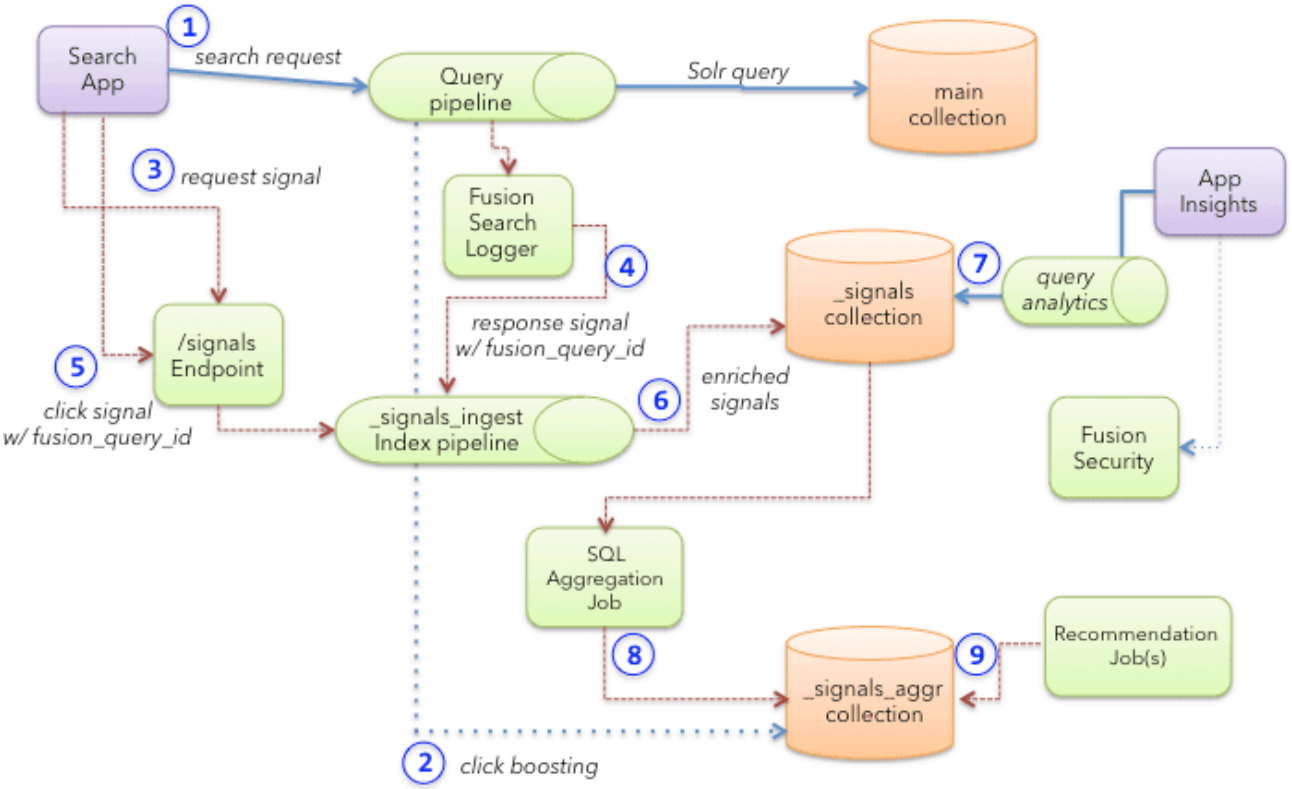
Signals are not automatically deleted by default, and over time they occupy an increasing amount of storage space.

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.

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.

   > 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 Structures

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 is 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 five built-in signal types:

- *annotation*
- *login*
- *request*
- *response*
- *click*

Annotation signals

Annotation signals are generated when a user bookmarks, likes, or comments on a document. Annotation signals are likewise generated when the user removes a bookmark, like, or comment.

> Annotation signals are generated by *App Studio*. If you are not using App Studio, this type of signal is not relevant to your search application.

Login signals

Login signals record information about specific users when they log in to an application. This includes a time stamp and various session details.

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:

```
[ { "id":"288fe4f7-6680-403e-8d18-27647cdd9989", "timestamp":1518717749409,
"type":"request", "params":{ "user_id":"admin", "session":"ef4e00cd-91bb-45b4-
be80-e81f9f9c5b27", "query":"USER QUERY HERE", "app_id":"SEARCH APP ID",
"ip_address":"0:0:0:0:0:0:0:1", "host":"Lucids-MacBook-Pro-5.local", "filter":[
"field1/value", ... ], "filter_field":[ "field1" ] } } ]
```

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.

> 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:

Field Name	Description	Example
`id`	The x-fusion-query-id generated by the query-pipeline used for associating click signals with queries in experiments and aggregation jobs.	`TwWCn3Dz`
`type`	Signal type	`response`
`response_type`	Used by Insights to determine if this query had results or was empty	`results | empty`
`session`	User session ID; the search app should pass the session ID in the query params for a query	`UUID`
`query`	The actual query string sent to Solr from Fusion	`ipad`
`query_orig_s`	The incoming query from the search app before it is enriched by the query pipeline	`ipad`
`query_id`	A hash generated from the session, query, and filters fields; used as a rollup key in Insights to group activity by a specific	`SHA1 hash`
`filters_s`	Filter queries sent to Solr; the Fusion SearchLogger component combines multiple fq parameters into a single value delimited by " $ "	`{!tag=format}format: (vhs) $ {!tag=type}type: (movie)`
`filter`	Reformatted filter queries for use by App Insights	`field1/value`
`user_id`	User ID; the search app should pass the user_id in the query params	`admin`
`doc_ids_s`	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	`123,456,789`
`pipeline_id`	Fusion query pipeline that processed this query	`_system`
`collection`	Fusion collection	`my_collection`
`qtime`	Query time from Solr, in milliseconds	`10`
`rows`	Number of rows requested for this query	`10`
`hits`	Total number of documents matching the query	`10000`
`totaltime`	Total processing time of this query in milliseconds, includes Solr qtime and Fusion query processing time	`15`

Field Name	Description	Example
`timestamp_tdt`	Timestamp when the query request was received by Fusion	`2018-02-15T18:17:42.560Z`
`res_offset`	Offset of results; this field is used by experiment metrics to calculate MRR	`0`
`res_pos`	Position of the clicked result within the list of results	`3`
`params.*`	Any other query param sent from the search app to Fusion that was not already mapped to a declared field	`params.defType_s=edismax`

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:

```
[ { "id":"SOME UUID HERE", "timestamp":1518725351750, "type":"click", "params":{
"fusion_query_id":"ABkaEA11", "user_id":"admin", "session":"b3a15101-9e30-4e28-
8a23-d1f663c2ee06", "query":"tiger woods", "ctype":"result", "res_offset":0,
"filter":[ "type/Game" ], "ip_address":"0:0:0:0:0:0:0:1", "host":"Lucids-MacBook-
Pro-5.local", "doc_id":"9502308", "app_id":"SEARCH APP ID", "res_pos":1,
"filter_field":[ "type" ] } } ]
```

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" "url":"http://localhost:8080/#/product/9502308", "label":"Tiger Woods PGA Tour
09 All-Play - Nintendo Wii",
```

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

Depending on how you use signals, certain fields are required. These are signals collection field names and not the JSON field names in the in-bound signals document. An example is when sending the **user id**, write it as `params.user_id`.

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

Field Name	Example Value	Description
`count_i`	1	Number of times an interaction event occurred with this item
`doc_id`	NMDDV	Product ID or Item ID
`id`	68f66808-6bfc-4d73-95f7-8a558529160b	The signal ID. If no ID is supplied, one will be automatically generated.
`query`	xwearabletech	A query string from the user
`session_id`	91aa66d11af44b6c90ccef44d055cf9a	Id for session in which user generated the signal
`type`	quick_view_click	Type of session the user used to interact with the platform
`user_id`	11506893	ID of user during the session
`timestamp_tdt`	2018-11-20T17:58:57.650Z	Time when signal was generated

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

Parameter suffixes

Fusion can add suffixes when fields are indexed. This table lists common suffix values.

Single Value Suffix	Multivalued Sufix	Type
`*_b`	`*_bs`	boolean
`*_d`	`*_ds`	double
`*_dt`	`*_dts`	date
`*_f`	`*_fs`	float
`*_i`	`*_ii`	int
`*_l`	`*_ls`	long
`*_s`	`*_ss`	string
`*_t`	`*_ts`	text

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.

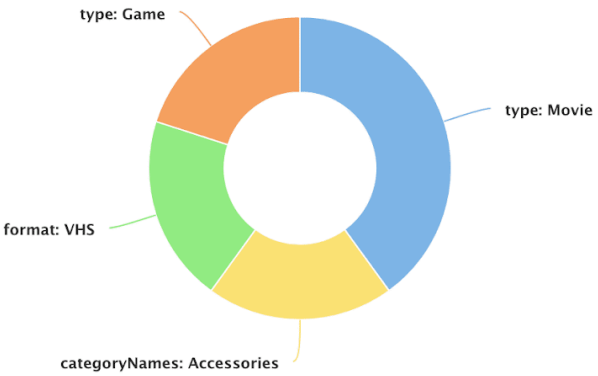
Field name	Query	Example number of documents
`ALL`	`*:*`	33,477,919
`count_i`	`count_i:[* TO *]`	11,101,165
`doc_id`	`doc_id:[* TO *]`	23,216,297
`id`	`id:[* TO *]`	33,477,919
`query`	`query:[* TO *]`	19,724,598
`session_id`	`session_id:[* TO *]`	11,101,165
`type`	`type:[* TO *]`	33,477,919
`user_id`	`user_id:[* TO *]`	26,117,399
`timestamp_tdt`	`timestamp_tdt:[* TO *]`	26,117,399

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 is 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*.

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. *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*.

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 a 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"
```

Legacy Aggregations

> 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 "legacy aggregations."

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 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.

SQL Aggregations

SQL aggregation is used for aggregating signals or other data. SQL is a familiar query language that is well suited to data aggregation.

> The aggregation approach available in prior Fusion releases is still available, though it is deprecated. 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 is 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 *(https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$)*. 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 *(https://en.wikipedia.org/wiki/Exponential_decay)* that Fusion uses for aggregations.
  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.

If you are using SQL to do aggregation but have not supplied a custom rollup SQL, 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.

Fusion's basic rollup is a SUM of numeric types (long, integer, double, and float) and can support `time_decay` for the weight field. If you are not using `time_decay` in your weight calculations, then the weight is calculated using `weight_d`. If you do include `time_decay` with your weight calculations, then the weight is calculated as a combination of timestamp, `halfLife`, `ref_time`, and `weight_d`.

The basic rollup SQL can be grouped by `DOC_ID`, `QUERY`, `USER_ID`, and `FILTERS`. The last GROUP BY field in the main SQL is used so it will ultimately group any newly aggregated rows with existing rows.

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 does not 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* do not 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 is not 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.

Time range in Fusion is equivalent to date range in Solr *(https://solr.apache.org/guide/8_8/working-with-dates.html)*.

Example values for time range:

- `[* TO NOW]` - all past events
- `[2000-11-01 TO 2020-12-01]` – specify by exact date
- `[* TO 2020-12-01]` – from the first data point to the end of the day specified
- `[2000 TO 2020]` - from the start of a year to the end of another year

SQL functions

A Spark SQL aggregation query can use any of the functions provided by Spark SQL *(https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$)*. 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 does not meet your needs, you can modify it. The `time_decay` function is implemented as a UserDefinedAggregateFunction *(https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.expressions.UserDefinedAggregateFunction)* (UDAF).

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

```
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`:

