amazon web services

Amazon CloudSearch

A fully managed, auto-scaling search service in the cloud

Get Started

Activate your Amazon CloudSearch free 30-day trial today.
http://aws.amazon.com/cloudsearch/free-trial

DZone Refcardz

Brought to you by:
amazon web services

CONTENTS INCLUDE:

Search Patterns

with Amazon CloudSearch

By: Jon Handler

Search is no longer just about finding content on the web. It's about searching proprietary data sets and displaying perfect matches that drive traffic, increase revenue, or reduce costs. This Refcard provides patterns for integrating cloud-based search with a variety of applications. Examples of these patterns are demonstrated using Amazon CloudSearch to abstract away the complexities of deploying and administering your own search servers, but the patterns apply to any powerful search system.

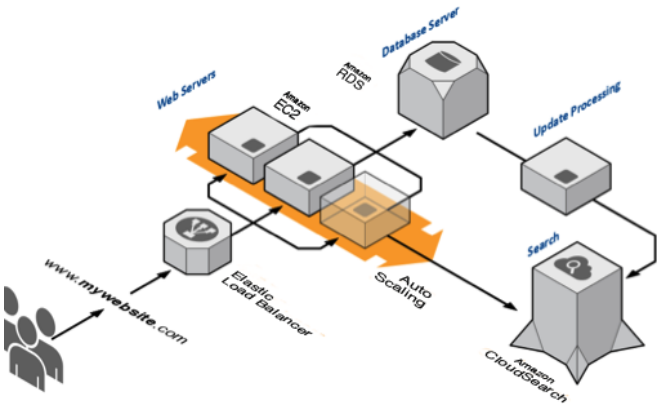
About Amazon CloudSearch

CloudSearch is a fully-managed service in the cloud that makes it easy to set up, manage, and scale a search solution. When you use CloudSearch, you create a search domain and the necessary hardware and software is automatically provisioned and deployed in the cloud. Each domain has a document endpoint and a search endpoint that persist for the life of the domain. You use RESTful APIs to POST documents to your domain and GET search results and CloudSearch takes care of the rest.

BUILDING SEARCH FROM THE GROUND UP

To build a search-centric application or website, you need to implement both a data pipeline and a query pipeline. On the data side, you **structure** your data, **upload** it to the search engine, and **configure** the search engine to parse and index your data. On the query side, you create a **user interface** that **translates** user input into queries against the search engine, **parses** the responses, and displays search results to the user. To deliver consistently good search results, you **tune** how the search engine computes the relevance of each match.

The following diagram shows how CloudSearch is integrated with a typical web application:



As user requests come in, they are load-balanced across an auto-scaled group of web servers. These servers manage the website's business and display logic and access data from a data store such as a database in the Amazon Relational Database Service. The web server sends search requests to the CloudSearch domain and constructs and displays pages of results. An additional server (or server fleet) collects updates from the database and uploads them to the CloudSearch domain.

Data Preparation and Upload

To use a search engine, you must structure your data as individual documents. For example, an eCommerce site like Amazon.com represents each product as a separate search document. A search query retrieves a list of products by finding all of the matching documents.

In CloudSearch, search documents are structured according to the Search Data Format (SDF). For example, the following JSON snippet shows an SDF batch of search documents. (You can also represent SDF batches in XML.)

```
[{"type":"add",
  "id": "b007oznzg0",
  "version": 1,
  "lang": "en",
  "fields": {
      "title":"Kindle Paperwhite",
      "description":"World's most advanced e-reader",
      "category": ["Electronics","eBook Readers"],
      "price":11900
} }, ...]
```

In JSON, each SDF batch is an array, [...]. In this example, the batch contains a single search document, which is represented as a JSON object, {...}.

Each document has a unique **id** and a **version number** that CloudSearch uses to sequence your updates. CloudSearch serves the highest version of a document that you have sent, in an eventually consistent manner. The version number is for sequencing updates only—you cannot retrieve the version number, search, or sort on it. A best practice for setting the version number when adding and deleting documents is to use the system clock expressed as a Unix epoch value, which provides an always-increasing integer.

