

much you'd like to bias search scores toward the best field for that search term, or toward a behavior closer to `most_fields`.

With these signal discordance issues, this is where we'll get off the query-parser train. Term-centric query parsers were invented to solve the albino elephant problem. They can get you far. But because they leave us short of a proper solution to signal discordance, we'll begin to focus on two other methods of term-centric search.

## 6.4 Solving signal discordance in term-centric search

Although dismax-style query parsers demonstrate the fundamental behaviors of term-centric techniques, often two other solutions are used. These two term-centric methods, which solve both the albino elephant and signal discordance problems, are *custom all fields* and *cross\_fields*. Custom all fields combine other fields together at index time into one all field, an inflexible arrangement, but one that certainly works. A *cross\_fields* search, on the other hand, works at query time by attempting to directly compensate for signal discordance by patching the document frequency of fields prior to searching them. This, as you'll see, yields greater flexibility at the cost of introducing inaccuracy and inscrutability.

### 6.4.1 Combining fields into custom all fields

Custom all fields solve the problem of term-centric search at index time. The approach does this by directly combining the fields you'd like to search into a single field. After all, you can't have problems with multifield, field-centric search if you have only one field! The name *all fields* comes from the idea that you can copy fields together into a single field referred to as an *all field*. This has also been called *copy fields* in the Solr and Elasticsearch communities. For example, you could combine our troublesome name fields `cast.name` and `directors.name` into a more general `people.name` field by using the search engine's ability to append multiple fields together at index time.

Seems a bit odd to do, though; what does this buy you? Well, consider your troublesome actor/director William Shatner, who has directed one movie yet starred in many. This signal discordance causes `directors.name` matches to score unexpectedly high. This doesn't jibe well with your users' more general expectations of how documents are structured. By combining `directors.name` and `cast.name` into a broader, derived `people.name` field, the signal becomes far more general, and perhaps much closer to what your users imagine when they search. This generality manifests itself mechanically in the scoring. Suddenly, by searching `people.name`, there's no particular bias toward whether William Shatner directed a movie or starred in one. The difference is washed away! The document frequencies for the associated terms are combined and reflect a more general notion of how common William Shatner is as a person associated with a movie instead of as a director or cast member, as shown in the table 6.1.

**Table 6.1** Relative document frequencies of the phrase `william shatner` in two source fields and the combined `people custom all` field

	william shatner document frequency
<code>directors.name</code>	1
<code>cast.name</code>	16
<code>people.name</code> (custom all field)	16

This field is searched just as any field would be searched by itself, using a Boolean query of `SHOULD` clauses. Remember that this form favors documents that include more results, thus continuing to defeat the dreaded albino elephant:

```
people.name:william people.name:shatner
```

Let's demonstrate by adding such a custom all field to our documents. You'll see that by eliminating the difference between `cast` and `director`, search results get closer to users' expectations.

To set up a custom all field, first you need to define the new field in the mapping. You define the field `people` just as you do any other. Then, in the mapping entry for each field that feeds into `people`, you use the `copy_to` option to copy the contents of the source field to a destination field. The mapping gets verbose, so we'll show you each one at a time. First there's the mapping for the `people` field, which looks just like the other name field mapping you saw in the previous chapter. We even leave in the bigram version of this field, as shown in the following listing.

#### Listing 6.10 Mapping for custom all field—`people`

```
mappingSettings = {
  "movie": {
    "properties": {
      "people": {
        "properties": {
          "name": {
            "type": "string",
            "analyzer": "english",
            "fields": {
              "bigrammed": {
                "type": "string",
                "analyzer": "english_bigrams"
              }
            }
          }
        }
      }
    }
  }
}
```

Destination people field,  
defined like any other

people expects to receive src  
fields with a "name" property

people continues to preserve  
the "bigrammed" property

Continuing the mapping, you copy the `cast.name` field over to the `people.name` field. You do the same for `directors`. The following listing adds `copy_to` to the `cast` mapping.

**Listing 6.11 Adding copy\_to from cast.name to people.name**

```

"cast": {
  "properties": {
    "name": {
      "type": "string",
      "analyzer": "english",
      "copy_to": "people.name",
      "fields": {
        "bigrammed": {
          "type": "string",
          "analyzer": "english_bigrams"
        }
      }
    }
  }
}

```

← **Appending this field to people.name**

you'll need to reindex all your data:

```
reindex(analysisSettings, mappingSettings, movieDict)
```

Great! Now what happens when you search this new field? With a new field, your options are exceedingly flexible. You can use it anywhere you'd list a field to be searched. For now, though, to see the utility of just this field, notice what happens in the following listing when you directly search only this new `people.name` field for our two actors.

**Listing 6.12 Simple use of a custom all field**

```

usersSearch = 'patrick stewart william shatner'
query = {
  "query": {
    "match": {
      "people.name": usersSearch,
    }
  }
}
search(query)

```

← **match query (searches a single field, people.name)**

← **User's query**

Num	Relevance Score	Movie Title
1	1.3773818	Star Trek: Generations
2	0.6629994	Showtime
3	0.65602106	The Wild
4	0.5989805	Bill & Ted's Bogus Journey
5	0.58601415	Star Trek V: The Final Frontier

First you notice that *Star Trek: Generations*, the movie containing both actors, comes to the top of the search results. Nothing surprising here; the Boolean query for the two actors continues to promote results that contain more search terms. The fact that the search occurs over a single field ensures that you don't encounter the albino elephant problem.

Signal discordance is solved in this case; the document frequency for a match on *william shatner* in *Star Trek V* for `people.name` is identical to the document frequency for *Star Trek: Generations*. So the scores come out closer to what users expect. The fact that William Shatner directed *Star Trek V* gets mostly washed away by the `people.name` field's combined document frequency. Also, notice how much higher the score is for *Star Trek: Generations*, a movie that satisfies all search criteria. The Boolean search of the combined fields amplifies the document that contains all the user's search terms.

How far should you take this? Should every field be copied into an `_all` field? Elasticsearch takes the approach far. By default, every field is copied to an overarching field called `_all`. Searching the `_all` field returns good results for our use case.

#### Listing 6.13 Searching `_all`

```
usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "match": {
      "_all": usersSearch,
    }
  }
}
search(query)
```

← User's  
query

Num	Relevance Score	Movie Title
1	0.9441141	Star Trek: Generations
2	0.577018	Star Trek: Insurrection
3	0.577018	Star Trek: First Contact
4	0.56560814	Star Trek V: The Final Frontier
5	0.5166054	Star Trek: Nemesis

Yet, repeat the same search with just `patrick stewart`, and the top result has nothing to do with Patrick Stewart:

1	0.4262765	Panic Room
2	0.33741575	Conspiracy Theory
3	0.33741575	The Wolverine
4	0.33741575	Vertigo
5	0.33741575	Star Trek: Insurrection

This happens because somewhere in the combined `_all` field of `panic room`, the text `patrick` is mentioned, and so is `stewart`. The signal generated from an `_all` search is vague, and doesn't map to meaningful information your users would recognize. The signal for the more carefully created `people.name` is probably closer to the user's sense of meaningful criteria; it measures relevance scores for people associated with a movie. Custom `_all` fields that vacuum up all the text (such as the `_all` field) tend to be far too general, and should generally be avoided.

Again, term-centric search isn't a panacea. Users likely *do* care about *what* is matching. To tightly control what is matching per field, you often focus on field-centric methods and the lessons in chapters 4 and 5. Later in this chapter, you'll begin to see how to have your cake and eat it too.

### 6.4.2 Solving signal discordance with cross\_fields

Custom all fields are static, created as documents are being indexed. In contrast, a `cross_fields` search is dynamic, addressing signal discordance at query time. It does this by becoming a `dismax`-style query parser on steroids. The ranking function of `cross_fields` remains identical to the query parser approach, with one important modification: the `cross_fields` query temporarily modifies the search term's document frequency, field by field, before searching. If the term `shatner` is particularly common in the fields being searched, except for one troublesome field such as `directors.name`, then that troublesome field will be lied to and given a larger document frequency. This attempts to solve signal discordance, albeit in a way that could be less accurate than an all field.

Say that instead of using a dismal query parser field, you execute a `cross_fields` search. So instead of something like this `dismax`

```
(cast.name:william | directors.name:william)
(cast.name:shatner | directors.name:shatner)
```

you get what's referenced by Elasticsearch with this `Blended` explain syntax, which blends the document frequencies of the listed fields before executing the search:

```
Blended(cast.name:william, directors.name:william)
Blended(cast.name:shatner, directors.name:shatner)
```

What would a `cross_fields` search combining all of these fields look like? It's a `multi_match` query that uses the `cross_fields` query type. `cross_fields` uses the field's common query analyzer. Much like the `dismax` query parser, `cross_fields` continues to suffer from field synchronicity. If fields don't share the same analyzer, `cross_fields` returns an error. With that in mind, you know that most of your fields have versions analyzed as English text. So you should be able to use `cross_fields` search over these English-language-analyzed results.

Running `cross_fields` with this in mind gets at our needed search results, as shown in the following listing.

#### Listing 6.14 cross\_fields search over useful fields

```
usersSearch = 'star trek patrick stewart william shatner'
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview", "cast.name", "directors.name"],
```

← User's  
query

```

    "type": "cross_fields",
  }
}
search(query)

```

← Uses a cross field search

## Results

Num	Relevance Score	Movie Title
1	1.9040859	Star Trek: Generations
2	1.6575186	Star Trek V: The Final Frontier
3	1.3508359	Star Trek: Nemesis
4	1.1206487	Star Trek: The Motion Picture
5	1.0781065	Star Trek: Insurrection

The results aren't identical to the preceding `_all` field search. It turns out you can't exactly calculate the combined document frequency of multiple fields at query time. The fields have limited information for this approximation. The `cross_fields` query can access only each term's document frequency for a field. It then must attempt to sensibly combine them, which can't be guaranteed to be exactly correct. Let's say `shatner` has a document frequency of 16 in `cast.name` and 1 in `directors.name`. By knowing those two facts, you can't tell whether the single occurrence of the term `shatner` in `directors.name` occurs in a film where William Shatner also stars, or if it occurs in a movie directed by Shatner but not starring him. In the former case, the combined document frequency would be 16. But in the latter, the true document frequency should be the result of summing the two fields' document frequencies: 17. The `cross_fields` approach takes the safe route, picking the max of the two fields' document frequencies: 16.

A custom all field that physically combines fields would capture the document frequency more accurately. A combined `people.name` field would have the term `shatner` exactly once, regardless of whether or not he directed. But this accuracy comes at a cost. All fields are built exactly once at index time. You can't decide at search time to combine another set of fields. Moreover, the space requirements sometimes aren't tolerable, especially in large-scale systems. Although not as accurate, `cross_fields` allows more ad hoc flexibility in blending fields. Moreover, `cross_fields` is fundamentally a term-by-term `dismax` query—the same as you saw with the previous query parser. So tuning options, field boosting, and the `tie_breaker` parameter continue to be available to you! Unfortunately, the same field synchronicity issues also apply.

## 6.5 **Combining field-centric and term-centric strategies: having your cake and eating it too**

You've seen that field-centric search often ignores users' basic search expectations. Yet term-centric search, because of field synchronicity, has its own problems. It puts strict enforcements on the underlying fields. And for this reason, delivers basic, unsophisticated relevance functionality. To put things in context: you spent chapter 4 using analyzers to build special-snowflake fields, capable of modeling anything that can be

tokenized. In chapter 5 you learned how to use those with field-centric search. And here you’ve learned that all that work is incompatible with term-centric search!

Sadly, there’s no silver bullet. Mastering how to combine both approaches is an ongoing struggle. In this section, we discuss strategies for trying to balance the strengths of both techniques to create relevance solutions that satisfy and delight. You’ll see that the way you use the two strategies together depends entirely on making the right compromises for your data and users. The struggle to get the balance right is a huge part of your ongoing tuning work as a relevance engineer.

### 6.5.1 Grouping “like fields” together

In this section, we examine one strategy for combining field-centric and term-centric effects: grouping similar, or *like*, fields together with a term-centric effect. These groupings are themselves combined with an outer field-centric search. Artfully grouping fields can be a way to sidestep field-centric search problems with just enough of a term-centric effect to prioritize more matches within the fields prone to albino elephant and signal discordance problems.

Note that if a user’s search term can match only a single field, then the problems of field-centric search go away. You can’t have albino elephant or signal discordance if search terms always find their ideal field. For example, a match for “Star Trek Patrick Stewart William Shatner” would be fine with field-centric search if you could guarantee that `star trek` matched in exactly one field (let’s call it `text`), while name-related terms such as `patrick stewart` and `william shatner` matched only in a field about people.

We call these groupings *like fields*. As we’ve discussed, your source data model doesn’t come with these straightforward groupings. It’s your job when signal modeling to group fields in such a way that albino elephant and signal discordance have little effect on the final solution. Your job too is to map signals to users’ general search expectations. By grouping like fields, you more closely approximate the higher-level signals users care about.

One way that a user is likely to think about ranking Star Trek matches, for example, is in groupings of like fields indicating that the ideal document

- SHOULD match people mentioned in the search string  
(inner term-centric `people:term1 people term2 ...` )
- SHOULD match text mentioned in the search string  
(inner term-centric `text:term1 text:term2...`)

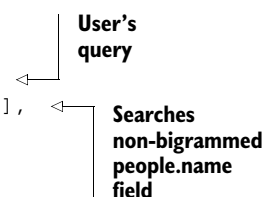
Given this specification, you can express the ideal document to Elasticsearch for your Star Trek search. One simple improvement, shown in the following listing, is to group people to more closely match the preceding specification by using the custom all field `people.name` as part of a `most_fields` search. You’ll note in the listing that we’re searching the non-bigrammed field to demonstrate the basic idea. Further precision could also be gained searching the bigrammed field.

**Listing 6.15 Search combining term-centric all field (people.name) with other fields**

```

usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["title", "overview", "people.name"],
      "type": "most_fields",
    }
  }
}
search(query)

```



User's query

Searches non-bigrammed people.name field

Num	Relevance Score	Movie Title
1	0.7104292	Star Trek: Generations
2	0.5998383	Star Trek IV: The Voyage Home
3	0.50374436	Star Trek: Nemesis
4	0.35599363	Star Trek
5	0.3373023	Star Trek: The Motion Picture

This is an improvement. By grouping people fields with other people fields (directors.name + cast.name -> people.name), you've eliminated a class of likely problems. Now you can search in a field-centric fashion, but with a term-centric effect on the custom all field. You carefully combined the various fields about people into a single field that provides a specific signal.

### 6.5.2 Understanding the limits of like fields

Is this it, then? Is this the solution to your search problems? Is grouping like fields the right way to balance the two approaches? The unfortunate reality is that search terms do spuriously crop up in fields we don't expect them to. So grouping like fields can't always save you. As we said, unfortunately there's no silver bullet to the yin and yang of field-centric and term-centric search. Like most programming problems, there are only carefully honed compromises based on the nature of your data and the demands of your users.

You can modify the strategy to account for reality. In our movie search, for example, names of actors crop up in movie descriptions. There are probably people named Star and even Trek that would score rather highly because of how rare they are as names. In the same way, names such as William could appear in titles or overview text. Grouping fields into like fields might not always be feasible.

Let's demonstrate the problem. Here `cross_fields` search is used instead of a custom all field to group text fields with text fields, and people fields with people fields. This is the strategy we described previously. Recall that `most_fields` expects a list of *fields* to search, not a list of *queries*. So to use `most_fields` with an inner `cross_fields` query, you need to implement the `most_fields` scoring behavior on your own. This is what the outer Boolean query does in the following listing.



Listing 6.16 Searching two field groupings—people and text

```

usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["directors.name", "cast.name"],
            "type": "cross_fields"
          }
        },
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["overview", "title"],
            "type": "cross_fields"
          }
        }
      ]
    }
  }
}
search(query)

```

**User's query**

**Boolean SHOULD clauses, replicating "most\_fields"**

**Grouping of like people fields into a term-centric search**

**Grouping of like text fields into a term-centric search**

Num	Relevance Score	Movie Title
1	1.1444862	Star Trek IV: The Voyage Home
2	0.75206727	Star Trek: Nemesis
3	0.7318188	Star Trek V: The Final Frontier
4	0.72360706	Star Trek: Generations
5	0.5002059	Star Trek

This, unfortunately, doesn't have the impact you want. For some reason, *Star Trek IV* shoots to the top of the list. Why? If you examine the explain, you'll see that surprisingly, *william shatner* is mentioned in the *overview* field of *Star Trek IV*:

```

0.18140785, max of:
  0.18140785, weight(overview:william in 474)
0.34648207, max of:
  0.34648207, weight(overview:shatner in 474)

```

As we suspected, names often crop up in many fields. You usually can't cleanly separate fields into like fields. The idea that "search terms will always be in bucket A or B" doesn't work always with our messy, unstructured data sets. Moreover, as the text-based `cross_fields` query gets biased heavily when more search terms match, the fact that names like William Shatner or Patrick Stewart do match these fields further amplifies the distortion of unexpected matches.

### 6.5.3 *Combining greedy naïve search and conservative amplifiers*

We said that term-centric search often satisfies but rarely delights. Another strategy combining term-centric and field-centric methods is to base relevance on a term-centric foundation. Using that baseline, smarter and discriminating per-field signals can be brought in to amplify documents in ways you feel confident will delight the user. You already know how to craft fields that when matched meet certain criteria with high confidence—for example, criteria such as “the user is searching for a person” or “the user is searching for a location.” Using this criteria, you can think of the ideal document for your TMDb example as follows.

The ideal document

- SHOULD have all search terms match the superset of fields being searched
- SHOULD have searched-for names match the people associated with the film

The first SHOULD serves as a basic text score over a superset of the plain-text versions of fields that occur in all other clauses. It matches rather greedily. The second SHOULD clause is much less greedy. It’s highly discriminating and searches only a subset of the fields from the first clause grouped into like fields (such as people, places, or things) in order to get specific signals.

Two factors are important with this approach:

- Any document that matches the second discriminating clause also matched the first “greedy” clause. This way, every document being considered has a base score.
- The nongreedy clauses should be high-quality signals erring on the conservative side to avoid overriding the base score with an unexpected match.

By having one wide-net base score and carefully selected discriminating amplifiers, you’re more likely to arrive at a place that at the very least satisfies the user. As your secondary signals improve, you can continue to use those as amplifiers with high confidence—sussing out additional highly discriminating signals. The secondary signals are conservative for important reasons: If those secondary signals are unintelligent, or let something “sneak in” such as a non-name match, you may override the user’s base assumptions of all their search terms matching, resulting in something rather silly. But if you miss the mark and don’t quite get to the precise match on the person named William Shatner, at least the basic term-centric search is backing you up so that you don’t arrive at something utterly terrible.

Let’s try that out, with a base `cross_fields` term-centric search over the text version of each field, and an additional SHOULD that brings up documents that match the subset of those fields corresponding to people’s names, modeled using the bigram analyzers.

Listing 6.17 Greedy term-centric paired with highly discriminating like fields

```

usersSearch = "star trek patrick stewart william shatner"
query = {
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["directors.name.bigrammed",
                      "cast.name.bigrammed"],
            "type": "cross_fields"
          }
        },
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["overview", "title",
                      "directors.name", "cast.name"],
            "type": "cross_fields"
          }
        }
      ]
    }
  }
}

search(query)
1  1.6669365      Star Trek: Generations
2  1.5123603      Star Trek V: The Final Frontier
3  1.0779369      Star Trek: Nemesis
4  0.9057324      Star Trek: The Motion Picture
5  0.8793935      Star Trek: Insurrection

```

**User's query**

**Outer Boolean query biases toward all matches of both general text and full people bigrammed names.**

**Searches names by bigrammed, discriminating like fields**

**Computes base text score over all searched fields**

Now you're getting somewhere! There are refinements to be made, but this pattern is a good start. The score for *Star Trek V* is close to that of *Star Trek: Generations*. Is it because the term frequency for the bigram *william shatner* is 2 (as we know he acts and directs)? Users don't care that there are two mentions of William Shatner! You could keep improving the quality of this signal, enhancing how much it measures the association of a person with a field. You could disable term frequency and get even closer to the ideal signal (or as you'll see in the next chapter, use a `constant_score` query to completely eliminate  $TF \times IDF$ ).

Another question is how much of a foundation should the base term-centric search create? How much should it impact the score as compared to these more specialized signals? What is the impact of the less specialized term-centric clause? With increasing **SHOULD** field-centric clauses, could you get back into an albino elephant scenario in which many of the nonfoundational, field-centric, special snowflake fields match strongly, but not all individual search terms match in the term-centric search? Yes, you can. And continuing to wrestle with the yin and yang of term-centric and

field-centric search takes a large portion of the relevance engineer's time. Luckily, every query in Elasticsearch can be boosted, tweaking the overall score toward whichever end—yin or yang—you need scoring to go.

### **6.5.4 *Term-centric vs. field-centric, and precision vs. recall***

The pattern in the previous section should remind you of the ever-present struggle between precision and recall. As you learned in chapter 4, high recall ensures that all the right matches are in the search results, and high precision ensures that few false-positive matches are included. You can modify the definition of precision and recall to think just about the top  $N$  (say, top 10) results that you'll show on the first page of search results. Practically speaking, this is what's important to get right!

A greedy term-centric search gives you high recall. It casts its net wide, ensuring that you've captured all the right search results. Your first search page contains a list of possibly relevant results—likely, a reasonable mix of fairly simple matches, though nothing particularly smart. Left alone, your users are likely to go through pages of search results every now and then, being forced to scan for what they want. Adding in highly discriminating field-centric signals improves the precision of the search results. It promotes increasingly more promising candidates to the first page only when certain exacting criteria are met.

What's great about the pattern from the previous section is that it lets you control how much term-centric or how much field-centric impact you'd like to have. If precise matching is more important, you can fine-tune those specific matches with boosts, letting those scores bubble up more easily. If this is less important, boosting field-centric signals matters much less than focusing on recall.

### **6.5.5 *Considering filtering, boosting, and reranking***

Although the pattern mentioned previously is a good starting point, the Query DSL is always being used in new and interesting ways. In the next chapter, you'll see how to carefully and explicitly use signals to manipulate ranking even further. You might, for example, want to fine-tune the base term-centric pattern discussed previously by filtering out results that don't match a minimum number of search criteria. Or figure out a way to make the number of search terms that match a primary sort, and call out the field-centric signals to be more of an inner reranking criteria. Often field-centric approaches can precisely let you filter or boost by using a specific signal. Perhaps you'd like to give an explicit nudge to restaurants close to the user, or to filter out restaurants far away. At the core, you may have a general relevance score. On the fringes, you massage and prod the relevance score with more carefully modeled signals.

