

Query API

Query overview

IBM Watson® Discovery offers powerful content search capabilities through search queries.

To retrieve data from Discovery after it is ingested, indexed, and enriched, submit a query.

As data is added to Discovery, a representation of each file is stored in the index as a JSON-formatted document. Enrichments that are applied to your collections identify meaningful information in the data and store it in new fields in these documents. To search your data, submit a query to return the most relevant documents and extract the information you're looking for.

Query types

Discovery accepts one of the following supported query types:

Query

Finds documents with values of interest in specific fields in your documents. Queries of this type use Discovery Query Language syntax to define the search criteria.

Parameter name: `query`

Natural Language Query (NLQ)

Finds answers to queries that are written in natural language. NLQ requests accept a text string value.

Parameter name: `natural_language_query`

Along with the query that you specify by using one of the supported query types, you can include one or both of the following parameters. The values for these parameters are also specified by using the Discovery Query Language (DQL) syntax:

- `filter`
- `aggregation`

For more information about the Discovery Query Language, see [DQL overview](#).

Queries that are submitted from the product user interface are natural language queries. A few other supported parameters are specified and given default values based on the project type in use. For more information, see *Default query settings*.



Note: Discovery does not log query request data. You cannot opt in to request logging.

Choosing the right query type

The following table summarizes the capabilities that are supported for each query type. Use it to help you determine which type of

query to submit.

Goal	Natural Language Query (NLQ)	Discovery Query Language (DQL)
Return passages from documents	✓	✓
Highlight terms in responses (unless passages per document is enabled)	✓	✓
Define custom stop words or query expansions	✓	✓
Search specific document fields or enrichments		✓
Use operators, such as boolean clauses in the search		✓
Enable spelling correction	✓	
Add curations to return hardcoded answers to certain questions	✓	
Use relevancy training	✓	
Enable answer finding to return a succinct answer from a passage	✓	
Use table retrieval	✓	

Query types comparison

Query analysis

When you submit a query, the query text string is analyzed. During query analysis, the root (or lemma) of each key term in the query is identified. Any stop words that occur in the original query string are removed and synonym expansions that are defined for any terms that occur in the original query string are added. This enhanced version of the query is what gets submitted to Discovery.

The same analysis is performed on all queries, whether they are submitted as natural language queries or by using Discovery Query Language syntax.

Query flow

The following diagram shows a conceptual illustration of how a search request is handled by Discovery.

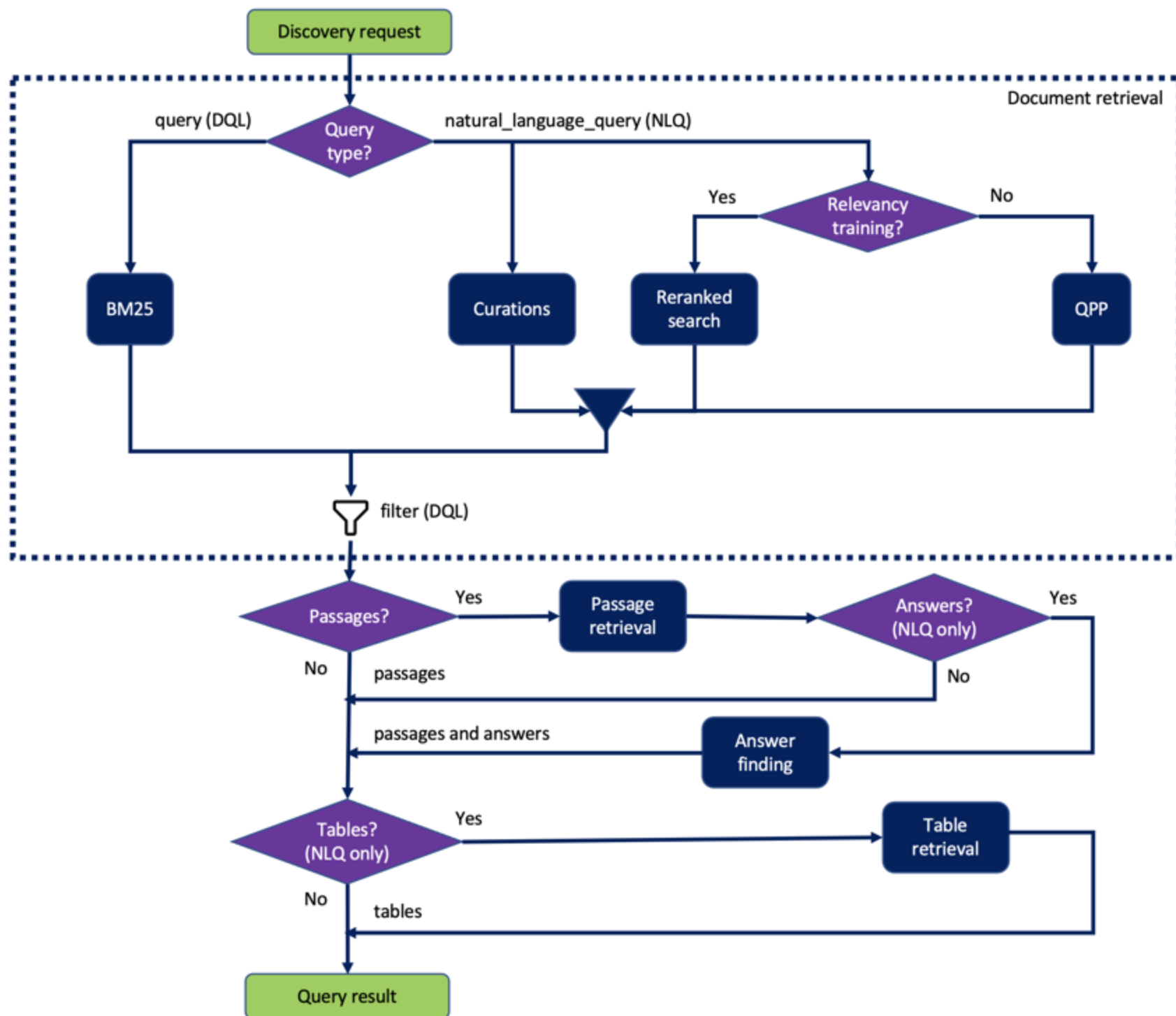


Figure 1. Flow chart that shows the processes that are used for Natural Language Queries versus Discovery Query Language queries

The following processes are shown in the flow diagram:

BM25

Uses Best Match 25 (a probabilistic information retrieval algorithm) to compute a relevance score for each document returned by search. The diagram shows that BM25 is applied to document results from the query requests, but it is not limited to query requests. It also is used along with other techniques as part of the relevancy training ranker process that is applied to natural language query results.

Curations

If the natural language query matches a predefined curation query, then certain documents and possibly a hardcoded snippet are returned. There is no query parameter to enable a curation. For curations to be used, you must define them programmatically ([Create curation method](#)). The output of any curations is merged with the output of the Relevancy training ranker or QPP results.

Relevancy training

A model that you can optionally define and apply to a project to score documents for relevance. There is no query parameter to enable relevancy training. For relevancy training to be used, you must successfully train the project either programmatically ([Create training query method](#)) or by using the product user interface.

QPP

A Query Performance Prediction algorithm that, given a query and a list of top results, produces a score that determines how relevant a document is. Used only if no Relevancy training ranker is available.

filter

The **filter** parameter can be passed along with **query** and **natural_language_query** requests to remove documents that don't meet certain criteria from the result set. The filter is shown as the last step within the document retrieval phase. However, it is used at different times in the flow. Its placement in the diagram is chosen to emphasize the fact that any documents that don't match the filter definition are excluded from the result set. The exclusion applies even to documents that might be specified in a curation.

Passage retrieval

Returns passages from documents **when** the `passages.enabled=true` parameter is included with a natural language query request.

Answer finding

When the `passages.find_answers=true` parameter is included with a natural language query request, returns succinct answers from passages along with the passages that are extracted from documents. If answer finding is enabled, then the final confidence score for each search result is a combination of the confidence scores from answer finding, passage retrieval, and QPP or Reranked search, whichever method is used.

Table retrieval

Returns information from tables in documents when the `table_results.enabled=true` parameter is included with a natural language query request.

Query limits

A query is any operation that submits a **POST** request to the `/query` endpoint of the API. Such operations include queries that are submitted by using the API. It does not include queries that are submitted from the search bar on the *Improve and customize* page of the product user interface.

A query is counted only if the request is successful, meaning it returns a response (with message code 200).

The number of search queries that you can submit per month per service instance depends on your Discovery plan type.

Plan	Queries per month per service instance
Cloud Pak for Data	
Premium	
Enterprise	
Plus (includes Trial)	
Number of queries per month	



Note: For Enterprise plans only, your bill labels requests that are generated from both query searches and analyze API calls as "Queries". For more information about Analyze API calls, see *Analyze API limits*.

The number of queries that can be processed per second per service instance depends on your Discovery plan type.

Plan	Concurrent queries per service instance
Cloud Pak for Data	
Premium	
Enterprise	
Plus (includes Trial)	
Number of concurrent queries	

For information about pricing, see *Discovery pricing plans*.

Estimating query usage

How to estimate the number of queries your application will use per month depends on your use case.

- For use cases that focus more on data enrichment and analysis or where the output from the document processing is not heavily searched, you can estimate query numbers based on the total number of documents.
- For use cases where many users interact with the application that uses Discovery, you can estimate by calculating the number of searches per user times the number of expected users. For example, 50% of the questions that are submitted by users to a virtual

assistant are likely to be answered by Discovery. With 100,000 users per month and an average of 3 questions per user, you can expect 15,000 queries per month. (10,000 users/mo * 3 queries/user * 50% to Discovery = 15,000)

Querying with document-level security enabled

IBM Cloud Pak for Data **IBM Cloud Pak for Data only**



Note: This information applies only to installed deployments.

If you enable document-level security for a collection, only documents that the current user has permission to access are returned in search results. For more information, see *Configuring document-level security*.