```
time_decay(count: Long, timestamp: Timestamp)
```

In this case, Fusion fills in these values for the omitted parameters: `halfLife` = `30 days`, `ref_time` = `NOW`, and `weight_d` = `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: `halfLife` = `30 days`, `ref_time` = `NOW`, and `weight_d` = `0.1`.

Parameters
Parameters for `time_decay` are:

Parameter	Description
`count`	Number of occurrences of the event. Typically, the increment is 1, though there is no reason it could not be some other number. In most cases, you simply pass `count_i`, which is the event count field used by Fusion signals, as shown in the *SQL aggregation examples*.
`timestamp`	The date-and-time for the event. This time is the beginning of the interval used to calculate the time-based decay factor.
`halfLife`	Half life for the exponential decay that Fusion calculates. It is some interval of time, for example, `30 days` or `10 minutes`. The `interval` prefix is optional. Fusion treats `30 days` as equivalent to `interval 30 days`.
`ref_time`	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 `ref_time` column to every row before executing the SQL.
`weight_d`	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 `weight_d`; see *Use Different Weights Based on Signal Types* for an example.

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.

Advanced Model Training Configuration for Smart Answers

This topic provides tips for training your *Smart Answers* deep learning model.

> Support for the Fusion 4.2 implementation of Smart Answers ended in December, 2020. Upgrade to Fusion 5 for ongoing Smart Answers support.

Model Base (Fusion 5.3 and up)

There are several types of model bases that can be used for training and fine-tuning:

- **word_en_300d_2M** are general pre-trained word embeddings. It is a good default choice to start with for English language.
- **bpe_{language}_{dim_size}_{vocab_size}** are general pre-trained BPE *(https://nlp.h-its.org/bpemb)* embeddings that are available for different languages, including CJK languages and multilingual. Also useful in scenarios when vocabulary is very big or when the data might have a lot of misspellings.
- **word_custom** or **bpe_custom** specifies that custom embeddings should be trained on users data via Word2Vec algorithm. It might be useful when your domain has a very unusual specific vocabulary.
- **transformer** based models such as **distilbert_{language}** and **biobert**. Much bigger and expensive models that might provide even better quality for FAQ, Chatbot and virtual Assistance use-cases. Also useful when the training data is limited.

If **word** or **bpe** based models are used, one or more RNN layers are added on top of the embeddings to be trained to capture contextual and semantic information. It is configurable in the **RNN Encoder Parameters** section. If you wish to use embeddings initialized on your data, refer to the **Custom Embeddings Initialization** to configure Word2Vec algorithm.

Transformer-based models already have specified fixed model architecture which is fine-tuned during the training procedure.

> Dimension size of vectors for Transformer-based models is 768. For RNN-based models it is 2 times the number units of the last layer. To find the dimension size: download the model, expand the zip, open the log and search for `Encoder output dim size:` line. You might need this information when creating collections in Milvus.

> We recommend to use Transformer-based models only if you can allocate GPU for the training job as these models are very computationally expensive.

At the end, Attention mechanism to aggregate output into final single vector for all models.

Auto hyperparameter tuning (Fusion 5.3 and up)

By default, training module tries to select the most optimal parameter values (for those left as blank) based on the training data statistics. Auto-tune can extend it by automatically finding even better training configuration through hyper-parameter search.

If **Perform auto hyperparameter tuning** is enabled, multiple models will be trained across several stages. On each stage the most impactful parameters are tuned to find the best configuration. All other parameters are used with default values or those specified on UI.

Although this is a resource-intensive operation, it can be useful to identify better RNN-based configuration. Transformer-based models are not used during auto hyperparameter tuning as they have a fixed architecture. They usually perform better on Q&A tasks yet they are much more expensive on both training and inference time.

Input/Output parameters (Fusion 5.3 and up)

Here you can specify the input data that should be used for training with possibility to filter or sample it.

> In Fusion 5.3 and later, you can also configure this job to read from or write to cloud storage. See *Configure An Argo-Based Job to Access GCS* and *Configure An Argo-Based Job to Access S3*.

If you have additional text data that can be used for **custom embeddings initialization** or to learn and capture bigger vocabulary when **word_en_300d_2M** is used, please provide it in the **Texts Data Path** field.

Model Replicas parameter allows to specify how many replicas of the model should be deployed. Auto-balancing mechanism is used to distribute queries between model replicas, so more replicas might provide faster indexing as well as higher QPS.

If you use aggregated signals data for training or have weights for each training pair, you can also specify **Weight Field**. It will be used for sampling positive answers for a particular query if there are more than one possible. It is useful for eCommerce use-cases when for one unique query there might be a lot of different paired products.

> **Weight Field** will not be used if **Use Labelling Resolution** is set on. These parameters are mutually exclusive.

Data pre-processing parameters

Labeling Resolution allows to find missing query/response pairs in the training data which helps in the training. When set on, a graph of all pairs connections is built. Then connected components are obtained to match missing query/response pairs. For example if there are three existing pairs: `q1-a1`, `q2-a1` and `q2-a2`. Then Labelling Resolution will match `q1-a2` as additional pair through `q2-a1` connection. This is useful in Q&A use-cases when there are not a lot of answers per unique question, otherwise too big connected components will be found. If you have data when for one query there might be a lot of different responses, like in eCommerce, it is better to leave it off.

If **Use Labelling Resolution** is set on, **Weight Field** is ignored. These parameters are mutually exclusive.

The **Maximum vocabulary size**, **Lower case all words** and **Apply unidecode decoding** parameters impact the vocabulary size if `word_en_300d_2M`, `word_custom` or `bpe_custom` model bases are used. Otherwise these parameters are ignored and model specific pre-processing is used. Default values should work in most cases, given enough RAM and time to train.

If you want to train custom embeddings for languages like CJK, disable **Apply unidecode decoding**.

If you see an out-of-memory error, try reducing the vocabulary size and/or the training batch size. The **Minimum number of words** and **Maximum number of words** parameters can help trim problematic documents.

Custom embeddings initialization parameters

If **word_custom** or **bpe_custom** model bases are chosen, then custom embeddings will be trained on the provided data.

If you want to use addition dataset to train custom embeddings, please specify **Texts Data Path** and **Text Fields** in the **Input/Output parameters**.

Additionally, commonly-used Word2vec training parameters are **Word2Vec Training Epochs**, **Size of Word Vectors** and **Word2Vec Window Size**. Default values should work in most cases.

Smaller word vectors size makes models smaller and more robust to overfitting. However, dimensions smaller than 100 may impact the quality.

Evaluation parameters

Validation Sample Size controls how much *unique* queries should be hold-out and used for validation. It is a fraction if the value below 1.0 or specific number of queries if it is integer value higher than 1.

During evaluation, all responses/answers are used. They form an index which is queried by unique validation queries. **Eval ANN Index** parameter controls should it be ANN index or brute-force search with auto value by default. If you notice that evaluation takes a lot of time, try to enable ANN index or reduce the number of evaluation queries.

Generally, this evaluation setup is similar to how it will work in index and query pipelines, so the evaluation results should provide good approximation of the quality. To evaluate the configured pipelines on the test data, please use *Evaluate a Smart Answers Query Pipeline job*.

A list of evaluation metrics is provided to monitor the training process and measure the quality of the final model:

- Mean Average Precision (MAP)
- Mean Reciprocal Rank (MRR)
- Recall

You can choose from the list in the **Metrics list** parameter. It uses all metrics by default.

You can also specify measuring the ranking position for each metric. For example, if you specify **Metrics@k list** as `[1,3]`, with **Metrics list** `["map","mrr","recall"]`, then the metrics `map@1`, `map@3`, `mrr@1`, `mrr@3`, `recall@1`, and `recall@3` will be logged for each training epoch and final model.

You can choose a particular metric at a particular `k` (controlled by the **Monitoring metric** parameter) to help decide when to stop training. Specifically, when there is no increase in the Monitoring metric value for a particular number of epochs (controlled by the **Patience during monitoring** parameter), then training stops.

During the training we evaluate the result using similar cold-start model (weighted average of word vectors) as a baseline. Look for the *Cold-start encoder validation evaluation* section of the logs, it is printed before first training epoch.

General Encoder parameters

Note that the following parameters are common across all model bases including RNN and Transformer architectures.

- **Fine-tune Token Embeddings** will allow to fine-tune embeddings (word vectors) layer to be updated during the training alongside with all other layers. It is disabled by default as it is usually one of the biggest layer in the network and updating it might lead to overfitting. It is useful to enable if your data have a lot of specific or misspelled words.
- **Max Length** controls the maximum context window that model can process. Texts longer than this value will be trimmed. The default value is the max value between three times the STD of question lengths and two times the STD of answer lengths. The longer the context the longer and harder it takes for model to process. This parameter is especially important for Transformer-based models as it affects training and inference time. Note that the maximum supported length for Transformer models is 512 tokens, so you can specify any value up to that.
- **Global Pool Type** specifies how token vectors should be aggregated to obtain final content vector. The default mechanism is self-attention which provides the best quality in most cases.
- **Number of clusters** and **Top K of clusters to return** are deprecated since 5.3 and will be removed in the following releases. There is no practical need to use them after Milvus vectors similarity search integration.

RNN Encoder parameters

We use RNN-based deep learning *(https://en.wikipedia.org/wiki/Recurrent_neural_network)* architecture for `word` and `bpe` model bases, with the flexibility to choose between `LSTM` and `GRU` layers with more than one layer. We don't recommend using more than three layers. The layers and layer sizes are controlled by the **RNN function list** and **RNN function units list** parameters.

Dropout ratio parameters provides regularization effect and is applied between embeddings layer and the first RNN layer.

Training parameters

These parameters controls the training procedure. Most of them are left blank so the robust default values can be determined by the training module based on the dataset statistics.

The learning rate scheduler has 3 stages. Firstly, it linearly increases the LR from **Minimum Learning Rate** value to **Base Learning Rate** value over **Number of Warm-Up epochs**. Then it stays consistent for **Number of Flat epochs**. And at the last stage Cosine Annealing is used for the remain number of epochs.

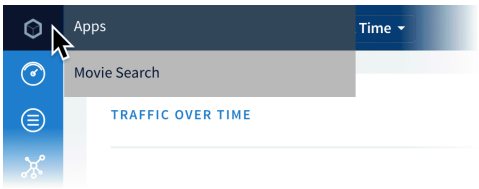
Use Mixed Precision parameter can enable mixed precision during the training for Transformer-based models if modern GPU are used (Turing and later). It helps to get more VRAM so bigger batch size can be used. As well as provides some performance boost in training time.

Cross-Batch Memory parameters allow to re-use encoded representations from the previous batches during loss computation, so loss function can process more positive and negative examples for the model update. It works well with Transformer-based models that consumes more VRAM and can be used with only with limited batch size. This is not necessary when the training batch size is large. When configured, this number needs to be greater than or equal to the training batch size.

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.

- 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**:



See also these subtopics:

- *Analytics*
- *App Insights Dashboards*
- *Events*
- *Experiment Results*
- *Sessions*

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.

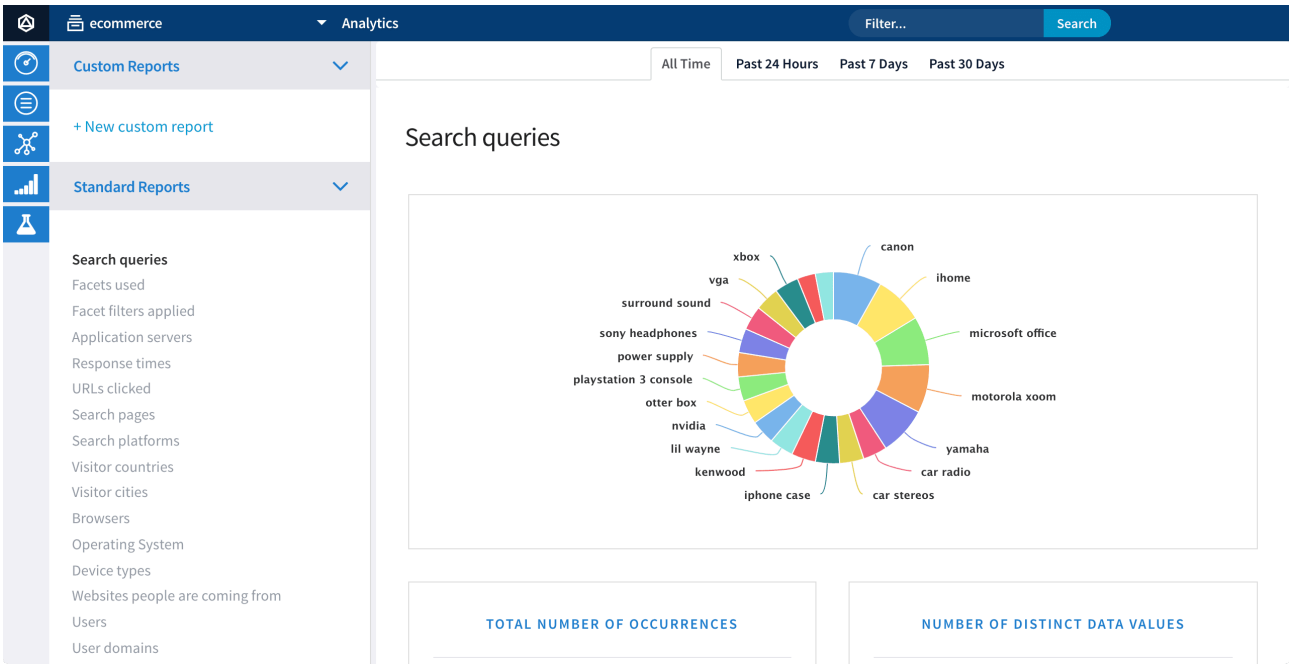
Reports

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.

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**.



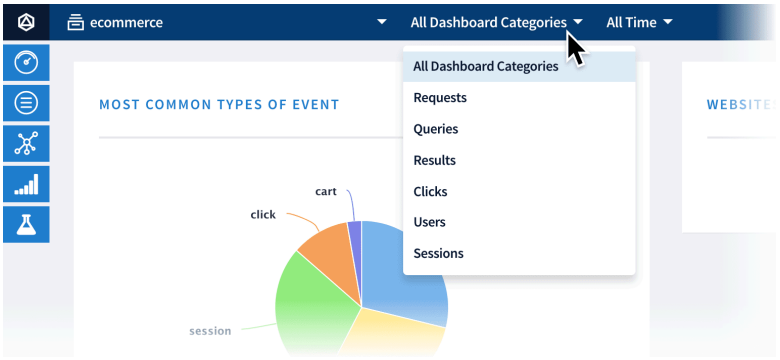
Standard reports

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
- Users
- User domains
- Types of event
- Head Tail analysis

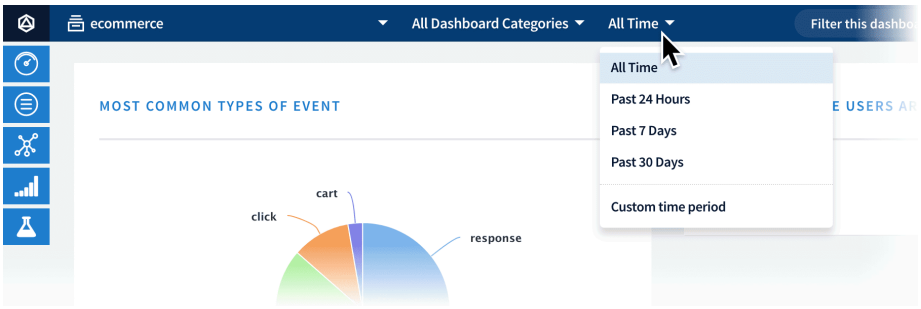
App Insights Dashboards

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:

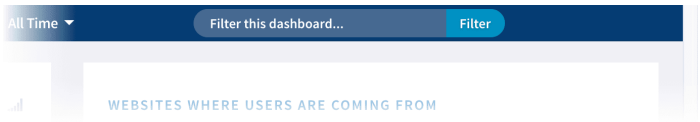


Dashboard filtering

You can filter by timeframe using the **All Time** menu:



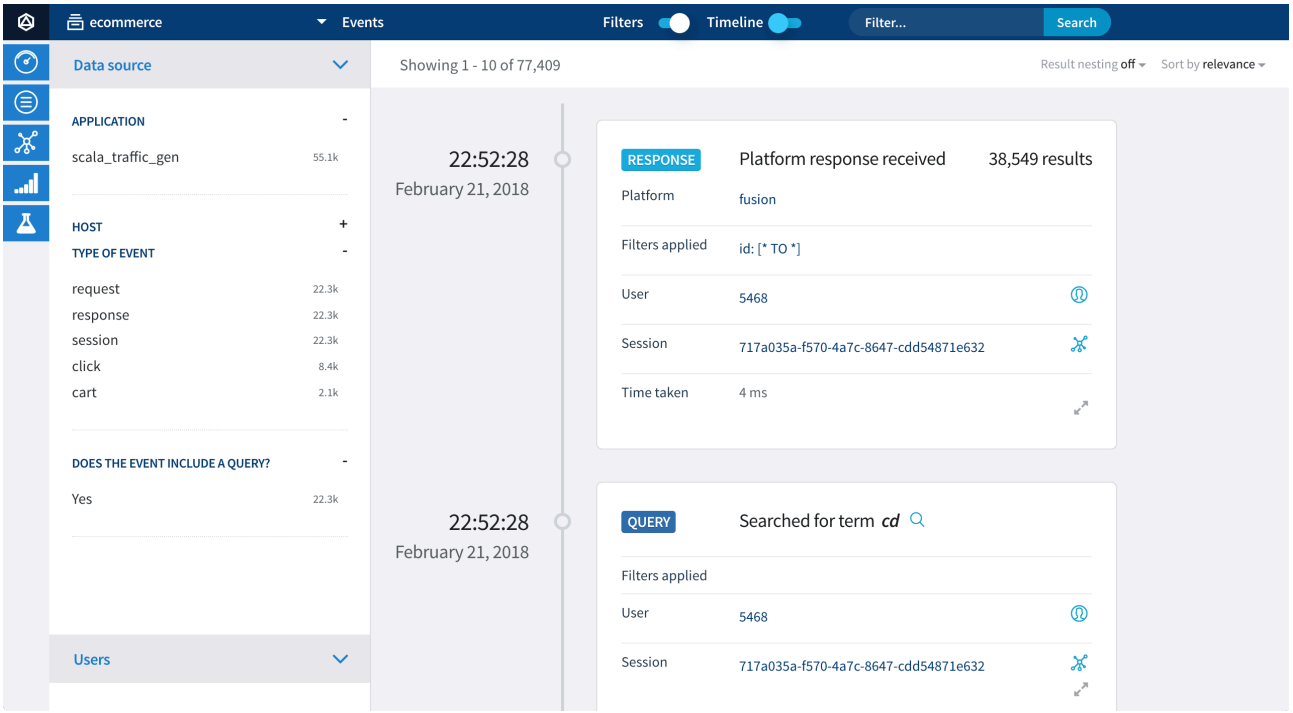
To filter by the content of the data, enter a string in the **Filter this dashboard…** text box:



Events

Click  **Events** to view histograms and timelines about events, filtered by:

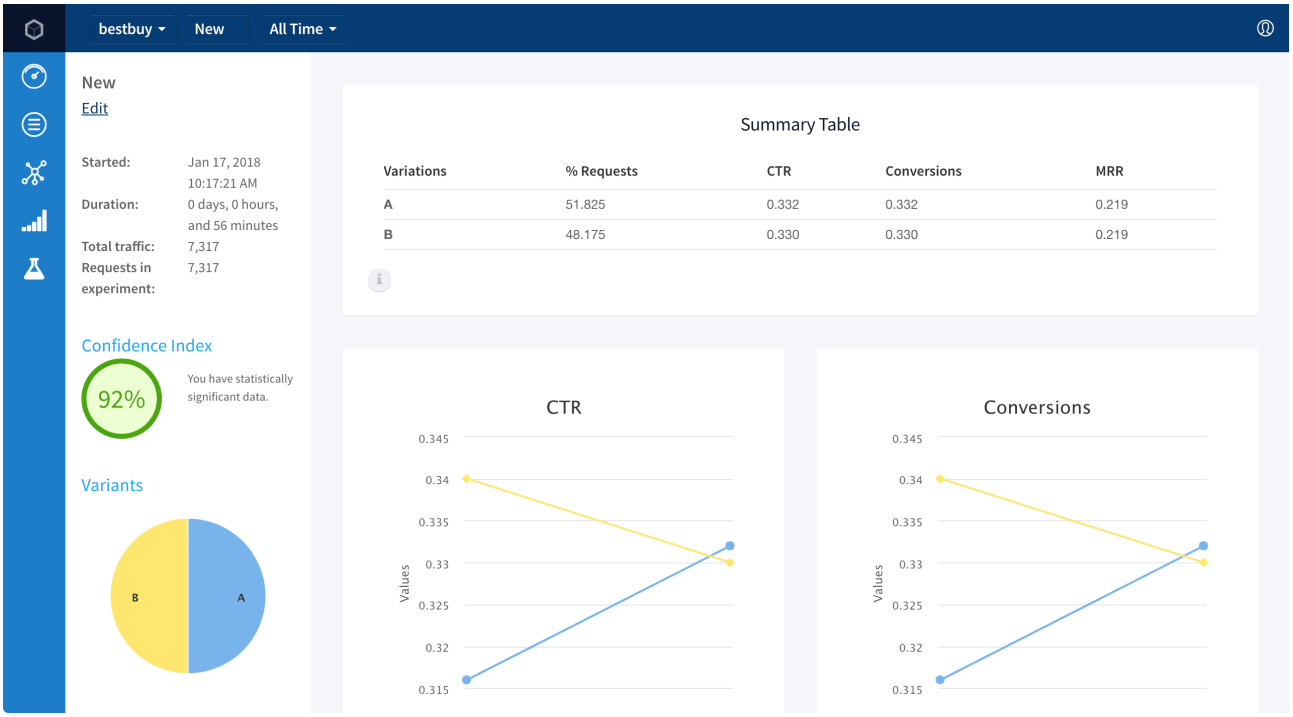
- event type
- users
- request/response information



Experiment Results

If you have configured an *experiment*, you can see visualizations about the results here.

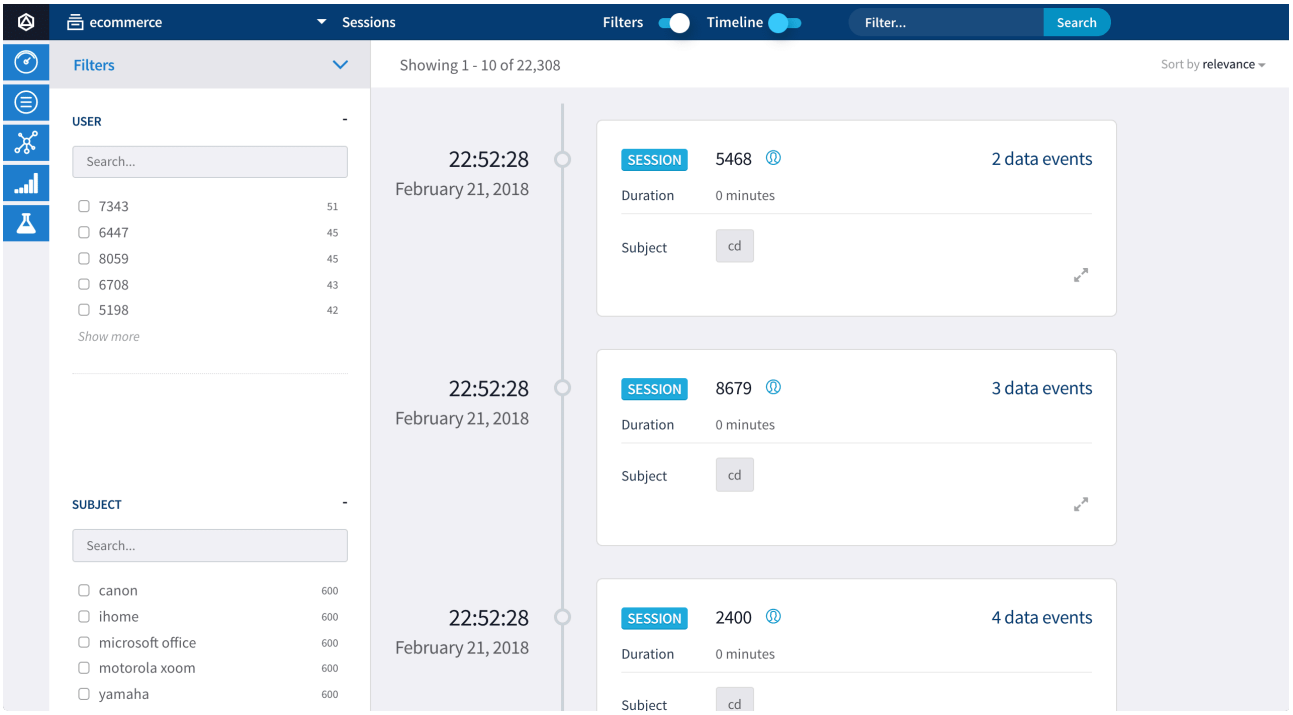
This example shows a 92% confidence index for the difference in click-through rate (CTR). Conversions use the same signal type (`click`), so the result is identical.



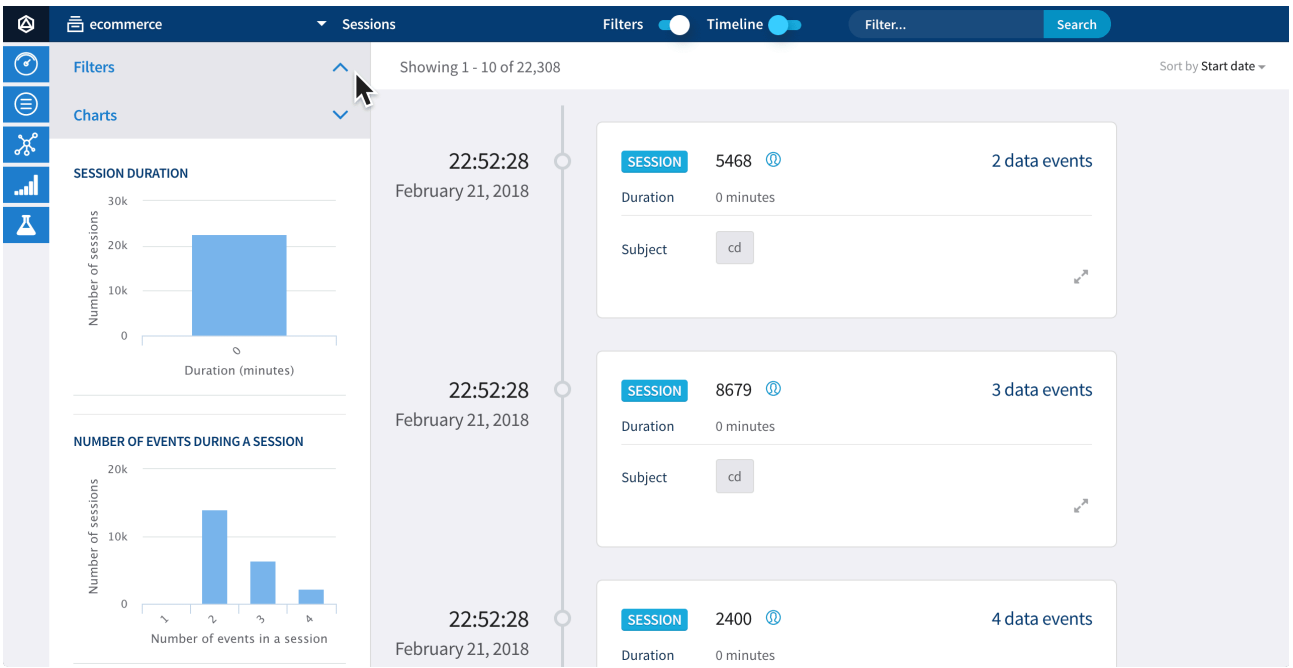
Sessions

Click  **Sessions** to view charts and timelines about search events, filtered by:

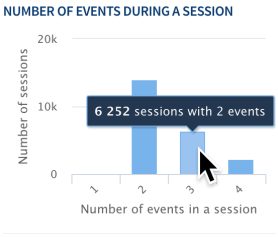
- user
- subject
- event duration
- events per session



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
20k
10k
0
Number of sessions
6 252 sessions with 2 events
1
2
3
4
Number of events in a session

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.

See also these subtopics:

- *Boost with Signals*
- *Getting Started with Recommendations and Boosting*
- *Items-for-Item Recommendations*
- *Items-for-Query Recommendations*
- *Items-for-User Recommendations*
- *Queries-for-Query Recommendations*
- *Methods for Recommendations and Boosting*
- *Users-for-Item 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:

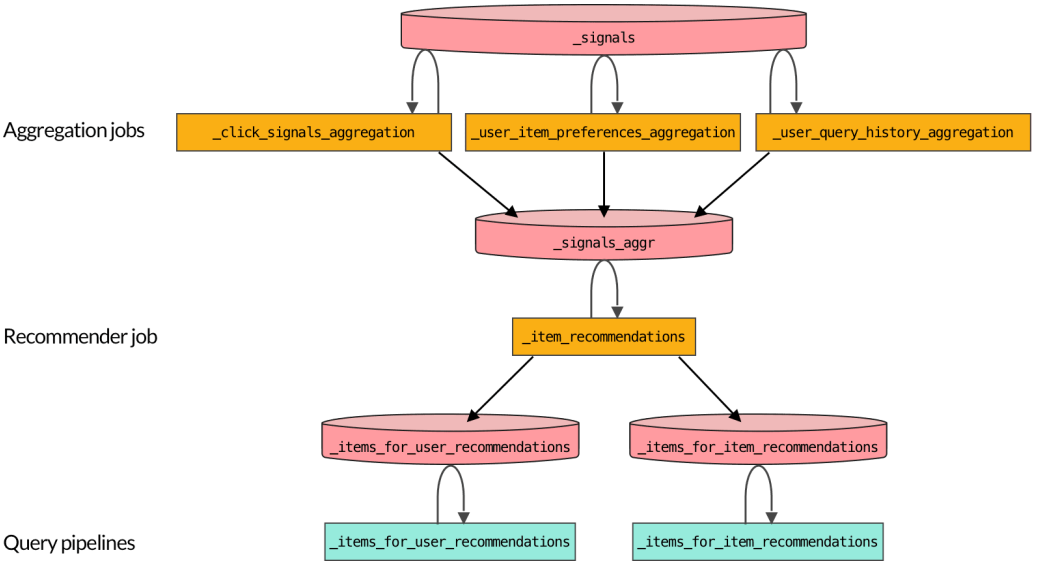
Recommendations	Boosts
Personalized search results from a special, automatic query, *whether or not the user has performed a query*. Recommendations can be based on a variety of criteria, as in the examples below: • "Based on your purchase history, you might be interested in these items." • "People who viewed this item also viewed these items." • "People who searched for 'ipad' also searched for these items."	Modifications to the *scores* of the items in a query's search results. The set of search results does not change, but their ranking does. For example: • Boost the most popular items • Boost items this user has clicked before • Boost seasonal items Boosts can also be configured manually, using the *Query Workbench*. 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 object for different *recommendation methods*.

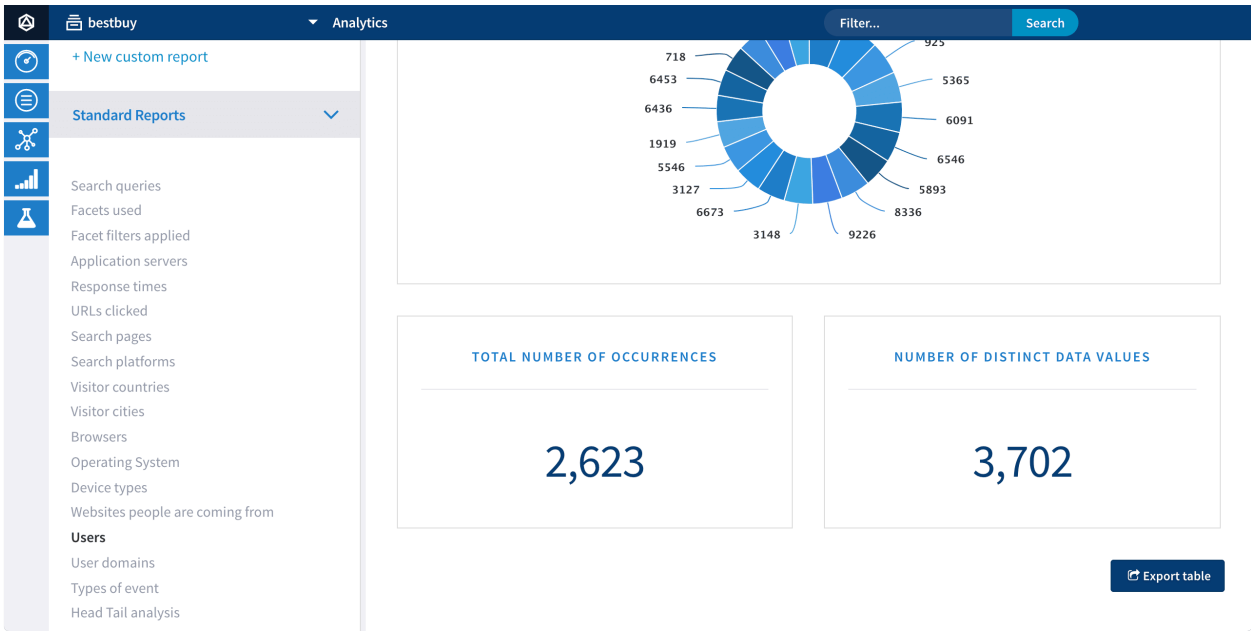


1. *Signals data* is aggregated into a format that can be consumed by recommender jobs.
   Your search application should be sending *well-formed signals data* to Fusion whenever there is user activity on your site.
   By default, the `_click_signals_aggregation` job runs every 15 minutes while the `_user_item_preferences_aggregation` 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.
   By default, the `_item_recommendations` job runs whenever the `_user_item_preferences_aggregation` job finishes successfully.
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:



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:

```
num_users * num_recs_per_user + num_items * 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.

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

Getting Started with Recommendations and Boosting

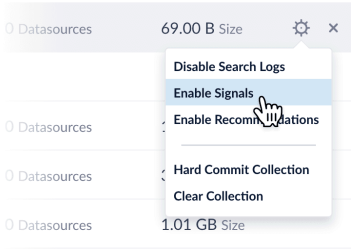
Signals provide the data that Fusion uses to generate *collaborative recommendations*. The simplest way to get started is to enable signals and recommendations in one of your primary collections.

Once you do this, Fusion automatically creates a set of *default objects* and begins creating and updating collections of recommendations on a regular schedule.

> *Content-based recommendations* can be used without enabling signals or recommendations, but they require manual configuration.

Enabling signals

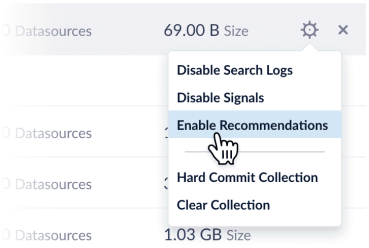
Signals are enabled by default for new collections when you have a Fusion AI license installed. You can enable or disable signals for any collection at **Collections** > **Collections Manager**.



Enabling signals automatically creates a set of *aggregation jobs* which create the input data for recommendations. See *Signals and Aggregations* for complete details.

Enabling recommendations

Recommendations are not enabled by default; you can do this at **Collections** > **Collections Manager**.



When you enable recommendations, this automatically enables the *items-for-user* and *items-for-item* recommendation methods. To use additional *recommendation methods*, you must configure them separately.

Default objects for recommendations

When recommendations are enabled, Fusion automatically creates a default set of collections, jobs, schedules, and query pipelines that provide basic functionality for recommendations.

You can tune the default jobs and pipelines as needed to refine the results, or create new ones, then configure your search application to request recommendations from the query pipelines.

See also the default objects created when you *enable signals*. These must already exist when you enable recommendations.

Collections

- `COLLECTION_NAME_items_for_item_recommendations`
  Collection to hold generated item-item similarities (by default 10 per item). No `user_id_s` data is present. A Recommend Items for Item query pipeline stage can use the similarities to return item recommendations. For example, a query in which `doc_id_s = docA` would return an ordered list of other `doc_id_s` values for documents that are similar to document `docA`, along with the similarities. For example: `[("docB", 0.83), ("docC", 0.55), ("docD", 0.43), …, ("docK", 0.22)]`.
- `COLLECTION_NAME_items_for_user_recommendations`
  Collection to hold recommended items for a user. By default the job creates 10 recommendations per user.

Job and schedule

Enabling recommendations creates one new *ALS Recommender* job, which consumes the output of the *signals aggregation jobs*.