## 6.6 Summary

- Because of the albino elephant problem, field-centric search doesn't satisfy users' relatively naïve sense of relevance, which relies on the premise that documents matching more of your search terms ought to be considered more relevant.
- Signal discordance creates unexpected ranking behavior as the search engine decomposes documents into many fields that are scored independently.
- Term-centric search pushes scoring toward users' naïve sense of relevance (prioritizing documents that match more search terms).
- Term-centric query parsers, such as `query_string` or Solr's `edismax`, solve the albino elephant problem, but not signal discordance.
- Creating a custom all field solves both the albino elephant problem and signal discordance most accurately, but increases the index size.
- Blended term-centric search methods solve the albino elephant problem and signal discordance at query time, but not as accurately as a custom all field.
- Term-centric search requires the search string to be analyzed before fields are searched. This eliminates the ability to specialize each field to measure different, smarter signals.
- Balancing generic term-centric scoring with smarter field-centric scoring requires careful work.
- Several strategies aim to take advantage of the strengths of field-centric and term-centric search (including like fields and highly discriminating boosts).
- Another way to look at term-centric versus field-centric search is that term-centric focuses on high recall, whereas field-centric is a tool to get higher precision.



# *Shaping the relevance function*

---

## ***This chapter covers***

- Bringing up relevant content through boosting
- Knowing when to use different forms of boosting
- Improving the ranking of popular or recent content
- Filtering out irrelevant or noisy content from search results
- Stoking your own creative uses of the search engine's querying features

As a relevance engineer, you tailor your users' search experience to their many unspoken ranking expectations. Unpacking, understanding, and finally implementing these expectations are key parts of your job. For example, almost everyone has a sense that a news search should show up-to-date articles about breaking events. Or that a restaurant search shouldn't take into account only the user's query but also the proximity of that user to the restaurant. Your search will probably have unique ranking needs that go beyond text matching. For instance, what if you're building a local news search? Should it focus on both proximity *and* freshness?

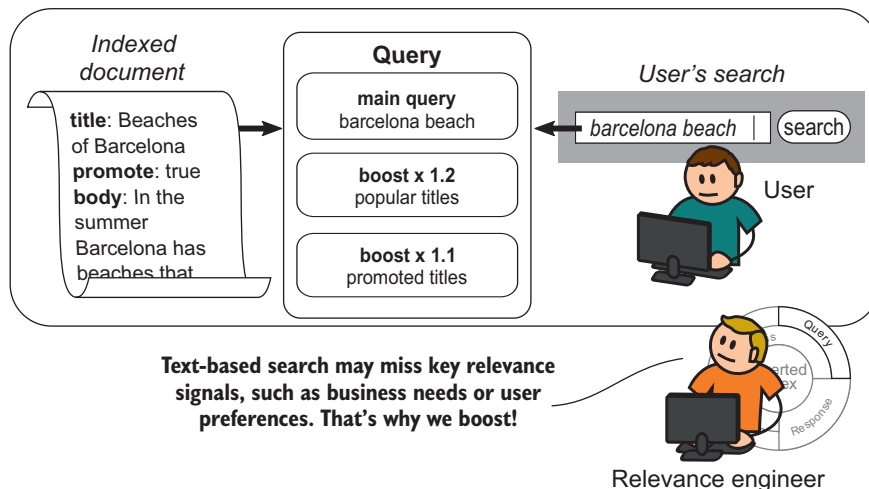
Or what about a global restaurant search for the jet-setting crowd? Should it focus on cities with major airports?

Previous chapters have shown you how to control the general form of results for common forms of search. In this chapter, you'll snip, craft, and shape the ranking function to get at your users' unique criteria. You'll see that you can become a master of your search engine's Query DSL, truly programming every corner of search ranking, in order to carefully boost, filter, rerank, and sort in exacting ways to implement unique ranking requirements.

This chapter is the search relevancy power hour, but our goal isn't to comprehensively teach you every trick for manipulating search. The number of permutations of every option of every query is far too large! Instead, we want to show you techniques to move the gears of your imagination. You'll see that using your search engine's Query DSL *is* programming. It's not a handful of knobs and dials to just tune. It's a language for customizing search ranking—with new techniques being constantly discovered! Indeed, if we're reading your blog post or book and learning a new technique from you in the years ahead, we'll have done our job in this chapter.

## 7.1 What do we mean by score shaping?

With *score shaping*, depicted in figure 7.1, relevance begins to take on the character of true programming. You use the tools within the Query DSL to snip the ranking function closer to your needs. The most important tools in the Query DSL for programming ranking are *boosting* and *filtering*. We've used these terms loosely in previous chapters, but they take on special importance in this chapter.



**Figure 7.1** Relevance engineers use score-shaping techniques to carefully craft the ranking, thus implementing custom, application-specific ranking rules.

So let's tighten up our definitions:

- *Boosting*—Given a base set of search results, boosting increases the relevance score of a subset of those search results.
- *Filtering*—Given the entire corpus of possible search results, filtering removes a subset of those documents from consideration by specifying a filter query.

Defining the subset of search results for boosting and filtering, and deciding how they modify relevance ranking, extends the skills you learned in the previous chapter:

- *Signals*—Signals measure important ranking criteria at search time. In this chapter, signals take a more quantitative tone: How recently was an article published? How close is the restaurant to me? The presence of a signal indicates when to filter or boost; the magnitude of the signal might control a boosting factor.
- *Ranking function*—Filtering and boosting directly adjust the ranking function. For example, a boost might apply a multiplier proportional to how recently a movie was released. A filter might limit the subset of search results that the ranking function is run against.

Although boosting and filtering are the core techniques that we cover in this chapter, quite a few methods are available to shape scores. Other methods of score shaping include:

- *Modifying the sort criteria* to not strictly be based on the relevance score, but based on other values—such as date, popularity, distance, or computed values
- *Negative boosting*, which drives down a set of search results (in contrast to standard, positive boosting)
- *Rescoring or reranking* by adding a second stage to ranking to tweak the order of the top  $N$  set of results with additional signals
- *Scripting the score via a custom score query*, which allows you to use a script to completely take control of the scoring for a base set of search results

Honing your expressive ability with the Query DSL will stoke your creative abilities as a relevance engineer. As you examine these techniques, remember: the goal is not just to explore all of these ideas, but as all algebra teachers tell their students, you learn this stuff to “teach you how to think.”

## 7.2 *Boosting: shaping by promoting results*

Boosting gives you the power to mathematically prioritize specific documents as more relevant than others. Carefully selecting the documents you'd like to boost and calibrating the impact of those boosts are key skills for programming relevance. How much you boost can be based on any number of criteria, including secondary text-relevance scores, simple constants, or content-quality metrics such as popularity, recency of publication, or user ratings.



You may recall that we discussed the boosts associated with various fields when going over multifield search in previous chapters. In those chapters, we spent time calibrating the relative weight of, say, a title or a body match in a multifield search. To be clear: those weights (what we refer to as *boost weights*) aren't what we mean here. The boosting work you'll see in this chapter isn't (just) tuning weights. Here you'll use secondary queries, or *boost queries*, to modify the overall ranking function.

For the boost to truly reflect an important, meaningful relevance signal, all the skills from previous chapters continue to matter a great deal. After all, without good signals to boost, you won't be shaping the score so much as polluting it with noisy scoring that promotes unexpected results to the top.

In this section, we'll introduce you to your first boost queries. You'll see the subdivisions used to break up forms of boosting. Initially, we'll teach you the basic forms of each. Knowing these basic forms is foundational for enhancing your boost-fu. Later sections demonstrate how to flex your skills to create search results that answer more real-world ranking questions through boosting.

### 7.2.1 Boosting: the final frontier

You're about to go on a wild boosting expedition, but before we tinker, explore, and prod, you need to pick up a basic example to drive your work.

What example should we choose? Well, boosting and score shaping is a bit of a final frontier—always ripe for discovery! So it makes sense to go back to the Star Trek fan movie search from previous chapters. One use case we haven't considered is a user's omission of "Star Trek" from the search. Fans who search for just "William Shatner" probably won't be happy with the TV show *TJ Hooker* as the first search result. Let's boost "Star Trek" titles to the top to get a feel for boosting basics.

First you need a base query. In the preceding chapter, you learned how `cross_fields` search forms a reasonable starting point for relevance, so let's start with it here. In the following listing, you're searching for "William Shatner Patrick Stewart," hunting for the film that stars both of them.

#### Listing 7.1 Base query to boost

```
usersSearch = "william shatner patrick stewart"
query = {
  "query": {
    "multi_match": {
      "query": usersSearch,
      "fields": ["overview", "title",
                "directors.name", "cast.name"],
      "type": "cross_fields"
    }
  }
}
search(query)
```

User's  
query

## Results

Num	Relevance Score	Movie Title
1	0.79947156	Star Trek V: The Final Frontier
2	0.67931885	Star Trek: Generations
3	0.4375222	The Wild
4	0.38154808	Dark Skies
5	0.32485005	Showtime

The results aren't perfect. *Star Trek: Generations* ought to be number one (recall that in the last chapter, we tweaked this a bit further). For our purposes, though, we're focused on a different problem. For the Star Trek fan, the last three results should be Star Trek movies. You need to boost Star Trek films! Let's explore the options.

### 7.2.2 When boosting—add or multiply? Boolean or function query?

You've seen a minor problem that requires boosting. Time to get to work bending the ranking function to your will! Where do you start? Let's make sense of the boosting landscape before deciding exactly how you'll bring (beam?) up those Star Trek title matches.

In particular, you'll see that when boosting, you need to make two key decisions:

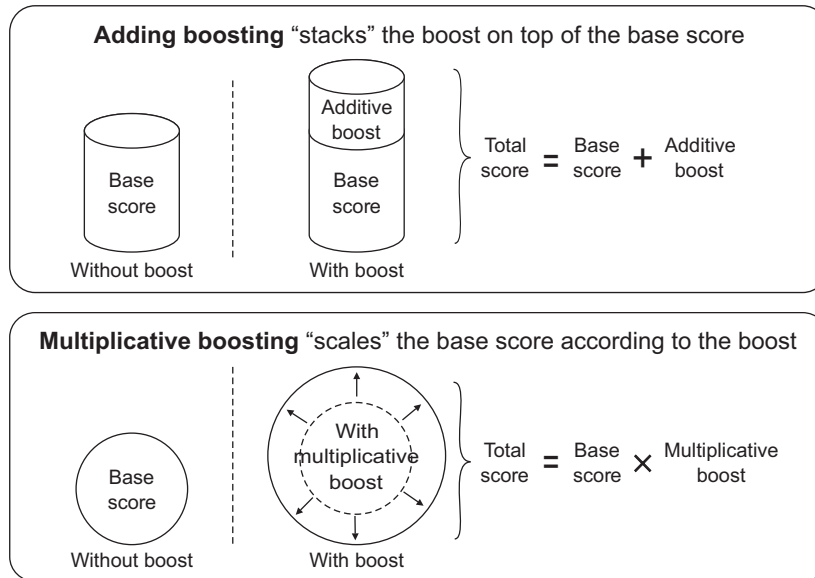
- *The math used to boost*—Should you add or multiply the boost score to the base query?
- *The query used to boost*—Some queries are explicitly geared toward boosting, but which should you use?

Let's discuss the math first. A document matching the boost query (Star Trek films, in our case) will have its score combined with the base query's. What are the consequences of choosing addition or multiplication?

- An *additive boost stacks* the boost on top of the base query. To be effective, the boost must layer on just enough oomph to matter in the final calculation. A 0.01 boost added to a base query score of 4 will hardly matter. A 100.0 boost added to that base query will effectively override the base query.
- A *multiplicative boost scales* the base query. A simple boost multiplier of 1.2 ensures, regardless of how the base query's score behaves, that boosted documents will get 20% more oomph than unboosted documents.

Figure 7.2 shows the implications of this choice. With the multiplicative boost, think of a document as a balloon. A multiplicative boost “inflates” the boosted balloon by the appropriate amount. Additive boosting forces you to consider how much you layer on the impact of a boost.

The second important decision is choosing the query to boost with. This choice dictates the form of the ranking function and the tools you'll have available to shape it to your needs.



**Figure 7.2** Additive boosting layers on extra criteria; multiplicative boosting inflates/deflates through multiplication.

With Elasticsearch, two queries are used for boosting:

- With a *Boolean query*, you boost via an additional Boolean clause on top of the base query, using Elasticsearch’s `bool` query.
- With a *function query*, you boost by directly modifying the ranking function by using Elasticsearch’s `function_score` query.

By using Boolean queries, much is abstracted away for you. Adding a boost means adding a simple `SHOULD` clause on top of the base query and tweaking a few weights. The ranking math is baked in (Boolean queries are always *additive boosts*), with a few knobs and dials to manipulate.

Function queries give you direct control of the ranking function. You combine the base query and any boost queries in mathematically arbitrary forms. There’s no baked-in “formula” as in Boolean queries. Therefore, function queries can’t cleanly be categorized as multiplicative or additive. Function queries are simply math! Given the lower level of control, there are things only function queries can accomplish.

Let’s get back to our problem. Where were we? Oh, right! You have an important search problem to solve with hordes of Trek fans raring to break down your door. Better choose a solution fast! Do you go through door A and solve your problem by applying an additional Boolean clause? Or do you go through door B and begin to get your hands dirty with function queries? Let’s see the consequences of both decisions.

### 7.2.3 You choose door A: additive boosting with Boolean queries

You choose door A: Boolean queries. To satisfy those Star Trek fans, you'll boost with an additional Boolean clause on top of the base `cross_fields` query. This will ask the search engine to prioritize documents with the phrase `star trek` in the title. You'll begin to see how to treat any boost as a signal to optimize. Further, combining the boost signal appropriately requires familiarizing yourself with the additive math underlying the Boolean query.

Let's see what a Boolean query boost looks like. A simple `match_phrase` query on "Star Trek" in the title is your first attempt to measure a signal that the film is a Star Trek film. Apply this alongside the base query within a `bool` query, as shown in the following listing.

**Listing 7.2** Boosting with an additional Boolean clause

```

usersSearch = "william shatner patrick stewart"
query = {
  "query": {
    "bool": {
      "should": [
        { "multi_match": {
          "query": usersSearch,
          "fields": ["overview", "title",
                    "directors.name", "cast.name"],
          "type": "cross_fields"
        } },
        { "match_phrase": {
          "title": {
            "query": "star trek",
          }
        } }
      ]
    }
  }
}
search(query)

```

**1** Base query—measures signal that more query terms match

**2** Boost query—measures signal that the film is a Star Trek film

**3** Combined through addition

User's query

Before we reveal the final search results, let's walk through what happens when the search engine scores this query. This will reveal exactly how and where you can modify the ranking function and boosting to your needs.

#### OPTIMIZING BOOSTS IN ISOLATION

With a Boolean query, first the boost and base scores are run in isolation (**1** and **2** in listing 7.2). Your boosting can be only as good as the underlying queries. In our case, your queries attempt to measure two signals:

- *Boost*—Is the film a Star Trek film? (**1** in listing 7.2)
- *Base*—Are all the query terms featured in the searched fields? (**2** in listing 7.2)

Do these queries accurately measure the needed signals? For instance, the boost query, a phrase query against the title field, relies on  $TF \times IDF$  scoring of phrases. Does that score help to measure whether a film is a Star Trek film? Is  $TF \times IDF$  appropriate to answer what seems to be a yes/no question?

What about the base query? Is this `cross_fields` search the right strategy? Are all the fields appropriately weighted? Should other fields be searched? What about other parameters of `cross_fields`, such as `tie_breaker`? The point is that all of these options are up for tweaking and experimentation. As a relevance engineer, you never rest when optimizing the signals underlying your queries.

### COMBINING BOOST AND BASE QUERY

The next step to consider is how the Boolean query combines these queries' scores (❸ in listing 7.2). Boolean queries do a bit more than add the scores of their component clauses. Additive boosting means you need to layer on the boost's influence carefully to not overwhelm or underwhelm the base query score.

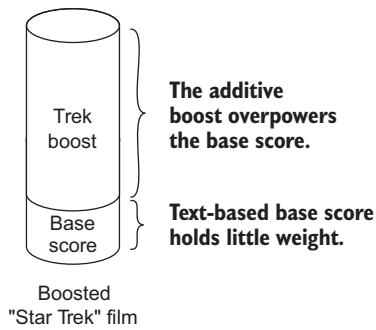
You know by now that Boolean queries do more than add. You also need to concern yourself with the coordinating factor (`coord`)—that strong bias toward documents that match all the clauses (here, both the boost and base queries). This can be disabled with the `disable_coord` option if you don't want this bias. You might want to do this if the boost query is more “extra credit” than “required.”

Finally, with all these factors, each clause has been scored and combined. Let's examine the search results from listing 7.2 to see whether they match our expectations:

Num	Relevance Score	Movie Title
1	4.363374	Star Trek
2	4.0461645	Star Trek: Generations
3	3.7096446	Star Trek V: The Final Frontier
4	3.6913855	Star Trek: Nemesis
5	3.653065	Star Trek: The Motion Picture

It definitely looks better! But recall that our search is for “William Shatner Patrick Stewart.” The first result doesn't star William Shatner or Patrick Stewart. Maybe the boost didn't work?

Examining these results, it appears that the Star Trek boost might be layered on too thick. You'll notice that instead of the desired result, *Star Trek: Generations*, we seem to be heavily favoring the shorter Star Trek match. This is suspicious, and leads us to think that perhaps the  $TF \times IDF$  relevance score of the Star Trek phrase-matching clause isn't quite measuring the right signal. Why might we think this? Recall,  $TF \times IDF$  has a strong bias toward shorter fields through field normalization, which scores the short Star Trek title match highly. Figure 7.3 shows the impact of each layer, with the Star Trek title boost overwhelming relevance scoring.



**Figure 7.3** The Star Trek title boost is layered on too thick, and needs to be tuned down so the base relevance score can contribute more appropriately.

There's too much boosting going on, and that boosting is based on factors you don't care about! The boost clause doesn't measure the right signal. You could correct this in various ways. The simplest option most reach for is to reduce the boost weight, lowering the boost from 1 to perhaps 0.25 or 0.1. This dampens the wonky  $TF \times IDF$  scoring. Let's examine the result of reweighting our boost to be more carefully applied (here we simply show the modified `match_phrase` clause):

```
{
  "match_phrase": {
    "title": {
      "query": "star trek",
      "boost": 0.1
    }
  }
}
```

This brings the results to a saner place. Notice how our top two results mirror the `cross_fields` search from our base search earlier. But what's improved is that subsequent matches bring up the Star Trek titles:

Num	Relevance Score	Movie Title
1	1.1662666	Star Trek V: The Final Frontier
2	1.0990597	Star Trek: Generations
3	0.6702043	Star Trek: Nemesis
4	0.62388283	Star Trek: The Motion Picture
5	0.6117288	Star Trek II: The Wrath of Khan

Nevertheless, this is a fragile solution. Simply tweaking a boost weight may postpone a problem for another search. Another, more sound option is to improve the precision of the signal associated with the Star Trek title matching. One way to do this is to disable field normalization for this field, trying to even out the Star Trek scoring.

The bigger question, however, is whether  $TF \times IDF$  even matters here. Users think of a "Star Trek" movie query as more of a yes/no, 1/0 sort of question. So  $TF \times IDF$  might not measure this signal correctly. We won't solve that problem right now. Later, when you dive into boosting strategies, you'll see one way to step away from  $TF \times IDF$  to get to yes/no scoring.

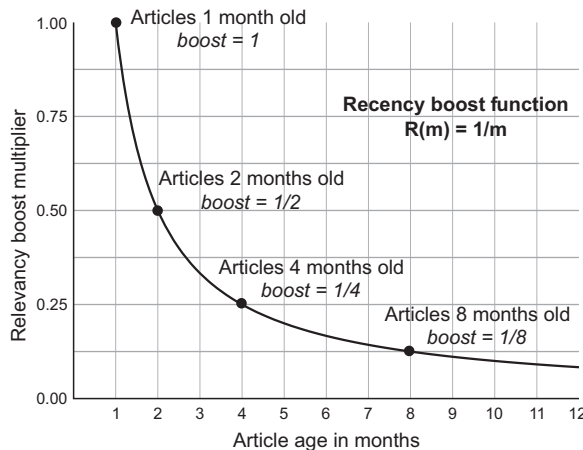
### 7.2.4 You choose door B: function queries using math for ranking

Time to try a different strategy: door B, function queries! In your work, you'll likely need numerical attributes, such as a product's profitability or an article's popularity, in relevance ranking. To incorporate these factors, you need to directly control the ranking function. Function queries give you this power. With function queries, you directly define the ranking function based on a combination of quantitative factors (such as popularity, publication date, and profitability) and other search queries.

Before solving our "star trek" problem, let's look at a classic example: prioritizing recently published news articles over old ones. When users search for "bad apple harvest" on a news site, a recent bad apple harvest is likely what they're after. Unfortunately, out of the box, the search engine has no notion that the recency of publication is an important factor. The apple harvest of 1901 could easily outrank the apple harvest of 2011!

You need to tell the search engine to prioritize news articles published recently. You need to get inside users' heads, modeling their expectations by using math.

To define a relationship between recency and the user's notion of relevance, let's start with a simple, relatively naïve formula, shown in figure 7.4. Let's define a function,  $R(m)$ , as  $1/m$ , where  $m$  is the months into the past this article was published. Let's use this function as a multiplicative boost: a single month into the past multiplies relevance by 1, two months knocks it down  $1/2$  (half as relevant), three months  $1/3$ , and so forth.



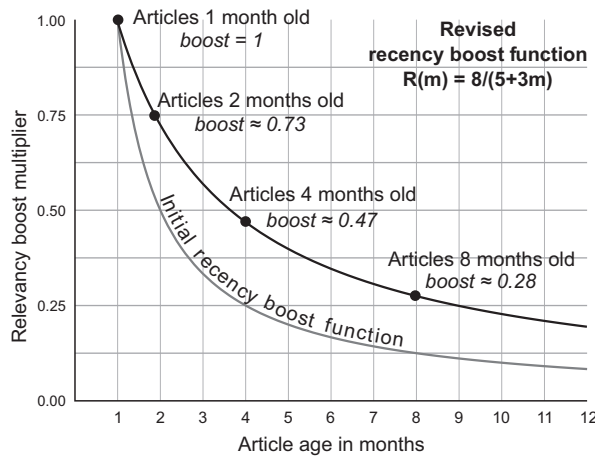
**Figure 7.4** Naïve recency boost for news articles

Now, as is often the case, this first pass might not model how news users prioritize this recency signal. Knocking down news four months old by a multiple of  $1/4$  might be far too aggressive. Then again, it might not be aggressive enough!

Just as when boosting with Boolean queries, the signal you're building might be inaccurate. To optimize the signal, you're tweaking the math itself, not just  $TF \times IDF$  and other text-scoring factors (what you might call *fudge factors*, we call *score shaping*).