To return search results that adhere to the security restrictions, the current user must meet these requirements:

- Have access to your Discovery instance.
- Have access to the data source.

If the current user does not meet these requirements, no search results are returned.

The username that is associated with your Discovery instance is used to generate an authorization token. The token is used to authenticate Discovery queries.

To generate each access token, run the following command:

```
$ curl -u "{username}:{password}" \
"https://{hostname}:{port}/v1/preauth/validateAuth"
```

Replace **{username}** and **{password}** with the user's Discovery credentials.

Use the bearer token that is associated with the user when you run the query.

```
$ curl -H "Authorization: Bearer {token}" \
'https://{hostname}/{instance_name}/v2/projects/{project_id}/collections/{Collection_ID}/query\?version\=2019-11-29'
```

DQL overview

The Discovery Query Language defines syntax you can use to filter, search, and analyze your data.

How to write a Discovery Query Language query

The Discovery Query Language leverages the structure of indexed documents. The following JSON snippet shows an indexed document from a collection where the **Entities** enrichment is applied. As a result of the enrichment, the JSON structure captures any mentions of known entities, such as city names, companies, or famous people.

In this example, the recognized entity is the company name **IBM**.

```
{
  "document": {
    "document_id": "f7f27ea30eb3e4c0ce21830618d9ee99",
    "enriched_text": [
      {
        "entities": [
          {
            "model_name": "natural_language_understanding",
            "mentions": [],
            "text": "IBM",
            "type": "Organization"
          }
        ]
      }
    ]
  }
}
```

To create a query that returns all of the documents in which the entity **IBM** is mentioned, use the following syntax:

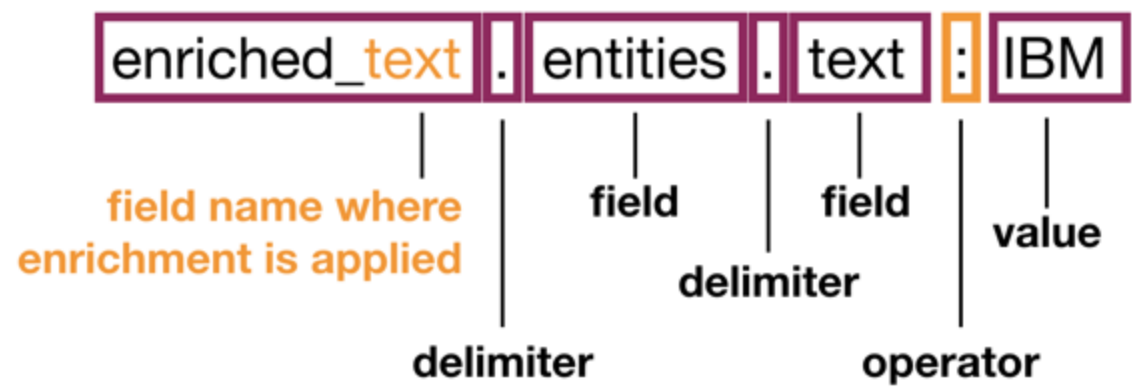


Figure 1. Example query structure

This basic query contains a nested path expression before the `:` operator. Each path element is the name of a field in the document separated by a period (`.`). The `:` operator indicates that the text that follows the operator must be included in the result.

The `::` operator indicates that the text must be matched exactly in the result. For more information, see [Query operators](#). You can see how the two operators are used in the following examples.

- To return matching documents in order of relevance, pass the following data object in the `POST` request:

```
{
  "query": "enriched_text.entities.text:IBM"
}
```

- To return matching documents in any order, pass the following data object in the `POST` request as the query body:

```
{
  "filter": "enriched_text.entities.text::IBM"
}
```

Using the filter and query parameters together

The `filter` parameter returns faster than the `query` parameter and its results are cached. If you submit queries that use the `filter` and `query` parameters separately on a small data set, each request returns similar (if not identical) results.

In large data sets, if you need results to be returned in order of relevance, combine the `filter` and `query` parameters. Using the parameters together improves performance because the `filter` parameter is applied first. It filters the documents and caches the results. The `query` parameter then ranks the cached results.

Filter example: Get a document by its ID

Query body:

```
{
  "filter": "document_id::b6d8c6e3-1097-421b-9e39-75717d2554aa"
}
```

If the document exists, the query returns 1 matching result. If it doesn't, the query returns no matching results.

Filter example: Find a document ID by its file name

If you don't know the `document_id` of a document, but you know the original `filename` of the document, you can use the `filter` and `return` parameters together to discover the `document_id`.

Query body:

```
{
  "filter": "extracted_metadata.filename::100674.txt",
  "return": [ "document_id", "extracted_metadata" ]
}
```

Response:

```
{
  "matching_results": 1,
  "results": [
    {
      "document_id": "b6d8c6e3-1097-421b-9e39-75717d2554aa",
      "extracted_metadata": {
        "sha1": "AD447F7592A17CDCBF0A589C4E6EC2087AF7H35F",
        "filename": "100674.txt",

```

```
      "file_type": "text"
    }
  }
]
```

Filter example: Find documents that mention an entity value

The query looks for documents that mention the entity `Gilroy` and finds 4 matching documents.

Query body:

```
{
  "filter": "enriched_text.entities.text::Gilroy"
}
```

Response:

```
{
  "matching_results": 4
}
```

Filtering nested values

You can nest one filter inside another to ensure that the documents that are returned match more than one condition.

In the documents used for these examples, the entity `"Gilroy"` appears as both a `"Location"` (a town in California) and as a `"Person"` (a surname) entity type. To find documents where `"Gilroy"` appears as a location, write a query that filters on two nested fields at the same time: the entity text must be `"Gilroy"` and the entity type must be `"Location"`.

One way to write the query is as follows:

```
{
  "filter": "enriched_text.entities.text::Gilroy,enriched_text.entities.type::Location"
}
```

This query matches documents where some path `enriched_text.entities.text` is `Gilroy` and some path `enriched_text.entities.type::Location` is `Location`. However, there is no guarantee that those two paths will be under the same `entities` object. For example, the query matches documents that have `Gilroy` as a `Person` entity type and, at the same time, have some other `Location` entity type object.

To accurately capture the nested semantics of this query, nest the filter values by using the following syntax:

Query body:

```
{
  "filter": "enriched_text.entities:(text::Gilroy,type::Location)"
}
```

This stricter query matches only those documents in which there is an `entities` object with `text` equal to `Gilroy` and `type` equal to `Location`.

As another example, if you want to match documents that contain an `entities` object with `text` equal to `Gilroy` but `type` not equal to `Location`, you can use the ***not equal*** operator in the query, for example:

```
{
  "filter": "enriched_text.entities:(text::Gilroy,type::!Location)"
}
```

You can also use aggregations to do more sophisticated filtering of the results. For more information about the available aggregation types, see [Query aggregations](#).

For more information about the Discovery Query Language, see the following topics:

- [Query parameters](#)
- [Query operators](#)

Query parameters

You can use these parameters when you write queries with the Discovery Query Language. For more information, see the Discovery [API reference](#). For an overview of query concepts, see the [Query overview](#).

Queries that are written in the Discovery Query Language can include both search and structure parameters.

The default values for query parameters can differ by project type. For more information about default values, see *Default query settings*.

Search parameters

Use search parameters to search your collection, identify a result set, and analyze the result set.

The **results set** is the group of documents that are identified by the combined searches of the search parameters. The results set might be significantly larger than the returned results. If an empty query is submitted, the results set is equal to all the documents in the collection.



Important: Documents that you do not have permissions to access are not returned in query results.

Answer finding

IBM Cloud The `find_answers` parameter is supported in managed deployments only.

By default, Discovery provides answers by returning the entire passage that contains the answer to a natural language query. When the answer-finding feature is enabled, Discovery also provides a "short answer" within the passage, and a confidence score to show whether the "short answer" answers the question that is explicit or implicit in the user query. Applications that use the answer-finding feature can display the short answer alone or can display the short answer emphasized in the context of the full passage. For most applications, displaying the short answer emphasized within the full passage is preferable, because answers generally make more sense in context.

The answer finding feature behaves in the following ways:



Note: In the passage examples that follow, the short answers are shown in bold font.

- Finds answers. It doesn't create answers. The answer must be part of the text; it can't be inferred.

"What was IBM's revenue in 2022?" can get a correct answer if you have a document that states what IBM's revenue was in 2022. However, if you have a document that lists what IBM's revenue was in each quarter of 2022, it doesn't add them up and give you a total.

- Handles synonyms and lexical variations if the answer is available.
 - Example question: "When did IBM purchase Red Hat?"
 - Passage: "IBM closed its \$34 billion acquisition of Red Hat in **July of 2019.**"
- Combines information across multiple sentences if they are close together (within approximately 2,000 characters).
 - Example question: "When did IBM purchase Red Hat?"
 - Passage: "IBM acquired Red Hat for \$34 billion. The deal closed in **July of 2019.**"
- Handles implicit questions similar to the way it would handle the equivalent explicit question.

Example questions:

- **company that developed the AS/400**
 - **What company developed the AS/400?**
- Works well with questions with longer phrase or clause answers.
 - Example question: How do I flip a pancake?
 - Passage: The key to getting a world-class pancake is flipping it properly. The best way to flip a pancake is to **stick a spatula under it, lift it at least 4 inches in the air, and quickly rotate the spatula 180 degrees.**
- Many how or why questions are only fully answered by much longer spans of text. The answer-finding feature does not return a whole document as the answer (and it doesn't summarize a document length answer).
- Handles yes or no questions that are factual and have a concise answer in the text
 - Example question: Is there a library in Timbuktu
 - Passage: Timbuktu's **main library, officially called the Ahmed Baba Institute of Higher Islamic Studies and Research**, is a treasure house that contains more than 20,000 manuscripts that cover centuries of Mali's history.
- Handles questions with very short answers, such as names and dates, especially when the type of answer that is required is explicit in the text.
- Handles opinion questions, but only by finding a statement of that opinion; it does not assess the validity of the opinion.
 - Example question: Should I try blue eyeshadow?