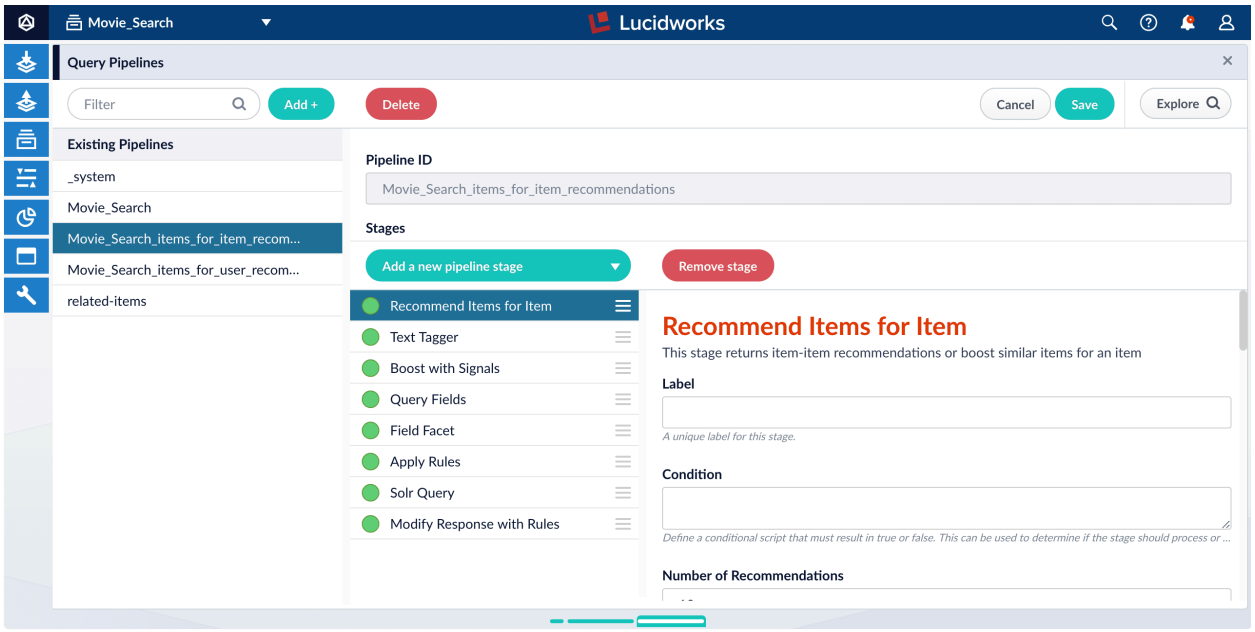
Job	`COLLECTION_NAME_item_recommendations`
Default input collection	`COLLECTION_NAME_signals_aggr`
Default output collections	`COLLECTION_NAME_items_for_user_recommendations` `COLLECTION_NAME_items_for_item_recommendations`
Default trigger	Successful completion of the `COLLECTION_NAME_user_item_preferences_aggregation` job

As suggested by the output collection names, this default job produces recommender data for *items-for-user* and *items-for-item* recommendations.

	The `COLLECTION_NAME_user_item_preferences_aggregation` job provides input data for this job and must run before it. See *SQL Aggregations* for details.

Query pipelines

- `COLLECTION_NAME_items_for_user_recommendations`
  Query pipeline to generate recommendations of items for a user.
- `COLLECTION_NAME_items_for_item_recommendations`
  Query pipeline to generate recommendations of items similar to an item.



Methods for Recommendations and Boosting

This topic provides an overview of different methods for generating recommendations and boosts. "Content-based" and "collaborative" recommendations are two broad categories. More specific recommendation methods are also explained.

Content-based vs collaborative recommendations

Recommendation methods can be divided into two broad categories: collaborative recommendations and content-based recommendations.

Collaborative recommendations always require signals. Some also require a job that pre-computes a matrix of values that can be queried. When these values are used for boosting, then certain query pipeline stages are needed as well.

Method	Requires aggregated signals	Required job	Boosting stage
Collaborative recommendations These methods analyze how users have interacted with documents.			
Items-for-item	✓	*ALS Recommender*	*Recommend Items for Item*
Items-for-user	✓	*ALS Recommender*	*Recommend Items for User*
Boost with signals	✓		*Boost with Signals*
Users-for-item recommendations	✓	*ALS Recommender*	
Content-based recommendations These methods analyze the content of documents or queries.			
Boost documents			*Boost Documents*
Parameterized boosting			*Parameterized Boosting*
More Like This			*Solr MoreLikeThis*

Fusion recommendation and boosting methods

This section provides an overview of each method, with links to more detailed topics.

Collaborative recommendations	Content-based recommendations
• *Items-for-item* • *Items-for-user* • *Items-for-query* • *Queries-for-query* • *Boost with signals* • *Users-for-item*	• *Boost Documents* • *Parameterized Boosting* • *More Like This*

When you *enable recommendations*, this automatically enables the items-for-user and items-for-item recommendation methods. To use additional recommendation methods, you must configure them separately.

Items-for-item recommendations

Items-for-item recommendations present items that are similar to a specified item. For example, when the user is viewing a BMX bicycle, Fusion can recommend other BMX bicycles. Similarity can be based on different criteria, such as click patterns, people who bought this also bought that, percentage match of document tags, and so on.

See *Items-for-Item Recommendations* for details.

Items-for-user recommendations

Items-for-user recommendations use the *Recommend Items for User* query stage to present items that are similar to ones in which the user has previously shown interest, based on the user's search history, browsing history, purchase history, and so on. This is one type of *collaborative recommendation*.

See *Items-for-User Recommendations* for details.

Items-for-query recommendations

The Items for Query recommender is the primary algorithm that supports recommendations on the search results page.

The items-for-query recommendations are based on how other users interacted with search results from the same query. This is one type of *collaborative recommendation*.

For example, if users have searched for "titanic" in the past and many of them clicked on the search result for the DVD of the movie *Titanic* (as opposed to books or memorabilia), then this method boosts the DVD item on subsequent searches for "titanic".

See *Items-for-Query Recommendations* for details.

Queries-for-query recommendations

Queries-for-query recommendations are queries performed by other users who also performed the current query. For example, when a user searches for "madonna", your app may also suggest searches for "evita", "a league of their own", "lady gaga", and so on.

See *Queries for Query* for details.

Boost with signals

The Boost With Signals query pipeline stage performs automatic boosts based on the contents of the aggregated signals collection. For this type of boosting, signals must be enabled but recommendations need not be.

See *Boost With Signals* for details.

Users-for-item recommendations

Users-for-item recommendations retrieve the set of users who have interacted with an item, weighted by the number of interactions. This can be useful for community-driven use cases or for marketing campaigns. For example, on your organization's intranet, clicking a search result could display the item plus a list of colleagues who have interacted with the same item.

See *Users-for-Item Recommendations* for details.

Boost documents

The Boost Documents query pipeline stage adds boosting parameters to matched documents based on user-defined rules. Boosts are defined with a term value to boost and the boost factor to add. The boosting parameters are added to the `boost` Solr query parameter.

See *Boost Documents* for details.

Parameterized boosting

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

See *Parameterized Boosting* for details.

More Like This

This stage uses the content of the current document to query for similar documents, using *Solr's MoreLikeThis component*.

This stage provides content-based recommendations. For collaborative recommendations, use the *Recommend Items for Item stage*.

See *Solr MoreLikeThis* for details.

Boost With Signals

The Boost With Signals query pipeline stage performs automatic boosts based on the contents of the aggregated signals collection. For this type of boosting, signals must be enabled but recommendations need not be.

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

Some of the important configuration parameters are discussed below. For complete details about all configuration parameters for this stage, see *Boost with Signals*.

Configuration overview

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

Number of Recommendations `numRecommendations`	Sets the `rows` query param in the main query as the maximum number of query results which will be boosted by this pipeline stage.
Number of Signals `numSignals`	Sets the `rows` query param in the query that searches the `_signals_aggr` collection, so only the specified number of aggregated signals are retrieved and used for boosting. When signals boosting is applied to a query, aggregated signals records are queried from the appropriate `_signals_aggr` collection to find out the popularity or boost weight for documents which have signals. `numSignals` limits the number of records to be queried from a `}}{{_signals_aggr` collection and used to calculate this boost.
Aggregation Type `aggrType`	A filter to retrieve aggregated signals in the `_signals_aggr` collection per each aggregated signal's `aggr_type_s` field value.
Solr Field to Boost On `boostId`	The document field in the main collection on which to perform boosting. Typically it should use default field, which is `id`. This field corresponds to the **Rollup Field**/`rollupField` field. Together, these two fields act like a `<field>:<value>` pair in the query modification for boosting.

Boost Method
 	
`boostingMethod`
	This adds a query parameter to the original query, either "query-param" or "query-parser". The result is `("query-param" or "query-parser") + Boost Param("boost" or `bq`)`, as in the examples below:
	
	"query-param"+"boost", result boost query param
	
	`boost="map(query({!field f='id' v='6239046,13026192}), 0, 0, 1, 27.1705)"`
	
	"query-parser"+"boost", result boost query param
	
	`bp_xxx_bbqx="map(query({!field f='id' v='6239046,13026192}), 0, 0, 1, 27.1705)"`
	
	When `Boost Param` uses `bq`, similar logic applies. When **Boost Param**/`boostingParam` uses "boost", it works with both "query-param" and "query-parser".

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

Rollup Weight Field `rollupWeightField`	Indicates the final boost weight used to calculate the new score for docs retrieved by the main query. Similar to **Rollup Field**/`rollupField` above, this should be set to the field in the aggregated signal collection that stores the final weight that was calculated. By default it's `weight_d`, because by default this is used in the *click_signal_aggr SQL job*.
Final Boost Weight Expression `weightExpression`	Calculates the final weight using the weight and score retrieved from the `_signals_aggr` collection. The default value is `math:log(weight_d + 1) + 10 * math:log(score+1)`.

Solr query parameters

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

`qf=query_t`	Defines which field to query. In the default case, the query searches on the `query_t` field of aggregated signal docs.
`pf=query_t^50`	Boosts docs within the set of retrieved docs using phrase matching.
`pf2=query_t^20`	`pf2` is similar to `pf`; the difference is that `pf2` works on bigram phrases.
`pf3=query_t^10`	`pf3` is similar to `pf`; the difference is that `pf3` works on trigram phrases.

FAQs

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

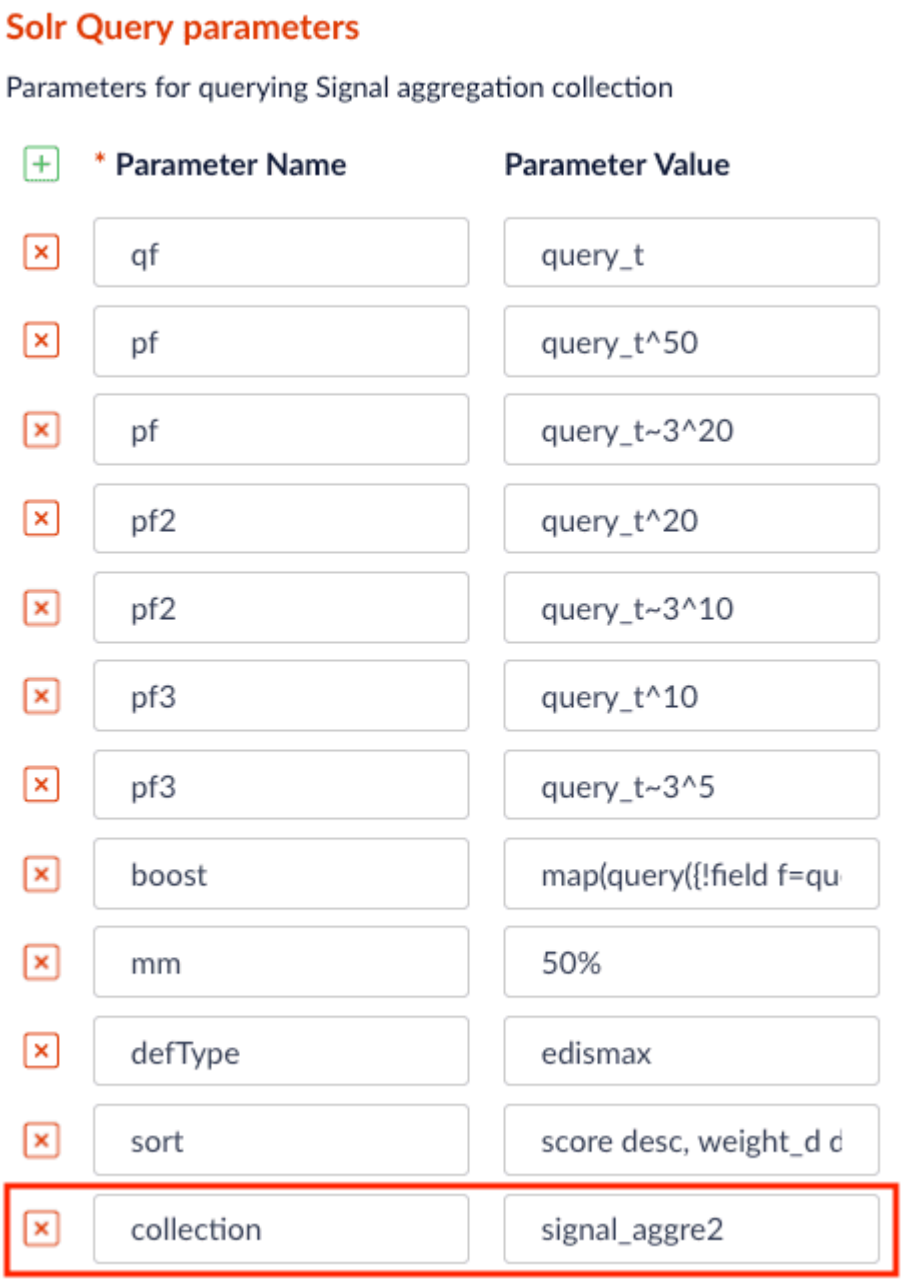
For example, if we have filter query param `fq=format:CD&fq=name:Latin`, this needs to be translated into `lw.rec.fq=filters_s:"format:cd $ name:latin"`. Values must be lowercase. The final main query should be:

```
http://localhost:8764/api/apollo/apps/demo_app/query-pipelines/demo_app/collections/demo_app/select?echoParams=all&wt=json&json.nl=arrarr&sort&start=0&q=apple&debug=true&rows=10&lw.rec.fq=filters_s:"format:cd $ name:latin"
```

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

If there are multiple `fq` values (for example, `format:cd` and `name:latin`), they are ordered alphabetically as strings and joined with " $ " (a $ with a space on each side). In the example, `"format:cd $ name:latin"`.

What if my aggregated signals are in a different collection?
You can point the Boost with Signals stage to a different signal collection by adding a `collection` parameter in the `Solr Query Parameters` section.

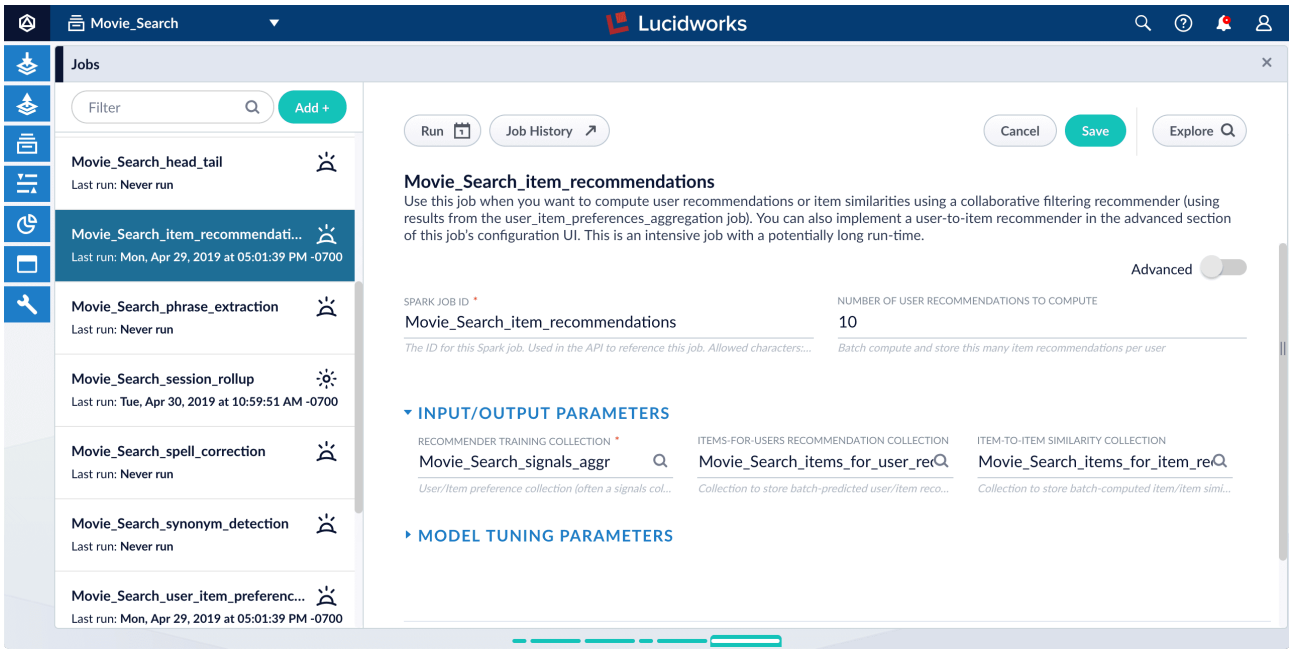


Items-For-Item Recommendations

Items-for-item recommendations present items that are similar to a specified item. For example, when the user is viewing a BMX bicycle, Fusion can recommend other BMX bicycles. Similarity can be based on different criteria, such as click patterns, people who bought this also bought that, percentage match of document tags, and so on.

Collaborative items-for-item recommendations

If you have enabled signals and recommendations for a collection, then the default `COLLECTION_NAME_item_recommendations` job is already created and configured to produce items-for-item recommendations (as well as *items-for-user recommendations*):



This is an *ALS Recommender* job.

> If you want to use different parameters for items-for-item recommendations and *items-for-user recommendations*, simply create separate jobs for each, where one job configuration includes an output collection for items-for-item recommendations only and the other includes an output collection for items-for-user recommendations only.

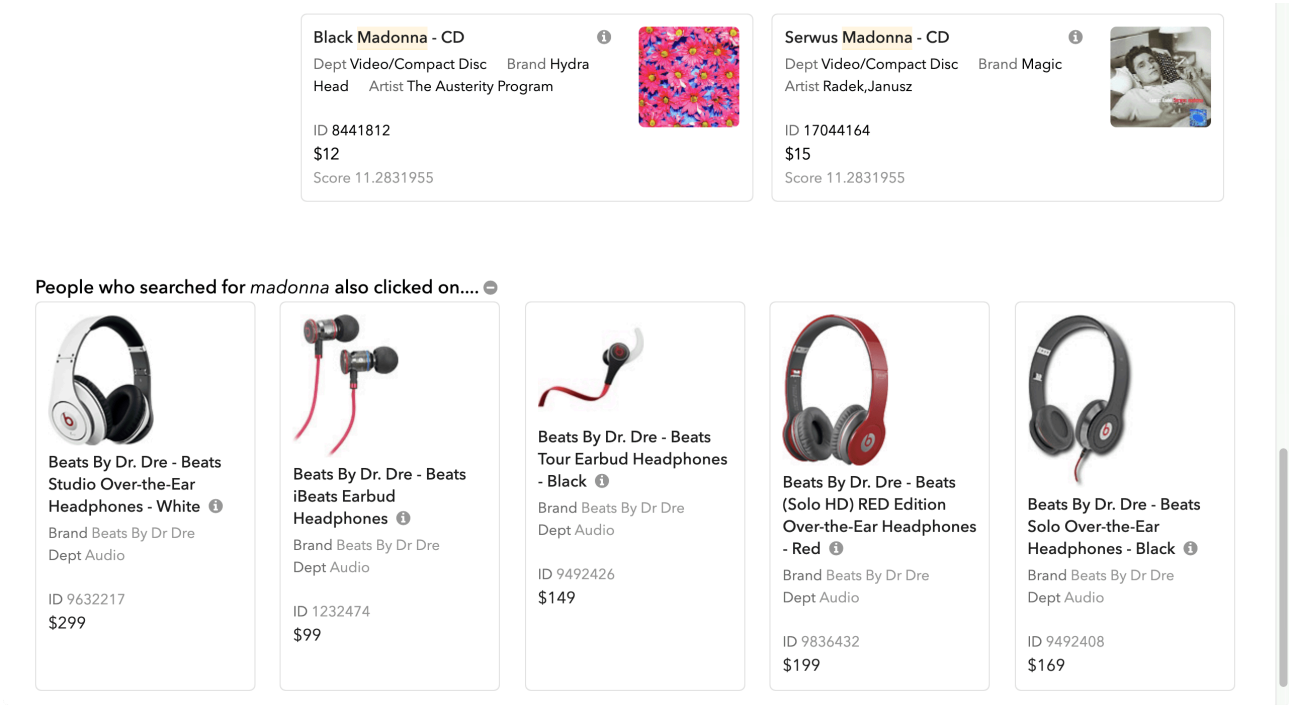
Items-For-Query Recommendations

The Items for Query recommender is the primary algorithm that supports recommendations on the search results page.

The items-for-query recommendations are based on how other users interacted with search results from the same query. This is one type of *collaborative recommendation*.

For example, if users have searched for "titanic" in the past and many of them clicked on the search result for the DVD of the movie *Titanic* (as opposed to books or memorabilia), then this method boosts the DVD item on subsequent searches for "titanic".

Below is an example of items-for-query recommendations based on a search for "madonna":



To use this type of boosting, signals must be enabled but recommendations need not be.

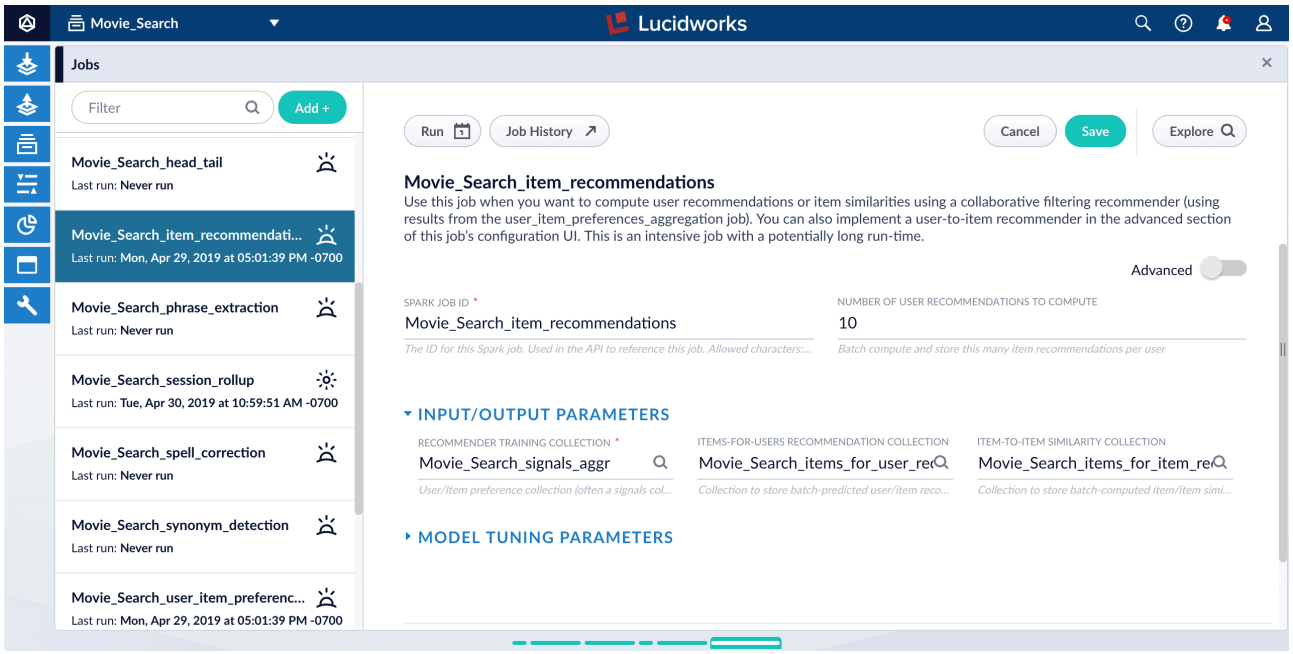
Query pipeline configuration

The *Boost with Signals* query stage is part of the default query pipeline, so usually you do not need to add it. It takes its input from the aggregated signals collection (`COLLECTION_NAME_signals_aggr`) at query time.