Yet even with the increased mathematical freedom, the imperatives are the same: you need to make sure your signals are accurate too. Here the pertinent questions are as follows: Does the recency effect degrade too fast into the past? Or too aggressively? Is the boost a large enough multiple? Should it degrade using a different mathematical formula? All of these questions matter in measuring how users are likely to prioritize recency.

For example, you may have learned that your users are news analysts. They prioritize recency but still need to occasionally research older news stories. The bad apple harvest last year still matters. Through some tweaking (and fudging!), perhaps a recency function such as  $R(m) = 8 / (5 + 3m)$  works better to make the recency signal less aggressive, and more attuned to the needs of your news analysts, as shown in figure 7.5.



**Figure 7.5 Tuned recency boost: a less aggressive recency boost for analysts**

Manipulating math to arrive at the correct signal comes with many of the same burdens as manipulating other relevance factors. Does the formula account for how users and the business prioritize recency? This is the identical discussion as in the Boolean query boosting, only the possibilities with arbitrary math become far more open, creating many more decision points.

### 7.2.5 Hands-on with function queries: simple multiplicative boosting

Let's switch back to our Star Trek example to get our hands dirty with function queries in the Query DSL. You need to see a function query in the context of something basic before tackling larger mathematical problems. To ease you in, we'll give you a starting point: solving our Star Trek title boost. Later in this chapter, you'll begin to tackle a larger, more challenging problem that extends beyond the simple function presented here.



In this case, you want documents with Star Trek titles to be multiplied by a factor of 2.5. The function you'd like to implement applies a straightforward multiplicative boost that, expressed mathematically, looks something like this:

$$B(\text{title}) = \text{"star trek" in title?} \begin{cases} \text{no: } 1 \\ \text{yes: } 2.5 \end{cases}$$

This straightforward function returns one value when a particular match occurs, and another when it doesn't.

Let's get to work! Elasticsearch's function queries are known as `function_score_query`, as shown in listing 7.3. At the heart of the `function_score_query` is the base query `query` parameter ❶—another full-fledged Query DSL query. Being combined with that query's relevance score are mathematical functions ❸. Here the only function is our simple star trek function ❷. By specifying a weight and a filter as shown in listing 7.3, all documents that satisfy the filter (in this case, a phrase query for star trek in the title) receive a value of weight (here, 2.5) for this function. The `function_score_query` lets you specify how to combine the functions, but here we leave it as the default: multiplication of all functions with the base query score.

Let's execute this listing and see what shakes out.

### Listing 7.3 Applying a multiplier for Star Trek movies

```
usersSearch = "william shatner patrick stewart"
query = {
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query": usersSearch,
          "fields": ["overview", "title",
                    "directors.name", "cast.name"],
          "type": "cross_fields"
        }
      },
      "functions": [
        {
          "weight": 2.5,
          "filter": {
            "query": {
              "match_phrase": {
                "title": "star trek"
              }
            }
          }
        }
      ]
    }
  }
}

search(query)
Results:
Num  Relevance Score  Movie Title
1    1.9986789       Star Trek V: The Final Frontier    5.4
2    1.6982971       Star Trek: Generations            6.5
```

❶ Base query

User's query

❸ Functions applied to the base query

❷ Function multiplier: base score × 2.5 when the phrase "star trek" occurs in a title

3	0.6236526	Star Trek: Nemesis	6.3
4	0.60909384	Star Trek II: The Wrath of Khan	7.1
5	0.5075782	Star Trek IV: The Voyage Home	6.7

The end result is a reasonable boosting effect: Star Trek movies are brought to the top. You'll also notice that whereas in the additive boosting example you had to struggle with whether the  $TF \times IDF$  score of the boost query made any sense, here that score isn't taken into account. In some ways, though more power and responsibility is in your hands, the end result is much simpler.

### 7.2.6 *Boosting basics: signals, signals everywhere*

You've seen the basic forms of boosting. One constant remains, regardless of the query form you choose to work with: the essential power of signals. Your signal modeling work is fundamental to relevance. With boosting, this is especially true. Does the boost for Star Trek layer on enough "oomph" to matter? Does  $TF \times IDF$  scoring measure what's important to users? Is the right query being used? The right mathematical function? Do these factors map to a user's intuitive priorities?

The better you get at shaping the features and signals to measure this information, the stronger your ability to solve ranking in terms that humans, not search engines, understand.

### 7.3 *Filtering: shaping by excluding results*

The final basic ingredient when shaping scores is *excluding* search results. Users often don't want to see certain results, and your job is to ensure that they're excluded. You carefully craft queries to express what should be excluded, and then use finely honed signals to control those exclusions. This way, you tune the precision by excluding known classes of irrelevant results. This section briefly introduces filters and their role in the overall ranking function.

You often think of filtering when implementing user-experience features. Filters remove a set of results from consideration. You can think of them as tools to declutter search. Allowing users to manually select filters helps guide them to relevant content through the user interface. Perhaps they self-selected a category, such as deciding to limit themselves to seeing only the DVDs in a movie search, preferring to ignore the digital streaming or Blu-ray options. Or they might filter down when shopping for a TV to a particular set of criteria: 50-inch, plasma, free shipping. Guiding users toward relevant content with filters is a topic of the next chapter.

Yet filters aren't simply a matter of controlling the user experience. Filters act as a gate. A carefully implemented filter helps more precisely control the precision of search results, eliminating content you've expressly declared as irrelevant.

A simple example can be taken from our Star Trek search. In the previous section, we focused on boosting the Star Trek results to the top. Another way to think about this is to remove search results that aren't Star Trek results. What would this look like?

In Elasticsearch, filters take the form of a filter clause within a Boolean query. Within the filter, you can define how you'd like to restrict the document set; in the following listing, we're doing it by using a phrase query.

#### Listing 7.4 Filtering instead of boosting Star Trek results

```
usersSearch = "william shatner patrick stewart"
query = {
  "query": {
    "bool": {
      "should": [
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["overview", "title",
                     "directors.name", "cast.name"],
            "type": "cross_fields"
          }
        }
      ],
      "filter": [{
        "query": {
          "match_phrase": {
            "title": "star trek"
          }
        }
      ]
    }
  }
}
search(query)
```

User's  
query

Include only  
"Star Trek"  
results

When scoring doesn't matter, and you only want to show or not show a set of search results, filtering can be a simpler solution. By scoring only a more limited set of search results, filtering allows your boosting to work with more precision. You eliminate obviously irrelevant results, letting you avoid corner cases in your relevance.

You ought to think about how you filter just as you think about boosting: in terms of signals. The queries you use to filter need to measure user and business criteria with high confidence. In the previous section, you concerned yourself with whether that phrase query measures the right criteria. The same is true here. If, for some reason, this isn't the right way to measure the signal "this is a star trek film," then you need to think through these queries. Filter queries can get as complex as boosting, as they try to include and exclude results.

As you continue to think through ranking problems in your career, be aware of all the tools in your toolbox. In some cases, other solutions aside from boosting might provide more straightforward functionality.

## 7.4 Score-shaping strategies for satisfying business needs

How should you use score-shaping tools like boosting and filtering to solve your specific problems? Remember, this chapter is about bending the Query DSL to your will. But in reality, it's not *your* will that you should be concerned about; it's all those challenging user and business ranking requirements you need to solve! This section helps you tackle real business and user needs with the tools you've learned thus far.

Score-shaping problems involve translating human, plain-English ranking priorities into boosts, filters, and queries that control the ranking function. Nontechnical bosses, content curators, and other colleagues like to ask for things such as “bring the movies with an exact-title match straight to the top” or “boost movies that came out in the last five years.” Your job requires translating these plain-English expectations into “search-engineese” by incorporating the right data and mastering your search engine's features.

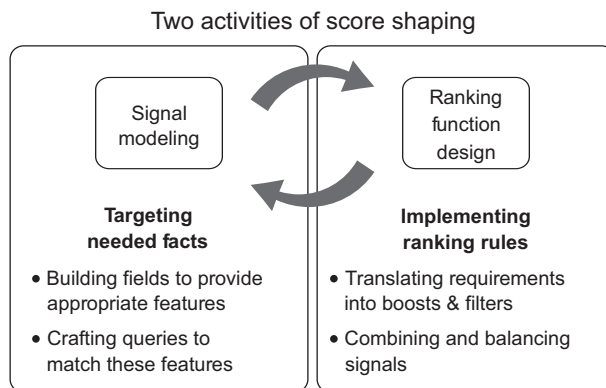
In this section, you'll explore strategies for manipulating the search engine to answer these questions. We introduce a framework for thinking through score-shaping problems based on two components.

First, our framework rests on a bedrock of high-precision boosting signals that rank based on questions in terms our users are familiar with: “the movie is an exact-title match” or “the movie came out > 5 years ago.” To build these signals, you incorporate the right data, carefully control the composition of fields, and govern how they're queried and scored.

Second, our framework considers how you combine expectations into a larger solution. You might want to override ranking one way based on the strength of a particular signal, otherwise falling back to a base query. On the other hand, you might have little nudge or tie-breaker boosts—nice-to-have, lower-priority boosts that nudge relevance up or down.

These two phases, shown in figure 7.6, are highly interdependent.

How much your boost nudges or overrides might depend exactly on the precision of the signal. Perhaps it's a text-based relevancy nudge. Or a nudge based on multiplying



**Figure 7.6** Two activities of score shaping are signal modeling (measuring pieces of ranking information) and the ranking function (combining the influence of different signals to implement business logic).

a carefully crafted function. You're always traversing both layers, optimizing signals, thinking how they'll be used, and modifying the ranking function accordingly. Search is highly iterative. With boosts, you have deep control over the ranking function—so this is where the iterations become fun. Let's see this philosophy in action by solving a real problem!

### 7.4.1 Search all the movies!

To flex the full power of this fully functional search engine, you're going to switch gears. You've done such a good job implementing search for those Star Trek fans that you've been promoted! Your next job is to implement movie search for everyone. You'll get a chance to implement some even more sophisticated ranking criteria to serve this broader audience. Let's take a second to consider what you're being told to implement for your search solution. Then you'll get your hands dirty with an interesting, nontrivial search solution that will mirror the tricky problems you're sure to face at your real job.

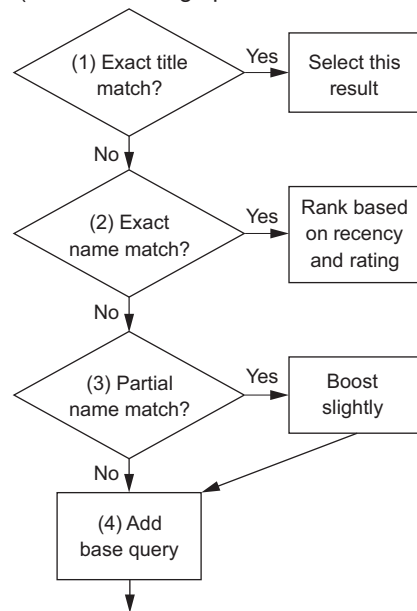
You've been told by your boss, "To get this search right, I think you generally need to implement the following logic in that search engine thing of yours. Here ya go:"

- 1 If the user's query is a full, *exact*-title match, such as a search for "Star Trek Generations," then that exact-title match should come straight to the top of the search results.
- 2 If the user's query is a full, *exact*-name match, such as a search for "William Shatner," then order results based on how recent and well rated the movie is.
- 3 Prioritize search results that include a person's full name, such as the query "William Shatner Star Trek."
- 4 Otherwise, base your scoring on a general text-based relevance of the user and a query.

Figure 7.7 captures these rules as a flowchart.

There's quite a bit to unpack. The first two criteria make it important to measure exact matches with high precision. You'll have to craft fields that can capture this ranking signal. Implementing step 2, the exact-name match, is interesting. You'll likely match many films when searching for a person, so your boss is thinking ahead when they

Flowchart of movie search ranking rules (from the vantage point of the business)



**Figure 7.7** Flowchart representation of the custom business ranking rules as understood/specified by the business

come up with how these ought to be ranked. Users, your boss is thinking, may want to see the latest and best offerings for that actor or director. You've been working on step 3 for about two chapters by now. So your boss must continue to have confidence in your abilities. You'll see how you can use this to boost relevance for a narrow set of matches, similar, as you may recall, to what you worked with at the end of the preceding chapter. Finally, step 4, your base relevance score, measures how many search terms match the overall document. Recall from the previous chapter, term-centric search implements this functionality.

Sounds like great marching orders, but hopefully alarm bells are going off in your head. How does the business know these are the right criteria? Why should these be the priorities for your ranking solution? These are important issues, but fret not, in future chapters you'll get answers to those precise questions. Getting the criteria right *is* important! But so is knowing how to translate these plain-English ranking rules to search-engineese, which you get to focus on in this chapter.

Let's get started! Your first step when given this criteria is looking at your fields to see whether you can implement the required signals. Once you have confidence that you can support the required signals—that your fields can turn into facts—then you begin to work on the second piece of boosting: crafting the ranking function to your needs.

### 7.4.2 *Modeling your boosting signals*

As a relevance engineer, when you're told to boost by something like “when the search is an *exact* match for the title,” your next question ought to be, how exactly do I measure *that*? That's what you'll tackle in this section. You'll translate the whims of human language into an implementation of a particular signal that you can work with in a larger ranking function. You've done this a few times already in this book. Yet this step is fundamental to being able to program relevance ranking rules. Without actionable information, how do you know whether your ranking function is prioritizing the right documents?

To implement your business requirements, you need to make sure that you can answer these important questions before crafting the overall ranking function:

- Is the user's query a full, exact match to the title?
- Is the user's query a full, exact match to the full name of an actor or director?
- Does the user's query contain the full name of an actor or director?

Your goal is to create highly discriminating signals. You need signals that act like snobby wine critics—carefully letting the essence of each document waft under their discerning noses and accepting a document only after careful consideration. You've begun this process: you have reasonable name-matching fields based on bigrams from previous chapters. These address your third criteria—whether part of the query matches a person on the film with high precision. We'll get back to our old name-matching friend, but first let's look at how to measure the other ranking criteria.

Two of these signals require exact, not partial, matching. To accomplish this, you can't simply wrap the user's query text into a phrase query. A search for the phrase "Star Trek" will match any document with Star Trek in the title, including *Star Trek: Generations*. This isn't an exact match between query and title. No, you want to boost *only* when the terms from the query line up exactly to the text in the title.

So now it's time for more signal modeling! You need to build a field that can measure this information with high precision. Astute readers might be thinking—exact matches? That's easy! Disable tokenization or use the keyword tokenizer to search on the exact-title string. This would generate exact tokens such as `star trek` or `star trek: insurrection`. Yes, that might work, but it rarely returns what you want. You want *some* analysis, even when doing so-called exact matching. For example, you'd like to remove that pesky colon in `star trek: insurrection` to help the user searching without the colon. You might also want some amount of stemming or any other form of per-term normalization.

Instead of disabling tokenization, you'll use a technique referred to as *sentinel tokens*. Sentinel tokens represent important boundary features in the text (perhaps a sentence, paragraph, or begin/end point). In this case, you'll inject the following begin and end sentinels into the title text:

```
"SENTINEL_BEGIN Star Trek SENTINEL_END"
```

These sentinels are tokens representing the beginning and end of the string. Then, when you search, you include the sentinels in the query, searching for the phrase query "SENTINEL\_BEGIN <user query> SENTINEL\_END." The result is a document that matches only when the phrase query matches *and* the correct sentinels are in the right place (in this case, when the beginning of the text and end of the text coincide).

You could implement this in a couple of ways. One intelligent way is to write a plugin for your search engine. As you may recall from chapter 4, token filters intercept and modify the token stream during analysis. You *can* write your own Lucene token filters to inject boundary features. Pick up a great Solr, Elasticsearch, or Lucene book to show you how. For brevity, you'll take the simpler road, injecting the sentinels outside the search engine.

First, you'll modify your TMDB index function to inject the sentinel characters before and after the title in a new field you'll call `title_exact_match`. To do this, you'll modify your TMDB reindex function to call a transform function before indexing. This will be your little hook to manipulate the document before going off to Elasticsearch.

```
for id, movie in movieDict.iteritems():
    ...
    esDoc = movie
    transform(esDoc)
    ...
```

Next, you'll define the transform function to create the field you need to answer our question with the sentinel tokens included.

#### Listing 7.5 Injecting begin/end sentinels for exact matching

```
SENTINEL_BEGIN = 'SENTINEL_BEGIN'
SENTINEL_END = 'SENTINEL_END'
def transform(esDoc):
    esDoc['title_exact_match'] = SENTINEL_BEGIN + ' ' + \
                                esDoc['title'] + ' ' + SENTINEL_END

reindex(analysisSettings, mappingSettings, movieDict)
```

I hope your head is swimming with all kinds of questions about how well this will work. Are all these settings, analysis, mappings, and so forth appropriate to measure this signal? There are many possible optimizations to think about. Remember the default analyzer? It's the entity whose job it is to break the string into tokens. We're set up to default to the English analyzer. Is that appropriate? The English analyzer will match "Stary Trek" to star trek because of stemming—do you want this behavior? Or do you want something as close as possible to character-by-character exact match? Your job is to sweat many of these details to ensure that you're measuring the ranking signal important to your users.

For now, let's see what happens when you issue a query for this exact-match signal. Does searching using the sentinels against this field give you the signal you need?

#### Listing 7.6 Isolated testing of your exact-match signal

```
usersSearch = "star trek"
query = {
    "query": {
        "match_phrase": {
            "title_exact_match": {
                "query": SENTINEL_BEGIN + ' ' + \
                        usersSearch + ' ' + SENTINEL_END,
                "boost": 0.1
            }
        }
    }
}
search(query)
Num  Relevance Score      Movie Title
1    7.172676             Star Trek
```

User's  
query with  
sentinels

Great, this works! You've built a field expressly for exact matching. As you grow in your skills, you'll keep being nagged by questions. Is  $TF \times IDF$  scoring important here? Will I use it as a signal of something? Likely not. As your skill increases, so will your awareness of all the factors that can impact the signal you're measuring.



Finally, you'll repeat the same process for your name field. For this field, you'll create a non-bigrammed, regular old text field, just like the preceding one.

#### Listing 7.7 Name and title exact-match fields

```
SENTINEL_BEGIN = 'SENTINEL_BEGIN'
SENTINEL_END = 'SENTINEL_END'
def transform(esDoc):
    esDoc['title_exact_match'] = SENTINEL_BEGIN + ' ' + \
                                esDoc['title'] + ' ' + SENTINEL_END
    esDoc['names_exact_match'] = []
    for person in esDoc['cast'] + esDoc['directors']:
        esDoc['names_exact_match'].append(SENTINEL_BEGIN + ' ' +
                                           person['name'] + ' ' +
                                           SENTINEL_END)
```

Adds title exact match field with sentinels

Adds name exact match field with sentinels

Now you have some ingredients to work with in your larger relevance solution. You've built and tested relevance signals in isolation. Next you'll look at how to incorporate the name and title exact matching signal into the larger ranking function. You can always come back to this piece if you need to improve any aspect of your ranking (or come up with a signal completely anew).

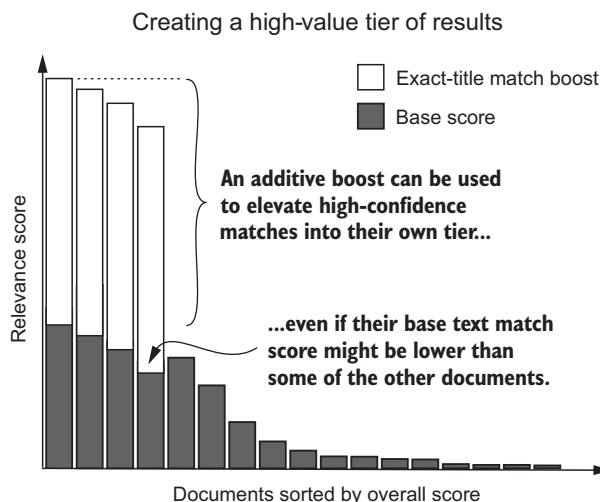
### 7.4.3 Building the ranking function: adding high-value tiers

With the pieces in place, the next step is to ask precisely how to combine them into a larger ranking function. The ranking function is where signals collide—hopefully into a harmonious whole, not a train wreck!

This section introduces one of our favorite shaping techniques: building *scoring tiers*. It's often useful to express ranking in terms of tiers based on your confidence in the information provided in the boosting signals. Particular signals that provide unequivocal information about what's being searched should not simply be “layered in” or “balanced against other factors.” If you have truly high-precision, discerning signals that definitively point at what's being searched for, then to even consider much else would be foolish. Instead, in this technique you let these higher-value matches sit on their own highly scoring tier, pushing their scores into a class all their own. Scoring at those tiers is dictated by what's important for those sets of search results, such as the exact matches of the previous section.

Figure 7.8 shows two kinds of scores. First there are the base, black scores. These match the base query. This query pulls in a wide net of possibly relevant results. In our example, you'll reuse the term-centric search from earlier in this chapter. Second there are enhanced, white bars. These results have been heavily boosted.

You could implement this high-value tier with a variety of boosting techniques. Listing 7.8 uses a Boolean query; you can see the boost query as the first Boolean



**Figure 7.8** High-recall base query in black augmented with a high-precision tier of unequivocally more valuable search results

clause. Notice how the boost weight ❸ is set astronomically high—all the way to 1,000.

#### Listing 7.8 Boolean boost on exact-title matching

```
query = {
  "query": {
    "bool": {
      "disable_coord": True,
      "should": [
        {
          "match_phrase": {
            "title_exact_match": {
              "query": SENTINEL_BEGIN + " " + \
                usersSearch + " " + SENTINEL_END,
              "boost": 1000,
            }
          }
        },
        {
          "multi_match": {
            "query": usersSearch,
            "fields": ["overview", "title",
                      "directors.name", "cast.name"],
            "type": "cross_fields"
          }
        }
      ]
    }
  }
}
```

❶ Boosting query—exact movie title match

❷ Base query

❸ High boost to create an overriding effect

User's query

```
search(query)
```

You can see that the results for a search for “Star Trek” are promising, bringing the target straight to the top:

Num	Relevance Score	Movie Title
1	7.1752715	Star Trek
2	0.0020790964	Star Trek: The Motion Picture
3	0.0020790964	Star Trek: Nemesis
4	0.0020790964	Star Trek: Insurrection
5	0.0020790964	Star Trek: First Contact

And here’s a search for “Good Will Hunting”:

Num	Relevance Score	Movie Title
1	7.914943	Good Will Hunting
2	0.0016986724	The Hunt
3	0.0012753583	Good Night, and Good Luck.
4	0.0011116106	As Good as It Gets
5	0.00058992894	Saw V

Here you’ve added a little rule to your ranking function that should work well for a common use case—exact-title matching. As you move forward with your problem, we’ll begin to call out specific patterns in the boosting.