Each SDF document also has a **collection of fields** that contain the document data. These named entities provide the basis for matching user queries and computing relevance. A field can have a single value or multiple values.

Element	Description
type	The operation type, *add* or *delete*. Use *add* to add new documents and update existing documents. Use *delete* to remove documents.
id	A globally unique string that identifies the document. Document IDs must begin with a letter or numeral, can contain the characters a-z (lower-case letters), 0-9, and _ (underscore), and must be at least 1 and no more than 128 characters long.
version	A positive integer for sequencing adds and deletes.
lang	English (*en*) is the only supported language.
fields	A set of named values. Field names must begin with a letter, can contain the characters a-z (lower case), 0-9, and _ (underscore, and must be at least 3 and no more than 64 characters.

You send document batches to a search domain's document endpoint using HTTP POST requests. The documents in the batch are indexed and become searchable in a few seconds to a few minutes. As you send updates, CloudSearch seamlessly adjusts your deployment to accommodate the volume of data in your domain.

Indexing

To quickly identify matches, search engines transform document field data into an **inverted index** that maps specific terms to the documents that contain them (much like the index of a book maps terms to the pages where they occur). How this transformation is carried out depends on the index schema and the text processing algorithms the search engine applies to each word. In CloudSearch, you can define your schema through the AWS Management Console for Amazon CloudSearch, or with the command line tools or Configuration API. CloudSearch supports three field types: text, literal, and unsigned integer (unit).

Field Type	Processing Applied
Text	Tokenization, Downcasing, Accent Folding, Stemming, Stopwords, Synonyms
Literal	Downcasing
Unit	Stored

Building Queries

You send a query to a search engine to get the set of documents that match particular search criteria, ranked according to how relevant the documents are to those criteria. Query criteria can include:

- Text that a user types in a search box
- Individual values that the user specifies by selecting UI elements
- Access restrictions you include based on the user's identity
- Other application-specific search criteria

To search a CloudSearch domain, you send a query to its search endpoint and specify the search criteria as URL parameters.

You use the q parameter to perform full-text matching. For example, specifying `q=harry` matches documents that have text fields that contain the word *harry*.

You use the *bq* parameter to build more complex queries using Boolean expressions.

For example, specifying `bq=(and 'harry' category:'books' year: 2000..` matches documents that contain the word *harry* in any *text* field, have the value *books* in their *category* field, and a value greater than or equal to *2000* in their *year* field.

Parameter	Uses
q	Search all text fields for a string. Example: `q=star` Results: Matches *Star Wars* as well as *A Star is Born.*
bq	Search particular fields of any type or all text fields. Use Boolean logic to define search criteria. Example: `bq=(and 'harry' category:'books)` Results: Matches documents have the word *harry* in any text field and books in their category field.

For example, if you populate a CloudSearch domain with the IMDb movies sample data set, the following HTTP GET request matches all movies that contain the words star wars:

```
http://<domain's search endpoint>/2011-02-01/search?q=star+wars
```

By default, CloudSearch returns the response in JSON (you can also get the response in XML):

```
{ "rank": "-text_relevance",
  "match-expr": "(label 'star wars')",
  "hits": {
    "found": 7,
    "start": 0,
    "hit": [{"id": "tt1185834"},
            {"id": "tt0076759"},
            {"id": "tt0086190"},
            {"id": "tt0120915"},
            {"id": "tt0121765"},
            {"id": "tt0080684"},
            {"id": "tt0121766"} ]
  },
  "info": {
    "time-ms": 4,
    "cpu-time-ms": 0
  }
}
```

[**Note**: To try this example out, you can easily configure and populate your own domain with the IMDb movies sample data through the **Create Domain Wizard** in the AWS Management Console for Amazon CloudSearch.]

The response to a search query specifies the rank function used to compute the relevance of each match, the logical expression (match-expr) used to identify matching documents, a collection of hits that contains the document ids of the ranked matches, and information about the time it took to process the request.