Items-For-User Recommendations

Items-for-user recommendations use the *Recommend Items for User* query stage to present items that are similar to ones in which the user has previously shown interest, based on the user's search history, browsing history, purchase history, and so on. This is one type of *collaborative recommendation*.

If you have enabled signals and recommendations for a collection, then the default `COLLECTION_NAME_item_recommendations` job is already created and configured to produce items-for-user recommendations (as well as *items-for-item recommendations*):

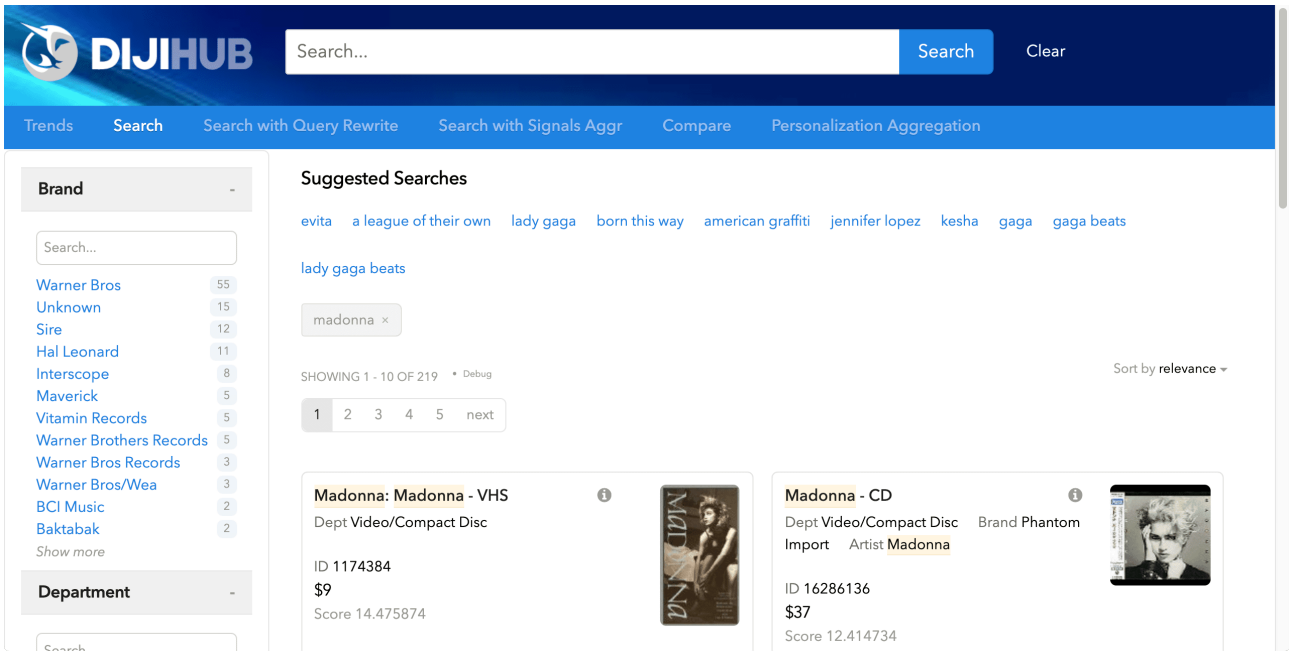


This is an *ALS Recommender* job. See *Items-for-User Recommendations Configuration (ALS)* for more details.

Queries-for-Query Recommendations

Queries-for-query recommendations are queries performed by other users who also performed the current query. For example, when a user searches for "madonna", your app may also suggest searches for "evita", "a league of their own", "lady gaga", and so on.

The "Suggested Searches" shown below are one example of queries-for-query recommendations:



Users-for-Item Recommendations

Users-for-item recommendations retrieve the set of users who have interacted with an item, weighted by the number of interactions. This can be useful for community-driven use cases or for marketing campaigns. For example, on your organization's intranet, clicking a search result could display the item plus a list of colleagues who have interacted with the same item.

This is one type of *collaborative recommendation*.

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*

Run an Experiment Tutorial

The *Run an Experiment* tutorial takes you through the steps needed to run an A/B experiment to compare metrics such as click-through rate (CTR) and query relevance for two differently configured query pipelines. You plan the experiment, create a Fusion app, index a datasource, and create a query profile that includes the configuration data needed for experiments. In Fusion, you start and stop the experiment. A search app uses the query profile for Fusion queries. Different users get different search results, but they are blissfully unaware that an experiment is going on.

A/B/n experiments

Fusion AI's experiments feature set implements A/B/n experiments *(https://en.wikipedia.org/wiki/A/B_testing)*, 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".

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

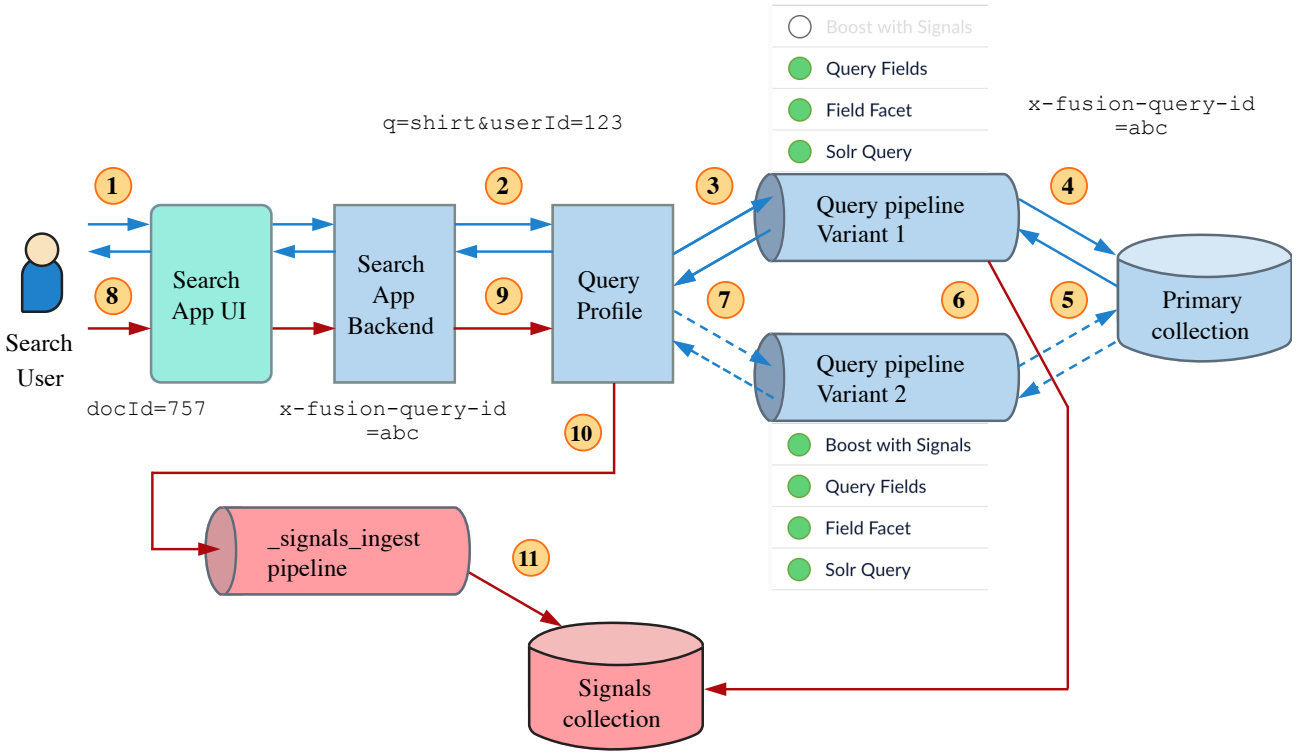
In an experiment:

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.

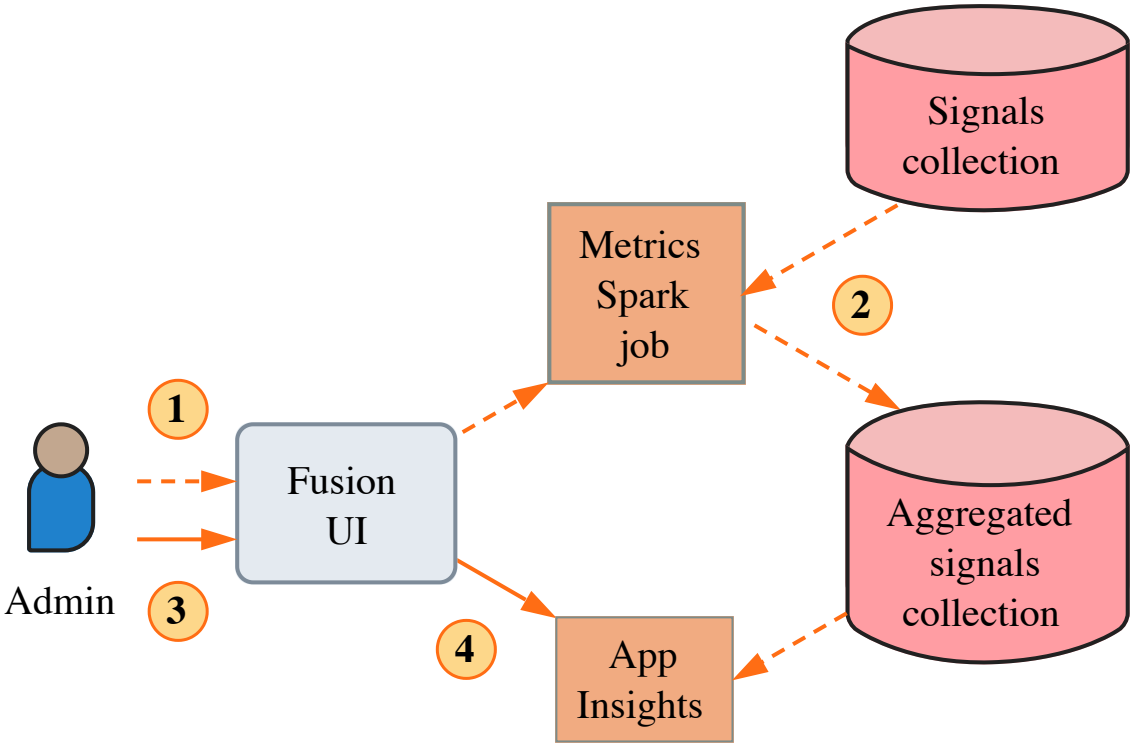


1. A user searches in a search app. For example, the user might search for `shirt`.
2. The search app backend appends a `userId` or other unique ID that identifies the user, for example, `userId=123`, to the query and sends the query to the query profile endpoint for the experiment.
3. Using information 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. Specifically, the click signal *must* include a field named `fusion_query_id` in the `params` object of the raw click signal whose value was returned in the response object in a header named `x-fusion-query-id`. If you are tracking queries and responses with App Studio, the `fusion_query_id` parameter will be passed with the click signal as long as you specify the appropriate response attribute in your `track:clicks` tag.

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 collection used for aggregated signals (`job_reports`).
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.

Machine Learning

See also these subtopics:

- *Machine Learning Models in Fusion*
- *Machine Learning Jobs*

Apache Spark *(http://spark.apache.org/)* 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*.

See *Machine Learning Jobs* for details about each pre-defined machine learning job in Fusion.

To schedule and run jobs on the nodes in the cluster, Spark uses Akka *(https://en.wikipedia.org/wiki/Akka_(toolkit))* which is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.

Further reading

- Machine Learning in Lucidworks Fusion *(https://lucidworks.com/post/machine-learning-in-lucidworks-fusion/)*
- Apache Spark Key Terms, Explained *(https://databricks.com/blog/2016/06/22/apache-spark-key-terms-explained.html)*
- Apache Spark on Wikipedia *(https://en.wikipedia.org/wiki/Apache_Spark)*
- Akka toolkit on Wikipedia *(https://en.wikipedia.org/wiki/Akka_(toolkit%29)*
- Reactive App presentation *(https://events.static.linuxfound.org/sites/events/files/slides/Reactive%20app%20using%20Actor%20model%20%26%20Apache%20Spark.pdf)*

Machine Learning Jobs

Fusion AI provides these job types to perform machine learning tasks.

Signals analysis

These jobs analyze a collection of signals in order to perform *query rewriting*, *signals aggregation*, or *experiment analysis*.

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

Query rewriting

These jobs produce data that can be used for query rewriting or to inform updates to the *synonyms.txt file*.

- *Head/Tail Analysis*
  Perform head/tail analysis of queries from collections of raw or aggregated signals, to identify underperforming queries and the reasons. This information is valuable for improving overall conversions, Solr configurations, auto-suggest, product catalogs, and SEO/SEM strategies, in order to improve conversion rates.
- *Token and Phrase Spell Correction*
  Detect misspellings in queries or documents using the numbers of occurrences of words and phrases.

Signals aggregation

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

Experiment analysis

- *Ranking Metrics*
  Calculate relevance metrics (nDCG and so on) by replaying ground truth queries against catalog data using variants from an experiment.
- *SQL-Based Experiment Metric* (deprecated)
  This job is created by an experiment in order to calculate an objective.

  SQL-Based Experiment Metric job is deprecated as of Fusion AI 4.0.2.

Collaborative recommenders

These jobs analyze signals and generate matrices used to provide *collaborative recommendations*.

- *ALS Recommender*
  Use this job when you want to compute user recommendations or item similarities using a collaborative filtering recommender. You can also implement a user-to-item recommender in the advanced section of this job's configuration UI. This job uses SparkML's Alternating Least Squares (ALS) *(https://spark.apache.org/docs/2.1.0/ml-collaborative-filtering.html)*.
- *Matrix Decomposition-Based Query-Query Similarity*
  Train a collaborative filtering matrix decomposition recommender using SparkML's Alternating Least Squares (ALS) *(https://spark.apache.org/docs/2.1.0/ml-collaborative-filtering.html)* to batch-compute query-query similarities.

Content-based recommenders

Content-based recommenders create matrices of similar items based on their content.

- *Item Similarity Recommender*
  Use this job when you want to compute user recommendations based on pre-computed item similarities.

Content analysis

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

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

- *Collection Analysis*
  Use this job when you want to compute basic metrics about your collection, like average word length, phrase percentages, and outlier documents (with very many or very few documents).
- *Levenshtein Spell Checking*
  Compute the edit distance between all values in a field.

  Levenshtein Spell Checking job is deprecated as of Fusion AI 4.1.0. Use the *Token and Phrase Spell Correction job* instead.
- *Logistic Regression Classifier Training*
  Train a regularized logistic regression model for text classification.
- *Outlier Detection*
  Use this job when you want to find outliers from a set of documents and attach labels for each outlier group.
- *Random Forest Classifier Training* (deprecated)
  Train a random forest classifier *(https://spark.apache.org/docs/2.1.0/ml-classification-regression.html#random-forest-classifier)* for text classification.
- *Statistically Interesting Phrases*
  Use this job when you want to identify phrases in your content.
- *Word2Vec Model Training* (Deprecated)
  Train a shallow neural model, and project each document onto this vector embedding space.

Data ingest

Machine Learning Models in Fusion

Fusion provides the following tools required for the model training process:

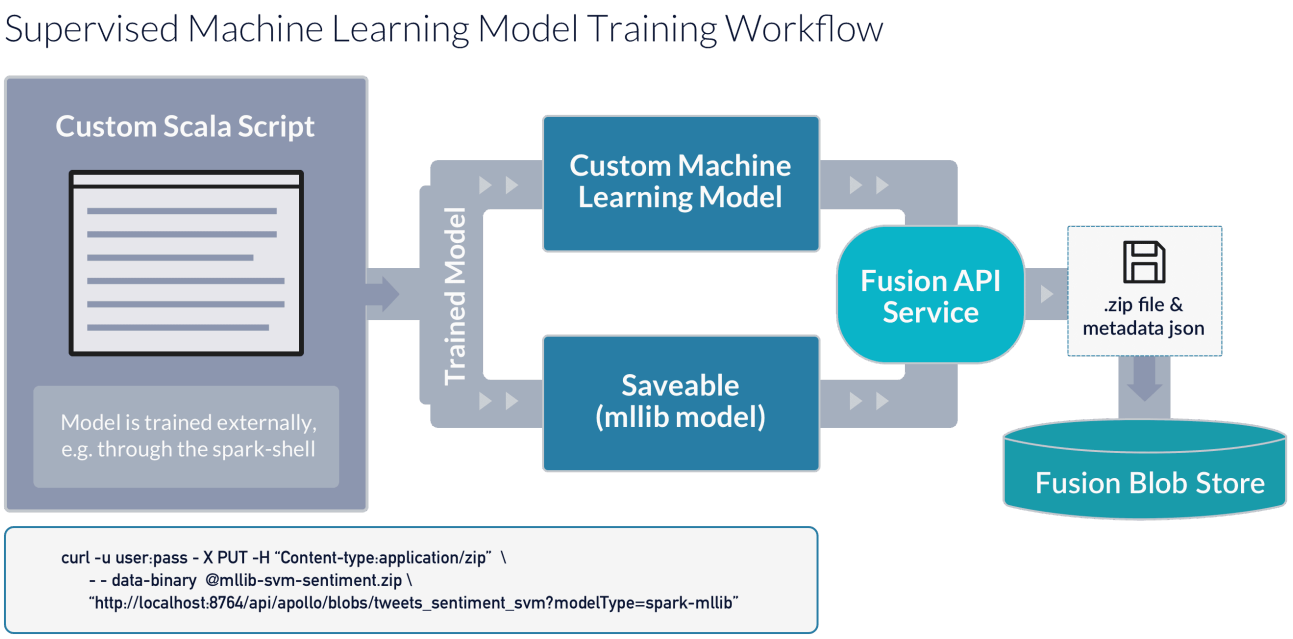
- Solr can easily store all your training data.
- Spark jobs perform the iterative machine learning training tasks.
- Fusion's blob store makes the final model available for processing new data.

Training Models

> The approach for training models explained in this section still works in Fusion 4.0. An alternative approach introduced in Fusion 3.1 lets you create model-training jobs in the Fusion UI. See Machine Learning in Lucidworks Fusion *(https://lucidworks.com/post/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` *(https://github.com/lucidworks/spark-solr/blob/master/src/main/scala/com/lucidworks/spark/example/ml/)* repository.

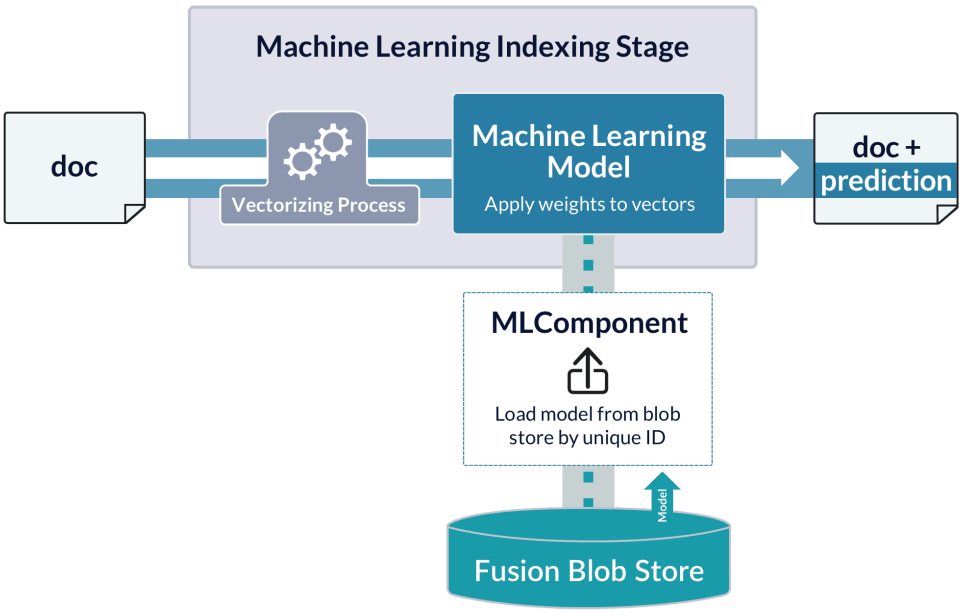
The following diagram depicts this process:



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:

Supervised Machine Learning Model Serving Workflow



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:

```
{ "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`:

```
{ "id":"tweets_sentiment_svm",
"modelClassName":"org.apache.spark.mllib.classification.SVMModel",
"featureFields":["tweet_txt"], "vectorizer":[ { "lucene-analyzer": { "analyzers":
[{ "name":"std_tok_lower", "tokenizer":{"type":"standard"}, "filters":
[{"type":"lowercase"}]}], "fields":[{"regex":".+","analyzer":"std_tok_lower"}]}
}, { "hashingTF":{"numFeatures":"1000000"} } ] }
```

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.

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` *(https://github.com/lucidworks/spark-solr/blob/master/src/main/scala/com/lucidworks/spark/example/ml/)* repository.

Reference Guides

- *API Reference*
- *Index Pipeline Stages Reference*
- *Query Pipeline Stages Reference*

Jobs Configuration

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

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

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

- *ALS Recommender*
  Use this job when you want to compute user recommendations or item similarities using a collaborative filtering recommender. You can also implement a user-to-item recommender in the advanced section of this job's configuration UI. This job uses SparkML's Alternating Least Squares (ALS) *(https://spark.apache.org/docs/2.1.0/ml-collaborative-filtering.html)*.
- *Cluster Labeling*
  Use this job when you already have clusters or well-defined document categories, and you want to discover and attach keywords to see representative words within those existing clusters. (If you want to create new clusters, use the *Document Clustering job*.)
- *Co-occurrence Similarity*
  Use this job when you only want to compute item-to-item similarities. This method is more lightweight than the generic Recommendations job.
- *Collection Analysis*
  Use this job when you want to compute basic metrics about your collection, like average word length, phrase percentages, and outlier documents (with very many or very few documents).
- *Ground Truth*
  Estimate ground truth queries using click signals and query signals, with document relevance per query determined using a click/skip formula.
- *Head/Tail Analysis*
  Perform head/tail analysis of queries from collections of raw or aggregated signals, to identify underperforming queries and the reasons. This information is valuable for improving overall conversions, Solr configurations, auto-suggest, product catalogs, and SEO/SEM strategies, in order to improve conversion rates.
- *Item Similarity Recommender*
  Use this job when you want to compute user recommendations based on pre-computed item similarities.
- *Levenshtein Spell Checking*
  Compute the edit distance between all values in a field.
  > Levenshtein Spell Checking job is deprecated as of Fusion AI 4.1.0. Use the *Token and Phrase Spell Correction job* instead.
- *Logistic Regression Classifier Training*
  Train a regularized logistic regression model for text classification.
- *Matrix Decomposition-Based Query-Query Similarity*
  Train a collaborative filtering matrix decomposition recommender using SparkML's Alternating Least Squares (ALS) *(https://spark.apache.org/docs/2.1.0/ml-collaborative-filtering.html)* to batch-compute query-query similarities.
- *Outlier Detection*
  Use this job when you want to find outliers from a set of documents and attach labels for each outlier group.

ALS Recommender Jobs

Use this job when you want to compute user recommendations or item similarities using a collaborative filtering recommender. You can also implement a user-to-item recommender in the advanced section of this job's configuration UI. This job uses SparkML's Alternating Least Squares (ALS) *(https://spark.apache.org/docs/2.1.0/ml-collaborative-filtering.html)*.

The `COLLECTION_NAME_user_item_preferences_aggregation` job provides input data for this job and must run before it. See *SQL Aggregations* for details.

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

When you *enable recommendations* for a collection, Fusion automatically creates an ALS Recommender job called `COLLECTION_NAME_item_recommendations`. This job generates both *items-for-user recommendations* and *items-for-item recommendations*, then stores the results in the `COLLECTION_NAME_items_for_user_recommendations` and `COLLECTION_NAME_items_for_item_recommendations` collections.

Basic job configuration

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

- `numRecs`/**Number of User Recommendations to Compute**
  This is the number of recommendations that you want to return *per item* (for items-for-item recommendations) or *per user* (for items-for-user recommendations) in your dataset.

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

  Think of this as generating a matrix where the rows are the users and the columns are the recommendations. If we choose 1000 items to recommend, the size of the matrix will be `(number of users) x (number of items to recommend)`. For instance, if there are 10,000 users and 1000 recommendations, then the size of the matrix will be `10,000x1000`.

Input/output parameters

- `trainingCollection`/**Recommender Training Collection**
  Usually this should point to the `COLLECTION_NAME_signals_aggr` collection. If you are using another aggregated signals collection, verify that this field points to the correct collection name.
- `outputItemSimCollection`/**Item-to-item Similarity Collection**
  Usually this should point to the `COLLECTION_NAME_items_for_item_recommendations` collection. This collection will store the `N` most similar items for every item in the collection, where `N` is determined by the `numSims`/**Number of Item Similarities to Compute** field described below. Fusion can query this collection after the job to determine the most similar items to recommend based on an item choice.

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

Model tuning parameters

- `numSims`/**Number of Item Similarities to Compute**
  This is similar to `numRecs`/**Number of User Recommendations to Compute** in the sense that this number of similar items are found for each item in the collection. Think of it as a matrix of size: `(number of items) x (number of item similarities to compute)`.

  This is not computationally expensive because it is just a similarity calculation (which involves no optimization). A reasonable value would be 30–250. It will also depend on the number of items displayed in your search application.
- `implicitRatings`/**Implicit Preferences**
  The concept of Implicit preferences is explained in *Implicit vs explicit signals*.

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

  If explicit ratings values are used (such as ratings from the user) then this box can be unchecked.
- `deleteOldRecs`/**Delete Old Recommendations**
  If you have reasons not to draw on old recommendations, then check this box. If this box is unchecked, then old recommendations will not be deleted but new recommendations will be appended with a different job ID. Both sets of

recommendations will be contained within the same collection.

Advanced job configuration

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

- `excludeFromDeleteFilter`/**Exclude from Delete Filter**
  If you have selected `deleteOldRecs`/**Delete Old Recommendations** but you do not want to completely delete all old recommendations, this field allows you to input a query that captures the data you want keep and removes the rest.
- `numUserRecsPerItem`/**Number of Users to Recommend to each Item**
  This setting indicates which users (from the known user group) are most likely to be interested in a particular item. The setting allows you to choose how many of the most interested users you would like to precompute and store.

  If one thinks of an estimated user-item matrix (after optimization), an item is a single column from the matrix, so if we wanted the top 100 users per item, we would sort the interest values in that column in descending order and take the top 100 row indices which would correspond to individual users.
- `maxTrainingIterations`/**Maximum Training Iterations**
  The Alternating Least Squares algorithm involves optimization to find the two matrices `(user x latent factor and latent factor x item)` that best approximate the original user-item matrix (formed from the signals aggregation).

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

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

Training data settings

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

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