#### ADDING A NEW TIER FOR MEDIUM-CONFIDENCE BOOSTS (THE BIGRAMS STRIKE BACK!)

So far you’ve added one tier to the larger Boolean query: the exact matches. Let’s examine another precise but less exacting case: a search that contains a full name as *part* of the search. This case calls for some prioritization, in its own tier, but not quite as much as exact-name matches. To incorporate this boost, you’ll work from two directions:

- You’ll use a smaller boost of 100 to create an intermediate tier.
- You’ll optimize the signal in the bigram matching, to ensure that you measure what’s important.

The second point is the most pressing. Let’s start with the bigram name matching from the previous chapter. You used a `cross_fields` search over `cast.name.bigrammed` and `directors.name.bigrammed`. In the following listing, we use this query as a boost alongside the base query. Unfortunately, it turns out that running the combined query doesn’t always end up with great results, as in this search for “Star Trek Patrick Stewart.”

**Listing 7.9 Adding a clause for bigrammed matches (base query not shown)**

```
'query': {
  'bool': {
    'should': [
      { 'multi_match': {
        'query': usersSearch,
        'fields': ['overview', 'title',
                  'directors.name', 'cast.name'],
        'type': 'cross_fields'
      } },
      { 'multi_match': {
        'query': usersSearch,
```

Base query

Mid-tier layer for name matching

User's query

```

        'fields': ['directors.name.bigrammed',
                   'cast.name.bigrammed'],
        'type': 'cross_fields',
        'boost': 100
    }]]}]

```

1	0.21988437	Star Trek: Insurrection	6.3	1998-12-10
2	0.21988437	Star Trek: First Contact	6.9	1996-11-21
3	0.19890885	Gnomeo & Juliet	5.9	2011-01-13
4	0.19890885	Excalibur	6.7	1981-04-10
5	0.19462639	Star Trek: Nemesis	6.3	2002-12-12

It's odd that *Gnomeo & Juliet* and *Excalibur* rank higher than *Star Trek* results. Why do these non-*Star Trek* Patrick Stewart results override the *Star Trek* ones? Turning to the explain, you see that, oddly, the *Excalibur* match on “Patrick Stewart” gets scored higher than *Star Trek: Nemesis*.

Here's *Excalibur*'s score:

```
0.5910474, weight(cast.name.bigrammed:patrick stewart in 315)
```

The *Star Trek: Nemesis* match receives a lower score:

```
0.5171665, weight(cast.name.bigrammed:patrick stewart in 631)
```

So why the difference? The *Excalibur* match receives a higher score because of the field norms. Remember this bias toward shorter fields? *Excalibur* has a smaller crew, so matches here appear more relevant to the search engine. But our users don't care about this; that factor doesn't matter in answering the question, “Does the movie star Patrick Stewart?”

Time to do more signal modeling. Just as with the previous sentinel tokens, you'll need to tightly control field matching and scoring. You can revisit the mapping, reindex with norms disabled, and rerun the search query. First, dig back into the deeply nested mapping to disable norms and reindex the movies:

```

mappingSettings['movie']['properties'] \
    ['cast']['properties'] \
    ['name']['fields']['bigrammed']['norms'] = {'enabled': False}

reindex(analysisSettings, mappingSettings, movieDict)

```

Searching again, you see that the intermediate tier makes more sense:

1	0.03920228	Star Trek: Insurrection	6.3	1998-12-10
2	0.03920228	Star Trek: First Contact	6.9	1996-11-21
3	0.03917096	Star Trek: Nemesis	6.3	2002-12-12
4	0.03917096	Star Trek: Generations	6.5	1994-11-17
5	0.03820324	Gnomeo & Juliet	5.9	2011-01-13

All Patrick Stewart movies move to the top. Within that set, the base query for “Star Trek Patrick Stewart” ranks more precisely.

**BUILDING A TIERED RELEVANCE LAYER CAKE**

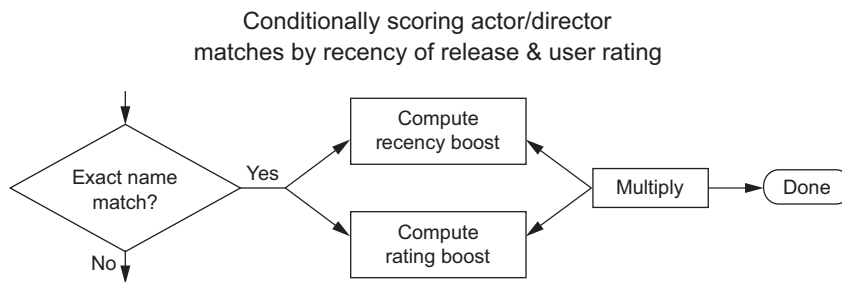
In the preceding examples, you implemented part of the ranking criteria asked for by your business. Exact-title matches were so compelling that your boss told you to ignore all other considerations. Placing exact-title matches in a higher-scoring tier does this. Similarly, name matches were compelling, but less overwhelmingly so.

Both of these boosts work well because you optimized their queries to more closely align to the needed signals. The boost query is assigned a boost weight in these situations not because the query has that level of priority, but because the signals are so unbelievably precise and discriminating. Matching each means winning the relevance lotto. For once, you happen to know exactly what's being searched for, so take advantage of that! You'll use another high-value strata in the next section as you think through exact-name-matching criteria, but the focus will shift to a more mathematical modeling of users' goals.

**7.4.4 High-value tier scored with a function query**

The next major challenge requires you to flex even deeper muscle in programming the ranking function. As you program relevance, you'll learn increasingly sophisticated techniques that build on each other. In this section, you'll take the high-value tier from the previous section and combine it with a second technique: the careful incorporation of a finely tuned function query based on features of the film.

The business has asked for specific ranking criteria when there's an exact-cast-name match. In these cases, your boss wants to show a combination of popular and recently released films. Doing this requires you to flex two sets of ranking muscles simultaneously, as depicted in figure 7.9.



**Figure 7.9** Flowchart representing the custom business ranking rules for exact-name (cast/director) matching

First, you need to bring those high-confidence, exact-name matches up into their own scoring tier. You know that these exact-name matches stand above the rest, so they should be selected aggressively. Second, and new to this section, that scoring tier

needs its own method of ranking. It needs to be based on two features of the films associated with those actors: the recency of their release and their user ratings.

You'll incorporate three parts in this next component of your query:

- The *exact-name matching* itself, used to trigger recency and rating
- The *recency* of the film, based on the release date
- The *user rating*, the 1–10 average rating from all the users' individual ratings of the movie (1 meaning terrible, 10 meaning best ever)

Your job in this section is to combine the influence of the latter two, but only when the first criteria (the exact-name matching) triggers.

As you build this part of the query, you'll become aware of how large these queries can get! You may be about to build one of the largest Query DSL queries you've ever worked with. Yet you're learning to cope with these behemoths in the same way you think through any large piece of code. Each scoring component has a purpose, a signal that it's lending to the overall ranking function. If you feel overwhelmed in your work, never hesitate to bring it down to a simpler level: working with the component queries in isolation before combining them.

How will you go about adding this functionality to the larger query? This query will be built into the Boolean query from the preceding section. You'll end up with three clauses: two you've already introduced, and a third, your new Boolean query. This Boolean query will look something like this:

- SHOULD: Base query (listing 7.8 ②)
- SHOULD: Full query exactly matches a title, boost 1,000 (listing 7.8 ①)
- SHOULD: Query contains full name, boost 100 (listing 7.9)
- SHOULD: Full query exactly matches a name (new to this section)

### 7.4.5 Ignoring $TF \times IDF$

You want to rank a film by its user rating and recency of release, but only when the query is an exact-name match. This ranking doesn't depend, at all, on the  $TF \times IDF$  of the exact-name text query. This isn't uncommon: frankly, many times you need a simple yes/no instead of the  $TF \times IDF$  score. You saw this earlier when working with the *star trek* title boost. Let's begin to dig into the exact-name function query, demonstrating a technique for omitting  $TF \times IDF$ .

As you might guess, because scoring for this query will be dominated by two numerical factors, recency and user ratings, you'll use a `function_score` query. Recall that with a function query, you specify a base query with the `query` argument. Your first attempt could be to insert the exact-name-matching phrase query as the base query:

```
query = {
  "query": {
    "function_score": {
      "query": {
        "match_phrase": {
```

```

    "names_exact_match": SENTINEL_BEGIN + \
                          " william shatner " + \
                          SENTINEL_END
  }},
  ...

```

You need to make one change here; you need to consider the fact that the  $\text{TF} \times \text{IDF}$  score of this query is unlikely to be useful. Remember, by default the `function_score` query will take the score for the query and multiply it by the resulting functions. This means multiplying the  $\text{TF} \times \text{IDF}$  score of the exact-name matching by the functions. In some cases, you need a way to ignore the  $\text{TF} \times \text{IDF}$  scoring. To do this, you can wrap your query in a `constant_score` query. This lets you hardcode the resulting query's score to a constant (the boost value). As you're focused on bumping this score into a discriminating higher tier, let's set this constant to a very high 1,000:

```

"function_score": {
  "query": {
    "constant_score": {
      "query": {
        "match_phrase": {
          "names_exact_match": SENTINEL_BEGIN + \
                              " william shatner " + \
                              SENTINEL_END
        }
      },
      "boost": 1000.0}}

```

Great! So now your base relevance score will be constant when the user's query matches exactly with an actor or director name.

To complete this query, you need to think through the functions themselves. You need to balance two considerations: the user rating and the recency. Each provides a signal to tune in isolation. So for a bit, you'll work on these in isolation. You'll see how each signal takes advantage of its own set of techniques to line up its influence with user expectations. Getting these signals precise and lined up to user expectations is crucial.

#### 7.4.6 Capturing general-quality metrics

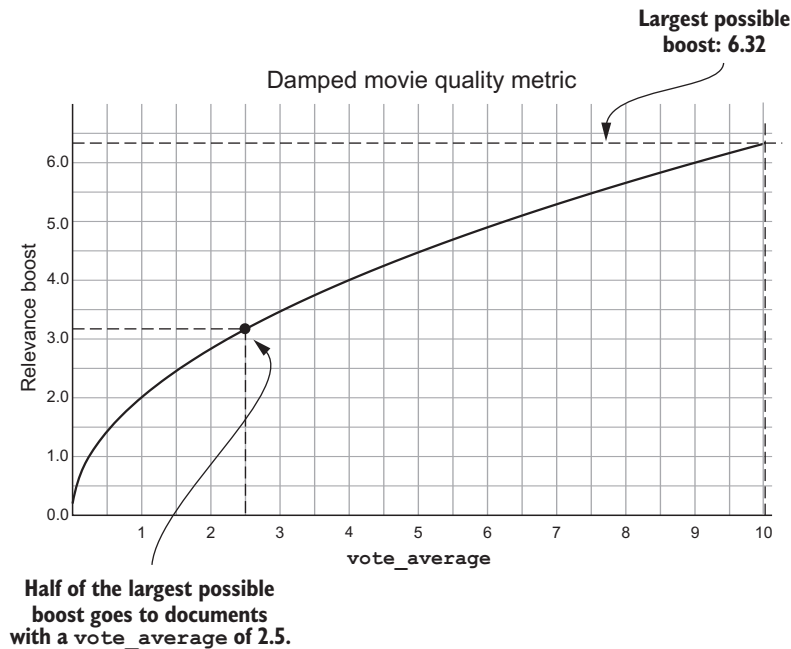
First, let's consider a boost on user rating. It's time to get back to thinking mathematically. The user rating field (called `vote_average`) is an example of a document property that directly measures the content's value. You'll often boost on other similar direct indications of quality: profitability, popularity, page views, and other factors. In this section, as you hone this signal, you'll explore what it means to include these general-quality features. Should you take its value directly or modify it somehow? Perhaps temper it down? Or should it be magnified? What do users think of the importance of this quality metric?

Incorporating `vote_average` depends on Elasticsearch's `field_value_factor` function. This function lets you take a field's value and use it directly. Optionally, you can apply a few simple modifiers and functions. In the following listing, for example, you take the square root of `vote_average` and multiply by 2.

#### Listing 7.10 Factoring in user's ratings of the movies

```
query = {
  "query": {
    "function_score": {
      "query": {
        "match_all": {}
      },
      "functions": [
        {
          "field_value_factor": {
            "field": "vote_average",
            "modifier": "sqrt",
            "factor": 2
          }
        }
      ]
    }
  }
}
```

This can be visualized in the graph shown in figure 7.10.



**Figure 7.10** Boosting user rating, an indication of content quality. The square root is taken to dampen the effect in line with how users perceive this metric's value.



Why take the square root? You're aiming for a signal that measures how important the user considers movie ratings. If you took the value directly, a rating-10 movie would be considered twice as valuable as a rating-5 movie. Users rarely perceive the influence of quality factors so starkly and directly. Taking the square root, as you can see by examining the graph, roughly ranks a rating-3 movie as half as important as a rating-10 movie.

These quality metrics are tricky, as you rarely want to incorporate them directly as a signal. Just as you did here, you have to ask how users perceive the value as a function of these features. For example, what if instead of a 1–10 user rating, this quality metric had been something like the number of page views of the movie's TMDb home page? Most pages have single-digit page views. A fractional amount have several hundred. A small handful get thousands, and one or two get millions of page views. Is a page that gets a million views a million times higher quality? No! It might be more reasonable to your users to measure it as twice as important. How you tone down the influence of these factors can be crucial. Users' considerations are more subtle than simply hammering in the quality metric directly.

#### 7.4.7 Achieving users' recency goals

You're still working on creating custom ranking rules when users match exactly on cast or director names. The second signal to consider for this case is the user's perception of the importance of a movie's recency. You'll see in this section that ranking is often about modeling how far a result is from an expressed goal. For example, users want to rank news items close to the present, or restaurants close to their home, or perhaps televisions near their target price. Elasticsearch's built-in *decay functions* rank based on how far documents are from target goals. You'll see in this section how to think through modeling a simple goal-based ranking problem based on a TMDb film's recency.

At the core, though, is how you express goals to the search engine with three primary variables: origin, decay, and scale. The *origin* is meant to indicate the user's goal (or ideal). At the origin point, scoring is at the highest (a 1.0). The *scale* is also a point, off in the distance. At this location, you can imagine the user declaring, "At this point from my goal, I consider the document this much less valuable." It's as if the user were saying, "If I have to drive 20 minutes from home to pick up food, that food is now half as interesting to me." How do you declare how much less valuable? Well, that's the *decay* parameter's job. It states the value at a certain location, given the scale. Taken together, you could rephrase the user's restaurant statement: "If I have to drive <SCALE> minutes from <ORIGIN> to pick up food, that food is now <DECAY> less interesting to me."

Given our query, you'll start by stating that movies released 900 days ago (Elasticsearch helps with same-date syntax) will have roughly a score of 0.5, or be considered half as valuable to users. This is expressed in the following isolated query.

**Listing 7.11** Gaussian decay from the user's recency goal

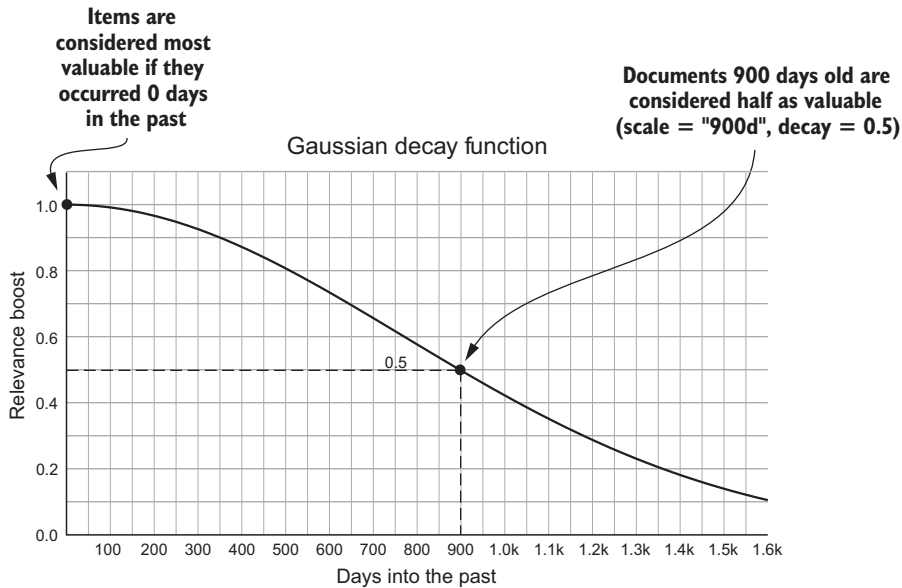
```

query = {
  "query": {
    "function_score": {
      "query": {
        "match_all": {}
      },
      "functions": [
        {
          "gauss": {
            "release_date": {
              "origin": "now",
              "scale": "900d",
              "decay": 0.5
            }
          }
        }
      ]
    }
  }
}

```

The equation for this Gaussian decay is complex, but to give you an idea of how it behaves, we've included the graph in figure 7.11. This graph demonstrates the user's perceived value for an actor's or director's film as a function of days into the past, showing that films 900 days into the past are half as valuable. As movies move five to six years into the past, the influence of this function begins to approach 0—near worthless!

Here, you might ask whether the decay function is too aggressive. Perhaps scale is too close in. Why's that? Consider one possible use case: a user searching for an actor

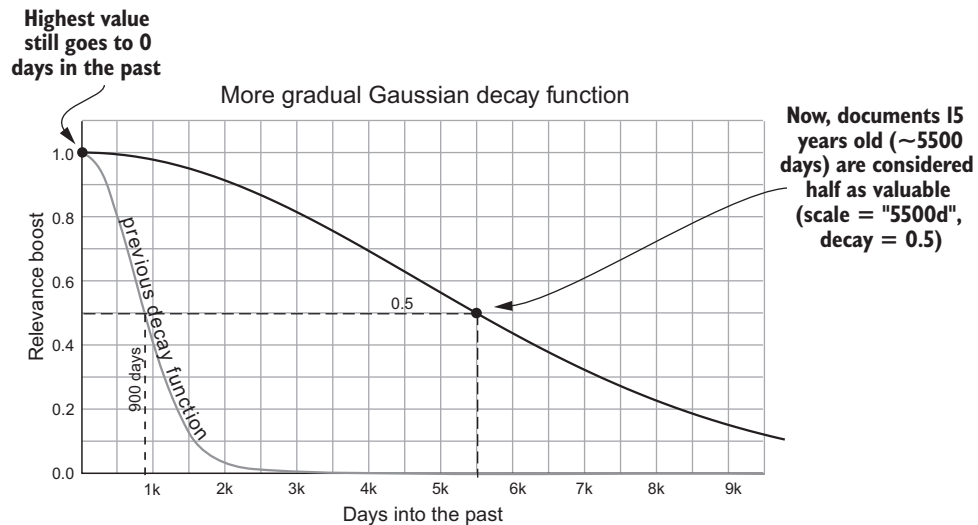


**Figure 7.11** A multiplicative boost controlling the impact of distance from your user's goal; here the goal is movies released "now," with movies released 900 days into the past considered half as valuable.

wants not only recent films but also films considered classics for that actor. You need to apply a careful balancing act between the conflicting goals of users who want the most recent films and those who want an understanding of what the really great films are, regardless of the recency.

Now this is an interesting point, because not all users are the same. As we discuss in a later chapter, one option could be to suss out which users have which priorities. For the purposes of this chapter, however, let's come up with a reasonable, generic solution. The proposed function considers films from even five to six years ago near worthless, approaching 0. It's unlikely, you decide, that the average user considers films from not this long ago that invaluable. For now, let's extend how far into the past this function decays. Instead of a scale of 900 days, let's use something far more conservative: 15 years.

The graph in figure 7.12 demonstrates this modified function. The x-axis measures thousands of days. The decay happens gradually, over the course of decades, instead of quickly over the course of a few short years.



**Figure 7.12** A tweaked multiplicative boost for movies; movies released 15 years into the past are deemed half as valuable.

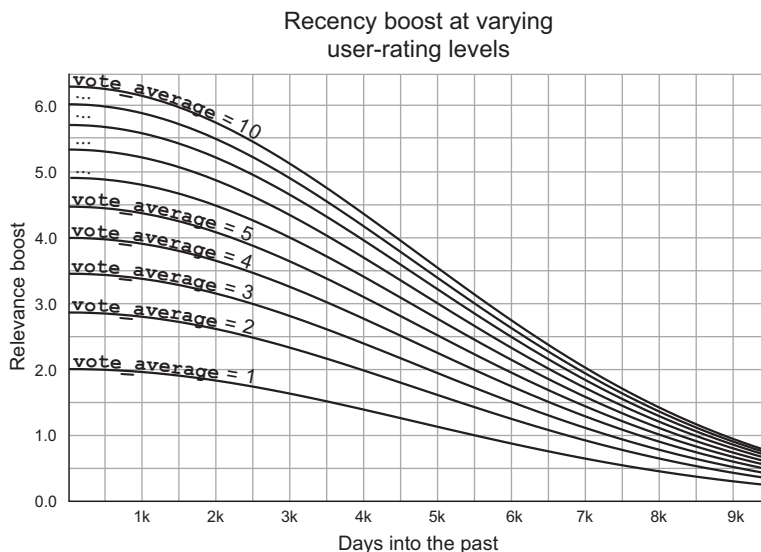
This function, you decide, is probably a better model of the average user's recency goals when matching on cast/director names. Now you've modeled the user as thinking that films released 15 years into the past are half as valuable. As is true with any signal, this is a starting point. You can always revisit this signal in isolation to assess whether it's still too aggressive. (Plenty of amazing movies came out more than 15 years ago!) Fine-tuning these knobs to generate the right signal is a matter of readjusting your understanding of the goals. Luckily you now have all the tools you need to calibrate relevance to users' goals.

These decay functions give you the ability to generate a signal based on how far an item is from a user's goal. This is a common pattern, not just something limited to dates and geography. Consider a real estate search in which ranking is based entirely on whether a home meets a user's goals. Factors such as school quality, commuting distance, price range, and proximity to parks can be ranked by how far an item is from where a user ideally would like to be. Sometimes these ranking problems aren't seen as search, but as more of a kind of machine-learning problem. But here the lines begin to blur. You're seeing how search engines are a framework for programming all kinds of ranking capabilities, based on far more than just text factors.

### 7.4.8 Combining the function queries

It's time to start building back up to our main query. Remember, you've been going down a path trying to model two signals that matter to users who search for exact actor/director names. You've been trying to consider how much the movie's rating and its distance into the past might matter to users. It's time to get back to programming the main ranking function. To do that, let's consider the impact of combining these two curves.