- Passage: We think **blue eye shadow is trending** this year.


How the answer-finding feature works

After a user submits a query, the query is analyzed by the Discovery service. Query analysis transforms the user's original query in ways that improve the chances of finding the best search results. For example, it lemmatizing words, removes stop words, and adds query expansions. The search is performed and the resulting documents and passages are returned.

Answer finding is applied to the returned passages. Up to 60 passages are sent to the answer-finding service. How these 60 passages are chosen differs based on the `passages.per_document` parameter value.

- If `passages.per_document` is `false`, the top 60 passages from all of the documents that are returned by search are chosen based on their passage scores only.
- If `passages.per_document` is `true`, the returned documents are ranked first, and then the top 60 passages from these top documents are chosen.

For example, if you set the query to return 100 documents (count=100) and ask for 2 passages from each document (passages.max_per_document=2), then 2 passages are chosen from each of the 30 top-ranked documents (2 x 30 = 60 passages) only. No passages are chosen from the remaining 70 documents.

 **Tip:** If your goal is to get the best 10 short answers, a good approach is to give the answer-finding feature various passages from more documents than just the top 10. To do so, set `passages.per_document` to `true`, and then request 20 documents and up to 3 passages from each document with the answer-finding feature enabled. The answer-finding feature searches for answers in up to 20*3 = 60 passages.

Answer finding does not use the transformed query string that is generated by query analysis. Instead, it uses a copy of the user's original input that is stored at query time to find the best short answer. If the answer-finding module is confident that it found an answer in one of the passages, the answer confidence score is combined with the document and passage scores to produce a final ranking, which can promote a document or passage that might otherwise be missed.

Answer-finding API details

The answer-finding API adds the following parameters to the `passage` section of the query API:

- `find_answers` is optional and defaults to `false`. If it is set to `true` (and the `natural_language_query` parameter is set to a query string), the answer-finding feature is enabled.
- `max_answers_per_passage` is optional and defaults to `1`. In this case, the answer-finding feature finds the number of answers that are specified at most from any one passage.

A section is also added to the return value within each `passage` object. That section is called `answers`, and it is a list of answer objects. The list can be up to `max_answers_per_passage` in length. Each answer object contains the following fields:

- `answer_text` is the text of the concise answer to the query.
- `confidence` is a number between `0` and `1` that is an estimate of the probability that the answer is correct. Some answers have low confidence and are unlikely to be correct. Be selective about what you do with answers based on this value. The confidence and order of documents in the search results are adjusted based on this combination if the `per_document` parameter of passage retrieval is set to `true` (which is the default).
- `start_offset` is the start character offset (the index of the first character) of the answer within the field that the passage came from. It is greater than or equal to the start offset of the passage (since the answer must be within the passage).
- `end_offset` is the end character offset (the index of the last character, plus one) of the answer within the field that the passage came from. It is less than or equal to the end offset of the passage.

To find answers across the entire project:

- Set `passages.enabled` to `true`
- Set `passages.find_answers` to `true`

To find answers within a single known document (for example, a document review application with long, complex documents):

- Set `passages.enabled` to `true`
- Set `passages.find_answers` to `true`
- Set `filter` to select the `document_id` for the document

The following example shows a query that uses this API:

```
POST /v2/projects/{project_id}/query{
  "natural_language_query": "Why did Nixon resign?",
  "passages": {
    "enabled": true, "find_answers":true
  }
}
```

Example response:

```
{
  "matching_results": 74, "retrieval_details": { "document_retrieval_strategy": "untrained"},
  "results": [
    {
      "document_id": "63919442-7d5b-4cae-ab7e-56f58b1390fe",
      "result_metadata": {"collection_id": "collection_id1234","document_retrieval_source":"search","confidence": 0.78214},
      "metadata": {"parent_document_id": "63919442-7d5b-4cae-ab7e-56f58b1390fg"},
      "title": "Watergate scandal",
      "document_passages": [
        {
          "passage_text": "With his complicity in the cover-up made public and his political support completely eroded, Nixon resigned from office on August 9, 1974. It is believed that, had he not done so, he would have been impeached by the House and removed from office by a trial in the Senate.",
          "field": "text",
          "start_offset": 281,
          "end_offset": 553,
          "answers": [
            {
              "answer_text": "his complicity in the cover-up made public and his political support completely eroded",
              "start_offset": 286, "end_offset": 373, "confidence": 0.78214
            }
          ]
        }
      ]
    }
  ]
}
```

natural_language_query

Use a natural language query to enter queries that are expressed in natural language, as might be received from a user in a conversational or free-text interface, such as IBM Watson Assistant. The parameter uses the entire input as the query text. It does not recognize operators.

The maximum query string length for a natural language query is **2048**.

Result confidence scores

When the query type is a natural language query, each result has a confidence score. The confidence score is a measure of the relevancy of the result. Each query result is evaluated and scored independently.

A variety of techniques are used to evaluate confidence. One important factor is the frequency of word matches between the query and the document.

Because a variety of techniques are used in different contexts to evaluate the result, the number range of result scores can vary widely from query to query. This variability means that comparing the confidence score to a static threshold value is an unsuitable method by which to delimit the results that are returned by your application. Results are ordered from highest to lowest confidence. You can find the best candidate answers by taking the top results, regardless of their confidence score values.

The **natural_language_query** parameter enables capabilities such as relevancy training. For more information, see *[Improving result relevance with training](#)*.

query

A query search returns all documents in your data set with full enrichments and full text in order of relevance. A query also excludes any documents that don't mention the query content.

aggregation

Aggregation queries return a count of documents that match a set of data values. For the full list of aggregation options, see [Query aggregations](#).

filter

A cacheable query that excludes any documents that don't mention the query content. Filter search results are **not** returned in order of relevance.

When you write a query that includes both a **filter**, and an **aggregation**, **query**, or **natural_language_query** parameter, the **filter** parameter runs first, and then any **aggregation**, **query**, or **natural_language_query** parameters run in parallel.

With a simple query, especially on a small data set, the **filter** and **query** parameters often return the exact same (or similar) results. If the **filter** and **query** calls return similar results, and you don't need the responses to be returned in order of relevance, use the **filter** parameter. Filter calls are faster and are cached. Caching means that the next time you make the same call, you get a much quicker response, particularly in a big data set.

Structure parameters

Structure parameters define the content and organization of the documents in the returned JSON. Structure parameters don't affect

which documents are part of the entire results set.

return

A comma-separated list of the portion of the document hierarchy to return. Any of the document hierarchies are valid values. If this parameter is an empty list, then all fields are returned.

count

The number of documents that you want to return in the response. The default is `10`. The maximum for the `count` and `offset` values together in any one query is `10000`.

offset

Index value of the position of the search result where the set of results to return begins. For example, if the total number of results that are returned is 10, and the offset is 8, it returns the last two results. The default is `0`. The maximum allowed value for the `count` and `offset` together in any one query is `10000`.

spell correction

In natural language queries, checks the query that is submitted for misspelled terms. The query is processed as-is. However, likely corrections to the original query, if any exist, are returned in the `suggested_query` field of the response. The suggestions are not used automatically, but your application can make use of them.

sort

A comma-separated list of fields in the document to sort by. You can optionally specify a sort direction by prefixing the field with `-` for descending order or `+` for ascending order. Ascending order is the default sort direction.

highlight

A Boolean value that specifies whether to include a `highlight` object in the returned output. When included, the highlight returns keys that are field names and values that are arrays. The arrays contain segments of query-matching text that is highlighted by using the HTML emphasis (``) tag.

This parameter is ignored if `passages.enabled` and `passages.per_document` are `true`, in which case passages are returned for each document instead of highlights.



Note: Currently, if the query searches for an `exact match` of an enrichment mention, only lowercase matches are highlighted. When the `includes` operator is used, upper- and lowercase matches are highlighted.

The output lists the `highlight` object after the `enriched_text` object, as shown in the following example.

```
$ curl -H "Authorization: Bearer {token}" \
'https://{hostname}/{instance_name}/v2/projects/{project_id}/collections/{collection_id}/query?version=2019-11-29&natural_language_query=Hybrid%20cloud%20companies&highlight=true'
```

The JSON that is returned has the following format:

```
{
  "highlight": {
    "extracted_metadata.title": [
      "IBM to Acquire Sanovi Technologies to Expand Disaster Recovery Services for <em>Hybrid</em> <em>Cloud</em>"
    ],
    "enriched_text.concepts.text": [
      "Privately held <em>company</em>",
      "<em>Cloud</em> computing"
    ],
    "text": [
      " Sanovi Technologies, a privately held <em>company</em> that provides <em>hybrid</em> <em>cloud</em> recovery, <em>cloud</em> migration",
      "IBM to Acquire Sanovi Technologies to Expand Disaster Recovery Services for <em>Hybrid</em> <em>Cloud</em>\n\nPublished",
      " undergoing digital and <em>hybrid</em> <em>cloud</em> transformation.\n\nURL: http://www.ibm.com/press/us/en/pressrelease/50837.wss",
      " and business continuity software for enterprise data centers and <em>cloud</em> infrastructure. Adding"
    ],
    "enriched_text.categories.label": [
      "/business and industrial/<em>company</em>/bankruptcy"
    ],
    "enriched_text.entities.type": [
      "<em>Company</em>"
    ],
    "html": [
      " Technologies, a privately held <em>company</em> that provides <em>hybrid</em> <em>cloud</em>\n recovery,
```

```

<em>cloud</em> migration and business",
  " Disaster Recovery Services for <em>Hybrid</em> <em>Cloud</em></title></head>\n<body>\n\n\n<p>Published: Thu,
27 Oct 2016 07:01",
  " digital and <em>hybrid</em> <em>cloud</em> transformation.</p>\n<p>URL:
http://www.ibm.com/press/us/en/pressrelease/50837.wss</p>\n\n\n\n</body></html>",
  " continuity software for \nenterprise data centers and <em>cloud</em> infrastructure. Adding these \ncapabilities"
]
}
}
}

```

passages

A Boolean that specifies whether the service returns a set of the most relevant passages from the documents that are returned by a query that uses the `natural_language_query` parameter. The passages are generated by sophisticated Watson algorithms that determine the best passages of text from all of the documents returned by the query. The default value for the parameter differs based on your project type. For more information about default values, see *Default query settings*.