  ```
  +doc_id:[* TO *] +user_id:[* TO *]
  ```

  This query returns all signals documents that have a `user_id` and `doc_id` field. Each query is separated by a space. The plus (`+`) sign is a positive request for the field of interest, meaning return signals with `doc_id` instead of signals without `doc_id` (negated or opposite queries are returned by prefixing with a negative (`-`) sign).
- `popularItemMin`/**Training Data Filter By Popular Items**
  The underlying assumption of this parameter is that the more users that view an item, the more popular that item is. Therefore, this value signifies the minimum number of interactions that must occur with the item for it to be considered a training data point.

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

  One way to speed up training is to increase this number along with the training data sampling fraction. A reasonable number is between 10 and 20 depending on the application and user base. For instance, a song may be played much more than a movie and both may have more interaction than purchasing an item.
- `trainingSampleFraction`/**Training Data Sampling Fraction**
  This value is the percentage of the signal data or training data that you want to use for training the recommender job. It is advised to set this value to 1 and reduce the training data size (while increasing quality) by increasing the **Training Data Filter By Popular Items** as well as increasing the weight threshold in the **Training Data Filter Query**.
- `userIdField`/**Training Collection User Id Field**
  The ALS algorithm needs users, items, and a score of their interaction. The user ID field is the field name within the signal data that represents a user ID.

- `itemIdField`/**Training Collection Item Id Field**
  The item ID field is the field name within the aggregated signal data that represents the item or documents of interest.
- `weightField`/**Training Collection Weight Field**
  The weight field contains the score representing the interest of the user in an item.
- `initialBlocks`/**Training Block Size**
  In *Spark*, the training data is split amongst the executors in unchangeable blocks. This parameter sets the size of these blocks for training, but it requires advanced knowledge of Spark internals. We recommend leaving this setting at -1.

Model settings

- `modelId`/**Recommender Model ID**
  The **Recommender Model ID** is assigned the field `modelId` in the `_items_for_item_recommendations` and `_items_for_user_recommendations` recommendations collections. This allows you to filter the recommendations by the recommender model ID. When the recommender job runs, a job ID is also assigned; therefore, you can see the results from different runs of the same job parameters. If you want to experiment with different parameters, it is advised to change the recommender model ID to reflect the parameters so that you can find the best parameters.
- `saveModel`/**Save Model in Solr**
  Saving the model in Solr adds the parameters to the `_recommender_models` collection as a document. Using this method allows you to track all the recommender configurations.
- `modelCollection`/**Model Collection**
  This is the collection to store the experiment configurations (`_recommender_models` by default).
- `alwaysTrain`/**Force model re-training**
  When the job runs, it checks to see whether the model ID for the job already exists in the model collection. If the model does exist, it uses the pre-existing model to get the recommendations. Otherwise, if the box is checked it will re-run the recommender job and redo the optimization from scratch. Unless you need to maintain this ID name, it is advisable to create a separate model ID for each new combination of parameters.

Grid search settings

- `initialRank`/**Recommender Rank**
  The recommender rank is the number of latent factors into which to decompose the original user-item matrix. A reasonable range is 50-200. Above 200, the performance of the optimization can degrade dramatically depending on computing resources.
- `gridSearchWidth`/**Grid Search Width**
  Grid search is an automatic way to determine the best parameters for the recommender model. It tries different combinations of parameters of equally spaced units within a parameter domain and takes the model that has the lowest cost function value. This is a long process because a single run can take several hours depending on the computing resources, so trying multiple combinations can take some time. Depending on the size of your training data, it is better to do a manual grid search to reduce the number of runs needed to find a suitable recommender model.
- `initialAlpha`/**Implicit Preference Confidence**
  The implicit preference confidence is an approximation of how confident you are that the implicit data does indeed represent an accurate level of interest of a user in an item. Typical values are 1-100, with 100 being more confident in the training data representing the interest of the user. This parameter is used as a regularizer for optimization. The higher the confidence value, the more the optimization is penalized for a wrong approximation of the interest value.
- `initialLambda`/**Initial Lambda**
  Lambda is another optimization parameter that prevents overfitting. Remember we are decomposing the user-item matrix by estimating two matrices. The values in these matrices can be any number, large or small, and have a wide spread in the values themselves. To keep the scale of the value consistent or reduce the spread of the values, we use a regularizer. The higher the lambda, the smaller the values in the two estimated matrices. A smaller lambda gives the algorithm more freedom to estimate an answer which can result in overfitting. Typical values are between 0.01 and 0.3.
- `randomSeed`/**Random Seed**
  When the two matrices are first being estimated, their values are set randomly as an initialization. As the optimization proceeds the values are changed according to the error in the optimization. When training it is important to keep the initialization the same in order to determine the effect of different values of parameters in the model. Keep this value the same across all experiments.

Item metadata settings

- `itemMetadataCollection`/**Item Metadata Collection**
  The main collection has very detailed information about each item, much of which is not necessary for training the recommender system. All that is important to train the recommender are the document IDs and the known users. If you have this metadata in a different collection than the main collection, enter that collection's name here. Once the training is complete, the document ID of

the relevant documents can be used to retrieve detailed information from the item catalog. The point is to train on small data per item and retrieve the detailed information for only relevant documents.

- `itemMetadataJoinField`/**Item Metadata Join Field**
  This is the field that is common to the aggregated signal data and the original data. It is used to join each document from the recommender collection to the original item in the main collection. Usually this is the `id` field.
- `itemMetadataFields`/**Item Metadata Fields**
  These are fields from the main collection that should be returned with each recommendation. You can add fields here by clicking the **Add** icon. To ensure that this works correctly, verify that `itemMetadataJoinField`/**Item Metadata Join Field** has the correct value.

Cluster Labeling Jobs

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

Co-occurrence Similarity Jobs

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

> Cooccurrence Similarity job is deprecated as of Fusion 4.1.

Collection Analysis Jobs

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

Document Clustering Jobs

Cluster a set of documents and attach cluster labels.

Ground Truth Jobs

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

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

Head/Tail Analysis Jobs

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

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

Head/tail analysis configuration

The job configuration must specify the following:

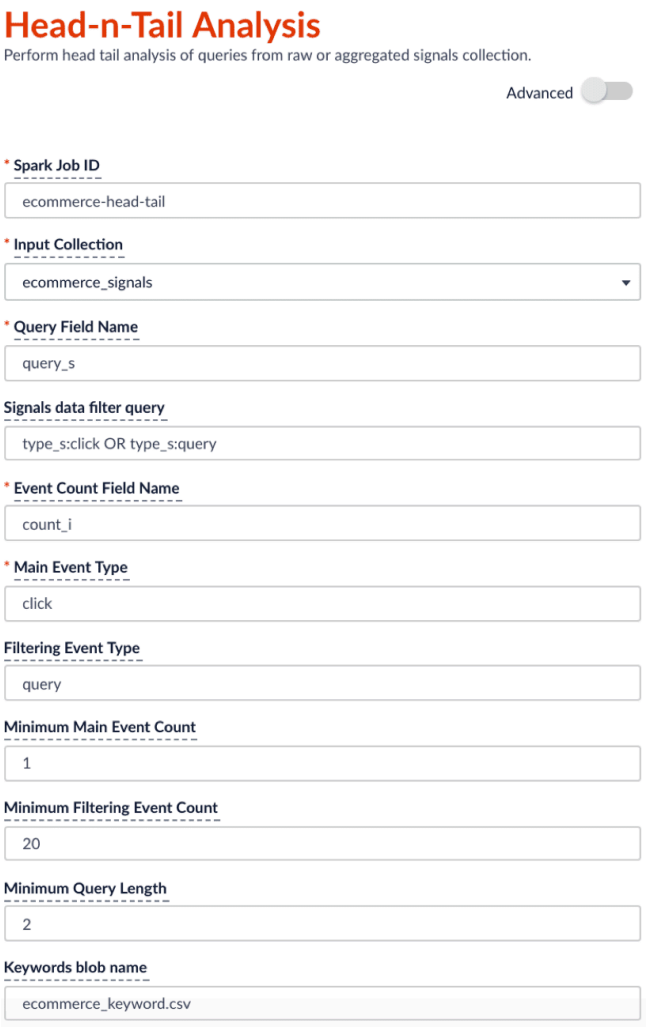
- The *signals* collection (the **Input Collection** parameter)
  Signals can be raw (the `_signals` collection) or *aggregated* (the `_signals_aggr` collection).
- The query string field (the **Query Field Name** parameter)
- The event count field
  For example, if signal data follows the default Fusion setup, then `count_i` is the field that records the count of raw signals and `aggr_count_i` is the field that records the count after aggregation.

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

- The main event (the `mainType`/**Main Event Type** parameter)
- The filtering/secondary event (the `filterType`/**Filtering Event Type** parameter)
  If you only have one event type, leave this parameter empty.

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

An example configuration is shown below:



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

Job output

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

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

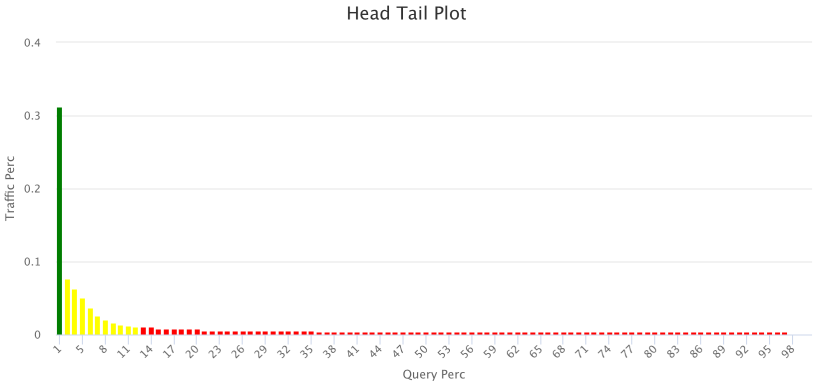
You can use *App Insights* to visualize each of these tables:

1. In the Fusion workspace, navigate to **Analytics** > **App Insights**.
   The App Insights dashboard appears.
2. On the left, click **Analytics**  .
3. Under **Standard Reports**, click **Head Tail analysis**.
   The Head/Tail Analysis job output tables appear. These are described in more detail below.

Head/Tail Plot (`overall_distribution`)

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

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



- Green = head
- Yellow = torso
- Red = tail

Summary Stats (`summary_stat`)

This user-configurable summary statistics table shows how much traffic is produced by various query groups, to help understand the head/tail distribution.

Summary Stats

Statistics	Value
total number of unique queries	22231
top 100 queries leads to ?% total events	21.24%
top 1.0% queries leads to ?% total events	31.06%
number of queries that constitute the top 75.0% of all events	3060
number of queries that constitute the top 50.0% of all events	680
number of queries that constitute the top 25.0% of all events	140
last 1.0% of total events spreaded in how many queries	1613
how many queries leads to zero events.	628
how many queries have less than 5 events	19734
events threshold for tail.	5
events threshold for head.	39

You can configure this table before running the job. Click **Advanced** in the Head/Tail Analysis job configuration panel, then tune these parameters:

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

Query Details (`queries_ordered`)

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

Query Details

Order ID	Query	Traffic (Click)	Traffic (Query)	Traffic Perc (Click)	Length	Num Tokens	Segment
1	ipad	860	862	00.873%	4	1	head
5	2622037 2127204 2127213 2121716 2138291	466	467	00.473%	39	5	head
6	tablet	435	437	00.442%	6	1	head
7	ipod	422	424	00.429%	4	1	head
8	beat	407	409	00.413%	4	1	head
9	touchpad	406	407	00.412%	8	1	head
10	kindle	332	334	00.337%	6	1	head
11	macbook	326	328	00.331%	7	1	head
12	iphone	320	322	00.325%	6	1	head
15	headphone	269	271	00.273%	9	1	head
17	projector	258	260	00.262%	9	1	head

Top Tokens (`tokens_ordered`)

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

Top Tokens

Tokens	Count
tv	675
the	613
samsung	609
sony	543
case	514
laptop	476
iphone	442
dvd	396
ipod	395
hp	376
car	359
wireless	356

Query Length (`queryLength`)

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

Query Length

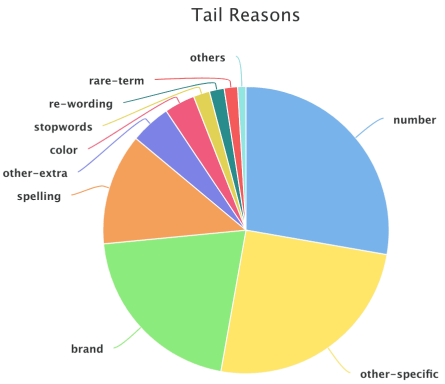
Avg String Length	One Word Perc	Two Words Perc	Three Words Perc	Four Words Perc	Five Words Perc	Six+ Words Perc
13	17.89	40.06	27.01	10.07	3.36	1.61

Tail Reasons table and pie chart (`tail_reasons`)

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

Tail Reasons

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



Pre-defined tail reasons

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

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

Custom dictionary

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

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

```
keyword,type a,stopword an,stopword and,stopword blue,color white,color
black,color hp,brand samsung,brand sony,brand
```

How to install a custom dictionary

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

Head Tail Similarity (`tail_rewriting`)

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

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

Item Similarity Recommender Jobs

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

Levenshtein Spell Checking Jobs

Compute the edit distance between all values in a field.

> Levenshtein Spell Checking job is deprecated as of Fusion AI 4.1.0. Use the *Token and Phrase Spell Correction job* instead.

Logistic Regression Classifier Training Jobs

Train a regularized logistic regression model for text classification.

Matrix Decomposition-Based Query-Query Similarity Jobs

Train a collaborative filtering matrix decomposition recommender using SparkML's Alternating Least Squares (ALS) *(https://spark.apache.org/docs/2.1.0/ml-collaborative-filtering.html)* to batch-compute query-query similarities.

Outlier Detection Jobs

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

Parameterized SQL Aggregation Jobs

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

Random Forest Classifier Training Jobs

Train a random forest classifier *(https://spark.apache.org/docs/2.1.0/ml-classification-regression.html#random-forest-classifier)* for text classification.

Ranking Metrics Jobs

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

SQL-Based Experiment Metric Jobs

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

> SQL-Based Experiment Metric job is deprecated as of Fusion AI 4.0.2.

Statistically Interesting Phrases Jobs

Use this job when you want to identify phrases in your content.

> In Fusion 4.1+, this job becomes the *Phrase Extraction job*.

Token and Phrase Spell Correction Job

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

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

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

For additional background, see the blog post Advanced Spell Check with Fusion 4 *(https://lucidworks.com/post/spellcheck-with-lucidworks-fusion/)*.

	Solr treats spelling corrections as synonyms. See the blog post Multi-Word Synonyms: Solr Adds Query-Time Support *(https://lucidworks.com/post/multi-word-synonyms-solr-adds-query-time-support//)* for more details.

1. Create a job

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

How to create a new job

1. In the Fusion workspace, navigate to  > **Jobs**.
2. Click **Add** and select the job type **Token and phrase spell correction**.
   The New Job Configuration panel appears.

2. Configure the job

Use the information in this section to configure the Token and Phrase Spell Correction job.

Required configuration

The configuration must specify:

- **Spark Job ID.** Used in the API to reference the job. Maximum 63 alphabetic characters, hyphen (-), and underscore (_).
- **INPUT COLLECTION.** The `trainingCollection` parameter that can contain signal data or non-signal data. For signal data, select **Input is Signal Data** (`signalDataIndicator`). Signals can be raw (from the `_signals` collection) or *aggregated* (from the `_signals_aggr` collection).
- **QUERY STRING.** The **Query Field Name**/`fieldToVectorize` parameter.
- **COUNT FIELD**
  For example, if signal data follows the default Fusion setup, then `count_i` is the field that records the count of raw signals and `aggr_count_i` is the field that records the count after aggregation.

Event types

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

- The main event (the **Main Event Type**/`mainType` parameter)
- The filtering/secondary event (the **Filtering Event Type**/`filterType` parameter)
  If you only have one event type, leave this parameter empty.

For example, if you specify the main event type to be `click` with a minimum count of 0 and the filtering event type to be `query` with a minimum count of 20, then the job:

- Filters on the queries that get searched at least 20 times.
- Checks among those popular queries to determine which ones didn't get clicked at all, or were only clicked a few times.

Spell check documents

If you unselect the **Input is Signal Data** checkbox to indicate finding misspellings from content documents rather than signals, then you do not need to specify the following parameters:

- Count Field
- Main Event Field
- Filtering Event Type
- Field Name of Signal Type
- Minimum Main Event Count

- Minimum Filtering Event Count

Use a custom dictionary

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

Example configuration

This is an example configuration:

Token and phrase spell correction

Provide token and phrase level spell correction which can be put into synonym list

Advanced

* Spark Job ID

spell_check

* Input Collection

ecommerce_signals

* Query Field Name

query_s

Signals data filter query

type_s:click OR type_s:query

Stopwords blob

stopword.txt

Dictionary Collection

ecommerce

Dictionary Field

description

* Count Field

count_i

* Main Event Type

click

Filtering Event Type

query

Minimum Main Event Count

1

Minimum Filtering Event Count

10

Minimum Prefix Match

1

Minimum Length of Misspelling

5

Maximum Edit Distance

3

Input is Signal Data ☑

When you have configured the job, click **Save** to save the configuration.

3. Run the job

> If you are finding spelling corrections in aggregated data, you need to *run an aggregation job* before running the Token and Phrase Spelling Correction job. You *do not* need to run a Head/Tail Analysis job. The Token and Phrase Spell Correction job does the head/tail processing it requires.

How to run the job

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

4. Analyze job output

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

An example record is as follows:

```
correction_s laptop battery mis_string_len_i 14 misspelling_s laptop baytery
aggr_job_id_s 162fcf94b20T3704c333 score 1 collation_check_s token correction
included corCount_misCount_ratio_d 2095 sound_match_b true id bf79c43b-fc6d-43a7-
931e-185fdac5b624 aggr_type_s tokenPhraseSpellCorrection aggr_id_s
ecom_spell_check correction_types_s phrase => phrase cor_count_i 68648960
suggested_correction_s baytery=>battery cor_string_len_i 14
token_wise_correction_s baytery=>battery cor_token_size_i 2 edit_dist_i 1
timestamp_tdt 2018-04-25T13:23:40.728Z mis_count_i 32768 lastChar_match_b true
mis_token_size_i 2 token_corr_for_phrase_cnt_i 1
```

For easy evaluation, you can export the result output to a CSV file.

5. Use spell correction results

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

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

Useful output fields

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

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

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

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

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

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

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

The `sound_match` and `lastChar_match` fields also provide useful information.

Job tuning

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

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

`trainingDataFilterQuery`/**Data filter query**
See *Event types* above, then adjust this value to reflect the secondary event for your search application. To query all data, set this to `*:*`.
`minCountFilter`/**Minimum Filtering Event Count**
Lower this value to include less-frequent misspellings based on the data filter query.
`maxDistance`/**Maximum Edit Distance**
Raise this value to increase the number of potentially-related tokens and phrases detected.
`minMispellingLen`/**Minimum Length of Misspelling**
Lower this value to include shorter misspellings (which are harder to correct accurately).

Query rewrite jobs post-processing cleanup

To perform more extensive cleanup of query rewrites, complete the procedures in *Query Rewrite Jobs Post-processing Cleanup*.

Word2Vec Model Training Jobs

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

Aggregation Properties

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

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

Aggregation Configuration Parameters

Aggregation configuration parameters

`groupingFields`	An array of strings specifying the fields to group on.
`signalTypes`	The signal types. If not set then any signal type is selected.
`selectQuery`	The query to select the desired signals. If not set then `*:*` will be used, or equivalent.
`sort`	The criteria to sort on within a group. If not set then sort order is by ID, ascending.
`timeRange`	The time range to select signals on, e.g., `[* TO NOW]`. See Solr date range for more options (https://solr.apache.org/guide/8_8/working-with-dates.html *(https://solr.apache.org/guide/8_8/working-with-dates.html)*).
`outputPipeline`	What pipeline to use to process the output. If not set then `_system` pipeline will be used.
`rollupPipeline`	Pipeline to use for processing results of roll-up. This is by default the same index pipeline used for processing the aggregation results.
`rollupAggregator`	The aggregator to use when rolling up. If not set then the same aggregator will be used for roll-up.
`outputCollection`	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.
`aggregator`	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.
`sourceRemove`	If true, the processed source signals will be removed after aggregation. Default is false.
`sourceCatchup`	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.
`outputRollup`	Roll-up current results with all previous results for this aggregation id, which are available in `outputCollection`.

`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 Examples

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:

```
{ "type" : "sumOfSquares", "sourceFields" : [ "params.position_s" ],
"targetField" : "sumOfSquares_position_d" }
```

count

A count of source values, as a long number.

Example:

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

geoMean

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

Example:

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

mean

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

Example:

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

min

The minimum numeric value.

Example:

```
{ "type" : "min", "sourceFields" : [ "params.position_s" ], "targetField" :
"min_position_d" }
```

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:

```
{ "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:

```
{ "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:

```
{ "type" : "split", "sourceFields" : [ "query_s" ], "targetField" :
"event:query_split", "params" : { "regex": "\\s+", "lower": true } }
```

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:

```
{ "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:

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

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:

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

ucollectCollect 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:

```
{ "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.

varianceThe 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:

```
{ "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:

```
{ "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:

```
{ "type" : "topk", "sourceFields" : [ "params.position_s" ], "targetField" :
"topk_position_ss", "params" : { } }
```

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 * (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:

```
{ "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 * (N - 1)` fields following the naming pattern 'sourceField1.sourceField2.targetField'.