You need to visualize how multiplying these two variables will impact the relevance score to determine whether their influence is appropriate. One way to visualize this is to draw a cool 3D graph. Yet sometimes overlaying a few 2D graphs, each of which varies one of the variables, can be a more useful way to answer critical questions. The graph in figure 7.13 represents the multiplication of the movie rating function and the decay function.



**Figure 7.13** Recency curves based on the user rating (`vote_average`) field for a movie, demonstrating how both recency and user rating impact the relevance score for exact-name matches.

The bottommost line corresponds to the recency graph of movies with a rating of 1. The topmost corresponds to movies with a rating of 10. What does this tell us about the combination of these two factors? One thing to notice is how movies of rating 1 relate to movies of rating 10. At what point into the past does a movie with a rating of 10 become less relevant than a movie with a rating of 1? This happens a little past day 6,000, or roughly 16.5 years from now.

You can reflect on the combination of these two factors. How do they work together to signal relevance? This is particularly true here: these functions *are* the relevance calculation for this case. Isolating how they work together can help you reason through their impact. Do these two factors have the right priority? What could be improved? Perhaps the recency should have a floor; users may want a slight nudge toward recent films, and care little about the difference between 80s and 70s films, for instance. Reflecting on and evaluating all of these factors is your constant job—and as you’ll see in chapter 10, the whole organization’s job!

Finally, with these functions taken together, let’s complete the person exact-matching query. This in turn will be folded into the larger query.

#### Listing 7.12 Clause for exact-name matching, ranking based on recency and user rating

```
usersQuery = "patrick stewart"
query = {
  "query": {
    "function_score": {
      "query": {
        "constant_score": {
          "query": {
            "match_phrase": {
              "names_exact_match": SENTINEL_BEGIN + " " + \
                                usersSearch + " " + \
                                SENTINEL_END
            }
          },
          "boost": 1000.0
        },
        "boost": 1000.0
      },
      "functions": [
        {
          "gauss": {
            "release_date": {
              "origin": "now",
              "scale": "5500d",
              "decay": 0.5
            }
          },
          "field_value_factor": {
            "field": "vote_average",
            "modifier": "sqrt"
          }
        }
      ]
    }
  }
}
search(query)
```

**Base query, score simply taken as the “boost”**

**Boost for recency**

**Boost for rating**

Results:

Num	Relevance Score	Movie Title	vote_average	release_date
1	2.762838	X-Men: Days of Future Past	7.7	2014-05-23
2	2.4984634	The Wolverine	6.4	2013-07-25
3	2.4377568	Ted	6.3	2012-06-29
4	2.2800508	Gnomeo & Juliet	5.9	2011-01-13
5	1.9779315	TMNT	6.0	2007-03-22

Interestingly, our search for “Patrick Stewart” results first in a relatively high-rated film that was released relatively recently. Notice too the difference between the fourth and fifth results. A 5.9-rated movie released in 2011 outranks a 6.0 released in 2007. These expectations seem to correspond to the preceding graph that demonstrates the impact of different ratings.

### 7.4.9 *Putting it all together!*

This query is simply a single clause in an even larger query! Yet we’ll spare you from seeing the full thing (you can view the examples on GitHub to see the full query in all its glory). Nevertheless, outlined in your overall query is a set of four Boolean clauses, including these criteria:

- SHOULD have full query exactly matching the title (listing 7.8, ❶)
- SHOULD have full query, exactly matching a director’s or cast member’s full name, scored by popularity and recency (listing 7.12)
- SHOULD have part of the query match a director’s or cast member’s full name (listing 7.9)
- SHOULD have all the user’s query terms match somewhere in the document (listing 7.8, ❷)

Combining these four SHOULD clauses in one `bool` query, you’ve done it! You’ve solved a particularly complex ranking problem. You’ve taken thoughts, whims, and requirements expressed in English and brought them to the search engine—bending the search engine to your will!

Now you can begin to ask some of the bigger questions. Just as in real programming, once you realize that you know *how* to program, you suddenly become conscious of the larger question: exactly *what* should you be programming? In the same way, as you become increasingly confident with programming relevancy, you’ll begin to see that this challenge applies to relevance as well. As you move into future chapters, finding the right ranking requirements will become front and center. How do you define what requirements/use cases are important to your search? How do you test to make sure you’re continuing to meet requirements? Here they’ve been given to you—a technical challenge. As you’ll see in future chapters, real life is never so simple!

## 7.5 Summary

- Score shaping, including techniques such as boosting and filtering, is about programming results ranking to satisfy your business/user needs.
- Boosting comes in several forms: Boolean and function queries, additive and multiplicative approaches.
- Boolean queries abstract ranking for you, providing a simple means to add a boost. Calibrating them correctly layering on additive boosts.
- Function queries allow you to take control of ranking math by boosting in arbitrary forms. Calibrating them means modeling user priorities mathematically.
- Filters often provide an alternative to boosting, removing low-priority results instead of trying to promote high-priority results.
- Score shaping depends on your ability to implement high-quality signals and incorporate them in the ranking function based on business rules.
- High-quality signals can be prioritized by placing them in their own scoring tier.
- You can implement a broad range of ranking forms, including modeling user goals, incorporating content quality metrics, and the like.

# 8

## *Providing relevance feedback*

---

### ***This chapter covers***

- The ongoing, multifaceted conversation between user and search
- Methods, besides relevance ranking, for getting users to relevant content, including
  - Guiding users toward better search queries
  - Correcting users' searches through spell-checking
  - Highlighting why documents are relevant to user searches
  - Explaining to users how their search is interpreted
  - Allowing users to filter out irrelevant content from the result set

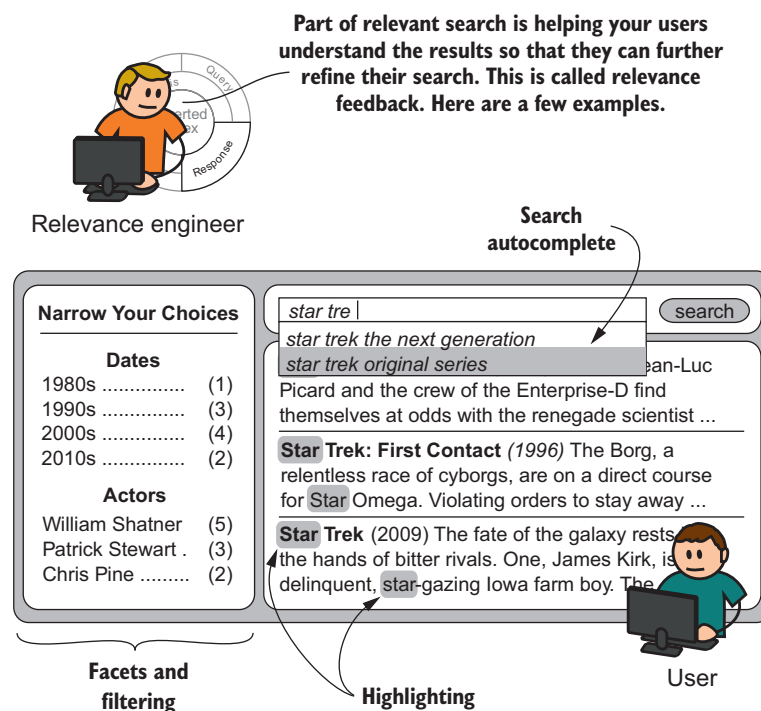
To this point, we've discussed how to deeply manipulate relevance ranking. In this chapter, you'll see that relevance ranking isn't the only way to guide users to relevant content. Search enables a multifaceted conversation between the user and the search engine. Because the relevance portion of this conversation is never perfect,



your users will always thank you for additional guidance toward relevant content. In this chapter, you'll steer the many layers of the search conversation beyond relevance ranking by building features that do the following:

- Explain to users how their query is being interpreted
- Correct mistakes such as typos and misspellings
- Suggest other searches that will provide better results
- Convey an understanding of how documents are distributed in the corpus
- Help users understand why a particular document is a match
- Help users efficiently understand the result set

We term these conversational aids *relevance feedback*. In this chapter, we provide an overview of relevance feedback across three areas of search user experience: the search box, browsing and filtering, and the search results. Figure 8.1 shows the activities covered in this chapter; the relevance engineer implements supporting capabilities to guide the user toward relevant content in the various parts of the search application.



**Figure 8.1** Relevance feedback helps users refine their searches and find the information they seek.

In your work, you'll see that guiding the user's search behavior can be an easier problem to solve than perfect relevance ranking. In this chapter, you'll find alternate paths for the user to reach relevant content. We won't be exhaustive. There are innumerable forms of relevance feedback, and countless methods for implementing them. Rather, we hope to get you started down a crucial path of the relevance engineer: enabling the two-way search conversation through any means possible.

## **8.1 *Relevance feedback at the search box***

The user and search application first interact at the search box. You may be surprised by the extent to which search box interactions provide the user with relevance feedback. By providing information to users while they type and immediately after they submit a query, you enable users to refine the query or sometimes even find what they need before submitting it. This section covers three common forms of relevance feedback at the search box:

- Search-as-you-type
- Search completion
- Postsearch suggestion

### **8.1.1 *Providing immediate results with search-as-you-type***

*Search-as-you-type* is just what you'd expect from the name: as the user types in keywords, the search engine proactively provides documents that match. The goals for search-as-you-type are twofold.

The first goal is relevance feedback. The user can glean the likely effectiveness of the query from search-as-you-type results. If the results are too broad, the user will continue to add search terms to further qualify the documents he wants to find. If the results are off the mark, the user can revise the current search.

Second, you may be able to bring relevant documents back to the user more quickly. Without search-as-you-type, users have to consciously submit a query before seeing results. With search-as-you-type, users can choose to quit typing and select a document that matches their requirements.

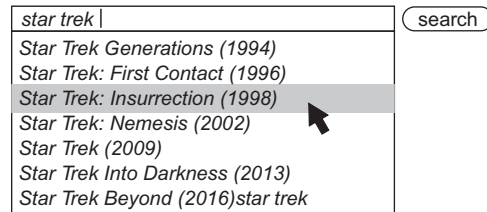
Implementation of search-as-you-type is straightforward. As the name indicates, the search application issues a series of searches as the user types. But there's at least a little nuance here. For example, if a user searches for "Star Trek the Next Generation" and types halfway through a keyword (for example, "Star Trek the Next Gener"), then you shouldn't include the trailing partial keyword in the search without indicating that it's a prefix rather than a completed word. Fortunately, Elasticsearch provides a `match_phrase_prefix` query that implements this approach. Consider the following query:

```
{ "query": {  
  "match_phrase_prefix" : {  
    "title" : "star trek the next gener"}}}
```

If this query is provided to the `/_validate/query?explain` endpoint, the query explanation shows that, as expected, the query is interpreted as a phrase query with the last term expanded via a trailing wildcard:

```
"explanation": "title:\"star trek the next gener*\""
```

As displayed in figure 8.2, applications typically present search-as-you-type matches concisely in a drop-down menu. This menu enables the user to easily select matching documents.



**Figure 8.2** Search-as-you-type is a means of providing users with immediate results. It's often presented as a drop-down menu below the search box.

But the drop-down menu presents some UI/UX challenges. For one thing, this area may be needed to provide suggested search terms to the user (search completions, described in the next section). Having both features share space can confuse users who must differentiate between two types of feedback. Also, the drop-down menu's limited space makes it impossible to give much information on a matching document. Unless search matches only titles, users won't always be able to determine why documents are relevant. Finally, the drop-down lacks application state. For instance, if a user selects a document from the drop-down menu and then immediately realizes that this document isn't relevant, how do they switch back to the previous search-as-you-type view? Typically this isn't possible, or is clumsy at best.

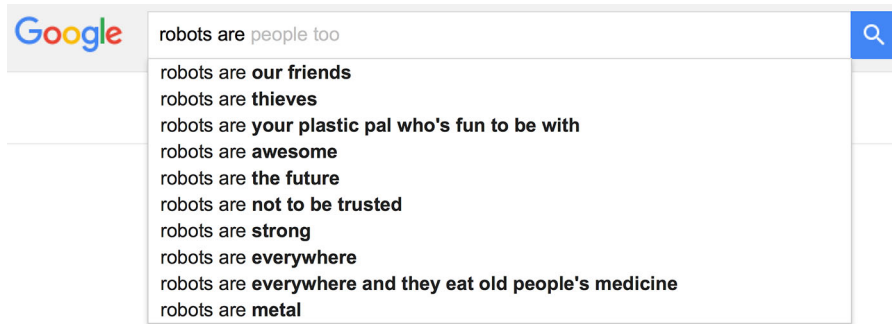
Google's search application uses an interesting search-as-you-type approach that avoids these problems. Rather than placing search-as-you-type results in a drop-down menu, Google presents them in the normal space for search results. This pattern helps eliminate possible user confusion. Placing results in their normal location reserves the drop-down menu for search-completion suggestions. Therefore, the user can focus attention on either the search results or search-completion suggestions. With search results in their normal location, there's more room to provide richer information per result than could be provided in a drop-down menu.

### 8.1.2 Helping users find the best query with search completion

*Search completions* usher users to better search queries and keywords. In effect, the search application brainstorms with the user about what they're looking for: "User, you typed *robots are*, do you want to search for *robots are our friends*? How about *robots are awesome*?" If you finely tune this interaction, it becomes almost subconscious,

with users iteratively typing, reformulating, and occasionally selecting the resulting completions.

As illustrated in figure 8.3, search applications usually present completions in a drop-down menu below the search box. Selecting any of the drop-down items will add the selected text to the current query.



**Figure 8.3** Search completions are typically presented in a drop-down menu below the search box. Selecting an item from the drop-down replaces the current query with the selection text.

Implementing search completion can be challenging. Users expect search completions to return shortly after they begin typing. If completions are too slow, users won't recognize that completions are available. The user then misses out on your guidance, searching unaided. Completions must also be highly relevant. Users will learn to ignore irrelevant suggestions and instead forge ahead unaided. Similarly, if users select a completion, but it leads to unexpected or 0 results, then the user will feel misguided.

As with most features discussed in this chapter, you can implement search completions in many ways. In the following discussion, we present three methods.

#### **BUILDING COMPLETIONS FROM USER INPUT**

As you prepare to build search completions, the first question you must ask yourself is "From what data source should I pull the completion text?" With enough traffic, you could base completions on past users' queries. This makes sense: allowing users to search with past users' searches should assure the current user that they are not far off the beaten path, right?

Maybe—but be careful; you need to consider a few gotchas. First, let's look at the assumption that you have enough traffic. Is your application a *short-tail* search application where a handful of queries (hundreds, perhaps) represent the majority of your search traffic? Or is yours a *long-tail* application where thousands of queries represent only a small portion of the search traffic? With too few queries, you may have insufficient data to build a satisfactory completion experience. Too many queries, and you'll have to prioritize what's most important from a large, diverse set of completion candidates.

Also consider whether old search traffic becomes obsolete for your application. Consider search completions for an e-commerce application that frequently changes inventory. Searches important a month ago may no longer match currently available products.

Finally, and perhaps most damagingly, old user queries are a distinct data set from the corpus. It's easy to find user queries that return 0 results! Suggesting such a query as a search completion would befuddle your users. If search completions lead to poor experiences, the user learns not to trust the search application.

Search completion based on past user queries *can* provide a good user experience; but the search engineer must ensure that the user's queries provide sufficiently rich and timely completions. Additionally, the search user should be reasonably assured that the completion will lead to relevant search results.

### BUILDING COMPLETIONS FROM THE DOCUMENTS BEING SEARCHED

A more straightforward approach for building search completions uses the text of the content being searched. This approach ensures that any completion recommended to the user corresponds to text in the corpus.

Let's look at how search completion might be implemented using the text in the TMDb data set. In your work, you'll first need to consider which fields could be used for completion. For TMDb, let's consider two: `title` and `overview`. Some users look for movies by their name. For these users, the `title` field has obvious utility. Other users, however, search by typing movie details. The richer `overview` provides better content for these users' search completions.

But with richer text comes complications. With so much text, the `overview` field contains quite a bit of variety and noise. The user might be disappointed with seemingly arbitrary completions pulled from `overview` text. For these verbose fields, you'll have to seek strategies for prioritizing which search completions ought to come first, which in itself is its own ranking and relevance problem. For now, let's keep things simple by using only the `title` field for a source of completion text.

The next thing to consider is how the text should be analyzed. In chapter 4, you explored several strategies for analyzing text in order to help users find what they're looking for. There, you used stemming or another form of token mutation to collapse several words, such as `happy`, `happier`, `happiest` into a single token: `happi`. These tokens won't suit as a search-completion data source. A user entering the partial word "happ" would be presented with the completion `happi`, which isn't a word!

Instead you'll preserve readability during analysis, as shown in the following listing.

#### Listing 8.1 Analysis setup for 2-gram completions

```
"settings": {
  "analysis": {
    "filter": {
      "shingle_2": {
        "type": "shingle",
        "output_unigrams": "false"
      }
    }
  }
}
```

1 Shingle filter  
for two-word  
bigrams

```

"analyzer": {
  "completion_analyzer": {
    "tokenizer":
      "standard",
    "filter": [
      "standard",
      "lowercase",
      "shingle_2"]]]}}}

```

Completion analyzer for generating completion text

You create an analyzer called `completion_analyzer`. The analyzer splits the text on punctuation and whitespace (using the *standard* tokenizer) and then lowercases the tokens. The lowercasing helps match/suggest regardless of the case of the content or search string. As an additional step, to support phrase suggestions, you use a shingle filter ❶ to generate two-word phrases. The analyzer transforms the string `star trek: into darkness` into the tokens `star trek`, `trek into`, `into darkness`.

Now that you have your `completion_analyzer`, let's use it on the title text. To do so, you'll copy the title field into a completion field that uses the `completion_analyzer`, as shown next.

#### Listing 8.2 Mappings for title-based completions

```

"mappings": {
  "movie": {
    "properties": {
      "title": {
        "type": "string",
        "analyzer": "english",
        "copy_to": ["completion"]},
      "completion": {
        "type": "string",
        "analyzer": "completion_analyzer"}}}}}

```

Duplicates title field to completion field

Completion field for holding completion text

And finally, after you index the documents, you're ready to construct your completion search. This search yields the most common two-word phrases that match the already typed query.

#### Listing 8.3 Query used to generate completions

```

{ "query": {
  "match_phrase_prefix": {
    "title": {
      "query": user_input}}},
  "aggregations": {
    "completion": {
      "terms": {
        "field": "completion",
        "include": completion_prefix + ".*"}}}}

```

❶ Limits candidates to titles that match the user's query so far

❷ Suggests search completions from title text

❸ Limits suggested completions to those that begin with "completion prefix"

Let's walk through how listing 8.3 works to deliver completions. Notice the two variables, `user_input` ❶ and `completion_prefix` ❸.

The `user_input` variable holds the full query string typed by the user. If the user types "Star Tr," `user_input` is then `star tr`. You place `user_input` into the same `match_phrase_prefix` query ❶ introduced earlier for search-as-you-type. This query forces completions to use titles that match the user's query. In our example, this prevents you from using `star wa movies` to suggest completions to Star Trek fans.

The completion prefix ❸ is the word currently being typed. For "Star Trek," this is `tr`. This limits completions to phrases the user could be trying to type (phrases that start with `tr`). Sometimes, especially when the last word typed is too short, it helps to include the previous term to limit further (phrases that start with `star tr`). If there's not even enough text for that (if the user just started typing), you may wish to omit completions altogether. There's too little context to aid the user.

Referring again to listing 8.3, you retrieve search completions by using an Elasticsearch `terms aggregation` ❷. Given a query (in this case, the `match_phrase_prefix` query), a terms aggregation collects a list of all terms that exist in the documents matching this query. Elasticsearch returns the list of terms, sorted according to the number of documents that contain the term. At ❸, you tell Elasticsearch to limit this to include only those terms that begin with `completion_prefix` (`tr`).

This strategy results in a list of the most common title phrases that match the already typed query. The response returned after issuing the search in listing 8.3 includes a completion section that looks like the following.

#### Listing 8.4 Example response for aggregation-based search completion

```
{'completion': {'buckets': [
  {'doc_count': 1, 'key': 'trek 3'},
  {'doc_count': 1, 'key': 'trek axanar'},
  {'doc_count': 1, 'key': 'trek first'},
  {'doc_count': 1, 'key': 'trek generations'},
  {'doc_count': 1, 'key': 'trek horizon'},
  {'doc_count': 1, 'key': 'trek ii'},
  {'doc_count': 1, 'key': 'trek iii'},
  {'doc_count': 1, 'key': 'trek insurrection'},
  {'doc_count': 1, 'key': 'trek into'},
  {'doc_count': 1, 'key': 'trek iv'}
], 'doc_count_error_upper_bound': 0, 'sum_other_doc_count': 4}}
```

Using aggregations to generate completions guarantees that completions match your documents within the context of what the user has typed. For instance, if a user supplies the partial query "Star Tre," the completion prefix in this case will be `tre*`. Based on only this prefix, you could return completions such as `treasure island` and `treading water`, but neither has anything to do with `star`. Because aggregations take the search context into account, the completions returned for `star tre` will include only context-appropriate items such as `trek generations` and `trek insurrection`.

Table 8.1 shows examples of completions that would be made as a user searches for a Star Trek movie.

**Table 8.1** Completions for a user searching for a Star Trek movie

User input	Completion prefix	Completions
st	(Not used—too little context)	(Not applicable)
star	star	star wars star trek starship troopers
star t	star t	star trek
star tr	tr	trek 3 trek axanar trek first trek generations

Another nice side-benefit of this method is that it can be easily combined with search-as-you-type because, as you can see in listing 8.3, you’re indeed issuing searches as the user types via the `match_phrase_prefix` ❶.

You can definitely improve this approach. The current strategy limits completions to two words. It could be more ideal to complete an entire movie title. This could be accomplished with a different tokenization strategy that generates longer completions. For instance, you could tokenize titles by using the `path_hierarchy` tokenizer with `delimiter` set to the space character and with `reverse` set to `true`. This splits movie titles on whitespace and saves the ends of movies as tokens. For example, *Star Trek: The Motion Picture* would be tokenized as `star trek the motion picture`, `trek the motion picture`, `the motion picture`, `motion picture`, and `picture`. These sound like great completions!

This approach has one sticking point, though: the single-word tokens are going to be much more prevalent than the longer tokens. But by reversing the terms aggregation to return completions from least to most prevalent, you can provide the user with the most specific and often the longest completion available.

Before you implement aggregation-based search completions, be aware that this method comes with drawbacks. First, heavy aggregations over large text fields can tax the search engine, especially in an application that uses distributed search. If you choose to use this method, ensure that response times are acceptably low; otherwise, your completions won’t keep up with users’ keystrokes. For this reason, this approach works best with smaller corpuses and with text fields with a relatively small set of unique terms.