Discovery attempts to return passages that start at the beginning of a sentence and stop at the end by using sentence boundary detection. To do so, it first searches for passages approximately the length specified in the `passages.characters` parameter (for most project types, the default is `200`). It then expands each passage to the limit of twice the specified length so as to return full sentences. If your `passages.characters` parameter is short or the sentences in your documents are long there might be no sentence boundaries close enough to return the full sentence without going over twice the requested length. In that case, Discovery stays within the limit of twice the `passages.characters` parameter, so the passages that are returned might not include the entire sentence and can omit the beginning, end, or both.

Since sentence boundary adjustments expand passage size, the average passage length can increase. If your application has limited screen space, you might want to set a smaller value for `passages.characters` or truncate the passages that are returned by Discovery. Sentence boundary detection works for all supported languages and uses language-specific logic.

Passages are grouped with each document result and are ordered by passage relevance. Including passage retrieval in queries increases the response time because it takes more time to score the passages.

You can adjust the fields in the documents for passage retrieval to search with the `passages.fields` parameter.

The `passages` parameter returns matching passages (`passage_text`), and the `score`, `document_id`, the name of the field that the passage was extracted from (`field`), and the starting and ending characters of the passage text within the field (`start_offset` and `end_offset`), as shown in the following example.

```

$ curl -H "Authorization: Bearer {token}"
'https://{hostname}/{instance_name}/v2/projects/{project_id}/collections/{collection_id}/query?version=2019-11-
29&natural_language_query=Hybrid%20cloud%20companies&passages=true&passages.per_document=false'

```

The JSON that is returned from the query has the following format:

```

{
  "matching_results":2,
  "passages":[
    {
      "document_id":"ab7be56bcc9476493516b511169739f0",
      "passage_score":15.230205287402338,
      "passage_text":"a privately held company that provides hybrid cloud recovery, cloud migration and business continuity
software for enterprise data centers and cloud infrastructure.",
      "start_offset":120,
      "end_offset":300,
      "field":"text"
    },
    {
      "passage_text":"Disaster Recovery Services for Hybrid Cloud</title></head>\n<body>\n\n\n<p>Published: Thu, 27 Oct
2016 07:01:21 GMT</p>\n",
      "passage_score":10.153470191601558,
      "document_id":"fbb5dcb4d8a6a29f572ebdeb6fbed20e",
      "start_offset":70,
      "end_offset":120,
      "field":"html"
    }
  ]
}

```

passages.fields

A comma-separated list of fields in the index that passages are drawn from. If this parameter is not specified, then passages from all root-level fields are included.

You can specify fields in both the `return` and `passages.fields` parameters. When you specify both parameters, each with different values, they are treated separately.

For example, the request might include the parameters `"return": ["docno"]` and `"passages":{"fields": ["body"]}`. The `body` field is specified in `passages.fields`, but not in `return`. In the result, passages from the document body are returned, but the contents of the body field itself is not returned.

passages.count

The maximum number of passages to return. The search returns fewer passages if the specified count is the total number found. The default value is `10`. The maximum value is `100`.

passages.characters

The approximate number of characters that any one passage can have. The default value is `200`. The minimum is `50`. The maximum is `2,000`. Passages that are returned can contain up to twice the requested length (if necessary) to get them to begin and end at sentence boundaries.

passages.max_per_document

One passage is returned per document by default. You can increase the maximum number of passages to return per document by specifying a higher number in the `passages.max_per_document` parameter.

similar

Finds documents that are similar to documents that you identify as being of interest to you. To find similar documents, Discovery identifies the 25 most relevant terms from the original document and then searches for documents with similar relevant terms.

If `similar.enabled` is `true`, you must specify the `similar.document_ids` field to include a comma-separated list of the documents of interest.



Note: In installed deployments, support for this parameter was added with the 4.6.0 release.

table retrieval

If **Table understanding** is enabled in your collection, a `natural_language_query` finds tables with content or context that match a search query.

Example query:

```
$ curl -H "Authorization: Bearer {token}" \
'https://{hostname}/{instance_name}/v2/projects/{project_id}/collections/{collection_id}/query?version=2019-11-29&natural_language_query=interest%20appraised&table_results=true'
```

The JSON that is returned from the query has the following format:

```
{
  "matching_results": 1,
  "session_token": "1_FDjAVkn9SW6oH9y5_9Ek3KsNFG",
  "results": [
    {}
  ]
}
{
  "table_results": [
    {
      "table_id": "e883d3df1d45251121cd3d5aef86e4edc9658b21",
      "source_document_id": "c774c3df0c90255191cc0d4bb8b5e8edc6638d96",
      "collection_id": "collection_id",
      "table_html": "html snippet of the table info",
      "table_html_offset": 42500,
      "table": [
        {
          "location": {
            "begin": 42878,
            "end": 44757
          },
          "text": "Appraisal Premise Interest Appraised Date of Value Value Conclusion\nMarket Value \\"As Is\\" Fee Simple Estate\nJanuary 12, 2016 $1,100,000\n",
          "section_title": {
            "location": {
              "begin": 42300,
              "end": 42323
            },
            "text": "MARKET VALUE CONCLUSION"
          },
          "title": {},
          "table_headers": [],

```

```

    "row_headers": [
      {
        "cell_id": "rowHeader-42878-42896",
        "location": {
          "begin": 42878,
          "end": 42896
        },
        "text": "Appraisal Premise",
        "text_normalized": "Appraisal Premise",
        "row_index_begin": 0,
        "row_index_end": 0,
        "column_index_begin": 0,
        "column_index_end": 0
      }
    ],
    "column_headers": [],
    "body_cells": [
      {
        "cell_id": "bodyCell-43410-43424",
        "location": {
          "begin": 43410,
          "end": 43424
        },
        "text": "Date of Value",
        "row_index_begin": 0,
        "row_index_end": 0,
        "column_index_begin": 2,
        "column_index_end": 2,
        "row_header_ids": [
          "rowHeader-42878-42896",
          "rowHeader-43145-43164"
        ],
        "row_header_texts": [
          "Appraisal Premise",
          "Interest Appraised"
        ],
        "row_header_texts_normalized": [
          "Appraisal Premise",
          "Interest Appraised"
        ],
        "column_header_ids": [],
        "column_header_texts": [],
        "column_header_texts_normalized": [],
        "attributes": []
      }
    ],
    "contexts": [
      {
        "location": {
          "begin": 44980,
          "end": 44996
        },
        "text": "Compiled by CBRE"
      }
    ],
    "key_value_pairs": []
  }
]
}

```

table_results.enabled

When `true`, a `table_results` array is included in the response with a list of table objects that match the `natural_language_query` value in order of scored relevance. For all project types, except *Document Retrieval for Contracts*, the default value is `false`.

table_results.count

This parameter specifies the maximum number of tables that can be included in the `table_results` array. Only returned if `table_results.enabled = true`. The default value is `10`.

Query operators

You can use operators when you write queries to submit to Discovery by using the Query API.

The types of operators that are supported differ by query type:

- *Natural language queries*

- [Discovery Query Language \(DQL\) queries](#)

Natural Language Query (NLQ) operator

The `natural_language_query` parameter accepts a string value.

"" (Phrase query)

Use quotation marks to emphasize a single word or phrase in the query that is most important to match. For example, the following request boosts documents that contain the term “nomination” in them.

```
{
  "natural_language_query": "What is the process for \"nomination\" of bonds?"
}
```

Specifying a quoted phrase does not prevent documents without the phrase from being returned. It merely gives more weight to documents with the phrase than those without it. For example, the query results might also contain documents that mention “bonds” or “process” and do not contain the word “nomination”.

The following request boosts the phrase “change in monetary policy” and also matches “change” or “monetary” or “policy”.

```
{
  "natural_language_query": "\"change in monetary policy\""
}
```

Single quotation marks (') are not supported. You cannot use wildcards (*) in phrase queries.

Discovery Query Language (DQL) operators

Operators are the separators between different parts of a query.

. (JSON delimiter)

This delimiter separates the levels of hierarchy in the JSON schema

For example, the following query argument identifies the section of the `enriched_text` object that contains entities and the text recognized as an entity.

```
enriched_text.entities.text
```

The JSON representation of this section looks as follows:



Figure 1. JSON representation of the `enriched_text.entities.text` field

: (Includes)

This operator specifies inclusion of the full query term.

For example, the following query searches for documents that contain the term `cloud computing` in the `text` field:

```
{
  "query": "enriched_text.entities.text:\"cloud computing\""
}
```

The **includes** operator does not return a partial match for the query term. If you want to find a partial match for a term, use a **wildcard** operator with the **includes** operator. For example, if you want to find any occurrences of `TP53` or `p53` in the `test_results` field, the following query will **not** find occurrences of both terms:

```
{
  "query": "test_results:P53"
}
```


Instead, include a wildcard in the request. For example, use the following query request. Because we are using the wildcard operator, we also changed the term to lowercase.

```
{
  "query": "test_results:*p53"
}
```

With this syntax, occurrences of `p53`, `tp53`, `P53`, or `TP53` are all returned.

"" (Phrase query)

Phrase queries only match occurrences of the whole phrase. The order of the words in the phrase must match.