Example:

```
{ "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:

```
{ "type" : "histogram", "sourceFields" : [ "params.position_s" ], "targetField" :
"histogram_position_ss", "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:

```
{ "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:

```
{ "type" : "when", "sourceFields" : [ "params.position_s" ], "params" : { "expr"
: "parseFloat(params_position_s) < 3" } }
```

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:

```
{ "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:

```
{ "type" : "script", "sourceFields" : [ ], "params" : { "aggregateScript" :
"result.addField('script_event_id_ss', event.getFieldValue('id'));" } }, { "type"
: "script", "sourceFields" : [ ], "params" : { "aggregateScript" :
"event.addField('position_rnd_1', event.getFieldValue('params.position_s') + 1.0
- Math.random());event.addField('position_rnd_2',
event.getFieldValue('params.position_s') + 5.0 - Math.random() * 10.0);" } }
```

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:

```
{ "type" : "expr", "sourceFields" : [ "query_s", "filters_s" ], "targetField" :
"expr_s", "params" : { "expr" : "v = ''; if (value != null) v = value + ' '; v +
query_s + '_' + filters_s" } }, { "type" : "expr", "sourceFields" : [
"params.position_s" ], "targetField" : "expr_d", "params" : { "expr" : "v = 0; if
(value != null) v = parseFloat(value); v + parseFloat(params_position_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:

```
{ "type" : "noop" }
```

Built-in SQL Aggregation Jobs

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:

Signals aggregation jobs

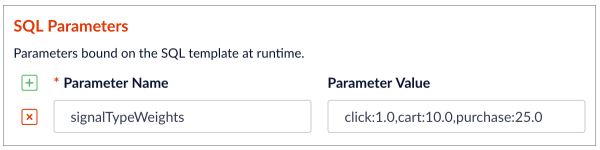
Job	Default input collection	Default output collection	Default schedule
`COLLECTION_NAME_click_signals_aggregation`	`COLLECTION_NAME_signals`	`COLLECTION_NAME_signals_aggr`	Every 15 minutes
`COLLECTION_NAME_session_rollup`	`COLLECTION_NAME_signals`	`COLLECTION_NAME_signals`	Every 15 minutes
`COLLECTION_NAME_user_item_preferences_aggregation`	`COLLECTION_NAME_signals`	`COLLECTION_NAME_signals_aggr`	Once per day
`COLLECTION_NAME_user_query_history_aggregation`	`COLLECTION_NAME_signals`	`COLLECTION_NAME_signals_aggr`	Once per day

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

COLLECTION_NAME_click_signals_aggregation

The `COLLECTION_NAME_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 signal types and weights for this job by changing the `signalTypeWeights` SQL parameter in the Fusion Admin UI.



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.

A user can configure the `COLLECTION_NAME_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:

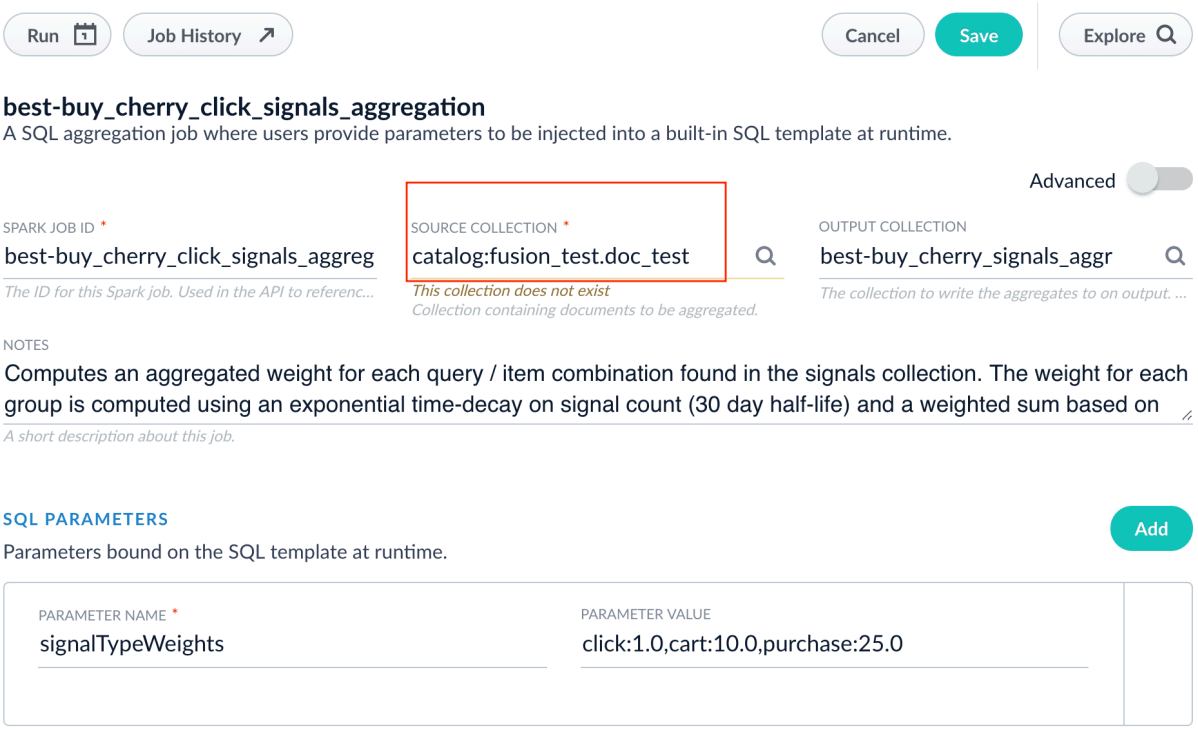
```
sample code: curl -u <username>:<pw> -X POST -H "Content-type:application/json"
--data-binary '{ "name": "fusion_test", "assetType": "project", "description":
"test", "cacheOnLoad": false }' http://localhost:{api-port}/api/catalog
```

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

```
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:{api-port}/api/catalog/fusion_test/assets
```

The parquet file listed above needs to have all the fields in the SQL script which the `COLLECTION_NAME_click_signals_aggregation` job is selecting/using.

3. In the `COLLECTION_NAME_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_NAME_session_rollup

The `COLLECTION_NAME_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_NAME_session_rollup` job uses signals as the input collection and output collection. Unlike other aggregation jobs that write aggregated documents to the `COLLECTION_NAME_signals_aggr` collection, the `COLLECTION_NAME_session_rollup` job creates session signals and saves them to the `COLLECTION_NAME_signals` collection.

COLLECTION_NAME_user_item_preferences_aggregation

The `COLLECTION_NAME_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.

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_NAME_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_NAME_user_query_history_aggregation

The `COLLECTION_NAME_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.

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 *(https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.expressions.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, `WHERE type_s = 'add'`, to provide fine-grained filters.
- The `time_decay` function uses an abbreviated function signature, `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 `q1` and document `1`:

```
[{ "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 `q1`:

```
{ "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":"15d4279d128T755e5137", "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 *(https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.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
```

Query Pipeline Stages

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

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

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

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

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

- *Analytics Catalog Query*
- *Boost Documents*
- *Boost with Signals*
- *Experiment*
- *Machine Learning*
- *Parameterized Boosting*
- *Recommend Items for User*
- *Recommend Items for Item*
- *Solr MoreLikeThis*

Analytics Catalog Stage

The Analytics Catalog query pipeline stage lets you define views of data stored in Fusion collections. You can query the objects in the Analytics Catalog using SQL, Solr Streaming Expressions *(https://cwiki.apache.org/confluence/display/solr/Streaming+Expressions)*, or regular Solr queries. See also the *Catalog API*.

Configuration

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

Boost Documents Stage

The Boost Documents query pipeline stage adds boosting parameters to matched documents based on user-defined rules. Boosts are defined with a term value to boost and the boost factor to add. The boosting parameters are added to the `boost` Solr query parameter.

The Boost Documents rule consists of the following elements:

- `field`. The document field to filter on.
- `keywords`. The words, phrases, or regex used as the filter.
- `mode`. Filtering logic applied to keywords, one of:
  - `exact`. The keyword and the query must match exactly. This is case-sensitive.
  - `phrase`. The phrase matching on the items in the keywords list.
  - `regex`. Treat items in the keywords list as a regex.
  - `match`. Must match every item in the keyword list, but does not require phrase matching.
- `boosts`. Consists of a list of pairs.
  - `value`. One or more terms used for boosting.
  - `boost`. The numeric boost value. Default `100`.

Configuration

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

Boost calculation

In Fusion 4.x.x, numbers entered into boosts are *added* to the score.

> This differs from Fusion 5.x.x, where numbers entered into boosts are *multiplied* with the score. This difference must be considered when you upgrade to Fusion 5.x.x. For more information, see *Fusion 5.x.x Boost Documents Stage*.

Boost with Signals Stage

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

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

See *Recommendation Methods* for more information.

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

Configuration overview

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

Number of Recommendations `numRecommendations`	Sets the `rows` query param in the main query as the maximum number of query results which will be boosted by this pipeline stage.
Number of Signals `numSignals`	Sets the `rows` query param in the query that searches the `_signals_aggr` collection, so only the specified number of aggregated signals are retrieved and used for boosting. When signals boosting is applied to a query, aggregated signals records are queried from the appropriate `_signals_aggr` collection to find out the popularity or boost weight for documents which have signals. `numSignals` limits the number of records to be queried from a `}}{{_signals_aggr` collection and used to calculate this boost.
Aggregation Type `aggrType`	A filter to retrieve aggregated signals in the `_signals_aggr` collection per each aggregated signal's `aggr_type_s` field value.
Solr Field to Boost On `boostId`	The document field in the main collection on which to perform boosting. Typically it should use default field, which is `id`. This field corresponds to the **Rollup Field**/`rollupField` field. Together, these two fields act like a `<field>:<value>` pair in the query modification for boosting.

Boost Method
 	
`boostingMethod`
	This adds a query parameter to the original query, either "query-param" or "query-parser". The result is `("query-param" or "query-parser") + Boost Param("boost" or `bq`)`, as in the examples below:
	
	"query-param"+"boost", result boost query param
	
	`boost="map(query({!field f='id' v='6239046,13026192}), 0, 0, 1, 27.1705)"`
	
	"query-parser"+"boost", result boost query param
	
	`bp_xxx_bbqx="map(query({!field f='id' v='6239046,13026192}), 0, 0, 1, 27.1705)"`
	
	When `Boost Param` uses `bq`, similar logic applies. When **Boost Param**/`boostingParam` uses "boost", it works with both "query-param" and "query-parser".

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

Rollup Weight Field `rollupWeightField`	Indicates the final boost weight used to calculate the new score for docs retrieved by the main query. Similar to **Rollup Field**/`rollupField` above, this should be set to the field in the aggregated signal collection that stores the final weight that was calculated. By default it's `weight_d`, because by default this is used in the *click_signal_aggr SQL job*.
Final Boost Weight Expression `weightExpression`	Calculates the final weight using the weight and score retrieved from the `_signals_aggr` collection. The default value is `math:log(weight_d + 1) + 10 * math:log(score+1)`.

Solr query parameters

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

`qf=query_t`	Defines which field to query. In the default case, the query searches on the `query_t` field of aggregated signal docs.
`pf=query_t^50`	Boosts docs within the set of retrieved docs using phrase matching.
`pf2=query_t^20`	`pf2` is similar to `pf`; the difference is that `pf2` works on bigram phrases.
`pf3=query_t^10`	`pf3` is similar to `pf`; the difference is that `pf3` works on trigram phrases.

FAQs

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

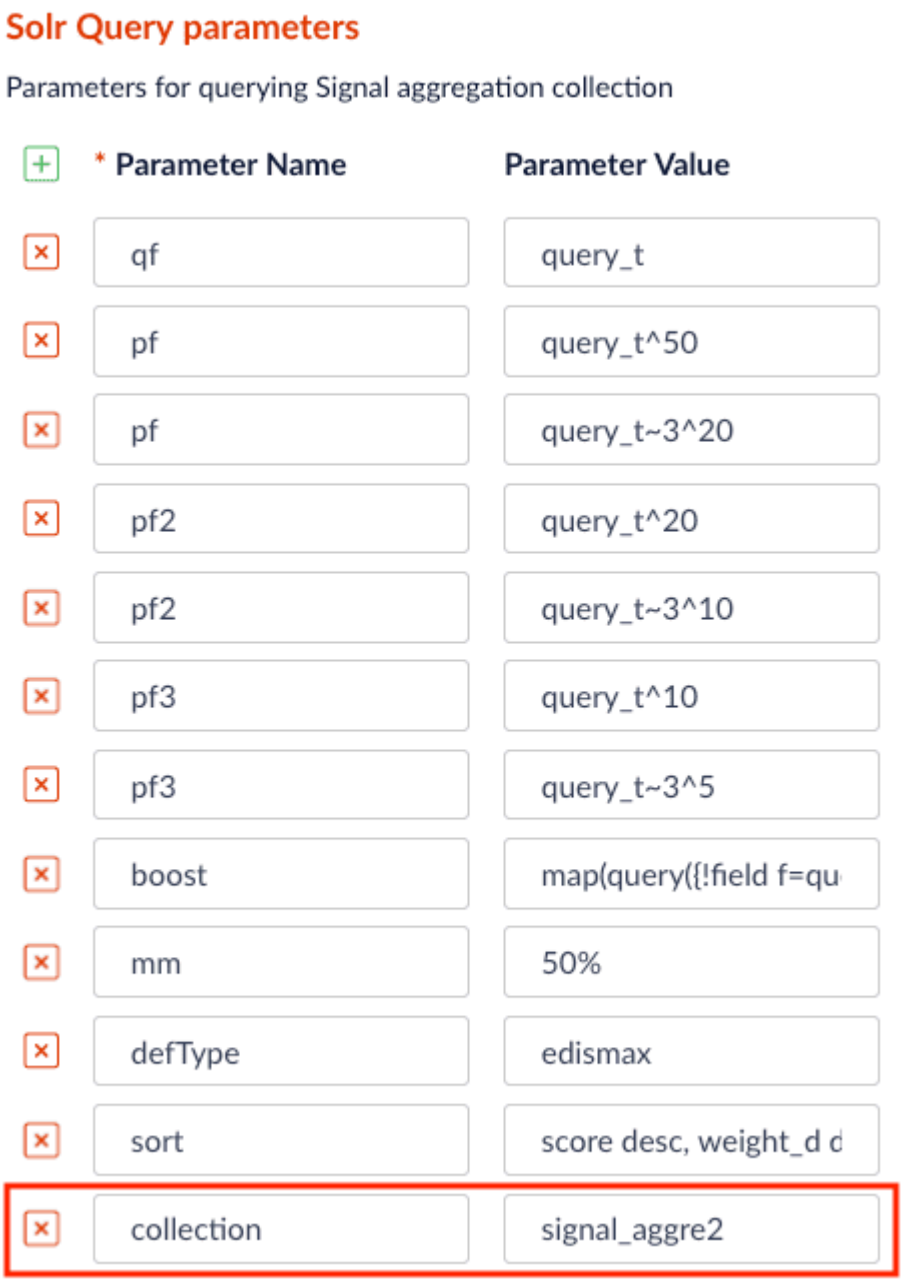
For example, if we have filter query param `fq=format:CD&fq=name:Latin`, this needs to be translated into `lw.rec.fq=filters_s:"format:cd $ name:latin"`. Values must be lowercase. The final main query should be:

```
http://localhost:8764/api/apollo/apps/demo_app/query-pipelines/demo_app/collections/demo_app/select?echoParams=all&wt=json&json.nl=arrarr&sort&start=0&q=apple&debug=true&rows=10&lw.rec.fq=filters_s:"format:cd $ name:latin"
```

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

If there are multiple `fq` values (for example, `format:cd` and `name:latin`), they are ordered alphabetically as strings and joined with " $ " (a $ with a space on each side). In the example, `"format:cd $ name:latin"`.

What if my aggregated signals are in a different collection?
You can point the Boost with Signals stage to a different signal collection by adding a `collection` parameter in the `Solr Query Parameters` section.



Configuration

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

Experiment Stage

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

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

Configuration

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

Machine Learning Stage

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

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

- Use a Fusion job that trains a model, like *Logistic Regression* or *Random Forest*.
- Train a model using Spark's MLlib API *(https://spark.apache.org/docs/latest/mllib-guide.html)* outside of Fusion, and upload this model into Fusion's *blob store*. Complete details are available in *Machine Learning Models in Fusion*.

Configuration

Parameterized Boosting Stage

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

Configuration

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

Recommend Items for Item Stage

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

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

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

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

Prerequisites

Enable recommendations:

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

- **Fusion UI** – In Collections Manager, click the settings icon next to the collection > **Enable Recommendations**.
- **Using the REST API** – Use this command to enable recommendations:

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

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

Using live signals

The **Estimate Recent Results** option uses live signals to augment items-for-user recommendations with real-time recommendations.

When this is enabled, Fusion first looks up items (from previously-generated recommendations) that are similar to the new items. If there are none then it looks up similar users (who also interacted with that item) to get a list of recommendations based on the new items. It then combines those new recommendations with the job-based recommendations already generated for that user (if any), to generate a final list of recommendations.

Configuration

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

Recommend Items for User Stage

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

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

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

Prerequisites

Enable recommendations:

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

- **In the Fusion UI** – With Query Workbench open, click **Settings > Enable Recommendations**.
- **Using the REST API** – Use this command to enable recommendations:
  `curl -u admin:<password> -X PUT http://<hostname>:<port>/api/v1/collections/<collection-name>/features/recommendations -H 'content-type: application/json' -d '{"enabled":true}'

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

Using live signals

The **Estimate Recent Results** option uses live signals to augment items-for-user recommendations with real-time recommendations.

When this is enabled, Fusion first looks up items (from previously-generated recommendations) that are similar to the new items. If there are none then it looks up similar users (who also interacted with that item) to get a list of recommendations based on the new items. It then combines those new recommendations with the job-based recommendations already generated for that user (if any), to generate a final list of recommendations.

Configuration

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

Solr MoreLikeThis Stage

This stage uses the content of the current document to query for similar documents, using *Solr's MoreLikeThis component*.

This stage provides content-based recommendations. For collaborative recommendations, use the *Recommend Items for Item stage*.

See *Recommendations and Boosting* for more information.

Tips

- The incoming query must include an `id` field in order to get recommendations from this stage. The stage returns documents similar to the one specified by this field.
- Since these secondary queries tend to be large, this stage can impact search performance. You can improve performance by first *clustering your documents*, then configuring this stage to query a specific document cluster instead of all documents.

Configuration

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

Boosting

Items-for-item boosting
Read about configuring items-for-item boosting.

Items-for-user boosting
Read about configuring items-for-user boosting.

Items-for-Item Recommendations Configuration (ALS)

Basic job configuration

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

- `numRecs`/**Number of User Recommendations to Compute**
  This is the number of recommendations that you want to return *per item* (for items-for-item recommendations) or *per user* (for items-for-user recommendations) in your dataset.

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

  Think of this as generating a matrix where the rows are the users and the columns are the recommendations. If we choose 1000 items to recommend, the size of the matrix will be `(number of users) x (number of items to recommend)`. For instance, if there are 10,000 users and 1000 recommendations, then the size of the matrix will be `10,000x1000`.

Input/output parameters

- `trainingCollection`/**Recommender Training Collection**
  Usually this should point to the `COLLECTION_NAME_signals_aggr` collection. If you are using another aggregated signals collection, verify that this field points to the correct collection name.
- `outputItemSimCollection`/**Item-to-item Similarity Collection**
  Usually this should point to the `COLLECTION_NAME_items_for_item_recommendations` collection. This collection will store the `N` most similar items for every item in the collection, where `N` is determined by the `numSims`/**Number of Item Similarities to Compute** field described below. Fusion can query this collection after the job to determine the most similar items to recommend based on an item choice.

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

Model tuning parameters

- `numSims`/**Number of Item Similarities to Compute**
  This is similar to `numRecs`/**Number of User Recommendations to Compute** in the sense that this number of similar items are found for each item in the collection. Think of it as a matrix of size: `(number of items) x (number of item similarities to compute)`.

  This is not computationally expensive because it is just a similarity calculation (which involves no optimization). A reasonable value would be 30–250. It will also depend on the number of items displayed in your search application.
- `implicitRatings`/**Implicit Preferences**
  The concept of Implicit preferences is explained in *Implicit vs explicit signals*.

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

  If explicit ratings values are used (such as ratings from the user) then this box can be unchecked.
- `deleteOldRecs`/**Delete Old Recommendations**
  If you have reasons not to draw on old recommendations, then check this box. If this box is unchecked, then old recommendations will not be deleted but new recommendations will be appended with a different job ID. Both sets of recommendations will be contained within the same collection.

Advanced job configuration

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

- `excludeFromDeleteFilter`/**Exclude from Delete Filter**
  If you have selected `deleteOldRecs`/**Delete Old Recommendations** but you do not want to completely delete all old recommendations, this field allows you to input a query that captures the data you want keep and removes the rest.
- `numUserRecsPerItem`/**Number of Users to Recommend to each Item**
  This setting indicates which users (from the known user group) are most likely to be interested in a particular item. The setting allows you to choose how many of the most interested users you would like to precompute and store.

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

- `maxTrainingIterations`/**Maximum Training Iterations**
  The Alternating Least Squares algorithm involves optimization to find the two matrices `(user x latent factor and latent factor x item)` that best approximate the original user-item matrix (formed from the signals aggregation).
  The optimization occurs at the matrix entry level (every non-zero element) and it is iterative. Therefore, the more iterations that are allowed during optimization, the lower the cost function value, meaning more accurate hyperparameters which lead to better recommendations.
  However, the bigger the data, the longer the job takes to run because the number of constraints to satisfy have increased. A value of 10 iterations usually leads to effective results. Above a value of 15, the job will begin to slow dramatically for above 25 million signals.

Training data settings

- `trainingDataFilterQuery`/**Training Data Filter Query**
  This query setting is useful when the main signals collection does not have the *recommended fields*. The two most important fields are `doc_id` and `user_id` because the job must have a user-item pairing. Note that depending on how the signals are collected the names `doc_id` and `user_id` can be different, but the concept remains the same.
  There are times when not all the signals have these fields. In this case we can add a query to select a subset of data that does have a user-item pairing. It is done with the following query:

```
+doc_id:[* TO *] +user_id:[* TO *]
```

  This query returns all signals documents that have a `user_id` and `doc_id` field. Each query is separated by a space. The plus (`+`) sign is a positive request for the field of interest, meaning return signals with `doc_id` instead of signals without `doc_id` (negated or opposite queries are returned by prefixing with a negative (`-`) sign).
- `popularItemMin`/**Training Data Filter By Popular Items**
  The underlying assumption of this parameter is that the more users that view an item, the more popular that item is. Therefore, this value signifies the minimum number of interactions that must occur with the item for it to be considered a training data point.
  The higher the number, the smaller amount of data available for training because it is unlikely that many users interacted with all of the items. However, the quality of the data will be higher.
  One way to speed up training is to increase this number along with the training data sampling fraction. A reasonable number is between 10 and 20 depending on the application and user base. For instance, a song may be played much more than a movie and both may have more interaction than purchasing an item.
- `trainingSampleFraction`/**Training Data Sampling Fraction**
  This value is the percentage of the signal data or training data that you want to use for training the recommender job. It is advised to set this value to 1 and reduce the training data size (while increasing quality) by increasing the **Training Data Filter By Popular Items** as well as increasing the weight threshold in the **Training Data Filter Query**.
- `userIdField`/**Training Collection User Id Field**
  The ALS algorithm needs users, items, and a score of their interaction. The user ID field is the field name within the signal data that represents a user ID.
- `itemIdField`/**Training Collection Item Id Field**
  The item ID field is the field name within the aggregated signal data that represents the item or documents of interest.
- `weightField`/**Training Collection Weight Field**
  The weight field contains the score representing the interest of the user in an item.
- `initialBlocks`/**Training Block Size**
  In *Spark*, the training data is split amongst the executors in unchangeable blocks. This parameter sets the size of these blocks for training, but it requires advanced knowledge of Spark internals. We recommend leaving this setting at -1.

Model settings

- `modelId`/**Recommender Model ID**
  The **Recommender Model ID** is assigned the field `modelId` in the `_items_for_item_recommendations` and `_items_for_user_recommendations` recommendations collections. This allows you to filter the recommendations by the recommender model ID. When the recommender job runs, a job ID is also assigned; therefore, you can see the results from

different runs of the same job parameters. If you want to experiment with different parameters, it is advised to change the recommender model ID to reflect the parameters so that you can find the best parameters.