Second, by default this method returns completion suggestions ordered from most commonly occurring to least commonly occurring. This isn’t always an appropriate metric. For instance, referring to listing 8.4, notice that all possible completions



occur exactly once in the index. Therefore, in this case it isn't possible to establish an ordering for the completions based on their prevalence in the index. To address these problems, we introduce our final search-completion strategy: specialized completion indexes.

### BUILDING FAST COMPLETIONS VIA SPECIALIZED SEARCH INDEXES

Because completion is such an important element of relevance feedback, Elasticsearch introduced a specialized component called the *completion suggester*. This component circumvents the performance problem referenced previously and allows for custom sorting of completion results (rather than sorting by occurrence). Effectively, the completion suggester is a specialized search index that's stored in parallel with the normal search index. It's backed by a compact data structure (a *finite state transducer*) that provides a fast prefix-lookup capability. In many ways, this approach is the ideal solution for completion, but as you'll see in a moment, it introduces a couple of problems of its own.

Setting up the completion suggester is simple: you just declare one of the fields to be of type `completion`.

#### Listing 8.5 Setting up Elasticsearch's completion suggester

```
{ "mappings": {
  "movie": {
    "properties": {
      "title": {
        "type": "string",
        "analyzer": "english",
        "completion": {
          "type": "completion"
        }
      }
    }
  }
}
```

Field of type "completion" to use Elasticsearch's completion suggester

In principle, you could copy the title field to the completion field and index the documents in the same way as demonstrated in the preceding section. But if you do, you forego one of the main benefits of using the completion suggester: the ability to directly specify the weight of the completion. This weight affects the order in which completions are suggested. In the following listing, we enrich the given document with a new completion field that uses the movie's title for completion text and the movie's popularity as the completion weight.

#### Listing 8.6 Enriching documents with a completion field using popularity for weight

```
doc = {
  "title": "Star Trek Into Darkness",
  "popularity": 32.15,
  /*...other fields*/
}

doc["completion"] = {
  "input": [doc["title"]],
  "weight": int(doc["popularity"])
}
```

Original document

Document enrichment

Weights must be integers

After you index these enriched documents, you can query for the completion. In the following listing, you use Elasticsearch's `_suggest` endpoint rather than the `_search` endpoint you used before. (Though if you like, you can include a `suggest` clause within a normal Elasticsearch search and get suggestions along with search results.)

**Listing 8.7 Retrieving search completions via the `_suggest` endpoint**

```
GET /tmdb/_suggest
{ "title_completion": {
  "text": "star tr",
  "completion" : { "field": "completion"}}
```

In this example, you search for a completion to the prefix text `star tr`. As in the previous case, the results are a list of Star Trek movies, but this time you sort by popularity rather than occurrence counts. This provides the user with much more relevant results. Additionally, the completion suggester can be configured to perform fuzzy matches so that appropriate completions can be returned despite a certain degree of user error.

The benefits of the completion suggester come at a cost. Because the completion suggester is implemented as a separate index internal to Elasticsearch, it isn't aware of the full context of search. For instance, consider a user who wants to find the Star Trek movie in which Spock dies (sorry, spoiler alert). What if the user searches for "Spock Dies Star Trek"? Obviously, no title completion will start with `spock dies star`, so a good completion implementation attempts to find completions starting with the second term—`dies star`—again, no completions. The completion implementation then moves on to find completions beginning with just `star`. Here, based on the context of the entire query, it would be unreasonable to return anything but Star Trek movies. But in the case of the Elasticsearch completion suggester, the top completion result is *Star Wars: Episode IV—A New Hope*. This demonstrates that the completion suggester is unaware of the search context.

Another unfortunate consequence with using a finite state transducer to back completion is that the data structure is immutable. If a document is deleted from the index, the completions for that document will still exist. Currently, the only remedy for this situation is a full-index optimization, which effectively rebuilds the completion index from scratch.

Despite the possible pitfalls, the completion suggester is an important tool for relevance feedback. The Elasticsearch completion suggester can occasionally lead users astray, but, it allows you to specify the criteria by which completions are sorted (in this case, popularity). And the completion suggester is typically the fastest method for retrieving completion suggestions. This rapid feedback to users can become an almost subconscious aid, helping to direct them to more targeted searches and ultimately to the documents that they're looking for.

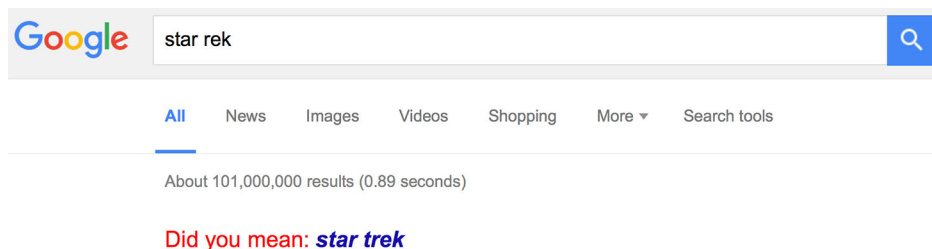
### WHICH SEARCH-COMPLETION METHOD IS BEST

In the preceding sections, we covered three search-completion methods: completions based on user input, completions based on the text in the index, and completions based on specialized indexes. Each technique helps bring your users closer to the information they seek. But as you've seen, there's no silver bullet; each technique has its share of benefits and drawbacks. Equipped with this knowledge, you'll at least have a better starting point for building your own completion solution.

#### 8.1.3 Correcting typos and misspellings with search suggestions

Search completions provide suggestions to users as they type. But another opportunity to provide relevance feedback arises immediately after the user's query is submitted. Users make mistakes—misspellings or typos—while entering their searches. After they submit a malformed query, the search engine can provide suggestions to modify and improve the query.

Here again Google serves as a good example of how post-search suggestion can provide relevance feedback to users. As displayed in figure 8.4, if Google receives a query that contains an obvious typo, it replaces the user's query with the correction that the user likely intended. Similarly, if the user's query likely contains a typo, but the situation is more ambiguous, then Google retrieves results corresponding to the user's original search, but also suggests a different query likely more aligned with the user's goals. In either case, the presentation of this information is paramount. If users are unaware that their query has been replaced, they may become disoriented and begin to mistrust the search application. Conversely, if users are never made aware of a typo mistake, they may similarly be disappointed, believing that the search application isn't able to find the anticipated results.



**Figure 8.4** Google uses post-search suggestions to provide users with relevance feedback.

You can implement post-search suggestions by using Elasticsearch's phrase suggester. To set up the phrase suggester, the mapping must contain a field specifically set aside for suggestion. Similar to the completions discussed previously, this field must be tokenized so that the tokens are still readable, correctly spelled words. For

this example, you'll copy the title field into a suggestion field and use the default standard analysis chain:

```
"mappings": {
  "movie": {
    "properties": {
      "genres": {
        "properties": {
          "name": {
            "type": "string",
            "index": "not_analyzed"
          }
        }
      },
      "title": {
        "type": "string",
        "analyzer": "english",
        "copy_to": ["suggestion"]
      },
      "suggestion": {
        "type": "string"
      }
    }
  }
}
```

To pull back suggestions, you again use the `_suggest` endpoint:

```
GET /tmdb/_suggest
{ "title_suggestion": {
  "text": "star trec",
  "phrase": {
    "field": "suggestion"
  }
}
```

This returns suggestions that look like this:

```
{ 'title_completion': [{ 'length': 9, 'offset': 0,
  'options': [
    { 'score': 0.0020846569, 'text': 'star three' },
    { 'score': 0.0019600056, 'text': 'star trek' },
    { 'score': 0.0016883487, 'text': 'star trip' },
    { 'score': 0.0016621534, 'text': 'star they' },
    { 'score': 0.0016162122, 'text': 'star tree' }
  ],
  'text': 'star trec'
}] }
```

Each suggestion is scored, representing how strongly the suggestion matches the user's intended query. But as seen here, this doesn't always work out—*star trek* isn't the first suggestion. Let's make some improvements.

#### Listing 8.8 Retrieving post-search suggestions in the context of a user's search

```
{ "fields": ["title"],
  "query": {
    "match": {"star trec"},
    "suggest": {
      "title_suggestion": {
        "text": "star trec",
        "phrase": {
          "field": "suggestion",
          "collate": {
            "query": {
```

**1** Collation added  
to main query

```
"inline": {
  "match_phrase": {
    "title" : "{{suggestion}}"}]]]]]]]]]]
```

← 2 Alternate query to issue for each suggestion

In this listing, you include the suggestion in the body of the corresponding search so that you don't have to make two separate requests. But you also address the problem of the poor suggestions by using collation ❶. With *collation*, the search engine is performing a sort of mini-search using each of the suggestions and then removing the suggestions that don't have a match. Within the *collate* section, you specify the search that determines which suggestions are returned; if documents match a suggestion, that suggestion will be included in the suggestions passed back to the user. Here Elasticsearch replaces the special `{{suggestion}}` parameter with the text of each suggestion.

Notice that you use a *match\_phrase* query for the collation query. You do this to more tightly constrain the possible suggestions to phrases that exist in the index. For instance, if you had used the *match* query at ❷ for the collation, then the suggestion *star three* would have remained in the *suggest* response, because some documents contain either *star* or *three*, even though none include the phrase *star three*. By filtering out weaker suggestions, you ensure that the user is presented only with suggestions that lead to meaningful results.

We present the final result of *star trec* query suggestions in the following listing.

#### Listing 8.9 Search and suggestion results after collations have been applied

```
{ 'title_completion': [{ 'length': 9, 'offset': 0, 'options': [
  { 'score': 0.0019600056, 'text': 'star trek'},
  { 'score': 0.0016621534, 'text': 'star they'}]]}]}
```

There are several things to note here. First, notice that several of the previous suggestions have been removed because of the collation *match\_phrase* search. Also notice that the top-ranking suggestion is now the most appropriate suggestion for this search.

Another tough question: what should the application do with the suggestions? Should it outright replace the user's query with the highest-scoring suggestion? Or should you present the suggestion as a "did you mean" suggestion? Unfortunately, the answer is rarely clear-cut. The suggestions provided by the phrase suggester are scored as a function of the text edit distance and the frequency of the suggested terms. But just because a suggestion is in some sense "more likely" than the user's query, the user's query isn't necessarily incorrect; the user might be looking for something very specific.

Practically speaking, you can never read the mind of an individual user. Therefore, a good solution will involve reading up on all of the Elasticsearch phrase suggester parameters, building a solution, and watching how it performs. But as a rule of thumb, it's usually safer to direct users toward a better search with a "did you mean" suggestion rather than replacing a search without asking them.

There is, however, one big exception to this rule: if the user's query returns no results, it's always better to provide results that might be relevant rather than leaving the user with no results at all. And in either case (search replacement or post-search suggestion), make sure that the UI readily conveys what's happening so that users won't become disoriented by an interaction that falls short of their expectations.

## 8.2 *Relevance feedback while browsing*

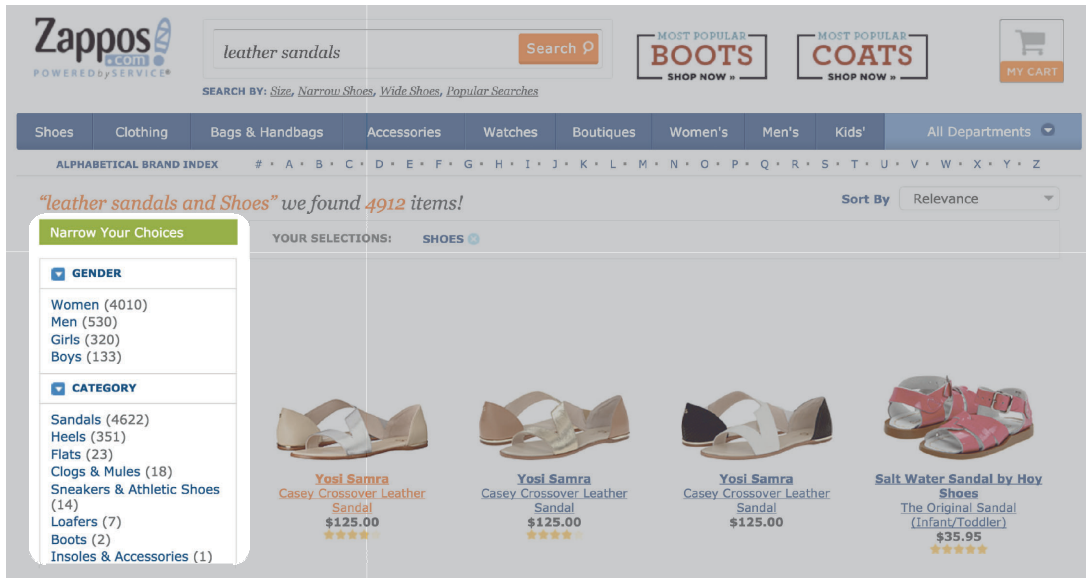
We've discussed several aspects of relevance feedback that can occur in or around the search box. But interaction at the search box is only a small part of the conversation between the search user and the search application. This section focuses on the *browse experience*.

A user can browse through the results of a search by selectively filtering those results, narrowing down to a small set that's most relevant to that user's information needs. Sometimes the user may even *start* by browsing through the documents rather than with a text query. The browse experience is different from interactions with the search box. Whereas search-box interactions provide subtle relevance feedback to users, the browse experience gives users a broad overview of how documents are distributed in the corpus. This lets users make intentional choices about how to filter through the documents.

*Faceted browsing* is the predominant method for facilitating browse behavior. Visit almost any e-commerce website and you'll see the same pattern: At the top of the page is the search box. Below the search box, and typically to the left side of the screen, is a list of categories. Within each category are items that can be selected to filter the results. These menu items are called *facets* (introduced in chapter 2). Often the facets include a count of the documents that match a particular filter. If a user selects a certain facet, the results are filtered according to that selection. As shown in figure 8.5, Zappos provides a great example of a faceted browsing interface.

Here a search for "leather sandals" returns a mix of sandals—mostly women's sandals, but you can also see a toddler's sandal. I (John) happen to be a grown man, so I'm uninterested in these. But looking at the categories on the left side of the screen, I see that I can filter these results according to product type, gender, and brand. Furthermore, within gender the products are divided into four groups: *Women*, *Men*, *Girls*, and *Boys*, and although the great majority of the products matching *leather sandals* are for women (4010 items), there are 530 items that I might be interested in. If I hadn't been presented with this information, I might have seen this initial set of products and assumed that Zappos wasn't into men's fashion. If I click the *Men* facet, the results will be filtered so that only men's sandals are displayed. From there, I may choose to further narrow my results by brand or by shoe size or by any other criteria that I choose.

Consider the great amount of relevance feedback that the faceted browsing interface is providing to users. With a quick scan of the information in the sidebar, users



**Figure 8.5** Faceted browsing helps Zappos users slice and dice the inventory and narrow the result set to include only items that most closely match their search goals.

gain a deep understanding of the products in the catalog and the metadata associated with these products. By looking at the document counts, users can also get a sense of how these documents are distributed within the categories. Finally, and most important, users can act on this knowledge by clicking a facet and narrowing the search results. In an e-commerce search, the ability for customers to quickly navigate the data and find the items they're looking for increases the likelihood that they'll make the purchase. These techniques also help you sidestep and simplify the relevance process. By giving users more options to guide themselves to what they want, you fret less about complex ranking and trying to read the user's mind.

In the following subsections, we discuss how faceted browsing is implemented in Elasticsearch. We also cover a couple of related topics. In the breadcrumb navigation section, we introduce a simple UI feature that helps users remain aware of the current facets they've selected. And in the result ordering section, we discuss how users can more closely control relevance ranking priorities.

### 8.2.1 Building faceted browsing

You can implement facets by using Elasticsearch's aggregation feature. For the purposes of this discussion, aggregations return a count for specific subdivisions of the current search results. Let's consider an example from TMDB—movie genre. All movies in the TMDB set contain a field `genres.name` that includes tags such as Adventure,

Comedy, and Drama. The following query counts the number of movies in the entire TMDb data set that fall within each genre:

```
GET tmdb/_search
{ "aggregations": {
  "genres": {
    "terms": {
      "field": "genres.name" }}}}
```

This instructs Elasticsearch to return an aggregation called `genres`. For each term in `genres.name`'s global term dictionary, the aggregation returns the number of documents with that term. In this instance, your terms correspond to movie genres. The aggregation response, shown in the following listing, can be used to populate a genre menu in your faceted browse interface.

#### Listing 8.10 Facet counts for movie genres

```
{ 'aggregations': { 'genres': { 'buckets': [
  { 'doc_count': 7546, 'key': 'Drama' },
  { 'doc_count': 5342, 'key': 'Comedy' },
  { 'doc_count': 3878, 'key': 'Thriller' },
  { 'doc_count': 3753, 'key': 'Action' },
  { 'doc_count': 2623, 'key': 'Romance' },
  { 'doc_count': 2165, 'key': 'Adventure' },
  { 'doc_count': 1981, 'key': 'Horror' },
  { 'doc_count': 1861, 'key': 'Crime' },
  { 'doc_count': 1640, 'key': 'Family' },
  { 'doc_count': 1597, 'key': 'Science Fiction' } ],
  'sum_other_doc_count': 7479 }
```

Scanning over this list, you can see that most movies in our set are labeled as dramas, followed by comedies, thrillers, and so forth. What's more, you can look at the `sum_other_doc_count` and see that 7,479 movies are in genres not listed in the top 10.

It's important to again draw attention to the analysis process. As in the previous sections in this chapter, you tokenize the documents so that the raw tokens can be presented back to the user. For this particular instance, it's important to *not* use the default standard analysis because it splits the input on whitespace. Splitting on whitespace would cause *science* and *fiction* to be considered as two different genres, which is clearly incorrect.

In this query, you don't constrain the document set, so the counts represent the distribution of all movies within the genres. This is fine; it's a great data set to present to users who haven't yet specified any other information. Users coming to the movie search application for the first time can quickly scan the distribution of movies across genres to decide where to look next.

But as users continue to interact with the search application, what do they do next? Presuming that they don't decide to leave (always a possibility), there are two options: search via the search box or click a facet item to narrow the focus. In either case, the



effective result is the same; they've narrowed the relevant documents to a subset of the corpus. The neat thing about aggregations is that the counts associated with each facet will be calculated based on this filtered set.

So let's say that the user clicks the Science Fiction genre, indicating a desire to see only movies tagged as science fiction. You can then filter the results set accordingly:

```
GET tmdb/_search
{"query": {
  "bool": {
    "filter": [{
      "term": {
        "genres.name": "Science Fiction"
      }]
    }
  },
  "aggs": {
    "genres": {
      "terms": {"field": "genres.name"}
    }
  }
}
```

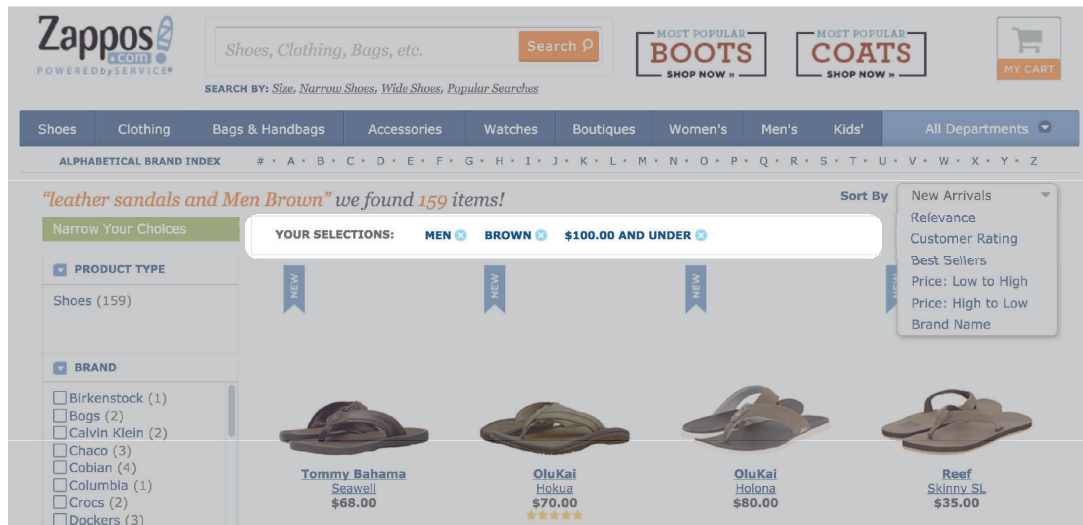
With this addition, the new search response contains updated facet counts corresponding to movies tagged as science fiction:

```
{ 'aggregations': { 'genres': { 'buckets': [
  { 'doc_count': 1597, 'key': 'Science Fiction' },
  { 'doc_count': 753, 'key': 'Action' },
  { 'doc_count': 502, 'key': 'Thriller' },
  { 'doc_count': 466, 'key': 'Adventure' },
  { 'doc_count': 337, 'key': 'Drama' },
  { 'doc_count': 336, 'key': 'Fantasy' },
  { 'doc_count': 327, 'key': 'Horror' },
  { 'doc_count': 299, 'key': 'Comedy' },
  { 'doc_count': 188, 'key': 'Animation' },
  { 'doc_count': 164, 'key': 'Family' } ],
  'doc_count_error_upper_bound': 0,
  'sum_other_doc_count': 361 } } }
```

And not only do the facet counts change, but the search results update to include only science fiction movies. Where do users go from here? Anywhere they please. They may choose to filter again within the same category by selecting another genre. They may filter based on other criteria, such as release date. Once they get an idea of what they're looking for, they may even abandon the browse interaction in favor of using the text box. No matter the case, the facet counts and the documents presented in the results will guide users to understand what's available and where to find it.

### 8.2.2 Providing breadcrumb navigation

You'll often want to give users feedback about how they've filtered the results. Without feedback, the user may be unaware that certain filters are still in place and therefore surprised to find no search results. *Breadcrumb navigation* is a commonly used technique for guarding against this problem. As shown in figure 8.6, breadcrumb navigation is typically presented at the top of the result set as a list of the currently selected



**Figure 8.6** Zappos uses breadcrumb navigation to remind users of the currently selected facets and allow them to easily deselect filters.

facets. Here we use the previous Zappos “leather sandals” search example. This time the result set is further filtered by men’s, under \$100, and brown.

As you can see, the breadcrumb navigation not only provides users with an awareness of which filters are in place, but also serves as an intuitive interface enabling users to remove filters of their choice.