Ranking

Search engines figure out which documents match the search criteria, but not all matches are equal—which match is "best" depends on application context and user intent. For example, searching for *SaaS* on Wikipedia returns about 4000 matches, from the main page on software as a service, to pages that mention *SaaS* in passing, and the page *Kublis (Rhaetian Railway Station)* where *SaaS* is mentioned as a stop on a railroad line in the Swiss Alps.

To identify the best matches, search engines compute a score for each matching document and rank the documents based on those scores. By default, CloudSearch uses a rank function called *text_relevance* to determine how relevant each document is to the search criteria and sort the results with the most relevant documents listed first. You can also create custom rank functions that use this relevance score in combination with values in the matching documents. (The search patterns discussed later include some examples of custom rank functions.)

Updating

The documents that you initially upload to your domain are a snapshot of your system's data. As users interact with your application and the underlying data changes, you have to update your search domain to keep the search results current.

CloudSearch provides a near-real-time update system that's based on document IDs and version numbers. To update a document, you add a new

document that has the same document id and a higher version number by specifying `"type":"add"`.

When you *add* a document with a higher version number, the new version entirely replaces the old one. CloudSearch does not support updating individual fields (partial updates).

To *delete* a document, you specify `"type":"delete"` and include the document id and a higher version number than the existing document. This removes all fields and values for the document (although the document itself is not immediately purged from the system).

Update Stream Patterns
The most common patterns for creating and maintaining an update stream are:

- **Write an update directly** to the search domain in response to a user action. While this is simple to implement, it does not take advantage of **batching**. Batching updates results in significant performance improvements, so it's well worth the extra effort.
- **Use external queuing technology** to gather updates into batches and upload them to your search domain. For example, you can use a database to store a status flag and the primary key of the object you want to update and run a process that periodically sweeps the table, generates SDF batches, and uploads them to your domain.
- **Use a system like Amazon Simple Queuing Service** (Amazon SQS) to maintain a queue of updates. This method has the advantage of being simpler to scale when there is a large volume of updates. Amazon SQS synchronizes the access of many processes to the queue.

SEARCH PATTERNS

Title-Body Search

Wikipedia illustrates the Title-Body pattern for search. This pattern is characterized by documents that have a descriptive title and a body that contains the main content. On this type of search-enabled website, you typically interact with the search function by entering terms in a **search field**. When you click the Search button, the UI submits a search query with those terms as the search criteria. The search engine then matches those terms against the documents in the corpus and returns a ranked list of matching documents.

For example, if you enter *software as a service*, a CloudSearch query could specify those terms with the q parameter:

```
https://<domain's search endpoint>/2011-02-01/search?q=software as a
service&return-fields=title...
```

The CloudSearch response contains the first set of matching documents, sorted by their text relevance scores (in descending order).

```
{ "rank": "-text_relevance",
  "match-expr": "(label 'software as a service')",
  "hits": {
     "found": 29166,
     "start": 0,
     "hit": [
       { "id": "wikipedia21560276",
         "data": { "title": ["Software business"] }
       },
              { "id": "wikipedia22947099",
         "data": { "title": [
                    "Service-oriented software engineering" ] }
       },
     ... ]
    },
    ...
}
```

However, this isn't a very good set of results. Ideally, the main Software as a Service page should be at the top. What happened?

The CloudSearch text relevance score is based on two components: *tf-idf* and *proximity*. Tf-idf measures the frequency of the query terms across the whole document corpus and scales by the number of occurrences in each matching document. A match scores higher if it has many occurrences of infrequently-occurring terms. Proximity measures the proximity of the query terms in each matching document. A match scores higher if it has many occurrences of the query terms in close proximity and in the correct order.

When ranking documents that follow the Title-Body pattern, you usually want to boost the value of matches in the document title. If you're searching for *software as a service*, the best matches probably contain *software* and *service* in their titles. Boosting title matches brings these documents to the top of the results.