For example, the following query returns only documents that contain a field named `quotation` with the text, `There's no crying in baseball`.

```
{
  "query": "quotation:There's no crying in baseball"
}
```

A document with a `quotation` field that says `Jimmy Dugan said there's no crying in baseball` is also returned. However, documents that only mention `baseball` or `crying` without the entire phrase are not matched. Neither is a document with `In baseball, there's no crying`. Documents that contain the right text in the wrong field also are not matched. For example, a document with the text `There's no crying in baseball` in the `text` field is not returned.

Single quotation marks (') are not supported. You cannot use wildcards (*) in phrase queries.

:: (Exact match)

This operator specifies an exact match for the query term. Exact matches are case-sensitive.

For example, the following query searches for documents that contain entities of type `Organization`:

```
{
  "query": "enriched_text.entities.type::Organization"
}
```

The entire content of the field that you specify must match the phrase you specify. For example, the following query finds documents in which only entity mentions of `IBM Cloud` are detected, not `IBM Cloud Pak for Data` or `IBM cloud` or `Cloud`.

```
{
  "query": "enriched_text.entities.text::\"IBM Cloud\""
}
```

:! (Does not include)

This operator specifies that the results do not contain a match for the query term.

For example:

```
{
  "query": "enriched_text.entities.text:!\"cloud computing\""
}
```

::! (Not an exact match)

This operator specifies that the results do not exactly match the query term.

For example:

```
{
  "query": "enriched_text.entities.text::!\"Cloud computing\""
}
```

Exact matches are case-sensitive.

\ (Escape character)

Escape character that preserves the literal value of the character that follows it.

For example, you can place an escape character before a quotation mark in query text to include the quotation mark in the query string.

```
{
  "query": "title::Dorothy said: \"There's no place like home\""
}
```

(), [] (Nested grouping)

Logical groupings can be formed to specify more specific information.

For example:


```
{
  "query": "enriched_text.entities:(text:IBM,type:Company)"
}
```

| (or)

Boolean operator for "or".

In the following example, documents in which **Google** or **IBM** are identified as entities are returned:

```
{
  "query": "enriched_text.entities.text:Google|enriched_text.entities.text:IBM"
}
```

 **Note:** The includes (`:`, `::`) and match (`:::`, `:::!`) operators have precedence over the **OR** operator.

For example, the following syntax searches for documents in which **Google** is identified as an entity or the string **IBM** is present:

```
{
  "query": "enriched_text.entities.text:Google|IBM"
}
```

It is treated as follows:


```
(enriched_text.entities.text:Google) OR IBM
```

, (and)

Boolean operator for "and".

In the following example, documents in which **Google** and **IBM** both are identified as entities are returned:

```
{
  "query": "enriched_text.entities.text:Google,enriched_text.entities.text:IBM"
}
```

 **Note:** The includes (`:`, `::`) and match (`:::`, `:::!`) operators have precedence over the **AND** operator.

For example, the following syntax searches for documents in which **Google** is identified as an entity and the string **IBM** is present:

```
{
  "query": "enriched_text.entities.text:Google,IBM"
}
```

It is treated as follows:

```
(enriched_text.entities.text:Google) AND IBM
```

<=, >=, >, < (Numerical comparisons)

Creates numerical comparisons of **less than** or **equal to**, **greater than** or **equal to**, **greater than**, and **less than**.

Only use numerical comparison operators when the value is a **number** or **date**.



Tip: Any value that is surrounded by quotations is a String. Therefore, `score>=0.5` is a valid query and `score>="0.5"` is not.

For example:

```
{
  "query": "invoice.total>100.50"
}
```

^x (Score multiplier)

Increases the score value of a search term.

For example:

```
{
  "query": "enriched_text.entities.text:IBM^3"
}
```

*** (Wildcard)**

Matches unknown characters in a search expression. Do not use capital letters with wildcards.

For example:

```
{
  "query": "enriched_text.entities.text:ib*"
}
```

~n (String variation)

The number of character differences that are allowed when matching a string. The maximum variation number that can be used is 2.

For example, the following query returns documents that contain `car` in the title field, as well as `cap`, `cat`, `can`, `sat`, and so on:

```
{
  "query": "title:cat~1"
}
```

The normalized version of the word is used for matching. Therefore, if the input contains "cats", the search looks for "cat", which is the normalized form of the plural cats.

When a phrase is submitted, each term in the phrase is allowed the specified number of variations. For example, the following input matches `cat dog` and `far log` in addition to `car hog`.

For example:

```
{
  "query": "title:\"car hog\"~1"
}
```

:* (Exists)

Used to return all results where the specified field exists.

For example:

```
{
  "query": "title:*"
}
```

:!* (Does not exist)

Used to return all results that do not include the specified field.

For example:

```
{
  "query": "title:!*"
}
```

For more information, see the Discovery [API reference](#).

For an overview of query concepts, see the [Query overview](#).

Query aggregations

Use aggregations to group, analyze, or compare results that are returned by a query request.

An aggregation is defined by an **aggregation** parameter that you can specify in the Query API. The input to the aggregation parameter is the document set that is returned from the **query**, **filter**, or **natural_language_query** parameter that is specified as a separate parameter in the same query request. Otherwise, the aggregation is applied to all of the documents in the project.

You can use an aggregation to do calculations from values in the result document set. For example, to get information about the highest dollar amount in the **order.total** field of the documents that are returned as query results, use **max(order.total)** as the value of the **aggregation** parameter.

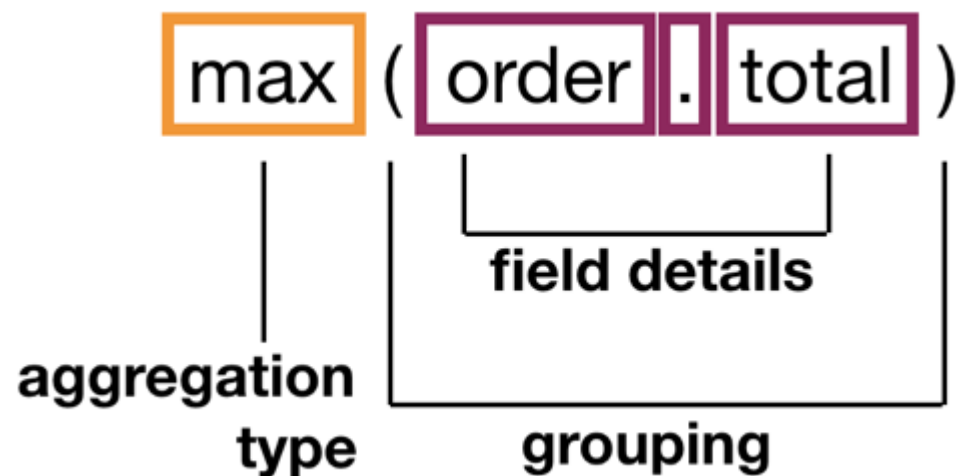


Figure 1. Aggregation query structure example

The aggregation parameter returns data about the field with the highest value.

```
"aggregations": [
  {
    "type": "max",
    "field": "order.total",
    "value": 100668.00
  }
]
```

Grouping documents

In addition to doing calculations, you can use an aggregation to group documents in the result set that match certain values, so you can count them or analyze them further. For example, you can use an aggregation to search a set of traffic incident reports for documents that mention the term **brake**. And from the returned documents, find reports from the US states with the most relevant mentions of the term.

In the following example request, the count parameter that returns only 3 aggregation results is included to make the example easier to follow.

```
{
  "query": "brake",
  "aggregation": "term(field:STATE,count:3,relevancy:true)"
}
```

The output of the aggregation parameter is returned in an **aggregations** object that is displayed before the **results** object, which contains the query results. A maximum of 50,000 values can be returned in the **aggregations** object for a single query.

The resulting **aggregations** object contains summary information about the query results. In this example, for instance, it shows that traffic incident reports from New York, California, and Florida have the most relevant mentions of the term **brake**.

```
{
  "matching_results": 9064,
  "retrieval_details": {
    "document_retrieval_strategy": "untrained"
  },
  "aggregations": [
    {
      "type": "term",
      "field": "STATE",
      "results": [
        {

```

```

      "key": "NY",
      "matching_results": 693,
      "relevancy": 1.1649531567631084,
      "total_matching_documents": 2156,
      "estimated_matching_results": 542
    },
    {
      "key": "CA",
      "matching_results": 1210,
      "relevancy": 1.1170819184294765,
      "total_matching_documents": 4017,
      "estimated_matching_results": 1011
    },
    {
      "key": "FL",
      "matching_results": 511,
      "relevancy": 0.828014956418841,
      "total_matching_documents": 2199,
      "estimated_matching_results": 553
    }
  ]
},
"results": []

```

Combining aggregation types

There are different types of aggregations that you can use to analyze or group the query results. And you can combine more than one aggregation in a request to do more targeted analysis.

The following example shows a request that is composed of two term operators. The first term aggregation groups the input documents by US STATE values and selects 3 groups. The second term aggregation applies to each of those 3 groups and groups them further by the value of CITY. Only 2 of those CITY subgroups are returned per STATE group.