### 8.2.3 *Selecting alternative results ordering*

During a browsing style search, what’s the appropriate ordering for the result set? Throughout the rest of this book, the focus has been on presenting results according to relevance. But when the user hasn’t yet specified a textual query, relevance has little meaning. Instead it’s up to the search application to define a default ordering. Depending on the domain, this may be based on popularity, location proximity, recency, or any number of things. The choice for the default ordering might even be a good opportunity for the search application to incorporate business concerns by, for instance, subtly promoting high-margin items toward the top.

You can also give the user control over the ordering of results. This is as simple as placing a drop-down selection box at the top of the result set. This commonly appears toward the right side of the search results, as again exemplified by Zappos search, shown in figure 8.7.

Zappos allows customers to sort results by the newest items, customer rating, popularity, and price. Allowing users to manipulate sort criteria is an excellent form

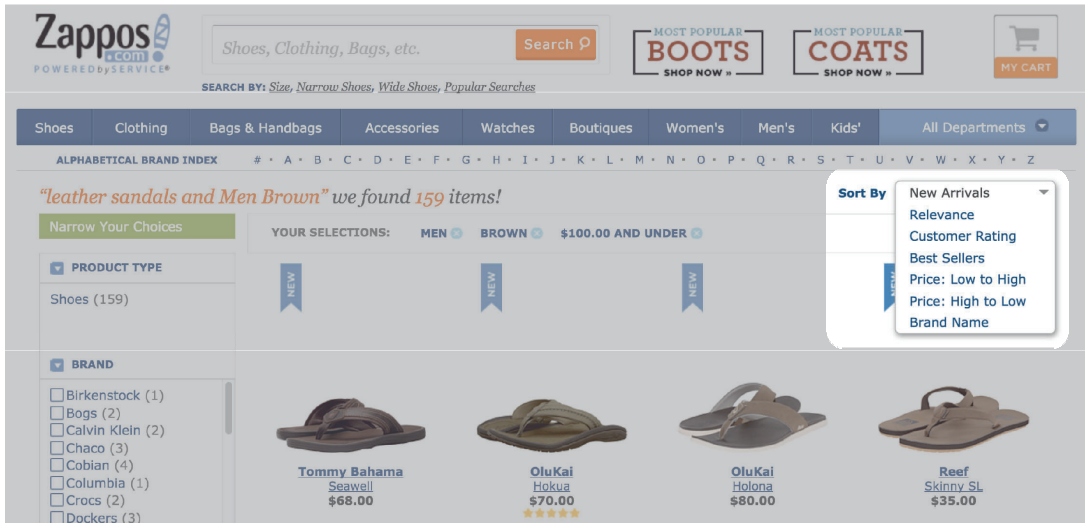


Figure 8.7 Zappos allows users to sort results according to various criteria.

of relevance feedback. This gives users a better understanding of the results and provides a means for users to organize the results to match their current search goals.

You might not always want to directly sort on these factors. A user requesting a sort on “popularity” may only want popular items prioritized; but not so strong that it overrides their search criteria. For example, a user searching for stylish “leather sandals” likely won’t be pleased if they see very popular flip flops. Refer to the boosting lessons of chapter 7 should you need to implement sorting this way.

### 8.3 Relevance feedback in the search results listing

In this chapter, you’ve followed the typical path of a user through a search application, starting with interactions at the search box and moving on to browsing and filtering with faceted search. These techniques give users many alternate paths to relevant content. Finally, let’s discuss relevance feedback and guidance in the search results themselves. Here users get an overview of the documents that the search application deems relevant. If users find a particular item interesting, they click through and investigate, hoping to find what they’ve been looking for. If users glance through the results listing and don’t find what they need, they’ll either modify their search and try again, or they’ll abandon it entirely.

The results listing is therefore an important part of relevance feedback. The items in this listing must convey the most important aspects of matching documents. This allows the user to judge whether to investigate further. In the sections that follow, you’ll consider what questions users ask when looking at results. You’ll examine how to answer those questions before users click into the search results. You’ll also see

how highlighted snippets can help users understand why a document matches a search. We cover a couple of approaches to document grouping that help decrease the cognitive burden for users. And we briefly discuss approaches for dealing with the case of no search results.

### **8.3.1 What information should be presented in listing items?**

Users will click a listing item if they think that it may meet their information needs. You must therefore consider what types of questions your users will be asking as they scan through the search results. The information presented in the result items should answer these questions as thoroughly and as concisely as possible.

The most obvious question to answer is “What is this?”; and if you’re lucky, answering this question may be as simple as placing the title of the document in the search results. But the title often isn’t a reliable semantic representation of the document it represents. For instance, in enterprise realms, the search engine may hold official forms and documents that have arcane titles, such as *Form I-130* or *401(k)*, which don’t readily convey the purpose of the form. Similarly, in e-commerce, products are often given titles intended to draw attention and convey a feeling, but not necessarily convey an accurate representation of the item. For instance, Oakley has a prominent line of women’s sunglasses call *Little Black Dress*—a funny name for sunglasses.

If you find yourself in a situation where there’s no meaningful title, you must look for other ways to convey information to the user. To answer the question “What is this?” in the realm of e-commerce, photos are often better than a title (or any other text, for that matter). A user seeking a certain product can glance through a page of images and quickly narrow the results that require more-focused attention. Even enterprise document retrieval can benefit by including imagery in the search results. For instance, including a small scanned image of the document in the search results could help users pinpoint a document that they’ve seen and used in the past.

Short descriptions also help the user answer “What is this?” But caution is warranted here. With too much text competing for users’ attention, they might miss important details, or worse yet, become overwhelmed by the surplus of information and abandon search altogether. But if the available title doesn’t readily convey an understanding of the item, a short description might be just the thing you need. If neither meaningful titles nor short descriptions are available, consider going back to the content provider and asking for a short description. If the content being searched is user provided, maybe you can place a required field for a short description. If an upstream vendor provides the content, perhaps you can work together to generate this extra bit of information that will make their products more saleable.

Besides the big question “What is this?” consider other questions that your users will ask. Are you building an e-commerce application? Then your users will be interested in knowing the price of a product. Does your search application revolve around events? Then date, time and location will be important. Place this information prominently in the search results so that users won’t have to bounce in and out of results to

judge whether a result is relevant. And if your search domain includes several subdomains, consider the questions important to users searching in those subdomains as well. If you know that users are shopping for cameras, place helpful camera-related details in the search results so that users can compare available choices in the results page itself.

### 8.3.2 Relevance feedback through snippets and highlighting

When the search domain involves text-heavy documents, snippets and highlighting provide an important form of relevance feedback. Snippets and highlighting mean exactly what you might imagine. If a user submits a keyword search, *snippets* are fragments of text from the matching documents that include the search keywords. *Highlights* are the keywords matches within the snippets, literally highlighted to the user. This can be done using bold text or altering the background color on the keyword match. Some search interfaces go as far as to use a different highlighting style per keyword.

The relevance feedback goal of snippet highlighting is to tell users why a particular document matches their query and where the match occurs. Users, especially search power users, will appreciate the ability to quickly read through matches in the context that they occurred. Without having to click into a particular search result, users get a sense of whether a document is a good match for their information needs. Furthermore, users may start to see patterns in the matching text and choose to further refine their search so that the appropriate documents will match. And occasionally, the answer to a user's question might be made immediately available in the snippets, thereby removing the need to even click through and look at the document where the match occurred.

Throughout this book, we've discussed considerations regarding text analysis. In chapter 4 we showed how to craft analysis to capture a semantic understanding of the words in a document. At various points, this chapter has seemingly reversed that notion to support human-readable tokens in facets and suggestions. With highlighting, we can safely return to the notion that tokens capture meaning. In other words, highlighted tokens don't have to be human readable. This is made possible by the fact that Elasticsearch—or more accurately, Lucene—tracks starting and ending character offsets of the original word prior to tokenization. During highlighting, Lucene can match on the token, look up the original character positions, and place the highlight appropriately in the original, stored text.

Elasticsearch provides three modes of highlighting that come with various trade-offs:

- The basic highlighter (the default mode)
- The postings highlighter
- The fast vector highlighter

The *basic highlighter* (the default mode) requires no special setup; you simply request with your search that Elasticsearch return highlighted snippets. The highlighted snippets will then be returned along with the search results. Unfortunately, the default

implementation must reanalyze documents in order to find the location of the matching terms within the document. For small documents, the time required to reanalyze the text is miniscule. But for larger documents (pages of text), this reprocessing comes with a significant performance hit.

The *postings highlighter* and the *fast vector highlighter* avoid query-time processing by storing extra information at index time. This comes at the cost of increasing the index size. But these two methods come with certain advantages. For instance, the postings highlighter is especially useful with natural language text (sentences and paragraphs as opposed to titles or single-term fields). It automatically breaks the input text such that snippets are returned to the user as complete sentences. The fast vector highlighter can highlight matching terms independently, allowing each keyword to be highlighted in its own color or style.

To demonstrate the utility of snippet highlighting, the following listing shows one of the highlighters, the fast vector highlighter. This listing showcases some of the interesting features that make highlighting a great source of relevance feedback.

#### Listing 8.11 Submitting a query with highlighting enabled

```
GET tmdb/_search
{
  "fields":["title","overview"],
  "query":{
    "match":{
      "title": "star trek"
    }
  },
  "highlight": {
    "fields": {
      "title": {
        "number_of_fragments": 0,
        "overview": {
          "fragment_size": 100,
          "number_of_fragments": 5,
          "no_match_size": 200
        }
      }
    },
    "pre_tags": ["<em class=\"hlt1\">","<em class=\"hlt2\">"],
    "post_tags": ["</em>"],
    "order": "score"
  }
}
```

④ Highlight request

① Fields to be highlighted

② Tags to mark up matches

③ Optional ordering of snippets

Prior to submitting this query, the documents were indexed with term vectors enabled for both the title and overview fields. (You do this by placing "term\_vector": "with\_positions\_offsets" in the field mappings.) Let's now take a look at some of the details in this query.

To enable highlighting, a new *highlighting* section is included ④. In the fields section of the highlight body ①, you specify the fields you want to highlight along with any other parameters you need per field. (You'll look more closely at these momentarily.) The optional *pre\_tags* and *post\_tags* parameters ② specify how to annotate the highlighted keywords in the results. By specifying several tags, you can style each

keyword in different ways. For instance, each keyword can be given its own color. This will allow users to glance at search results and quickly understand just why a given document matches their query. The final order parameter ③ is particularly useful for search involving large documents. Ordering the snippets by score causes highlighting to take the extra step of sorting the snippets so that the best matching snippets are returned first.

Now that highlighting has been enabled, let's take a look at a portion of the result for the "Star Trek" query in the following listing.

**Listing 8.12 Partial response for a search for "Star Trek" with highlighting enabled**

```
{ '_id': '193', '_index': 'tmdb',
  '_score': 5.0279865, '_type': 'movie',
  'fields': {
    'overview': ['Captain Jean-Luc Picard ...'],
    'title': ['Star Trek: Generations']},
  'highlight': {
    'overview': ['renegade scientist Soran who is destroying entire
      <em class="hlt1">star</em> systems. Only one man
      can help Picard stop Soran\'s'],
    'title': ['<em class="hlt1">Star</em> <em class="hlt2">Trek</em>:
      Generations']},
  '_id': '201', '_index': 'tmdb',
  '_score': 5.0279865, '_type': 'movie',
  'fields': {
    'overview': ['En route to the honeymoon of William Riker ...'],
    'title': ['Star Trek: Nemesis']},
  'highlight': {
    'overview': ['En route to the honeymoon of William Riker to Deanna Troi
      on her home planet of Betazed, Captain Jean-Luc'],
    'title': ['<em class="hlt1">Star</em> <em class="hlt2">Trek</em>:
      Nemesis']}]}
```

In this response, the new highlight sections of each listing item are shown in bold text. The first thing to notice is that, thanks to the use of the *fast vector highlighter* with the specified pre- and post-tags, the keyword *star* and the keyword *trek* are encapsulated in different tags.

Now, referring to listing 8.11, let's dig into some of those field-level highlighting specifications. For short fields, such as the title, you don't want just a snippet to come back; you want the entire text. By specifying `number_of_fragments` returned to be 0, you signal Elasticsearch to highlight and return the entire text of the field. Next, with the overview field, you see that the length and number of snippets can be specified via the `fragment_size` and `num_of_fragments` parameters, respectively.

The next parameter, `no_match_size`, can be particularly useful. Often a document that's a perfectly good match for a search may not have a match in the field you're using for highlighting. And, typically, this means that no snippets will be returned for this field. But rather than returning nothing, why not at least return a portion of the

leading text in the field so that the user receives some context. This is precisely where the `no_match_size` parameter comes in handy. After specifying `no_match_size` to be 200 here, any document that has no keyword matches in the `overview` field will return the first 200 characters rounded down to the nearest whole word. Notice that this is exactly the case with the second document in listing 8.12. Because neither *star* nor *trek* occur in this field, the first portion is returned instead.

One final thing to note about listing 8.12: the snippets are broken at arbitrary locations. If you opted to use the *postings highlighter* rather than the *fast vector highlighter*, the snippets would be broken at sentence boundaries, which arguably provides a better user experience by making more-understandable snippets. But the *postings highlighter* can't differentiate between the keywords so that they can be presented differently in the search results. Thus, as is so often the case, you must understand and carefully weigh the trade-offs when choosing one implementation over another.

### 8.3.3 *Grouping similar documents*

Within the results listing, another way to provide users with relevance feedback is to group documents in such a way that the information is easier for users to process. This comes in two flavors: document grouping and field collapsing.

First, concerning document grouping, documents often fall into several natural buckets within the corpus. For example, the movies in the TMDb corpus have a `status` field indicating where the movies are in the production pipeline: *rumored*, *planned*, *in production*, and *released*. For certain conceivable search applications, it may be helpful to present search results pregrouped according to the available buckets. If your users think first in terms of document groupings, then this feature can greatly reduce their cognitive burden.

Document grouping can be accomplished with the use of aggregation—in this case, a combination of *terms* aggregation and *top hits* aggregation.

#### Listing 8.13 Using a combination of terms and top hits aggregation for doc grouping

```
GET /tmdb/_search
{ "query": {
  "match": {
    "title": "star trek"
  },
  "aggs": {
    "statuses": {
      "terms": { "field": "status" },
      "aggs": {
        "hits": {
          "top_hits": {}
        }
      }
    }
  }
}
```

This query instructs Elasticsearch to find all documents matching *Star Trek* and then group the documents according to status. The corresponding result set will contain a `buckets` section that looks like the following listing.



**Listing 8.14 Documents grouped with a combination of terms and top hits aggregation**

```
{ 'buckets': [
  { 'doc_count': 82,
    'key': 'released',
    'hits': { 'hits': { 'hits': [
      { '_id': '13475',
        '_index': 'tmdb',
        '_score': 6.032624,
        '_source': {
          'name': 'Star Trek: Alternate Reality Collection',
          'popularity': 2.73003887701698, ... },
        /* more documents */ },
    ] } } },
  { 'doc_count': 4,
    'key': 'in production',
    'hits': { 'hits': { 'hits': [
      { '_id': '13475',
        '_index': 'tmdb',
        '_score': 6.032624,
        '_source': {
          'name': 'Star Trek: Axanar',
          'popularity': 3.794237016983887, ... },
        /* more documents */ },
    ] } } },
  /* more groups */ ],
  'doc_count_error_upper_bound': 0,
  'sum_other_doc_count': 0 }
```

A note on group ordering: by default, the terms aggregations are sorted according to the number of matching documents, in descending order. If the default top-level sorting isn't appropriate for your application, take some time to read through the Elasticsearch documentation and see if you can construct a more appropriate ordering for the top-level groups. (Listing 8.15 demonstrates an example of implementing a custom sorting of the top-level groups.)

The other common form of document grouping is known as *field collapsing*. Say you have several documents that are near duplicates of one another. For instance, what if the TMDB collection contains multiple, separate entries for the *Star Trek* movie, one for each language that the movie has been translated into? Rather than filling a user's search results with page after page of effectively the same result, you can use field collapsing to group near-duplicate documents and then present the user with only an exemplar document from this set.

To build such a search response, you again use a combination of terms and top hits aggregation. Getting the details right takes finesse. A bit of setup is required for field collapsing: you need a field that indicates which documents are near duplicates of one another. Though the TMDB set doesn't have a good example, let's again assume that the *Star Trek* movie has multiple, separate entries, one for each language that the movie has been translated into. Furthermore, presume that no matter the language, each entry contains a unique identifier for the original English version. This unique identifier field is a good candidate for field collapsing. By combining all movies with the

same value in this field, you'll have successfully de-duplicated the movies that have multiple translations.

The next finicky bit is modifying top-level group ordering. Recall that the top-level groups are by default sorted according to the number of documents that they contain. In this case, the top-level groups correspond to the original version of movies, and you want the results to still be sorted according to relevance. To achieve this, sort the top-level groups by the score of each group's most relevant document. The following listing is similar to listing 8.13, but it highlights the changes necessary to correctly implement field collapsing.

#### Listing 8.15 Sorting groups according to the most relevant document in the group

```
GET /tmdb/_search
{
  "query": {
    "match": {
      "title": "star trek"
    }
  },
  "aggs": {
    "original_versions": {
      "terms": {
        "field": "original_id",
        "order": { "top_score": "desc" }
      },
      "aggs": {
        "hits": {
          "top_hits": { "size": 1 },
          "top_score": {
            "max": { "script": "_score" }
          }
        }
      }
    }
  }
}
```

**1** Collapses on original\_id, ordered by top\_score

In particular, the top-level terms aggregation now refers to the `original_id` field; the terms aggregation is ordered according to `top_score`; and adjacent to the top hits aggregation, the `top_score` **1** is defined as the maximum scoring document associated with each top-level bucket.

### 8.3.4 Helping the user when there are no results

Sometimes your users make a request for which no results can be found. This can either be because they make a mistake such as a misspelling in the request or because they have so constrained the request that no documents are available. In this situation, the worst thing that you can do is to present the user with an empty results page. This leaves the user with no recourse other than to abandon search and fulfill their needs elsewhere.

To guard against this, always consider backup methods to bring users closer to satisfying their search goals. In section 2.1.3 we discussed how to use suggestions to correct user mistakes. If there are no search results, you can take this a step further by replacing the user's query with the best suggestion. Or, if there's no obvious mistake on the part of the user, perhaps you can automatically remove the user's last filter so that the search will be less constrained. Finally, if you have nothing else to go on, show

users popular documents. Whatever the case may be, if the search application modifies the search on behalf of the user, make sure to inform the user about how the search has been changed so that the user won't become disoriented with results that don't match expectations.

## 8.4 Summary

- Relevance feedback facilitates a conversation between the search application and the user.
- When designing the search experience, consider the users' typical workflow. They often start with a text search and then, as they review the results, they engage in filtering and refining their criteria. Finally, they find items of interest and click through to see the details page.
- Help users find items more quickly by updating search results as they type.
- Assist users in building successful queries by suggesting search completions.
- Correct user mistakes with "did you mean" suggestions.
- Faceted search helps users understand the distribution of items in the index and allows them to filter and drill down to the subset of the results that they want.
- The breadcrumb UI component can make users aware of how their results are being filtered. It also gives them an obvious way to remove existing filters.
- Highlighted search results and details pages draw the users' attention to *why* a particular item is deemed important.
- Result grouping and sorting can reduce user cognitive load by allowing users to focus on the most interesting results.
- Relevance feedback makes it easier for users to engage with search and find what they're looking for.



# *Designing a relevance-focused search application*

---

## ***This chapter covers***

- Gathering information before building a new search application
- Designing and implementing a complete search application
- Designing a query as a composite of subqueries
- Balancing query parameters
- Deploying, monitoring, and improving search
- Knowing when further relevance tuning is no longer advantageous

In the previous chapters, we laid out all the ingredients for good search:

- Extracting features from the text of the documents through proper tokenization
- Defining important signals and creating search fields to represent them
- Crafting queries that take into account both user needs and business requirements
- Providing feedback to users in order to guide them to more-relevant results

Now, this chapter is the rest of the recipe—the set of instructions that organizes these ingredients and lays out the methodology for building a relevance-focused search application.

Previously, we looked deep into the details of the search engine itself and described how to provide users with a relevant search experience. In this chapter (and the chapters beyond), we shift from low-level details to a higher-level view of search application development. In this chapter, we present a systematic approach for building a search application based on the simple flowchart in figure 9.1. At a high level, designing a search application requires three steps: gathering search requirements, designing the application, and maintaining a deployed application. In the subsequent sections, we provide a detailed understanding of each step so that, upon completing this chapter, you'll have a solid framework to build your own search application.

As with all of our chapters, we motivate the discussion with a fun example. And having completely wrung all educational and comedic value from the Star Trek examples, we turn to a different one. You're going to build a new and exciting start-up called *Yowl*, and search will be its central feature.

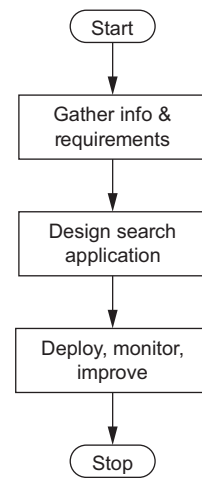
## 9.1 Yowl! The awesome new start-up!

First things first, you need a tough problem to start to solve, as in figure 9.2.

Luckily, Doug has come up with a great idea! Just imagine it: you hop off a plane in a city you've never been to before and, having subsisted on pretzels and ginger ale for the past five hours, you're dying for something more substantive—a big, juicy hamburger. But how do you find a good burger joint? There's no product on the market that can help you find good restaurants nearby.

Yowl to the rescue! With Yowl's smartphone application, you can search for and quickly find nearby restaurants fit for your cravings. Or maybe you're not in a new city, but you're interested in exploring the foodscape in your hometown. Yowl will help you discover restaurants that you never knew existed. Yowl will find restaurants perfectly matched to your tastes. Yowl will even help you explore new tastes! The possibilities are endless.

Though you might have to do some legwork up front, you hope that as Yowl catches on, all the restaurant data will come from restaurant patrons and owners. The restaurant patrons will write reviews for restaurants they've dined at. And restaurant



**Figure 9.1** Relevant search is built in three phases: information gathering, design, and deployment. This chapter covers all three phases in detail.



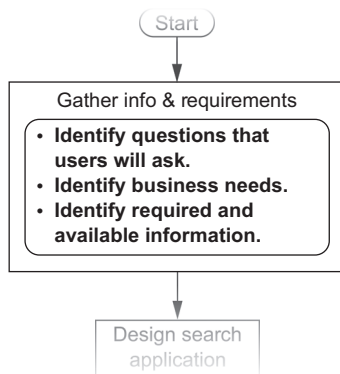
**Figure 9.2** At the start of any good search application is a problem that needs to be solved. Identifying and thinking through that problem is an important step in building the search application.

owners, encouraged by the new traffic created by Yowl, will update their restaurant data so that their restaurant can easily be found.