In CloudSearch, you can define custom rank functions statically using the console, command line tools, or API, or dynamically within the query itself. To define a rank function within a query, you use the *rank-<name>* parameter. To select the rank function you want use to rank matches, you use the *rank* parameter. For example:

```
https://search endpoint>/2011-02-01/search?q=software as a service&return-
fields=title&rank-title_boost=cs.text_relevance({weights:{"title":4.0}})&ra
nk=-title_boost
```

The `rank-title_boost= cs.text_relevance({weights:{"title":4.0}})` URL parameter defines a new rank function called *title_boost* that multiplies the relevance of title matches by 4. By boosting the title, we get better matches listed first, including the main software-as-a-service page right at the top:

```
{ "rank": "-title_boost",
  "match-expr": "(label 'software as a service')",
  "hits": {
     "found": 29166,
     "start": 0,
     "hit": [{ "id": "wikipedia2262333",
               "data": { "title": [ "Software as a service" ] }
             },
             {
               "id": "wikipedia29176559",
               "data": { "title": [
                 "Nsite Software (Platform as a Service)"] }
},...} ... }
```

Social Search Patterns

Many of today's applications have a social component that enables users to find and follow their friends and rate, review, and share content. These users expect search results to reflect their connections and contributions.

One of the ways that social data can influence search is through surfacing the most popular content. First, an application must **record** the user-generated popularity in the data source. For example, if your application supports "liking" a post, you increment a counter in the database each time a user likes a post.

Next, you need to add a corresponding *number_of_likes* field to your index schema. In CloudSearch, you configure this field as an integer. Each time a post's like counter is incremented, you update the corresponding document so its *number_of_likes* field reflects the current popularity.

To reflect the popularity in search results, you **build a rank function** for the domain that takes into account the number of likes and use that function to rank the results:

```
rank_likes=text_relevance+log10(number_of_likes)*50
```

When defining rank expressions, keep in mind that the text_relevance function computes a score in the range 0 to 1000. You often need to scale the contribution from document fields to avoid overwhelming the text relevance score. For example, if the *number_of_likes* field ranges into the millions, use a log or normalizing constant. In *rank_likes,* we take the base-10 log of the number of likes to scale it roughly from 0 to 6 and then multiply by 50 to put it in the range of about 0 to 300. This enables it to influence result ranking without dominating.

When tuning relevance for your application, gather some common and long-tail queries that you expect or are already receiving. You can use the CloudSearch *return-fields* parameter to get the computed text_relevance score or the computed value for any of your own rank expressions. For example:

```
q=star+wars&return-fields=text_relevance
```

This returns the computed relevance value for each document:

```
"rank": "-text_relevance",
  "match-expr": "(label 'star wars')",
  "hits": {"found": 7, "start": 0,
    "hit": [{
      "id": "tt1185834",
      "data": {"text_relevance": ["306"]}] ...
```

Mobile Search Patterns

Information about a user's immediate context can improve result relevance. For example, searching for *pizza parlor* in a location-aware application should find nearby pizza parlors. In CloudSearch, query-time rank expressions enable you to incorporate a user's current location into the rank function so you can boost matches that are located near the user.

First, you need to associate a location with each document. You can do this by transforming the location's latitude and longitude to unsigned integers and embedding those values with the document data (see the CloudSearch

```
"fields": {
      "name":"Joe's Pizzeria",
      "latitude":"12785",
      "longitude":"5751",
```

documentation for details):

You can use this embedded location information to perform bounding-box searches with integer range searches of the latitude and longitude fields. For example, you can search within a box that contains San Francisco with the query:

```
bq=(and 'pizzeria' latitude:12700..12900 longitude:5700..5800)
```

In CloudSearch, ranges can either be open- or closed-ended. 12700..12900 restricts matches to documents from 12700 to 12900, inclusive. To specify an open-ended range, you simply omit the upper or lower bound. For example, 1000 matches documents with values greater than or equal to 1000.