- `saveModel`/**Save Model in Solr**
  Saving the model in Solr adds the parameters to the `_recommender_models` collection as a document. Using this method allows you to track all the recommender configurations.
- `modelCollection`/**Model Collection**
  This is the collection to store the experiment configurations (`_recommender_models` by default).
- `alwaysTrain`/**Force model re-training**
  When the job runs, it checks to see whether the model ID for the job already exists in the model collection. If the model does exist, it uses the pre-existing model to get the recommendations. Otherwise, if the box is checked it will re-run the recommender job and redo the optimization from scratch. Unless you need to maintain this ID name, it is advisable to create a separate model ID for each new combination of parameters.

Grid search settings

- `initialRank`/**Recommender Rank**
  The recommender rank is the number of latent factors into which to decompose the original user-item matrix. A reasonable range is 50-200. Above 200, the performance of the optimization can degrade dramatically depending on computing resources.
- `gridSearchWidth`/**Grid Search Width**
  Grid search is an automatic way to determine the best parameters for the recommender model. It tries different combinations of parameters of equally spaced units within a parameter domain and takes the model that has the lowest cost function value. This is a long process because a single run can take several hours depending on the computing resources, so trying multiple combinations can take some time. Depending on the size of your training data, it is better to do a manual grid search to reduce the number of runs needed to find a suitable recommender model.
- `initialAlpha`/**Implicit Preference Confidence**
  The implicit preference confidence is an approximation of how confident you are that the implicit data does indeed represent an accurate level of interest of a user in an item. Typical values are 1-100, with 100 being more confident in the training data representing the interest of the user. This parameter is used as a regularizer for optimization. The higher the confidence value, the more the optimization is penalized for a wrong approximation of the interest value.
- `initialLambda`/**Initial Lambda**
  Lambda is another optimization parameter that prevents overfitting. Remember we are decomposing the user-item matrix by estimating two matrices. The values in these matrices can be any number, large or small, and have a wide spread in the values themselves. To keep the scale of the value consistent or reduce the spread of the values, we use a regularizer. The higher the lambda, the smaller the values in the two estimated matrices. A smaller lambda gives the algorithm more freedom to estimate an answer which can result in overfitting. Typical values are between 0.01 and 0.3.
- `randomSeed`/**Random Seed**
  When the two matrices are first being estimated, their values are set randomly as an initialization. As the optimization proceeds the values are changed according to the error in the optimization. When training it is important to keep the initialization the same in order to determine the effect of different values of parameters in the model. Keep this value the same across all experiments.

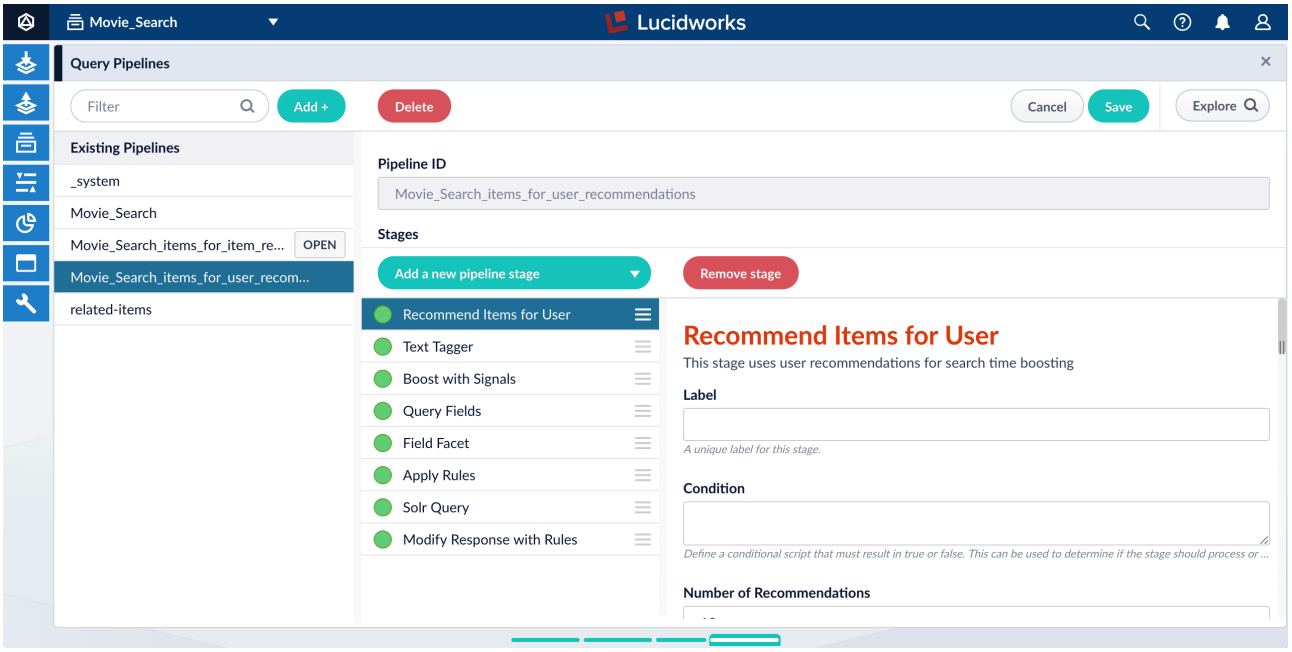
Item metadata settings

- `itemMetadataCollection`/**Item Metadata Collection**
  The main collection has very detailed information about each item, much of which is not necessary for training the recommender system. All that is important to train the recommender are the document IDs and the known users. If you have this metadata in a different collection than the main collection, enter that collection's name here. Once the training is complete, the document ID of the relevant documents can be used to retrieve detailed information from the item catalog. The point is to train on small data per item and retrieve the detailed information for only relevant documents.
- `itemMetadataJoinField`/**Item Metadata Join Field**
  This is the field that is common to the aggregated signal data and the original data. It is used to join each document from the recommender collection to the original item in the main collection. Usually this is the `id` field.
- `itemMetadataFields`/**Item Metadata Fields**
  These are fields from the main collection that should be returned with each recommendation. You can add fields here by clicking the **Add**  icon. To ensure that this works correctly, verify that `itemMetadataJoinField`/**Item Metadata Join Field** has the correct value.

Items-for-User Recommendations Configuration (ALS)

Query pipeline configuration

If you have enabled signals and recommendations for a collection, then the default `COLLECTION_NAME_items_for_user_recommendations` query pipeline is already created and configured to serve items-for-user recommendations:



You can tune the *Recommend Items for User query stage* as needed to refine the output.

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 should not 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 are not predefined.)

For example, if you are 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`.

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:

Field	Description
`id`	Response signal ID set by a query pipeline and returned to the client application using the `x-fusion-query-id` response header
`variant_id`	Experiment variant this response signal is associated with
`query_doc_ids`	Comma-delimited list of document IDs returned in the response, in ranked order
`query_timestamp`	ISO-8601 timestamp for the time when Fusion executed the query
`query_user_id`	User associated with the query. The front-end application must supply this.
`query_rows`	Number of rows returned for this query, that is, the page size
`query_hits`	Total number of documents that match this query, that is, the number of documents that were found
`query_offset`	Page offset
`query_time`	Total time to execute the query (in milliseconds)

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:

```
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 the 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 us 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 are not 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:

```
    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$ *(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 *does not* 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`.

> You must run the `groundTruth` job by hand the first time. Query Relevance `rankingMetrics` jobs that run before the `groundTruth` job runs do not 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:

Query	Document ID	Weight
hammer	123	0.9
hammer	456	0.8
hammer	789	0.7
masking tape	234	0.85
masking tape	567	0.82
masking tape	890	0.76

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 is not 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 *(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 *(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).

REST APIs

Fusion API services are designed to be accessed via Fusion's authentication proxy module which is part of the Fusion UI service (default port 8764). All applications should use this method to access the API service:

```
https://FUSION_HOST:8764/api/<endpoint>
```

List all Fusion component services

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

```
curl -u USERNAME:PASSWORD https://FUSION_HOST:8764/api/introspect
```

REST API Reference Pages

- *Experiments API*
- *Recommendations API*
- *Signals API*

Experiments API

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

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

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

This API requires a *Fusion AI license*.

Experiment types

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

A/B testing

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

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

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

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

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

```
{ "id": "sample-experiment", "uniqueIdParameter": "userId",
"baseSignalsCollection": "bestbuy", "variants": [ { "name": "control-a1", }, {
"name": "control-a2", }, { "name": "b", "queryPipeline": "with-recommendations",
} ], "enabled": true, "metrics": [ { "type": "ctr", "name": "CTR", "primary":
true }, { "type": "conversion-rate", "name": "purchase rate", "signalType":
"purchase", } ] }
```

As users interact with this experiment, we expect the results of metrics for the first two variants to be quite similar, since they are configured identically. With a small sample size, metrics could vary somewhat based on random chance but, as more users interact with the experiment, we expect the metrics to converge on identical results. The third variant, however, will likely perform differently, and its performance will tell us whether we should be using the `with-recommendations` pipeline for all search traffic.

API specification

Recommendations API

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

This API requires a *Fusion AI license*.

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

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

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

Recommendation types

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

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

Output

The output includes the following sections:

`header`	The query parameters (in a section named `queryParams`) and the total time it took to process the query.
`items`	Depending on the recommendation type: • `itemsForQuery` The document IDs and the weights of aggregated events that match the query. This type supports a `debug` option that adds a `debug` section to the output. • `queriesForItem` The queries and the weights of aggregated events that match the document ID.

Examples

Below are examples for each recommendation type.

itemsForQuery

Get the top items for the query 'ipod':

INPUT

```
curl -u USERNAME:PASSWORD
https://FUSION_HOST:8764/api/recommend/lucidworks102/itemsForQuery?
q=ipod&fq=count_d:4&debug=true
```

OUTPUT

```
{ "header" : { "queryParams" : { "aggrType" : "*", "rows" : 10, "collection" :
"lucidworks102", "aggrRows" : 100, "debug" : true, "q" : "ipod", "fq" : [
"count_d:4" ] }, "totalTime" : 5 }, "items" : [ { "weight" : 1.0726584E-11,
"docId" : "8771929" }, { "weight" : 3.865899E-12, "docId" : "9225439" }, {
"weight" : 9.230597E-12, "docId" : "3109302" } ], "debugInfo" : { "aggrTime" : 1,
"queryTime" : 4, "solrParams" : { "mm" : [ "50%" ], "pf" : [ "query_t^3",
"query_t~2^7", "query_t~0^1" ], "fl" : [ "id", "doc_id_s", "weight_d" ], "sort" :
[ "score desc,weight_d desc" ], "q" : [ "ipod" ], "qf" : [ "query_t" ],
"collection" : [ "lucidworks102_signals_aggr" ], "fq" : [ "aggr_type_s:*",
"count_d:4" ], "rows" : [ "100" ], "defType" : [ "edismax" ] } } }
```


queriesForItem

INPUT

```
curl -u USERNAME:PASSWORD
https://FUSION_HOST:8764/api/recommend/lucidworks102/queriesForItem?docId=9225439
```

OUTPUT


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


API specification

Signals API

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

This API requires a *Fusion AI license*.

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

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

You can manage aggregation jobs in the *Jobs manager* and *Scheduler* in the Admin UI, or with the *Spark jobs API*. We recommend using the Admin UI.

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

Examples

Send two signal events to record user clicks:

REQUEST

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

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

```
{ "responseHeader":{ "status":0, "QTime":1, "params":{ "indent":"true",
"q":"docId: 2125233", "wt":"json"} }, "response":
{"numFound":1198,"start":0,"docs":[ { "id": "7aee7b1f-5cde-4957-b73c-
c15881f559ec", "filters_s": "abcat0100000 $ abcat0101000 $ abcat0101001 $
cat00000", "query_orig_s": "Televisiones Panasonic 50 pulgadas", "params.user_s":
"000000df17cd56a5df4a94074e133c9d4739fae3", "docId": "2125233",
"params.query_time__s": "2011-09-01T23:43:59.75Z", "query_t": "televisiones
panasonic 50 pulgadas", "query_s": "televisiones panasonic 50 pulgadas",
"filters_orig_ss": [ "abcat0100000", "abcat0101000", "abcat0101001", "cat00000"
], "flag_s": "EVENT", "type_s": "click", "attr_params.filterQueries_": [
"cat00000", "abcat0100000", "abcat0101000", "abcat0101001" ], "timestamp_dt":
"2011-09-01T23:44:52.53Z", "_version_": 1478892846857584600 }, { "id": "6789a209-
f5b5-457e-9df6-8033b8f7f317", "filters_s": "abcat0100000 $ abcat0101000 $
abcat0101001 $ cat00000", "query_orig_s": "Sharp", "params.user_s":
"000001928162247ffaf63185cd8b2a244c78e7c6", "docId": "2009324",
"params.query_time__s": "2011-09-05T12:25:01.18Z", "query_t": "sharp", "query_s":
"sharp", "filters_orig_ss": [ "abcat0100000", "abcat0101000", "abcat0101001",
"cat00000" ], "flag_s": "EVENT", "type_s": "click", "attr_params.filterQueries_":
[ "cat00000", "abcat0100000", "abcat0101000", "abcat0101001" ], "timestamp_dt":
"2011-09-05T12:25:37.42Z", "_version_": 1478892846859681800 } ] } }
```

API specification

Index Pipeline Stages

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

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

- *Format Signals*
- *OpenNLP NER Extraction*
- *Machine Learning*

Format Signals Index Stage

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

Date/time parsing and formatting

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

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

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

Output document will carry the following two fields:

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

This stage does not define a list of allowed types.

Example Stage Specification

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

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

Configuration

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

Machine Learning Index Stage

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

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

- Use a Fusion AI job that trains a model, like *Logistic Regression* or *Random Forest*.
- Train a model using Spark's MLlib API *(https://spark.apache.org/docs/latest/mllib-guide.html)* outside of Fusion, and upload this model into Fusion's *blob store*. Complete details are available in *Machine Learning Models in Fusion*.

Configuration

OpenNLP NER Extraction Index Stage

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

This stage uses Apache OpenNLP *(http://opennlp.apache.org/)* project's Named Entity Recognition tool *(http://opennlp.apache.org/docs/#tools.namefind.recognition)* (the Name Finder tool). The OpenNLP documentation states:

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

Fusion 4.x.x contains a common set of NER models for English that include sentence, token, and part-of-speech models. These models are:

Model	Purpose
`nlp/models/en-sent.bin`	Sentence model to detect sentences
`nlp/models/en-token.bin`	Tokenizer model for tokenization of sentences
`nlp/models/en-ner-date.bin`	Date name finder model
`nlp/models/en-ner-location.bin`	Location name finder model
`nlp/models/en-ner-money.bin`	Money name finder model
`nlp/models/en-ner-organization.bin`	Organization name finder model
`nlp/models/en-ner-percentage.bin`	Percentage name finder model
`nlp/models/en-ner-person.bin`	Person name finder model
`nlp/models/en-ner-time.bin`	Time name finder model

See OpenNLP 1.5 series *(http://opennlp.sourceforge.net/models-1.5/)* for additional pre-trained OpenNLP models.

To use these models, upload to Fusion using the *Fusion Blob Store service*. Here is an example of how to upload the sentence model file using the `curl` command-line utility, where "admin" is the name of a user with admin privileges, and "pass" is the password:

```
curl -u USERNAME:PASSWORD -X PUT --data-binary @data/nlp/models/en-sent.bin -H
'Content-type: application/octet-stream' http://localhost:8764/api/blobs/en-
sent.bin
```

Example Specification

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

```
{ "type":"nlp-extractor", "id":"iqtr", "rules":[ { "source":[ "body_t" ],
"target":"organizations", "writeMode":"append",
"sentenceModelLocation":"nlp/models/en-sent.bin",
"tokenizerModelLocation":"nlp/models/en-token.bin", "entityTypes":[ {
"name":"organization", "definition":"nlp/models/en-ner-organization.bin" } ] }, {
"source":[ "body_t" ], "target":"persons", "writeMode":"append",
"sentenceModelLocation":"nlp/models/en-sent.bin",
"tokenizerModelLocation":"nlp/models/en-token.bin", "entityTypes":[ {
"name":"person", "definition":"nlp/models/en-ner-person.bin" } ] } ],
"type":"nlp-extractor", "skip":false, "label":"Extract Entities",
"licensed":true, "secretSourceStageId":"iqtr" }
```


Configuration

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

Insights Reports and Signal Data Requirements

Although there are many fields that are relevant to *signals*, this topic describes which fields are **required** for each type of signal. Without these fields, reports and dashboards within *App Insights* may not function as expected.

Signal Types

There are five types of signals:

- **Annotation.** Annotation signals are generated when a user bookmarks, likes, or comments on a document. Annotation signals are likewise generated when the user removes a bookmark, like, or comment.

  > Annotation signals are generated by *App Studio*. If you are not using App Studio, this type of signal is not relevant to your search application.

- **Click.** Click signals are generated when a user clicks on a page element that is being monitored by the search app. Click signals are sent from the search app to Fusion.
- **Login.** Login signals record information about specific users when they log in to an application. This includes a time stamp and various session details.
- **Request.** 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.
- **Response.** Response signals are automatically generated by a query pipeline when the signals feature is enabled for a collection.

  > Because response signals and their fields are automatically generated, this topic does not cover what response signal fields are required.

Required Fields

Signals

The following table describes which fields are required for annotation, click, login, and request signals.

> Requests (or queries) can also require additional, available user data for the search.

Field	Type	Description	Example	Required
`id`	string	Unique ID for the signal.	`b0ee5307-6223-4150-ac5a-d0d8113aa480`	✅ Annotation ✅ Click ✅ Login ✅ Request
`doc_id`	string	Product ID or Item ID of the clicked result.	`NMDDV`	✘ Annotation ✅ Click ✘ Login ✘ Request
`date`	timestamp	Timestamp of when the signal was generated. This timestamp follows Unix epoch time formatting.	`1518717749409`	✅ Annotation ✅ Click ✅ Login ✅ Request
`user_id`	string	Unique ID for the user that generated the signal.	`admin`	✅ Annotation ✅ Click ✅ Login ✅ Request
`session`	string	Unique ID for the user's browser session.	`ef4e00cd-91bb-45b4-be80-e81f9f9c5b27`	✅ Annotation ✅ Click ✅ Login ✅ Request
`host`	string	Host name of the server which is hosting the app that is generating the signal.	`x.y.z`	✅ Annotation

Field	Type	Description	Example	Required
				✅ Click ✅ Login ✅ Request
`ip_address`	string	IP address of the user that generated the signal.	`80.6.99.35`	✅ Annotation ✅ Click ✅ Login ✅ Request
`app_id`	string	Name of the application that is generating the signal.	`myApp`	✅ Annotation ✅ Click ✅ Login ✅ Request
`annotation_type`	string	Type of the annotation signal, either "comment", "bookmark", or "like". Required fields for "comment": • `comment`. The comment itself. • `target`. The target that the comment refers to. Required fields for "bookmark": • `title`. The bookmark title. • `url`. The bookmark url. Required fields for "like": • `` - • `` -	`bookmark`	✅ Annotation ✘ Click ✘ Login ✘ Request
`query`	string	Terms of the query.	`ipad`	✘ Annotation ✅ Click ✘ Login ✅ Request
`fusion_query_id`	string	Unique ID for the `query` that is automatically generated from the Fusion response signal.	`ABkaEA11`	✘ Annotation

Field	Type	Description	Example	Required
				✅ Click ✘ Login ✘ Request
`filter`	array of string	List of filters associated with the query, which in turn is associated with signal.	`["type/tablet","category/electronics"]`	✘ Annotation ✅ Click ✘ Login ✅ Request
`ctype`	string	Type of click.	`result`	✘ Annotation ✅ Click ✘ Login ✘ Request
`res_pos`	number	Position of the clicked result within the list of results.	`3`	✘ Annotation ✅ Click ✘ Login ✘ Request
`res_offset`	number	Result page.	`2`	✘ Annotation ✅ Click ✘ Login ✘ Request
`url`	string	URL of the page that the signal originated from.	`http://localhost:8080/products/search`	✘ Annotation ✘ Click ✘ Login ✅ Request

Field	Type	Description	Example	Required
`path`	string	URL path of the page that the signal originated from.	`/search`	✘ Annotation ✘ Click ✘ Login ✅ Request
`page_title`	string	Title of the page that the signal originated from.	`Search Page`	✘ Annotation ✘ Click ✘ Login ✅ Request

Insights Reports

The table below describes which fields are required to generate the various reports within *App Insights*:

Report	Signal Types Used	Required Field	Type	Description	Example
Search queries	✘ Annotation ✅ Click ✘ Login ✅ Request ✅ Response	`query`	string	Terms of the query.	`physics`
Facets used	✘ Annotation ✘ Click ✘ Login ✅ Request ✘ Response	`filter_field`	array of string	List of filters associated with the query, which in turn is associated with signal.	`["categories_s"]`
Facet filters applied	✘ Annotation ✘ Click ✘ Login ✅ Request ✘ Response	`filter`	array of string	List of **applied** filters associated with the query, which in turn is associated with signal.	`["categories_s/math.OA"]`
Application servers	✘ Annotation ✅ Click ✘ Login ✅ Request ✘ Response	`host`	string	Host name of the server that is hosting the app, which in turn is generating the signal.	`lucidworks`

Report	Signal Types Used	Required Field	Type	Description	Example
Response times	✘ Annotation ✘ Click ✘ Login ✘ Request ✅ Response	`time`	number	Response time (in milliseconds).	`17`
URLs clicked	✘ Annotation ✅ Click ✘ Login ✘ Request ✘ Response	`url`	string	URL of the page that was selected from the search results. This typically results from a click signal.	`http://arxiv.org/pdf/astro-ph/0611688v1`
Search pages	✘ Annotation ✘ Click ✘ Login ✅ Request ✘ Response	`page_title`	string	Title of the page that the signal originated from.	`Search Preview`
Search platforms	✘ Annotation ✅ Click ✘ Login ✅ Request ✅ Response	`platform`	string	Name of the search platform.	`fusion`

Report	Signal Types Used	Required Field	Type	Description	Example
Visitor countries	✘ Annotation ✅ Click ✘ Login ✅ Request ✘ Response	`geo_country_name`	string	Originating country of the user.	`United States`
Visitor cities	✘ Annotation ✅ Click ✘ Login ✅ Request ✘ Response	`geo_city_name`	string	Originating city of the user.	`San Francisco`
Browsers	✘ Annotation ✅ Click ✘ Login ✅ Request ✘ Response	`browser_name`	string	Name of the browser that generated the signal.	`Firefox`
Operating System	✘ Annotation ✅ Click ✘ Login ✅ Request ✘ Response	`os_name`	string	Name of the operating system used by the user.	`Ubuntu`

Report	Signal Types Used	Required Field	Type	Description	Example
Device types	✘ Annotation ✅ Click ✘ Login ✅ Request ✘ Response	`os_device`	string	Type of the device used by the user.	`Computer`
Websites people are coming from	✘ Annotation ✅ Click ✘ Login ✘ Request ✘ Response	`referrer_domain`	string	URL of the page that the signal originated from.	`http://www.google.com`
Users	✘ Annotation ✅ Click ✘ Login ✅ Request ✘ Response	`user_id`	string	Unique ID for the user that generated the signal.	`admin`
User domains	N/A	`domain`	string	Domain of the user that generated the signal.	`lucidworks.com`

Report	Signal Types Used	Required Field	Type	Description	Example
Types of event	✅ Annotation ✅ Click ✅ Login ✅ Request ✅ Response	`type`	string	Type of the signal.	`click`
Head Tail analysis	✘ Annotation			This report is populated by running a head tail job in Fusion. With a collection called `foo`, for example, run the `foo_head_tail` job.	N/A
	✅ Click	type	string		N/A
	✅ Click	query	string		N/A
	✘ Login				N/A
	✘ Request				N/A
	✅ Response	count_i	number		N/A

Fusion jobs

The following table specifies required fields for specific Fusion jobs:

Fusion job	Field	Required
click_signal_aggr	`user_id`	no
	`query`	yes
	`doc_id`	yes
	`count_i`	yes
	`fusion_query_id`	depends
	`type`	yes
	`session_id`	optional
	`id`	N/A
	`res_offset`	optional
	`res_pos`	optional
	`filters`	optional
	`type = response`	depends
	`dependency`	N/A
phrase_extraction	`user_id`	no
	`query`	yes
	`doc_id`	no
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	N/A
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A

Fusion job	Field	Required
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
spell_correction	`user_id`	no
	`query`	yes
	`doc_id`	no
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	N/A
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
synonym_detection	`user_id`	no
	`query`	yes
	`doc_id`	yes
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	N/A

Fusion job	Field	Required
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	phrase, misspelling
head_tail	`user_id`	no
	`query`	yes
	`doc_id`	yes
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	N/A
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
_user_item_prefs_agg	`user_id`	yes
	`query`	no
	`doc_id`	yes
	`count_i`	yes

Fusion job	Field	Required
	`fusion_query_id`	no
	`type`	yes
	`session_id`	optional
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
user_query_history_aggr	`user_id`	yes
	`query`	yes
	`doc_id`	no
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	optional
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
als_recommender	`user_id`	yes

Fusion job	Field	Required
	`query`	yes
	`doc_id`	yes
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	optional
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
bpr_recommender	`user_id`	yes
	`query`	yes
	`doc_id`	yes
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	optional
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A

Fusion job	Field	Required
	`type = response`	N/A
	`dependency`	N/A
query_query_collaborative	`user_id`	yes
	`query`	yes
	`doc_id`	no
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	optional
	`id`	N/A
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
query_query_session_based	`user_id`	no
	`query`	yes
	`doc_id`	yes
	`count_i`	yes
	`fusion_query_id`	no
	`type`	yes
	`session_id`	yes
	`id`	N/A

Fusion job	Field	Required
	`res_offset`	N/A
	`res_pos`	N/A
	`filters`	N/A
	`type = response`	N/A
	`dependency`	N/A
experiments	`user_id`	yes
	`query`	yes
	`doc_id`	yes
	`count_i`	yes
	`fusion_query_id`	yes
	`type`	yes
	`session_id`	yes
	`id`	yes
	`res_offset`	yes
	`res_pos`	yes
	`filters`	yes
	`type = response`	yes
	`dependency`	N/A

Licensing

Fusion Server requires a valid license. Depending on the details of your contract, your license may also enable optional *connectors* or *Fusion AI*.