The relevancy parameter is being excluded to make the results easier to read.

```

{
  "query": "brake",
  "aggregation": "term(field:STATE,count:3).term(field:CITY,count:2)"
}

```

The response contains city information from each state.

```

{
  "matching_results": 9064,
  "retrieval_details": {
    "document_retrieval_strategy": "untrained"
  },
  "aggregations": [
    {
      "type": "term",
      "field": "STATE",
      "count": 3,
      "results": [
        {
          "key": "CA",
          "matching_results": 1210,
          "aggregations": [
            {
              "type": "term",
              "field": "CITY",
              "count": 2,
              "results": [
                {
                  "key": "LOS ANGELES",
                  "matching_results": 77
                },
                {
                  "key": "SAN DIEGO",
                  "matching_results": 66
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "key": "NY",
      "matching_results": 693,
      "aggregations": [

```

```

    {
      "type": "term",
      "field": "CITY",
      "count": 2,
      "results": [
        {
          "key": "BROOKLYN",
          "matching_results": 35
        },
        {
          "key": "NEW YORK",
          "matching_results": 21
        }
      ]
    }
  ]
},
{
  "key": "FL",
  "matching_results": 511,
  "aggregations": [
    {
      "type": "term",
      "field": "CITY",
      "count": 2,
      "results": [
        {
          "key": "JACKSONVILLE",
          "matching_results": 33
        },
        {
          "key": "TAMPA",
          "matching_results": 29
        }
      ]
    }
  ]
}
]
}
],
"results": []

```

The order in which you specify the aggregations matters. For example, if you reverse the order of the term aggregations from the previous example, you get different results.

```

{
  "query": "brake",
  "aggregation": "term(field:CITY,count:3).term(field:STATE,count:1)"
}

```

The new order produces results that surface Chicago, a city that wasn't included in the previous set of results. When the request starts by grouping by state, Illinois, which has only one city with a high number of traffic incident reports, is not included in the results. New York and Florida, which both have more than one city with many incident reports, produce a higher number of statewide matches and therefore, were returned. When you group by city first, the results change.

```

{
  "matching_results": 9064,
  "retrieval_details": {
    "document_retrieval_strategy": "untrained"
  },
  "aggregations": [
    {
      "type": "term",
      "field": "CITY",
      "count": 4,
      "results": [
        {
          "key": "LOS ANGELES",
          "matching_results": 77,
          "aggregations": [
            {
              "type": "term",
              "field": "STATE",
              "count": 1,
              "results": [
                {
                  "key": "CA",
                  "matching_results": 77
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}

```

```

    }
  ]
}
],
{
  "key": "SAN DIEGO",
  "matching_results": 66,
  "aggregations": [
    {
      "type": "term",
      "field": "STATE",
      "count": 1,
      "results": [
        {
          "key": "CA",
          "matching_results": 66
        }
      ]
    }
  ]
},
{
  "key": "CHICAGO",
  "matching_results": 59,
  "aggregations": [
    {
      "type": "term",
      "field": "STATE",
      "count": 1,
      "results": [
        {
          "key": "IL",
          "matching_results": 59
        }
      ]
    }
  ]
}
]
},
"results": []

```

Using aggregations to explore enrichments

The `term()` aggregation is especially useful for analyzing results to find out how many enrichments are recognized in the documents. For example, to count how many times each entity type is recognized in the filtered documents, you can submit the following query parameters:

```

{
  "filter": "enriched_text.entities:(text::Gilroy,type::Location)",
  "aggregation": "term(enriched_text.entities.type)"
}

```

The query first selects the documents that have at least one entity of type `Location` and whose text is `Gilroy`. This action returns 3 documents. From the returned documents, the aggregation then counts the number of documents in which each entity type appears.

```

{
  "matching_results": 3,
  "retrieval_details": {
    "document_retrieval_strategy": "untrained"
  },
  "aggregations": [
    {
      "type": "term",
      "field": "enriched_text.entities.type",
      "results": [
        {
          "key": "Location",
          "matching_results": 3
        },
        {
          "key": "Person",
          "matching_results": 3
        },
        {
          "key": "Company",
          "matching_results": 2
        }
      ]
    }
  ]
}

```



```

    },
    {
      "key": "GeographicFeature",
      "matching_results": 2
    },
    {
      "key": "Organization",
      "matching_results": 2
    },
    {
      "key": "Quantity",
      "matching_results": 2
    },
    {
      "key": "Facility",
      "matching_results": 1
    },
    {
      "key": "PrintMedia",
      "matching_results": 1
    }
  ]
}
]
}

```

The 3 matching documents all have a **Location** and a **Person** entity type (**"matching_results": 3**). However, only 2 of the matching documents have a **Company** entity type.

By default, the top 10 matches are returned, sorted by relevance. You can change the number of results by adding the **count** parameter to the aggregation.

```

{
  "filter": "enriched_text.entities:(text::Gilroy,type::Location)",
  "aggregation": "term(enriched_text.entities.type,count:20)"
}

```

Add a filter

Use the **filter()** in the aggregation clause to filter results. For example, you can specify the same filter that was submitted separately in the previous example directly in the **aggregation** clause.

```

{
  "aggregation": "filter(enriched_text.entities:(text::Gilroy,type::Location)).term(enriched_text.entities.type)"
}

```

In this case, the **filter().term()** aggregation finds the same result as the earlier example with the separate **filter** and **aggregation** clauses. However, results are ranked differently when the **filter** clause is used. You can leverage this difference by using the **filter()** clause within the **aggregation** clause to filter results from a sequence of expressions, as shown in the next example.

Start with nested objects

In the previous examples, the **"matching_counts"** value represents the number of documents that match the filter and aggregation. You might want to count how many *nested* objects are present in the query response. The **nested()** aggregation allows you to change the set of documents that is used as input to other aggregation terms.

For example, in the following query the **nested()** segment selects all **enriched_text.entities** nested objects as the input used by the **filter()** and **term()** segments.

```

{
  "aggregation":
  "nested(enriched_text.entities).filter(enriched_text.entities.type::Organization).term(enriched_text.entities.text,count:3)"
}

```

The query results in an **aggregations** object that looks as follows:

```

{
  "aggregations": [
    {
      "type": "nested",
      "path": "enriched_text.entities",
      "matching_results": 1993,
      "aggregations": [
        {

```

```

    "type": "filter",
    "match": "enriched_text.entities.type::Organization",
    "matching_results": 645,
    "aggregations": [
      {
        "type": "term",
        "field": "enriched_text.entities.text",
        "count": 3,
        "results": [
          {
            "key": "IBM",
            "matching_results": 36
          },
          {
            "key": "Docker",
            "matching_results": 12
          },
          {
            "key": "OpenShift",
            "matching_results": 12
          }
        ]
      }
    ]
  }
]
}

```

The `nested()` segment of the query found 1993 `enriched_text.entities` nested objects. The filter was applied to those objects and found 645 `enriched_text.entities` of type `Organization`.

Terminal operations

For most aggregation types, when you construct a query with multiple aggregation operations, the first operation is applied to the documents. Then, the output of that operation is used as the input for the next operation. However, a subset of the aggregation types are ***terminal operations***. The output of a terminal operation is not used as input for the next aggregation. Instead, the output is returned in a discrete group.

For an example of a request that combines aggregation types and includes an aggregation that performs a terminal operation, see the second [example](#) for the `average` aggregation type.

Aggregation types

The following types of aggregations are supported:

- [average](#)
- [filter](#)
- [group_by](#)
- [histogram](#)
- [max](#)
- [min](#)
- [nested](#)
- [pair](#)
- [sum](#)
- [term](#)
- [timeslice](#)
- [top_hits](#)
- [trend](#)
- [topic](#)
- [unique_count](#)

For Document Retrieval project types, when you don't include an aggregation parameter in a query request, a default aggregation request is applied. For more information, see *Document Retrieval project aggregations*.

For more information about how to submit a query, see the Discovery [API reference](#).

average

Returns the mean of values of the specified field across all matching documents.

Syntax

average(field)

Example

Product	Price
I Series	200
J Series	450
X Series	325

Table 1. Sample product prices

When the **average** aggregation type is applied to a set of documents in which the **price** field contains the values that are shown in Table 1, the result is **325**.

average(price)=325

This aggregation type performs a terminal operation. When combined with other aggregations, the output is not used as input for the next aggregation. The output is returned in a discrete group.

```
{
  "query": "brake",
  "aggregation": "term(field:STATE,count:3).average(field:VEH_SPEED).term(field:CITY,count:2)"
}
```

For each state returned by the first **term** aggregation operation, the response shows the average vehicle speed specified in the incident reports. Notice that the second **term** aggregation uses the output from the first **term** aggregation, not the **average** aggregation, as its input.

```
{
  "matching_results": 9064,
  "retrieval_details": {
    "document_retrieval_strategy": "untrained"
  },
  "aggregations": [
    {
      "type": "term",
      "field": "STATE",
      "count": 3,
      "results": [
        {
          "key": "CA",
          "matching_results": 1210,
          "aggregations": [
            {
              "type": "average",
              "field": "VEH_SPEED",
              "value": 26.239653512993264
            },
            {
              "type": "term",
              "field": "CITY",
              "count": 2,
              "results": [
                {
                  "key": "LOS ANGELES",
                  "matching_results": 77
                },
                {
                  "key": "SAN DIEGO",
                  "matching_results": 66
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

filter

A modifier that narrows the document set of the aggregation query that it precedes.

Syntax

```
filter(field)
```

Example

The following example filters the matching document set to include only documents that mention `IBM`.

```
filter(enriched_text.entities.text:IBM)
```

When combined with other aggregations, filters the matching documents set to include only those documents that meet the condition you specify.

```
{
  "query":"brake",
  "aggregation": "filter(VEH_SPEED>50).term(field:STATE,count:3).term(field:CITY,count:2)"
}
```

The query response shows cities where incidents happen that involve the brakes and the vehicle speed is over 50.

```
{
  "matching_results": 9064,
  "retrieval_details": {
    "document_retrieval_strategy": "untrained"
  },
  "aggregations": [
    {
      "type": "filter",
      "match": "VEH_SPEED>50",
      "matching_results": 1075,
      "aggregations": [
        {
          "type": "term",
          "field": "STATE",
          "count": 3,
          "results": [
            {
              "key": "CA",
              "matching_results": 176,
              "aggregations": [
                {
                  "type": "term",
                  "field": "CITY",
                  "count": 2,
                  "results": [
                    {
                      "key": "FONTANA",
                      "matching_results": 6
                    },
                    {
                      "key": "ALTA LOMA",
                      "matching_results": 5
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

group_by

Separates results into groups that you define.

Syntax

```
group_by(condition:[(condition 1),(condition 2)...])
```

Each condition must be specified as a valid Discovery Query Language expression surrounded by parentheses. For example, `(age<20)` or `(flavor:chocolate)`. The maximum number of conditions that you can define is 50.