To sort results by distance from the user, embed the user's current location in a query time rank function and use it to sort the results. The Cartesian distance function provides a fast-to-compute sorting function for user queries:

$$\sqrt{((x - x_0)^2 + (y - y_o)^2)}$$

You can insert the user's current location as the (x_0,y_0) point and use the document's latitude and longitude as the (x,y) point in a rank function. For example:

```
rank-geo=sqrt(pow((userlat-latitude),2)+pow((userlon-longitude),2)&rank=geo
```

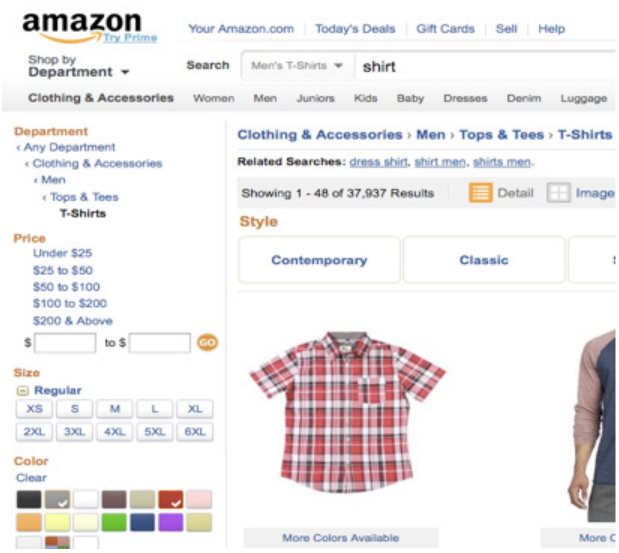
(Of course, you replace *userlat* and *userlon* with the user's actual latitude and longitude and URL-encode the query before submitting it.)

Note that the *rank* parameter specifies the *geo* function to sort in *ascending* order, closest first. In other examples, we've used a negative function (*-text_relevance*) to sort in *descending* order.

eCommerce Patterns

Search is a central component of eCommerce applications, responsible for connecting users with the products that they want to buy. In a typical eCommerce application, users narrow text-based search results using **faceted drilldown** to zero in on what they want.

A **facet** is a single attribute of a product, such as brand, size, or color. In a product search application, the UI typically shows the values for the facet below the facet name, often with a count of documents containing that value.



CloudSearch makes it easy to create this UI. First, add a field for the facet to each SDF document. Then, enable the Facet option for the field in the domain's indexing options. Through the AWS Management Console for Amazon CloudSearch, this is simple:

Now when you search, you can use the *facet* parameter to request facet counts for the field:

```
http://<search endpoint>/search?q=shirt&facet=color
```

In the response, the facet information is included after the hits:

```
{"rank": "-text_relevance",
 "match-expr": "(label 'shirt')",
 "hits": { "found": 389, "start": 0,
   "hit": [ ... ]
 },
 "facets": {
   "color": {
     "constraints": [
       { "value": "Black", "count": 30 },
       { "value": "White", "count": 28 },
       { "value": "One Color", "count": 14 },
     ] } }, ...
```

When a user selects a facet in the UI, you restrict the search results by adding the facet's value as a filter to the query:

```
http://<search endpoint>/search?bq=(and 'shirt' color:'White')
```

This query returns documents that match *shirt* and have the value *White* in the *color* field.

Sometimes facets comprise a hierarchical set of categories. For example, you might have a three-level hierarchical categorization scheme for products: department / sub-department / leaf department. A single product would be placed in one leaf department like Clothing & Accessories/Men/Tops & Tees. By creating three fields in the search document, one for each level of the tree, you can provide users with result counts at all levels and enable them to narrow their searches:

```
"fields": {
      "dept":"Clothing & Accessories",
      "sub_dept":"Clothing & Accessories/Men",
      "leaf_dept":"Clothing & Accessories/Men/Tops & Tees"
```

To retrieve counts across the search results at each level of the hierarchy, specify `facet=dept,sub_dept,leaf_dept` in the query. When displaying these counts to the user, you simply remove the prefix path. You can restrict searches to any level of the hierarchy by including the appropriate filter parameter:

```
bq=(and 'shirt' sub_dept:'Clothing & Accessories/Men')
```

This query retrieves all matches for the string shirt in any leaf category below Clothing & Accessories/Men. To support exact matching, be sure to configure these fields as *literal* fields.

