


How to work with search results in Azure Cognitive Search

04/01/2020 • 6 minutes to read •  +6

In this article

[Result composition](#)

[Paging results](#)

[Ordering results](#)

[Hit highlighting](#)

[Next steps](#)


This article explains how to get a query response that comes back with a total count of matching documents, paginated results, sorted results, and hit-highlighted terms.

The structure of a response is determined by parameters in the query: [Search Document](#) in the REST API, or [SearchResults Class](#) in the .NET SDK.

Result composition

While a search document might consist of a large number of fields, typically only a few are needed to represent each document in the result set. On a query request, append `$select=<field list>` to specify which fields show up in the response. A field must be attributed as **Retrievable** in the index to be included in a result.

Fields that work best include those that contrast and differentiate among documents, providing sufficient information to invite a click-through response on the part of the user. On an e-commerce site, it might be a product name, description, brand, color, size, price, and rating. For the hotels-sample-index built-in sample, it might be fields in the following example:

HTTP	 Copy

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
  "search": "sandy beaches",
  "select": "HotelId, HotelName, Description, Rating, Address/City"
  "count": true
}
```

ⓘ Note

If want to include image files in a result, such as a product photo or logo, store them outside of Azure Cognitive Search, but include a field in your index to reference the image URL in the search document. Sample indexes that support images in the results include the **realestate-sample-us** demo, featured in this [quickstart](#), and the **New York City Jobs demo app**.


Paging results

By default, the search engine returns up to the first 50 matches, as determined by search score if the query is full text search, or in an arbitrary order for exact match queries.

To return a different number of matching documents, add `$top` and `$skip` parameters to the query request. The following list explains the logic.

- Add `$count=true` to get a count of the total number of matching documents within an index.
- Return the first set of 15 matching documents plus a count of total matches: `GET /indexes/<INDEX-NAME>/docs?search=<QUERY STRING>&$top=15&$skip=0&$count=true`
- Return the second set, skipping the first 15 to get the next 15: `$top=15&$skip=15`. Do the same for the third set of 15: `$top=15&$skip=30`


The results of paginated queries are not guaranteed to be stable if the underlying index is changing. Paging changes the value of `$skip` for each page, but each query is independent and operates on the current view of the data as it exists in the index at query time (in other words, there is no caching or snapshot of results, such as those found in a general purpose database). Following is an example of how you might get duplicates. Assume an index with four documents:

text	 Copy
<pre>{ "id": "1", "rating": 5 } { "id": "2", "rating": 3 } { "id": "3", "rating": 2 } { "id": "4", "rating": 1 }</pre>	

Now assume you want results returned two at a time, ordered by rating. You would execute this query to get the first page of results: `$top=2&$skip=0&$orderby=rating desc`, producing the following results:

text	 Copy
<pre>{ "id": "1", "rating": 5 } { "id": "2", "rating": 3 }</pre>	

On the service, assume a fifth document is added to the index in between query calls: `{ "id": "5", "rating": 4 }`. Shortly thereafter, you execute a query to fetch the second page: `$top=2&$skip=2&$orderby=rating desc`, and get these results:

text	 Copy
<pre>{ "id": "2", "rating": 3 } { "id": "3", "rating": 2 }</pre>	

Notice that document 2 is fetched twice. This is because the new document 5 has a greater value for rating, so it sorts before document 2 and lands on the first page. While this behavior might be unexpected, it's typical of how a search engine behaves.

Ordering results

For full text search queries, results are automatically ranked by a search score, calculated based on term frequency and proximity in a document, with higher scores going to documents having more or stronger matches on a search term.

Search scores convey general sense of relevance, reflecting the strength of match as compared to other documents in the same result set. Scores are not always consistent from one query to the next, so as you work with queries, you might notice small discrepancies in how search documents are ordered. There are several explanations for why this might occur.

Cause	Description
Data volatility	Index content varies as you add, modify, or delete documents. Term frequencies will change as index updates are processed over time, affecting the search scores of matching documents.
Multiple replicas	For services using multiple replicas, queries are issued against each replica in parallel. The index statistics used to calculate a search score are calculated on a per-replica basis, with results merged and ordered in the query response. Replicas are mostly mirrors of each other, but statistics can differ due to small differences in state. For example, one replica might have deleted documents contributing to their statistics, which were merged out of other replicas. Typically, differences in per-replica statistics are more noticeable in smaller indexes.
Identical scores	If multiple documents have the same score, any one of them might appear first.

Consistent ordering

Given the flex in results ordering, you might want to explore other options if consistency is an application requirement. The easiest approach is sorting by a field value, such as rating or date. For scenarios where you want to sort by a specific field, such as a rating or date, you can explicitly define an [\\$orderby expression](#), which can be applied to any field that is indexed as **Sortable**.

Another option is using a [custom scoring profile](#). Scoring profiles give you more control over the ranking of items in search results, with the ability to boost matches found in specific fields. The additional scoring logic can help override minor differences among replicas because the search scores for each document are farther apart. We recommend the [ranking algorithm](#) for this approach.

Hit highlighting



Hit highlighting refers to text formatting (such as bold or yellow highlights) applied to matching terms in a result, making it easy to spot the match. Hit highlighting instructions are provided on the [query request](#).

To enable hit highlighting, add `highlight=[comma-delimited list of string fields]` to specify which fields will use highlighting. Highlighting is useful for longer content fields, such as a description field, where the match is not immediately obvious. Only field definitions that are attributed as **searchable** qualify for hit highlighting.

By default, Azure Cognitive Search returns up to five highlights per field. You can adjust this number by appending to the field a dash followed by an integer. For example, `highlight=Description-10` returns up to 10 highlights on matching content in the Description field.

Formatting is applied to whole term queries. The type of formatting is determined by tags, `highlightPreTag` and `highlightPostTag`, and your code handles the response (for example, applying a bold font or a yellow background).

In the following example, the terms "sandy", "sand", "beaches", "beach" found within the Description field are tagged for highlighting. Queries that trigger query expansion in the engine, such as fuzzy and wildcard search, have limited support for hit highlighting.

HTTP	 Copy
<code>GET /indexes/hotels-sample-index/docs/search=sandy beaches&highlight=Description?api-version=2020-06-30</code>	
HTTP	 Copy


```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
  "search": "sandy beaches",
  "highlight": "Description"
}
```

New behavior (starting July 15)

Services created after July 15, 2020 will provide a different highlighting experience. Services created before that date will not change in their highlighting behavior.

With the new behavior:

- Only phrases that match the full phrase query will be returned. The query "super bowl" will return highlights like this:

HTML	 Copy
'super bowl is super awesome with a bowl of chips'	

Note that the term *bowl of chips* does not have any highlighting because it does not match the full phrase.

When you are writing client code that implements hit highlighting, be aware of this change. Note that this will not impact you unless you create a completely new search service.

Next steps

To quickly generate a search page for your client, consider these options:

- [Application Generator](#), in the portal, creates an HTML page with a search bar, faceted navigation, and results area that includes images.

- [Create your first app in C#](#) is a tutorial that builds a functional client. Sample code demonstrates paginated queries, hit highlighting, and sorting.

Several code samples include a web front-end interface, which you can find here: [New York City Jobs demo app](#), [JavaScript sample code with a live demo site](#), and [CognitiveSearchFrontEnd](#).

Is this page helpful?

 Yes  No