You can optionally include the `relevancy` parameter and set it to `true` to return the relevancy value of the set of documents that meet the specified condition. When `true`, the results are sorted by relevance. When `false`, the results are sorted by the highest number of `matching_results`.

Example

The following request looks for documents that mention the term `engine`, and groups them by car manufacturing year. The documents are sorted into 3 groups, one group of traffic incident reports involving cars that were manufactured before 2000, one group for cars manufactured in 2000, and one group for cars manufactured after 2000.

```
{
  "query":"engine",
  "aggregation": "group_by(condition:[(YEARTXT<2000),(YEARTXT=2000),(YEARTXT>2000)],relevancy:true)"
}
```

The results might look like this:

```
{
  "type": "group_by",
  "results": [
    {
      "key": "YEARTXT<2000",
      "matching_results": 2034,
      "relevancy": 1.0,
      "total_matching_documents": 2034,
      "estimated_matching_results": 2034
    },
    {
      "key": "YEARTXT=2000",
      "matching_results": 1738,
      "relevancy": 1.0,
      "total_matching_documents": 1738,
      "estimated_matching_results": 1738
    },
    {
      "key": "YEARTXT>2000",
      "matching_results": 32708,
      "relevancy": 1.0,
      "total_matching_documents": 32708,
      "estimated_matching_results": 32708
    }
  ]
}
```

histogram

Creates numeric interval segments to categorize documents.

Syntax

```
histogram({field},{interval})
```

Uses field values from a single numeric field to describe the category. The field that is used to create the histogram must have a number data type, such as `integer`, `float`, `double`, or `date`.

Nonnumber types such as `string` are not supported. For example, `"price": 1.30` is a number value that works, and `"price": "1.30"` is a string, so it doesn't work.

Use the `interval` argument to define the size of the sections for the results to be split into. Interval values must be whole, nonnegative numbers. Choose a value that makes sense for segmenting the typical values from the field.

Histograms can process decimal values that are specified in a field, but the interval must be a whole number.

You can optionally include a custom name by including a `name` parameter.

Example

For example, if your data set includes the price of several items, like: `"price": 1.30`, `"price": 1.99`, and `"price": 2.99`, you might use intervals of `1`, so that you see everything that is grouped in the range `1 - 2`, and `2` and `3`. You do not want to use an interval of `100` because then all of the data ends up in the same segment.

```
histogram(product_price,interval:1)
```

max

Returns the highest value in the specified field across all matching documents.

Syntax

```
max(field)
```

Example

Product	Price
I Series	200
J Series	450
X Series	325

Table 2. Sample product prices

When the `max` aggregation type is applied to a set of documents in which the `price` field contains the values that are shown in Table 2, the result is `450`.

```
max(price)=450
```

This aggregation type performs a terminal operation. When combined with other aggregations, the output is not used as input for the next aggregation. The output is returned in a discrete group.

min

Returns the lowest value in the specified field across all matching documents.

Syntax

```
min(field)
```

Example

Product	Price
I Series	200
J Series	450
X Series	325

Table 3. Sample product prices

When the `min` aggregation type is applied to a set of documents in which the `price` field contains the values that are shown in Table 3, the result is `200`.

```
min(price)=200
```

This aggregation type performs a terminal operation. When combined with other aggregations, the output is not used as input for the next aggregation. The output is returned in a discrete group.

nested

Applying `nested` before an aggregation query restricts the aggregation to the area of the results that are specified.

For example, `nested(enriched_text.entities)` means that only the `enriched_text.entities` components of any result are used to aggregate against.

The following example checks how many mentions are returned per model type.

```
nested(enriched_text.entities).term(enriched_text.entities.model_name)
```

The result shows that there are a total of 50 recognized entities and all of them are of type NLU.

```
"aggregations": [
```

```

{
  "type": "nested",
  "path": "enriched_text.entities",
  "matching_results": 50,
  "aggregations": [
    {
      "type": "term",
      "field": "enriched_text.entities.model_name",
      "results": [
        {
          "key": "natural_language_understanding",
          "matching_results": 50
        }
      ]
    }
  ]
}
]

```

For another example, see [Starting with nested objects](#).

pair

Analyzes relationships between two fields.

Syntax

```
pair(first:{aggregation},second:{aggregation})
```

The first and second `{aggregation}` values must be one of the following aggregation types:

- `term`
- `group_by`
- `histogram`
- `timeslice`

The `relevancy` parameter from the `term` or `group_by` aggregation is ignored. The `pair` aggregation type calculates relevancy values by using combinations of document sets from the results of the two aggregations.

Only one pair aggregation can be used per query request, and it cannot be combined with any other aggregations.

Example

For example, you might specify `term(model_name)` as the first aggregation and `term(component_name)` as the second. Each of the aggregations returns the following values as keys of aggregated document sets:

- `term(model_name)`: Accord, CR-V
- `term(component_name)`: engine, brake, radiator

The calculated relevancy values of combinations of each of the document sets might look like this:

- Accord x engine
- Accord x brake
- Accord x radiator
- CR-V x engine
- CR-V x brake
- CR-V x radiator

The response defines a two-dimensional array of aggregation results, which can be represented in a table.

Car model	Component: engine	Component: brake	Component: radiator
Accord	Accord x engine	Accord x brake	Accord x radiator
CR-V	CR-V x engine	CR-V x brake	CR-V x radiator

Table 4. Pair aggregation example

Each array of columns and rows of the table is sorted in the same order of the results of the first and second aggregations. For example, if you specify the `term` aggregation as the first argument, the resulting column arrays are sorted by frequency of terms. If you use the `timeslice` aggregation as the second argument, the row arrays are sorted by date or time.

sum

Adds the values of the specified field across all matching documents.

Syntax

```
sum(field)
```

Example

Product	Price
I Series	200
J Series	450
X Series	325

Table 6. Sample product prices

When the `sum` aggregation type is applied to a set of documents in which the `price` field contains the values that are shown in Table 6, the result is `975`.

```
sum(price)=975
```

This aggregation type performs a terminal operation. When combined with other aggregations, the output is not used as input for the next aggregation. The output is returned in a discrete group.

term

Indicates the frequency of a term or set of terms in a set of queried documents.

Syntax

```
term(field:{ field_name})
```

You can optionally specify the following parameters:

- `count`: Specifies the maximum number of terms to return.
- `name`: You can optionally include a custom name. Not returned if relevancy information is included in the request.
- `relevancy`: Boolean value that indicates whether to include relevancy information in the result. You can use relevancy to get a score that indicates the level of relevancy between the term and keywords in the query. This parameter is `false` by default. If set to true, the following fields are returned also:
 - `total_matching_documents`: Number of documents in the collection where the term is mentioned in the specified field.
 - `estimated_matching_results`: Number of documents that are estimated to have the term in the specified field in the set of documents that are returned by the query.

Example

The following example returns the text from the recognized entities in the document, and specifies to return a maximum of 10 terms.

For example:

```
term(enriched_text.entities.text,count:10)
```

When `relevancy` is set to `true`, a relevancy score is shown in the results. Relevancy measures the level of uniqueness of the frequency count compared to other documents that match your query. If the relevancy shows 2.0, it means that the number of times that the two data points intersect is 2 times larger than expected.

For more examples, see [Grouping documents](#) and [Combining aggregation types](#).

timeslice

A specialized histogram that uses dates to create interval segments.

Syntax

The syntax is `timeslice({field},{interval},{time_zone})`.

- The field that you specify must have a **date** data type. For more information about date field, see [How dates are handled](#).
- Valid interval values are **1second** or **{n}seconds**, **1minute** or **{n}minutes**, **1hour** or **{n}hours**, **1day** or **{n}days**, **1week** or **{n}weeks**, **1month** or **{n}months**, and **1year** or **{n}years** where {n} is a number.
- You can optionally include a custom name by including a **name** parameter.

Example

The following example shows the number of matches for each day value.

```
timeslice(field:DATEA,interval:1day)
```

The results look as follows.

```
"aggregations": [
  {
    "type": "timeslice",
    "field": "DATEA",
    "interval": "1d",
    "results": [
      {
        "key": 1262304000000,
        "key_as_string": "2010-01-01T00:00:00.000Z",
        "matching_results": 5
      },
      {
        "key": 1262390400000,
        "key_as_string": "2010-01-02T00:00:00.000Z",
        "matching_results": 18
      },
      {
        "key": 1262476800000,
        "key_as_string": "2010-01-03T00:00:00.000Z",
        "matching_results": 38
      },
      {
        "key": 1262563200000,
        "key_as_string": "2010-01-04T00:00:00.000Z",
        "matching_results": 66
      }
    ]
  }
]
```

top_hits

Returns the documents ranked by the score of the query or enrichment. Can be used with any query parameter or aggregation.

Syntax

```
{aggregation}.top_hits({n})
```

Example

The following example returns the top hit for the term **halt** per city.

```
{
  "query":"halt",
  "aggregation": "term(CITY).top_hits(1)"
}
```

The response contains the top query results for the term **halt** grouped by cities mentioned in documents where the term is most mentioned. Ten results are returned by default. For each of the 10 cities, the document with the top score is returned as the **hit** object. The content for each **hit** in the **hits** array matches the content in each **result** in the **results** array. Only the order of the results is different.

```
"aggregations": [
  {
    "type": "term",
    "field": "CITY",
    "results": [
      {
        "key": "LOS ALTOS",
        "matching_results": 3,
```

```

    "aggregations": [
      {
        "type": "top_hits",
        "size": 1,
        "hits": {
          "matching_results": 3,
          "hits": [
            {
              "document_id": "2bed19a9069442fd82542827ebe260d5_7015",
              ...
            }
          ]
        }
      }
    ]
  },
  {
    "key": "ANDOVER",
    "matching_results": 2,
    "aggregations": [
      {
        "type": "top_hits",
        "size": 1,
        "hits": {
          "matching_results": 2,
          "hits": [
            {
              "document_id": "2bed19a9069442fd82542827ebe260d5_18329",
              ...
            }
          ]
        }
      }
    ]
  },
  ...
  {
    "key": "ACTON",
    "matching_results": 1,
    "aggregations": []
  }
  ...