For attributes like color, you usually present all choices and allow the user to select more than one option. If you pass in a query with a filter such as `color:'green'`, you won't receive correct counts for any of the other values—the results will all have green for their color. Instead, you can submit two queries, one to get the search results with green selected, and a second to get the facet counts without any selection for color.

If you have a second multi-select facet, such as size, then you run one query with both the size and color selections to get the query results, a second query with just the size selections to get the correct color counts, and a third query with just the color selections to get the correct size counts.

You can also enable users to narrow their search by price. To narrow matches to a range of prices, include the price as a field for each of the products. (In CloudSearch, uint fields are the only type of numeric field, so store prices in cents by multiplying the price by 100.)

Specifying an integer field as a facet returns the min and max values in that field across all documents instead of counts for individual values. This means you can get the full range of prices for a search with the *facet* parameter:

```
http://<search endpoint>/search?q=shirt&facet=price
```

This query returns:

```
{"rank": "-text_relevance",
  ...
  "facets": {
    "price": {
      "min": 1,
      "max": 349900
    }
 } ...
```

To get counts for particular price ranges, you need to specify the ranges using the *facet-<field name>-constraints* parameter. For example, this query requests counts for prices from $0 to $25, $25 to $50, $50 to $100, $100 to $150, and $150 and up:

```
http://<search endpoint>/search?q=shirt&facet=price&facet-price-constrain
ts=..2500,2500..5000,5000..10000,10000..15000,15000..
```

In the response, the constraint information is included with the facet information:

```
{...
 "facets": {
   "price": {
     "min": 1,
     "max": 349900,
     "constraints": [
       { "value": "..2500", "count": 11194 },
       { "value": "2500..5000", "count": 3888 },
       { "value": "5000..10000", "count": 1526 },
       { "value": "15000..", "count": 603 },
       { "value": "10000..15000", "count": 452 }
     ]
   }
```

To narrow subsequent searches, you include a filter in the query. For example, if the user selects the $25 to $50 price range, you add the constraint to the bq parameter:

```
http://<search endpoint>/search?bq=(and 'shirt' price:2500..5000)
```

Mixed Data Source Patterns

If you have documents of different types, you often want to enable users to search one type of document at a time. For example, if you're presenting a collection of hotels, cars, and flights, you want to support searching only hotels, only cars, or only flights.

Two main patterns enable you to manage and search a mix of document types:

- **Create a separate search domain for each document type.** With this approach, you use the search type to channel queries to the appropriate domain. This is the best solution when you have a large collection of each document type, a limited number of document types, and heterogeneous fields within each type of document. The key advantage of using separate domains is that each type's domain is updated and scaled independently.

- **House all document types in a single search domain.** To do this, you add a **type field** to every document that identifies the document type:

```
... "fields" : {
        "type" : "flight",
         ...
```

In each query, you include a filter to restrict the matches to a particular document type:

```
bq=(and type:'flight' ...)
```

The advantage of using a single domain is that the combined index can be ideally packed into memory and efficiently partitioned across multiple hosts. However, it means that scaling is less granular—a single highly-active type can force scaling that the other types don't need. It also means that configuration changes for one type require rebuilding the index for every type. Combining types in a single domain also has an impact on relevance ranking. The text_relevance function depends on the distribution of tokens in the index—when there are multiple types of documents, the text_relevance score is aggregated across types, which can result in less accurate ranking.

Access Control Patterns

You can use a pattern similar to the single-domain mixed document pattern to restrict access to documents based on the identity of the user. You add a user_id field to each of the documents in the search domain and add a filter to the query to restrict the results to the documents the user owns.