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.

Visit our company page to try Fusion *(https://lucidworks.com/try)*.

Release Notes

Release notes provide detailed descriptions of the relevant changes included with each Fusion Server release, including new features, improvements, bug fixes, and more.

Upgrades

Keeping your Fusion deployment upgraded to the latest version enables you to take advantage of the latest features, improvements, bug fixes, and addressed security issues.

Process

To upgrade to the latest version of Fusion, see:

- **From Fusion Server 4.2.x**
  - *To Fusion Server 4.2.y*
  - *To latest SP*
- **From Fusion Server 4.1.x**
  - *To Fusion Server 4.2.y*
  - *To Fusion Server 4.1.y*
- **From Fusion Server 4.0.x**
  - *To Fusion Server 4.2.y*
  - *To Fusion Server 4.1.y*
  - *To Fusion Server 4.0.y*
- **From Fusion 3.1.x**
  - *To Fusion Server 4.2.y*
  - *To Fusion Server 4.1.y*
  - *To Fusion Server 4.0.y*

Deprecations and Removals

Every Fusion release adds or improves Fusion features and functionality. Occasionally, we also remove features and functionality. Before doing so, we "deprecate" them. Deprecated features are no longer officially supported by Lucidworks and will not be updated or maintained. In many cases, new features are introduced which replace and/or improve the functionality of the deprecated feature.

See *Deprecations and Removals* for a list of active deprecations and removals.

How long do I have before the feature is removed?
We give a minimum of one minor release cycle before removing a deprecated feature. For example:

- A feature is deprecated in Fusion 4.1.0, so it will not be removed before Fusion 4.2.0
- A feature is deprecated in Fusion 4.1.1, so it will not be removed before Fusion 4.2.0

	Deprecations are typically made in major and minor releases only. Maintenance release deprecations are less common.

Fusion AI 4.2.6 Release Notes

Release date: 31 March 2020

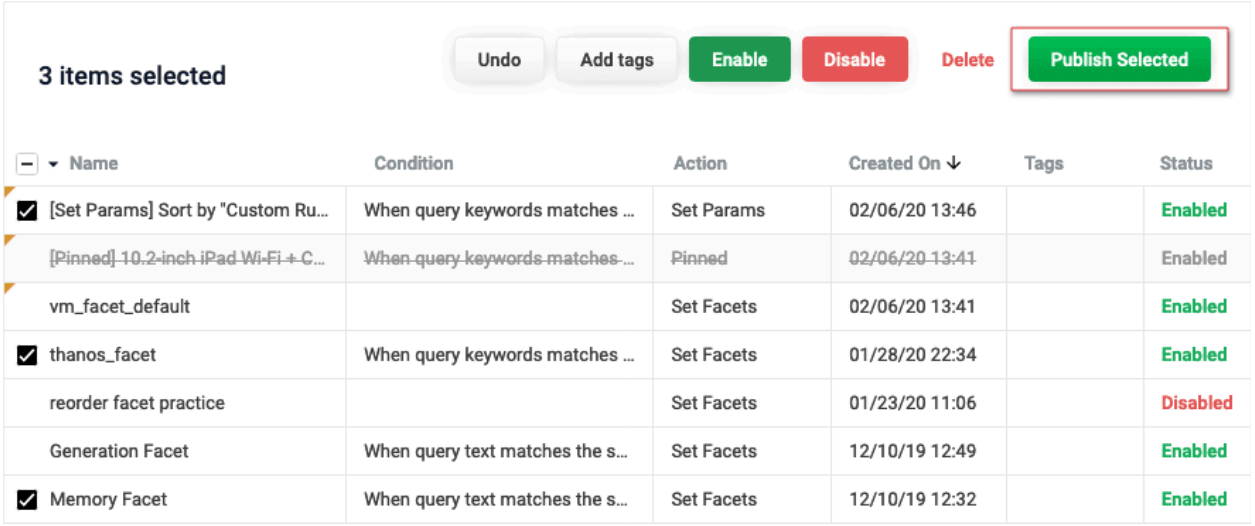
Component versions:

Solr 7.7.2	ZooKeeper 3.4.13	Spark 2.3.2	Jetty 9.4.19.v20190610	Ignite 2.6.0

More information about support dates can be found at *Lucidworks Fusion Product Lifecycle*.

New features

- Support is added for individually selecting multiple rules to publish in the Fusion UI:



Improvements

- Fields in the Business Rules edit modal now flex to show more information when the panel is expanded.

Bug fixes

- Fixed a bug that caused *head/tail analysis jobs* to overwrite published rules.
- Fixed a bug in the Rules Editor that prevented more than 20 rules from being deleted at once.
- Fixed a bug in the Rules Editor that sometimes caused UI instability when adding a synonym to an existing rule.

Fusion AI 4.2.5 Release Notes

Release date: 22 October 2019

Component versions:

Solr 7.7.2	ZooKeeper 3.4.13	Spark 2.3.2	Jetty 9.4.19.v20190610	Ignite 2.6.0

More information about support dates can be found at *Lucidworks Fusion Product Lifecycle*.

New features

Lucidworks Fusion Smart Answers

Fusion 4.2.5 introduces *Lucidworks Fusion Smart Answers*, an AI-driven question answering system which combines the power of Solr and neural dense vectors to recommend solutions to users.

Smart Answers incorporates semantic and contextual information into query understanding to generate recommendations from frequently asked questions (FAQ) records. If FAQ data is not available, Smart Answers can adopt a "cold start" methodology, which uses our word vector (Word2vec) training module to learn about the vocabulary in the search results.

Example use cases:

Call Center or IT Support
Fusion Smart Answers can be implemented on a help or contact us page, or embedded in a virtual assistant, to reduce the workload for call centers or IT support. Smart Answers can also increase efficiency for the customer support team by helping them find answers to problems that have already been solved.

E-commerce
Smart Answers can help shoppers answer their questions about products, search user manuals for solutions to their problems, or point them to relevant discussions other customers have had about the product.

Improved Search for Long Queries
With the cold start solution, Smart Answers uses various word embedding methods to capture semantic and contextual information for long queries or natural language processing.

New Pipeline Stages

To support Lucidworks Fusion Smart Answers, Fusion 4.2.5 introduces several new pipeline stages. These stages are documented in detail in *Smart Answers*.

See the *Fusion Server 4.2.5 release notes* for other changes.

Fusion AI 4.2.2 Release Notes

Release date: 17 May 2019

Component versions:

Solr 7.5	ZooKeeper 3.4.13	Spark 2.3.2	Jetty 9.4.12.v20180830	Ignite 2.6.0

More information about support dates can be found at *Lucidworks Fusion Product Lifecycle*.

New features

No new features were introduced in Fusion AI 4.2.2.

See the *Fusion Server 4.2.2 release notes* for other changes.

Improvements

- *Synonym pairs* are now grouped and treated as one term when for better compatibility with *Solr's "minimum match" (`mm`) query parameter*.
- *Query rewriting* accuracy is improved in this release.
- *Parallel Bulk Loader (PBL) jobs* and *Script jobs* can now be configured to set environment variables. In the Fusion UI, click **Advanced** to see this option.
- Recommender jobs are no longer scheduled by default; they must now be scheduled manually or run on demand. (Previously, default recommender jobs were automatically scheduled to run after the `_user_item_preferences_aggregation` job.)

Other changes

- The Spark driver now writes `error.log` and `output.log` files to `var/log/api` instead of `var/api/work`.
- *Query Rewriting UI* bug fixes:
  - Fixed an issue in Fusion Server 4.2.0 and 4.2.1 that caused the UI's search feature to return a `Problem with underlying storage` when a query was `:` followed by an additional query term.
  - Fixed a bug that caused the Query Rewriting UI to display an error window that could not be closed.
- Signals from the *Simulator* are no longer sent to the signals collection.
- Fixed an issue in the *Text Tagger query stage* which prevented query rewrites from working after **Save Tags in Context** had been enabled.
- *Business rules* defined with tags are no longer triggered by queries that contain no `tags` parameters.

Fusion AI 4.2.1 Release Notes

Release date: 5 April 2019

Component versions:

Solr 7.5	ZooKeeper 3.4.13	Spark 2.3.2	Jetty 9.4.12.v20180830	Ignite 2.6.0

More information about support dates can be found at *Lucidworks Fusion Product Lifecycle*.

New features

No new features were introduced in Fusion AI 4.2.1.

See the *Fusion Server 4.2.1 release notes* for other changes.

Improvements

- In the `_user_query_history_aggregation` SQL aggregation job, the default value of the `signalTypeWeights` SQL parameter has been changed from `click:1.0,add-to-cart:10.0,purchase:25.0` to `request:1.0,click:5.0,cart:10.0,purchase:25.0` to add request signals so that all signal types are included in these aggregations.
- Query Rewriting UI improvements:
  - Facet groups can now be collapsed and expanded for easier viewing.
  - Query rewrites are now faceted by tag.
  - The **Apply Rules** query pipeline stage has a new parameter, **Partially Matched Filter Queries Will Trigger the Rule**/`matchPartialFilterQueries`, which allows a rule to fire when it is configured with multiple **Field Value** conditions and only some of those conditions are matched.
  - The **Misspelling Detection** page now has a **Published** column.
- Manually-created query rewrites are now automatically assigned a review value of "Approved" and the **Review** field is no longer editable in the query rewrite creation interface.
- Fixed an issue which broken rules with banner actions when the **Banner Zone** field value was an integer.
- Better checks for duplicate rewrites.
- Uni-directional synonyms are now working correctly in the `query_rewrite_staging` collection and the Simulator.
- A manually-created head/tail query improvement now has its `action` field correctly populated.
  - A variety of minor UI issues were fixed.

Fusion AI 4.2.0 Release Notes

Release date: 28 February 2019

Component versions:

Solr 7.5	ZooKeeper 3.4.13	Spark 2.3.1	Jetty 9.4.11.v20180605	Ignite 2.3.0

More information about support dates can be found at *Lucidworks Fusion Product Lifecycle*.

New features

- **Smarter relevancy with query rewriting**
  Query rewriting turns ineffective queries into more relevant results by automatically correcting misspelled terms, expanding queries to include synonyms, boosting known phrases, and applying your own business rules.
  4.2.0 ties together existing *query rewriting* features and adds new features to make query rewriting easier to configure:
  - Built-in support for *business rules*
    Business rules provide versatile, fine-grained control over query/response rewriting. You can create, edit, deploy, and organize rules using this new Query Rewriting UI. New *rules-based query stages* are automatically updated based on the changes you make in the rules editor.
  - New *Query Rewriting UI*
    Now you can view, create, modify, and publish multiple types of query rewritings by navigating to **Relevance > Query Rewriting** in the Fusion UI. A new *Simulator* lets you preview search results to test all enabled query rewriting strategies, including unpublished query rewrites.
  - New *Synonym detection* job
    The new *Synonym and Similar Queries Detection job* takes input from signals data, the *Token and Phrase Spell Correction job*, the *Phrase Extraction job*, and keyword lists in the *blob store* to automatically detect and store synonyms for use in *query rewriting*.
  - New `query_rewrite_staging` and `query_rewrite` collections are dedicated to AI-generated content that can be used to rewrite queries:
    - *Business Rules*
    - *Head/Tail analysis job* results
    - *Synonym and Similar Queries Detection job* results
    - *Token and Phrase Spell Correction job* results
    - *Phrase Extraction job* results
  - New query pipeline stages apply the rules and results from the `query_rewrite` collection:
    - *Text Tagger*
    - *Apply Rules*
  - A new `rules_simulator` query profile allows you to experiment with rules and other query rewrites in the `_query_rewrite_staging` collection using the *Simulator* before deploying them to the `_query_rewrite` collection.
- **Refine the final search results with response rewriting**
  Similar to query rewriting, response rewriting can apply machine learning, business rules, or other criteria to Solr's response, refining the final set of search results before Fusion sends them to the search application. *Response rewriting* can be performed using *rules* and a set of new query pipeline stages that fall into two categories:
  - **Distribute clicks more evenly among the top N results**
    These stages act on the whole set of documents in the response:
    - *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.
  - **Manipulate specific search result items**
    These stages act on individual documents in 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.
- **More Natural Language Processing (NLP) power in pipelines**
  This release includes a new NLP Annotator *index stage* and *query stage* that leverage the popular John Snow NLP library for Spark *(https://nlp.johnsnowlabs.com)*, introducing these new NLP features in addition to the existing Named Entity Extraction (NER) functionality:
    - Sentence detection
    - Part-of-Speech (POS) tagging

  See *Natural Language Processing* for an overview of Fusion AI's NLP capabilities.

See the *Fusion Server 4.2.0 release notes* and the *App Studio 4.2.0 release notes* for other changes.

Improvements

- Experiments can now be configured as *multi-armed bandits*, by selecting the new **Automatically Adjust Weights Between Variants** option when *setting up the experiment*.
- A Part-of-Speech (POS) model is now available in the blob store by default, as `en-pos-maxent.bin`, for use by the *Phrase Extraction job*.
- When a new app is created, these jobs are now automatically created and scheduled to run daily, beginning 15 minutes after app creation, in the following order:
  1. *Token and Phrase Spell Correction*
  2. *Phrase Extraction*
     This job runs if the Token and Phrase Spell Correction job succeeds.
  3. *Synonym and Similar Queries Detection*
     This job runs if the Phrase Extraction job succeeds.

Known issues

- In Underperforming Query Rewriting and Misspelling Detection, new or modified query rewrites cannot be saved when any of their values include trailing or leading whitespace. Remove any trailing or leading whitespace to save the query rewrite.
- The Query Rewriting UI's search feature joins multiple search terms using OR instead of AND. For example, searching for a rule called "Test 1" returns "Test 1", "Test 2", and "1 Rule".
- The Query Rewriting UI's search feature may return a `Problem with underlying storage` if your query is `*:*` followed by an additional query term. This is an invalid query; use only `*:*` to search for all query rewrites.
- In Underperforming Query Rewriting, job-generated query improvements do not preserve the original query's uppercase and lowercase characters. For example, an underperforming query containing "brandX" may be rewritten to contain "brandx". You may need to manually modify the query improvements to preserve the correct cases.
- After selecting multiple business rules where some rules have tags, adding more tags to the selected rules deletes their existing tags. To work around this, add tags to individual rules instead of adding them in bulk.

See also the *Fusion Server 4.2.0 known issues*.

Fusion AI 4.1.2 Release Notes

Release date: 17 December 2018

See also the *Fusion Server 4.1.2 release notes* and the *App Studio 4.1.2 release notes*.

No changes to the AI feature set were introduced in this release.

Fusion AI 4.1.1 Release Notes

Release date: 7 November 2018

See also the *Fusion Server 4.1.1 release notes*.

Improvements

- *Phrase Extraction job* improvements
  - The job now trims low-confidence phrases based on likelihood.
  - The job adds a `review` tag on the result to facilitate the review process based on beginning and ending POS and likelihood.
  - The output now connects phrase tokens with an underscore (`_`) to make a single token per phrase so that complete phrases can be used as facets.
  - New metadata fields:
    - `phrases_count` shows how many times the phrases appear in the documents.
    - `word_num` shows how many words are in the phrase.
  - This release includes optimizations to the default Lucene analyzer configuration.
  - Output names are updated for clarity, such as `llr` to `likelihood` and `ngram` to `phrases`.
- The *Head/Tail Analysis job* now processes large data sets twice as fast.
- The *Ranking Metrics job* now correctly accepts values for the `queryPipelines` property in the Fusion UI.
- The *Solr Query pipeline stage* has two new properties to configure *signals*:
  - `responseSignalsEnabled`
    Disable this option to prevent the stage from generating a *response signal* containing metadata about the response from Solr.
    Response signals are used by *App Insights* and *experiments*.
    In auto-complete pipelines, disable this option to avoid generating a response signal for each keystroke.
  - `excludeResponseSignalMatchRules`
    If `responseSignalsEnabled` is "true", then you can prevent generating a response signal based on specific parameters in the query, such as to enable response signals in general but to disable them for auto-complete queries.

Other changes

- The Spark driver now cleans up `$FUSION_HOME/var/spark/Spark-workDir-*` directories and shaded jars correctly on Windows to prevent excessive disk consumption.
- For installations that were upgraded from 3.1.x to 4.1.0, upgrading to 4.1.1 resolves an issue that prevented successful signal aggregations with the Parameterized SQL Aggregation job.

Fusion AI 4.1.0 Release Notes

Release date: 17 July 2018

See also the *Fusion Server 4.1.0 release notes*.

New features

- *SparkSQL datasource loader job*

Improvements

- **SQL Engine Improvements**
  - Added virtual table join support to the Solr/SQL pushdown strategy (SQL 360 Collections)
  - Added Kerberos authentication and SSO authentication support
  - Fixed scaling bugs with Tableau
  - Added robust Tableau support (better than MySQL and PostGres on cross table joins)
- **Experiment management improvements**
  - New "default response time in ms" default metric
- The *Machine Learning index stage* and the *Machine Learning query stage* now support multiple values when specifying field names, in this format: `field1:weight,field2:weight,field3:weight`
- **Job improvements**
  - Custom stopword lists can now be uploaded to the blob store and utilized by the Head/Tail Analysis, Document Clustering (formerly the Bisecting KMeans Clustering job), Cluster Labeling, Phrase Extraction (formerly the Statistically Interesting Phrases job), and Token and Phrase Spell Correction jobs.
  - The Levenshtein Spell Checking job has been removed. Use the *Token and Phrase Spell Correction job* instead.
  - Add schema groups and categories by default on all Spark Jobs
  - Some jobs have been renamed:

4.0 Job	4.1 Job
Co-occurrence Similarity	*Legacy Item Similarity*
Item Similarity Recommender	*Legacy Item Recommender*
Matrix Decomposition Based Query - Query Similarity	*Query to Query Similarity Computation*
Statistically Interesting Phrases	*Phrase Extraction*

Other changes

- When you disable signals for a collection, the associated jobs are now deleted.
- The Boost with Signals query pipeline stage can now be configured to point to an alternative collection for aggregated signals.

Fusion AI 4.0.2 Release Notes

Release date: 22 May 2018

See also the *Fusion Server 4.0.2 release notes*, which include known issues in this release.

New features

None.

Improvements

- The `weightExpression` parameter for the *Boost with Signals* query pipeline stage now supports math expressions in JEXL syntax *(http://commons.apache.org/proper/commons-jexl/reference/syntax.html)*.
  Examples: `weight_d * 1000` `weight_d * score` ** `weight_d + math:max(score, 1)`

Other changes

- The *SQL-Based Experiment Metric* job is no longer available in the Jobs panel of the UI.
  The *Custom SQL experiment metric* performs an equivalent function. You can configure this metric when you *Set up an experiment*.

Known issues

None.

Fusion AI 4.0.1 Release Notes

Release date: 28 February 2018

See also the *Fusion Server 4.0.1 release notes*.

New features

None.

Improvements

- **Session rollup job improvements**
  Session rollup jobs now include the `app_id`, `host`, `ip_address`, and `user_token` fields, for use in App Insights.

Other changes

None.

Fusion AI 4.0.0 Release Notes

Release date: 21 February 2018

Fusion AI is a license-controlled subset of Fusion features. Starting with Fusion 4.0, a Fusion Server license enables the basic Fusion feature set, while a Fusion AI license enables these additional features:

- Signals
- App Insights (new in 4.0)
- Smart jobs
  - ALS Recommender Jobs
  - Bisecting KMeans Clustering Jobs
  - Cluster Labeling Jobs
  - Co-Occurrence Similarity Jobs
  - Collection Analysis Jobs
  - Document Clustering Jobs
  - Ground Truth Jobs
  - Head/Tail Analysis Jobs (new in 4.0)
  - Item Similarity Recommender Jobs
  - Levenshtein Spell Checking Jobs
  - Logistic Regression Classifier Training Jobs
  - Matrix Decomposition-Based Query-Query Similarity Jobs
  - Outlier Detection Jobs
  - Random Forest Classifier Training Jobs
  - Ranking Metrics Jobs
  - SQL-Based Experiment Metric Jobs
  - Statistically Interesting Phrases Jobs
  - Token and Phrase Spell Correction Jobs (new in 4.0)
- REST APIs
  - Experiments API
  - Recommendations API
  - Signals Aggregator API
  - Signals API
- Index pipeline stages
  - Machine Learning index stage
  - OpenNLP NER Extractor index stage
  - Signal Formatter index stage
  - Update Experiment index stage
- Query pipeline stages
  - Advanced Boosting query stage
  - Analytics Catalog query stage
  - Experiment query stage
  - Machine Learning query stage
  - More Like This query stage
  - Recommend Items for Item query stage
  - Recommend Items for User query stage
  - Recommendation Boosting query stage

See also the *Fusion Server 4.0.0 release notes*.

New features

- **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.

  > Pre-4.0 signals data will not produce useful visualization in App Insights. New signals generated with Fusion 4.0 will produce the best results.

- **New jobs**
  - *Head/Tail Analysis*
    Perform head/tail analysis of queries from collections of raw or aggregated signals, to identify underperforming queries and the reasons. This information is valuable for configuring better synonyms, auto-suggest, recommendations, and so on, in order to improve conversion rates.

- *Token and Phrase Spell Correction*
  Extract tail tokens (one word) and phrases (two words) and find similarly-spelled head tokens and phrases. If several matching heads are found for each tail, the job can compare and pick the best correction using multiple configurable criteria.

Improvements

- **Experiment management features**
  The Fusion UI now includes interfaces for running and managing *experiments*.
- **SQL for aggregations**
  By default, *aggregation jobs are now defined using Spark SQL*. To use the old aggregation configuration scheme in the Fusion UI, click **Advanced** in the Aggregation job configuration panel.

Other changes

- **Bisecting Kmeans Clustering Jobs are now called *Document Clustering Jobs***
- **The data structure for signals has changed**
  See *Signals* for complete details about the new structure.
- **The Experiments API is no longer experimental**
  The *API* and other *experiments* features are now fully functional.
- **Aggregator API is deprecated**
  The `/aggregator` endpoints are deprecated in this release. Aggregations are now managed through the `/spark` endpoints.