```

This aggregation type performs a terminal operation. When combined with other aggregations, the output is not used as input for the next aggregation. The output is returned in a discrete group.

trend

Detects sharp and unexpected changes in the frequency of a keyword value in a specified time period based on the past frequency changes of the keyword value.

Syntax

```
trend(facet:{aggregation},time_segments:{aggregation})
```

The first (**facet**) aggregation must be one of the following types of aggregations:

- **term**
- **group_by**

The **relevancy** parameter from the **term** or **group_by** aggregation is ignored.

The second (**time_segments**) aggregation must be an aggregation of type **timeslice**.

You can alternatively include the following parameters:

- **show_estimated_matching_results:true**: Indicates whether to include the **estimated_matching_results** information in the result. This field contains the number of documents that are estimated to have the term in the specified field or meet the conditions in the specified aggregation for the specified time interval in the set of documents that are returned by the query.
- **show_total_matching_documents:true**: Indicates whether to include the **total_matching_documents** information in the result. This field contains the number of documents in the collection where the term is mentioned in the specified field or the condition is met.

Only one trend aggregation can be used per query request, and it cannot be combined with any other aggregations.

Example

The following example calculates the *trend indicator* or *trend index* by using combinations of results from the following aggregations:

- term(flavor): vanilla, chocolate, mint
- timeslice(date, 1month): Jan 2020, Feb 2020, Mar 2020, Apr 2020, May 2020, Jun 2020

```
trend( facet: aggregation(<parameter>...), time_segments: timeslice(<parameter>...)),
show_estimated_matching_results: <true_or_false>, show_total_matching_documents: <true_or_false> )
```

The resulting matrix can be represented in a table.

Month in 2020	Flavor: vanilla	Flavor: chocolate	Flavor: mint
Jan	vanilla x Jan	chocolate x Jan	mint x Jan
Feb	vanilla x Feb	chocolate x Feb	mint x Feb
Mar	vanilla x Mar	chocolate x Mar	mint x Mar
Apr	vanilla x Apr	chocolate x Apr	mint x Apr
May	vanilla x May	chocolate x May	mint x May
Jun	vanilla x Jun	chocolate x Jun	mint x Jun

Table 5. Trend aggregation example

In the following sample response, the key information is the `trend_indicator` value. The trend indicator measures the increase ratio of the frequency of a given facet value for a given time interval compared to the expected average frequency. The expected average frequency is calculated based on the changes in the past time interval frequencies of the given facet value, using a weighted arithmetic mean.

If the standardized residual value is less than -2, the observed frequency is less than the expected frequency. If it is greater than 2, the observed frequency is greater than the expected frequency. If the standardized residual is greater or less than the expected frequency by 3 or more, then something unusual is happening and suggests that there might be an anomaly that is worth investigating.

For example, the expected number of feedback submissions for the `vanilla` flavor in May is calculated from the number of feedback submissions that were received previously (from Jan to Apr). The result is `5.341`. The actual number of feedback submissions in May is `10`. The results indicate that the vanilla flavor got about twice the number of feedback submissions as expected. The standardized residual value is `2.016`, which is greater than expected, but not unusually so.

```
{
  "aggregations": [
    {
      "type": "trend",
      "facet": "term(flavor)",
      "time_segments": "timeslice(date, 1month)",
      "show_estimated_matching_results": true,
      "show_total_matching_documents": true,
      "results": [
        {
          "aggregations": [
            {
              "type": "term",
              "field": "flavor",
              "results": [
                {
                  "key": "vanilla",
                  "matching_results": 36,
                  "aggregations": [
                    {
                      "type": "timeslice",
                      "field": "date",
                      "results": [
                        {
                          "key": 1577836800000,
                          "key_as_string": "2020-01-01T00:00:00.000Z",
                          "matching_results": 4,
                          "trend_indicator": 0.0,
                          "total_matching_documents": 7,
                          "estimated_matching_results": 0.0
                        },
                        {
                          "key": 1588291200000,
                          "key_as_string": "2020-05-01T00:00:00.000Z",
                          "matching_results": 10,
                          "trend_indicator": 2.016,
                          "total_matching_documents": 7,
                          "estimated_matching_results": 5.341
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

```

        "matching_results": 10,
        "trend_indicator": 2.016106745,
        "total_matching_documents": 12,
        "estimated_matching_results": 5.340760209
      },
      {
        "key": 1590969600000,
        "key_as_string": "2020-06-01T00:00:00.000Z",
        "matching_results": 5,
        "trend_indicator": -0.763212711,
        "total_matching_documents": 11,
        "estimated_matching_results": 7.022515985
      }
    ]
  },
  {
    "key": "chocolate",
    "matching_results": 10,
    "aggregations": [...]
  },
  {
    "key": "mint",
    "matching_results": 25,
    "aggregations": [...]
  }
  ...
}

```

topic

Detects how much the frequency of a keyword value deviates from the expected average for the specified time period. This aggregation type does not use data from previous time periods. It calculates an index by using the averages of frequency counts of other keyword values for the specified time period.

Syntax

```
topic(facet:{aggregation},time_segments:{aggregation})
```

The first (**facet**) aggregation must be one of the following types of aggregations:

- **term**
- **group_by**

The **relevancy** parameter from the **term** or **group_by** aggregation is ignored.

The second (**time_segments**) aggregation must be an aggregation of type **timeslice** .

You can alternatively include the following parameters:

- **show_estimated_matching_results:true**: Indicates whether to include the **estimated_matching_results** information in the result. This field contains the number of documents that are estimated to have the term in the specified field or meet the conditions in the specified aggregation for the specified time interval in the set of documents that are returned by the query.
- **show_total_matching_documents:true**: Indicates whether to include the **total_matching_documents** information in the result. This field contains the number of documents in the collection where the term is mentioned in the specified field or the condition is met.

Only one topic aggregation can be used per query request, and it cannot be combined with any other aggregations.

Example

```

{
  "query": "like",
  "aggregation": "topic( facet: term(flavor), time_segments: timeslice(date, 1month), show_estimated_matching_results: true, show_total_matching_documents: true )"
}

```

With the same data set and aggregation as is used in the term aggregation example, the results might look as follows.

Notice that the **topic_indicator** values are different from the **trend_indicator** values that are returned by the **trend** aggregation. While both are calculated from the actual and expected frequencies, they differ because their expected frequencies are computed differently. In the **trend** aggregation, the expected frequency of the feedback submissions for vanilla-flavored ice cream in May is computed from the number of feedback submissions that were received for vanilla previously (from Jan to Apr) and the total number of feedback submissions received for all of the flavors in May. However, in the **topic** aggregation, the expected frequency of feedback submissions for vanilla-flavored ice cream in May is calculated from the number of feedback submissions that were received for vanilla and the total number of feedback submissions received for all of the flavors in May. In this example, the expected frequency

result is **12.169**, the actual frequency is **10**, and the **topic_indicator** is **-0.621777032**.

```
{
  "aggregations": [
    {
      "type": "topic",
      "facet": "term(flavor)",
      "time_segments": "timeslice(date, 1month)",
      "show_estimated_matching_results": true,
      "show_total_matching_documents": true,
      "results": [
        {
          "aggregations": [
            {
              "type": "term",
              "field": "flavor",
              "results": [
                {
                  "key": "vanilla",
                  "matching_results": 36,
                  "aggregations": [
                    {
                      "type": "timeslice",
                      "field": "date",
                      "results": [
                        {
                          "key": 1577836800000,
                          "key_as_string": "2020-01-01T00:00:00.000Z",
                          "matching_results": 4,
                          "topic_indicator": -0.027972712,
                          "total_matching_documents": 7,
                          "estimated_matching_results": 4.056338028
                        },
                        {
                          "key": 1588291200000,
                          "key_as_string": "2020-05-01T00:00:00.000Z",
                          "matching_results": 10,
                          "topic_indicator": -0.621777032,
                          "total_matching_documents": 12,
                          "estimated_matching_results": 12.16901408
                        },
                        {
                          "key": 1590969600000,
                          "key_as_string": "2020-06-01T00:00:00.000Z",
                          "matching_results": 5,
                          "topic_indicator": -0.787665504,
                          "total_matching_documents": 11,
                          "estimated_matching_results": 7.098591549
                        }
                      ]
                    }
                  ]
                }
              ]
            },
            {
              "key": "chocolate",
              ...
            },
            {
              "key": "mint",
              ...
            }
          ]
        }
      ]
    }
  ]
}
```

unique_count

Returns a count of the unique instances of the specified field in the collection.

Syntax

```
unique_count(field)
```

Example

The following aggregation requests the number of unique enrichment types that are recognized in the query.

```
unique_count(enriched_text.keyword.type)
```

The result indicates that there are 17 matching results. In those 17 documents, 14 entity types are mentioned.

```
{
  "matching_results": 17,
  "retrieval_details": {
    "document_retrieval_strategy": "untrained"
  },
  "aggregations": [
    {
      "type": "unique_count",
      "field": "enriched_text.entities.type",
      "value": 14.0
    }
  ],
  "results": []
}
```

This aggregation type performs a terminal operation. When combined with other aggregations, the output is not used as input for the next aggregation. The output is returned in a discrete group.

In the following example, the aggregation parameter requests for the results to show the first 45 most-frequently mentioned entities. Per entity, it indicates how many documents mention the term and how many times in total that the term occurs.

```
term(enriched_text.entities.text,count:45).unique_count(enriched_text.entities.type)
```

The results include several aggregations such as the following group for the term **PostgreSQL**. The aggregation indicates that the term appears in 4 documents and is mentioned 12 times.

```
{
  "key": "PostgreSQL",
  "matching_results": 4,
  "aggregations": [
    {
      "type": "unique_count",
      "field": "enriched_text.entities.type",
      "value": 12.0
    }
  ]
}
```