It gets more complicated when users have access rights to documents that they do not own. The simplest solution is to maintain a list of users with access to each document as part of the document data:

```
... "fields" : {
        "userid" : ["12345", "67890"],
         ...
```

The drawback to this approach is that you have to update each document whenever a user gains or loses access. In the worst case, you might have to update every document in the domain.

A better solution is to organize users in groups or documents in folders and manage user access outside of the search engine. In this case, you maintain a list of groups with access to each document as part of the document data:

```
... "fields" : {
        "groups" : ["11111", "12345", "87654"]
         ...
```

Your search queries then filter using ORs of the groups to which the user belongs:

```
bq=(and 'query string' (or group:'11111' group:'22222'...))
```

When user permissions change, it happens outside of the search engine and doesn't require configuration changes or reindexing. However, if you have a large number of groups, your queries become more complicated, which can degrade performance.

TUNING PERFORMANCE AND RELEVANCE

Ultimately, the goal of adding search to your application is to enable users to find what they are looking for. To meet your users' search goals, you have to test query performance and tune how results are ranked.

To get started with tuning, you should log:

- User queries
- Responses to user queries
- Query latencies

Often, you can simply process your weberver's logs to pull user queries. To tune relevance, look at the most common queries in your logs, run those queries, and review the results to see if they match your expectations.

In cases where results are not sorted correctly, you can use the **Compare Rank Expressions** tool in the AWS Management Console for Amazon CloudSearch to adjust your rank expressions and see the results in real time.

Another important source of information is **user queries that have no results** from your domain. Examine these queries to see which user-specified terms have no matches in your indexed data. You can add synonyms for common terms to increase matches for those terms—do you have a *Kids* department, but users are searching for Children's *socks*? Add a synonym!

To delight your users, you also need to consider the total time it takes to process and display search results—**the search latency**. CloudSearch returns processing time information in every search response:

```
{ ... "info": {"time-ms": 4, "cpu-time-ms": 0 }}
```

The *time-ms* value represents the total wall time spent processing the query. This value is the sum of the time spent calculating the search results, plus the overall request latency. Reducing this time minimizes the total time it takes to display results to the user.

Once you are in production, you can cull the slowest queries from your logs and work to reduce their latency. Queries against text fields that specify just a few terms are fastest. Queries that contain complex logical expressions perform most slowly, especially ones that contain multiple ORs and NOTs.

If you have frequent, slow queries that are submitted over a relatively short period of time, consider caching results using Amazon ElastiCache or a similar mechanism. Use the query as the cache key and the full CloudSearch response as the cache value and check the cache before sending a new query to CloudSearch.

You should also use a **benchmarking** tool like **Apache JMeter** to test your search domain's response time. Use a broad test set that includes at least *1000 different searches* and varies widely in query complexity. Your actual query logs are the best source for your test set, but you can also generate test queries based on a statistical sampling of terms contained in your documents.

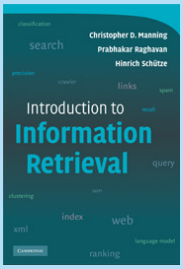
ADDITIONAL RESOURCES

- **Amazon CloudSearch** http://aws.amazon.com/cloudsearch
- **Amazon CloudSearch Developer Guide** http://docs.aws.amazon.com/cloudsearch/latest/developerguide
- **Search User Experience** Designing the Search Experience: The Information Architecture of Discovery, Russell-Rose, T., and Tate, T., Morgan Kaufman, 2013

ABOUT THE AUTHOR

As a Solutions Architect for Amazon CloudSearch, **Jon Handler** works with customers planning and implementing a wide range of search and information retrieval projects using CloudSearch. Previously Jon was Engineering Manager at Shopping.com, an eBay company. He holds a PhD in Artificial Intelligence from Northwestern University.

RECOMMENDED BOOK



Explains how search engines work from a theoretical perspective. Includes chapters on indexes, scoring, text classification, relevance feedback, clustering, machine learning on documents, and more.

BUY NOW

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. ""DZone is a developer's dream"," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1
50795
9 781936 502806

$7.95