And the best thing is that Yowl has absolutely no competition in this market. Amazing, right? How has no one thought of this before? *Doug said he looked for it on Google, and nothing came up!* Yelp, folks, as soon as we get this new search application built, we're all going to be rich. Oh, and a final note on the name; you might think that Yowl is the sound that a cat makes when you step on its tail. Maybe so. But we think it's catchy. Think of Yowl as the sound that your stomach makes when you're really, really, *really* hungry. So when you're most hungry, you'll immediately think of our app!

## 9.2 Gathering information and requirements

Before building a search application, take time to consider the needs and expectations of all stakeholders. This is the next step in our process, as shown in figure 9.3. Most obviously, search should be easy to use and helpful for users. But Yowl is a search-driven business, so it must also meet business needs. In this section, we look at both user and business needs. We identify the information required to meet these needs and we describe potential sources for this information.



**Figure 9.3** Before building search, consider the needs of all parties involved, identify the information required to fulfill those needs, and determine where this information can be found.

### 9.2.1 Understand users and their information needs

A useful exercise is to create personas for the various types of users who are expected to interact with the application. A *persona* is a description of a fictional character that epitomizes a particular type of user behavior. Here's our first cut at Yowl user personas:

- *Jet-setter Jenny*—Jenny regularly travels and looks to Yowl to find places to eat when she's in a new city. Sometimes she's looking for a quick and convenient bite; other times she wants to find a fancier restaurant for meals with business associates.
- *Connoisseur Courtney*—Courtney is interested in the finer things in life, and when it comes to food, she has exacting tastes. One night she'll seek the finest French cuisine. Another night she'll look for the restaurant that serves the best Chirashizushi (a Japanese dish).

- *Explorer Evan*—Evan has a broad palate and is always interested in trying something new. This includes new types of food, new restaurants, and new parts of town.
- *Discount Danny*—Danny is a price-conscious user. When choosing a restaurant, he wants to find the best value possible. He’s looking for low prices and high ratings. If there’s a discount or meal deal available, he’ll use it. He prefers nearby restaurants—you know, to save on gas.

Building out user personas might seem artificial at first, but they make it possible to identify with users on a more concrete level. Personas also form a shared vocabulary that allow you to more easily discuss the entire spectrum of user behaviors. When creating personas for your project, make sure to cover all of the predominant behaviors you expect to see, and try to avoid overlapping behaviors.

Let’s use the preceding personas to get to know your users. You can do this by enumerating the questions that these imaginary users might ask of your service. Three of your consumers are interested in restaurant location, though in different ways. They might ask the following questions:

- Jet-setter Jenny: “Where are all the nearby delis?”
- Discount Danny: “Where are the nearest restaurants that have discounts?”
- Explorer Evan is still interested in location but rather than “nearby” or “nearest,” he’s interested in a slightly different question: “What areas of town have new and highly reviewed restaurants?”

The next most prevalent concern appears to be the users’ tastes for a particular type of cuisine. Here, the questions are as follows:

- Jet-setter Jenny may require only a generic representation of cuisine type. She may ask, “I’m heading out with business colleagues for dinner; what Italian restaurants are nearby?”

Or perhaps, being in an unfamiliar city, Jet-setter Jenny might long for familiar chain restaurants where she knows what she can expect: “My tummy has the grumbles for cheap Mexican. Is there a Taco Belly nearby?”

- Connoisseur Courtney needs Yowl to understand cuisine with the acumen of a professional chef. She won’t ask for Chinese food; instead she’ll ask more specifically: “Where’s the finest Szechuan dining in the city?”

This indicates that you might need to keep track of cuisine types as a hierarchy—for instance, with Hunan and Szechuan cuisines being a subtype of Chinese cuisine. But Courtney, being a true connoisseur, may even go so far as to ask for specific dishes: “Where can I find the Szechuan dish called *bon bon chicken*?”

- Explorer Evan, having a broad palate, won’t constrain search results by a particular cuisine, but he’s interested in knowing what types of food are available. He’ll ask exploratory questions, such as “What kinds of restaurants are located in this part of the city?”

Finally, all of the personas appear to have additional criteria that they might use to filter their results:

- Explorer Evan is interested in “What’s new?”
- Connoisseur Courtney is interested in “What’s highly reviewed?”
- Discount Danny is interested in “What’s cheap and yummy (low price, high rating)?”

Danny might also appreciate knowing about any available discounts. Hmm ... you might even use the discount thing as a business angle.

Identifying your typical users and their information needs gives you a good start at identifying the information necessary to satisfy these requirements. Read back through your users’ questions and consider: What types of information will you need in order to answer all of your users’ questions? Where will you get this information? Before getting to the bottom of this, let’s look at the other side of the equation: the needs of the business itself.

### 9.2.2 *Understand business needs*

If Yowl is going to help users for very long, you need a sustainable business model; someone has to pay you for the service you provide. No sweat—you have the perfect idea: restaurants will pay a subscription fee in order to become one of Yowl’s *promoted restaurants*. Whenever a user searches for a type of cuisine or a dish that’s a match for a promoted restaurant, that restaurant will get an extra boost in its relevance score, which will ensure that the restaurant is listed higher in the search results. On top of this, you’ll further boost promoted restaurants that have available discounts.

There are also nonmonetary aspects to the business. Yowl is a content-driven company, and by and large you’ll depend on restaurants to provide information. Therefore, whenever a restaurant provides Yowl with useful information, Yowl will temporarily provide it with a boost as a *thank you* for the help.

Notice that at this point you have several competing concerns to balance. If you boost too far in favor of the restaurants, you’ll lose users because the search results will be less relevant. If you boost too far in favor of the users, you offer no benefit to promoted restaurants and will soon lose them as customers. Relevance engineering is tricky!

### 9.2.3 *Identify required and available information*

Having considered both user needs and business needs, let’s boil all this knowledge down. You’re looking to answer three main questions right now:

- What information is required?
- What will the information be used for?
- How do you get it?

After answering, you’ll have a clearer understanding of the application. And you can identify feasibility risks that may arise as a result of hard-to-get information or information that doesn’t quite answer your users’ questions.



Let's first consider the content being searched—restaurants. At a minimum, a restaurant is represented as a name and an address. Usually this information is publicly available. Restaurant names and addresses that are more accurate and machine readable can be purchased from a business database provider.

Restaurant names will be of obvious use, both in search and in the presentation to the user on the details page. You need to use the address for location information. You'll use a geolocation service to convert restaurant addresses into their corresponding latitude-longitude coordinates. At search time, the latitude-longitude location will be useful for filtering, for finding all restaurants within a given bounding box. And the latitude-longitude coordinates will also be useful in boosting restaurants that are located close to the user making the search request.

Even more than location, Yowl users want to know about the food that restaurants serve. Often users will search for restaurants by cuisine (Mexican, Italian, Chinese, and so forth). Some users will come to Yowl looking for a particular dish. And when users review search results, they'll appreciate having cuisine and menu information nearby so that they can be sure their search is on the right track. Gathering this information is tricky. You do expect users to provide restaurant reviews; maybe you can also ask them to provide missing details about the restaurants. And once you bring significant users to more restaurants, the restaurant owners will have the incentive to provide this information. But for now, you'll probably have to roll up your sleeves and research and enter this information yourself.

After location and cuisine, users will filter and sort results according to secondary criteria: price, review rating, and whether a discount is available. The overall price and rating will come from the users. And the discounts will be supplied by the restaurants. Restaurants can also provide descriptions of themselves. This will be nice to present to users in the details pages, but it might also carry rich, searchable text.

And finally, don't forget that Yowl's success depends on subscribed restaurants. This is a simple Boolean value that controls whether restaurants will get an extra nudge toward the top of the search results. Similarly, for a limited time, you're boosting *engaged* restaurants—those that are helping you by providing information.

In review, the information you need, the sources you'll draw this information from, and the uses for this information are presented in table 9.1.

**Table 9.1** Yowl information requirements, sources, and uses

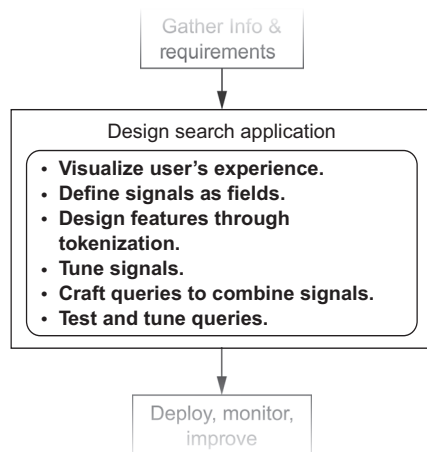
Information	Example	Source	Uses
Restaurant name	Taco Belly	Public	Search, display (listing, details)
Address	1234 Cordon Ave. Blacksburg TN 23913	Public	Source of location, display (details)
Location (lat/long)	35° 24' 30" N 88° 31' 20" W	Geolocated from address	Filtering, display, source for distance

**Table 9.1** Yowl information requirements, sources, and uses (*continued*)

Information	Example	Source	Uses
Distance	12 mi	Calculated from user and restaurant locations	Boosting, sorting, display (listing)
Cuisine	Mexican; fast food	Restaurant, user	Search
Menu items	Burrito Ultimate \$4.95	Restaurant	Search, display (details), source of price information
Price	\$, \$\$, \$\$\$, \$\$\$\$	Menu items, user	Filter, display (listing)
Review rating	★,★★,★★★,★★★★	User	Filter, display (listing)
Discounts	(Varies)	Restaurant	Filter, display (listing, details)
Promoted restaurants	Boolean	Restaurant	Boost
Completed form	Boolean	Restaurant	Boost
Description	“Tasty, Fast, Cheap, Addictive”	Restaurant	Display (listing, details), search

### 9.3 *Designing the search application*

With all of the ingredients assembled, it's time to pull together your recipe for restaurant search. As indicated in figure 9.4, you'll first design the user experience based on expected user needs. Next you'll rely on knowledge of the available information to identify the fields that you need to index; these will serve as your base signals. You'll determine analysis requirements to ensure that the text is properly tokenized and the appropriate features are extracted. Once you're sure that the base signals function well in isolation, you'll build a query that balances the signals together.



**Figure 9.4** A systematic approach for designing a search application involves creating the appropriate fields and queries so that the users' questions are met.

### 9.3.1 Visualize the user's experience

Leaning on our understanding of user and business needs, let's think through the user experience. From a search relevance perspective, you have two goals. The primary goal is to help users fulfill their information needs as quickly as possible (this is the goal of the book, after all). The secondary goal is to provide users with relevance feedback (as in chapter 8) to help them understand why they're seeing the current search results and how to adjust their searches accordingly.

When users come to Yowl, they immediately begin browsing for restaurants through a user interface similar to the one shown in figure 9.5.

Some users start with a text search and enter a restaurant name, a cuisine type, or specific dish. Users with less-specific needs filter nearby restaurants according to price or rating. Many users search and filter. As users modify their search text or their filter selections, the map updates with markers for nearby restaurants that meet the search criteria. The map itself can be used as a location-based filter of restaurants, and users can pan and zoom to various parts of the map in order to see restaurants in a different location.

After you satisfy users' search criteria, they move on to one of the two search result displays. The first display, the map view, is part of the browse interface. As shown in figure 9.6, the browse components can be dismissed to reveal a full-size map.

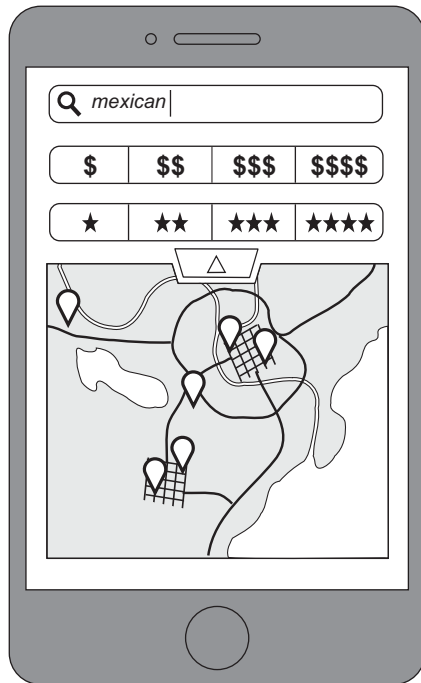


Figure 9.5 Yowl's browse interface

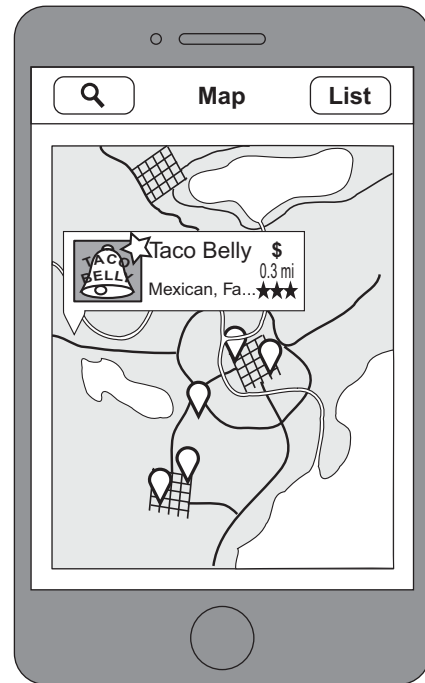


Figure 9.6 Yowl's map view, one of the two search result displays

The visual presentation of a map provides relevance feedback, assuring users that they're searching in the correct location. When a user clicks a restaurant marker, a restaurant card appears, displaying relevant information:

- Restaurant name
- Distance from the user to the restaurant
- Rating (by number of stars)
- Price (by number of \$ characters)
- Cuisine type
- An image for the restaurant

The goal of the restaurant cards is to present the most critical information as compactly as possible on the screen. This allows the user to efficiently scan the display and click through to the details only when the restaurant is a good match.

As shown in figure 9.7, you also present search results as a tabular listing of matching restaurants. The list view presents each search result as a restaurant card that looks just like the restaurant cards used in the map view. The list view isn't as good as the map view for displaying location information, but it allows users to focus on the other metadata—cuisine type, rating, price, and distance. The results can be sorted by relevance or distance; and in order to retrieve more results, the user can scroll further down in the list.

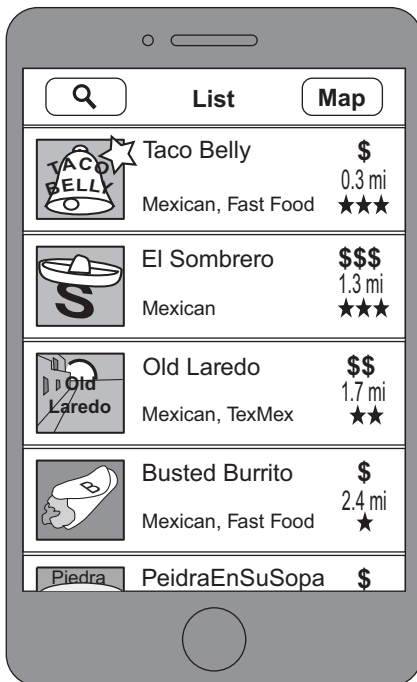


Figure 9.7 The list view, a tabular view of the search results

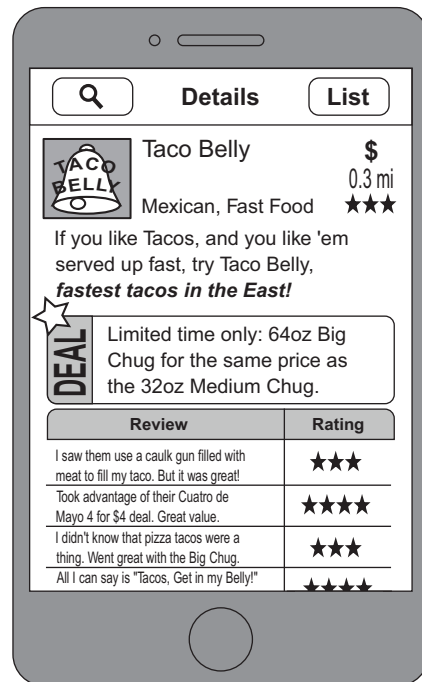


Figure 9.8 Yowl's restaurant details view

Finally, if a user clicks a restaurant listing, you present the restaurant details view, shown in figure 9.8. This page provides rich information for that restaurant, including:

- Everything from the restaurant card (restaurant name, distance, rating, price, and so forth)
- Restaurant description
- User reviews
- Details about available deals
- A link to the menu
- Hours that the restaurant is open
- Address and phone number

### 9.3.2 Define fields and model signals

You now know what information is available and how it will be used within Yowl. It's finally time to start building this search application! This starts with defining your fields. Table 9.2 uses the information from table 9.1 to describe how each piece of information will be analyzed and indexed into its own field. Nothing should be too surprising here; everything is a straightforward application of the ideas discussed in chapter 4.

Although we're brushing over some details, don't be fooled into thinking that modeling the signals is an easy task. In practice, defining fields can be iterative. You'll often run into unexpected behavior and have to go back and tune the analysis to handle the corner cases as they pop up.

**Table 9.2** Given the required information, you form fields and determine how to appropriately analyze the content.

Field name	Information (from table 9.1)	Analysis
location	Location (lat/long)	geo_point (geolocated from address)
name	Restaurant name	Standard tokenized, possessive and plural stemmed, lowercased, diacritics removed
cuisine_hifi	Cuisine (from restaurants/ curators)	Standard tokenized, synonyms (used to normalize the vocabulary)
cuisine_lofi	Cuisine (from user)	
menu	Menu items	Standard tokenized, lowercased, bigram shingled, multivalued (ETL will involve OCR and curation)
description	Restaurant description	English tokenization and stemming, lowercased
price	Price	One of {,\$,\$\$,,\$\$\$,\$\$\$\$\$} as determined from menu items or user reviews

**Table 9.2** Given the required information, you form fields and determine how to appropriately analyze the content. (*continued*)

Field name	Information (from table 9.1)	Analysis
rating	Review rating	One of {★,★★,★★★,★★★★} as determined by average user reviews
has_discount	Discounts	Boolean
promoted	Promoted restaurants	Boolean
engaged	Completed form	Boolean

At this point, it's good to start thinking about how base signals (the fields in table 9.2) naturally group together. Later, when it's time to build your queries, you can think about these groups as logical chunks rather than having to consider each base signal as a completely independent entity. A few sections from now, you're going to build out Yowl's main query. You'll see that thinking of fields in groups greatly simplifies the process of balancing signals and tuning the overall query.

As you can see in table 9.2, Yowl's fields fall into four fairly natural groups: location, content, user preferences, and business concerns. These are defined as follows:

- *Location*—The location field is just that: a `geo_point`-encoded set of latitude-longitude pairs.
- *Content*—This includes text fields: `name`, `cuisine_hifi`, `cuisine_lofi`, `menu`, and `description`. Note that you split your cuisine information into two fields: a high-fidelity field, `cuisine_hifi`, for cuisine information supplied by Yowl curators and restaurant owners; and a low-fidelity field, `cuisine_lofi`, for information that comes from users reviewing the restaurants.
- *User preference*—User preference is represented in the `price` and `rating` fields. Both fields have similar UI/UX that allows users to filter through the restaurants. Both fields are represented similarly in the index. They differ only in meaning.
- *Business concerns*—This includes `has_discount`, `promoted`, and `engaged` fields. These Boolean fields define groups of restaurants that Yowl wants to boost as part of your business strategy.

### 9.3.3 *Combine and balance signals*

By this point, you're confident that fields correctly capture the important features from the data that they encode. You're now ready for the sometimes-arduous task of creating queries that combine and balance these signals. In this section, we lay out a bottom-up approach whereby the base signals (fields) combine according to their logical grouping. Once these higher-level signals are in place, you combine them to create the final query used for Yowl search.

**BUILDING QUERIES FOR RELATED SIGNALS**

You’ve already seen how the fields naturally group into location, content, user preferences, and business concerns. Now let’s consider how each of these higher-level signals might be stated in terms of Elasticsearch queries.

First, let’s look at the location signal. You need two queries to handle location. One is a bounding-box filter that will be used on the map screen to geographically limit the area that needs to be searched, as shown in the following listing.

**Listing 9.1 Bounding-box geographic filter**

```
{  "filter": {
    "geo_bounding_box": {
      "location": {
        "top_left": <northwest corner of user display>,
        "bottom_right": <southeast corner of user display>}}}}
```

The other location query scores the restaurants according to their geographic distance from the user’s current location, as shown in the next listing.

**Listing 9.2 Geo-query scoring restaurants near to the user more highly**

```
{  "query": {
    "function_score": {
      "functions": [{
        "gauss": {
          "location": {
            "origin": {
              "lat": <user-lat>,
              "lon": <user-lon>},
            "offset": "0km",
            "scale": "10km"}}}}}]}}
```

This query ensures that nearby restaurants are boosted above the distant ones in Yowl’s list view. Determining a score based on distance can be computationally expensive, so every time you use this query, you’ll also apply the geo-filter of listing 9.1. Otherwise, you’ll inadvertently be calculating distance scores for restaurants so far away that the users wouldn’t be interested in them anyway.

As you build up the subqueries corresponding to your signals, it’s important to test each query in isolation, as you did in chapter 7. This simplifies the work required when combining these pieces into an overarching query, because you’ll be able to focus on the higher-level signal rather than worrying that the base signals aren’t appropriately balanced. Test queries by indexing a realistic set of documents and then running through each query to ensure that it measures the expected information. In this case, you test the location filter and query by making sure that only restaurants within the bounding box are returned, and that they’re returned in order of increasing distance.

Next, you'll build a query to handle the content signal. This is your main text-based query.

### Listing 9.3 Query associated with content

```
{ "query": {
  "multi_match": {
    "query": <user query>,
    "fields": [
      "name^10",
      "cuisine_hifi^10",
      "cuisine_lofi^4",
      "menu^2",
      "description^1"],
    "tie_breaker": 0.3}}}
```

← **User's  
search**

As discussed in detail in the previous chapters, a lot of strategies are available for searching text fields: the query can be field-centric or term-centric, the scoring can be a simple summation of subscores or something more complicated such as a disjunction-maximum score with a tie parameter. Or you can forego all the complexity and just dump the text into a single content field and have a single score to worry about. In the preceding case, you use a field-centric approach by applying a `multi_match` query across the `name`, `cuisine_hifi`, `cuisine_lofi`, `menu`, and `description` fields. By default `multi_match` uses a `best_fields` approach, which is good here because the users will likely search for a restaurant name *or* a cuisine type *or* a menu item—but not all three at once. But to soften the `best_fields` behavior a bit, you'll also introduce a `tie_breaker` with a value of 0.3.

The fields in listing 9.3 are associated with boosts that indicate the level of importance placed on each field. It's good to think through these boosts and get a representative value down for each field. These boosts are subject to change as you fine-tune your relevance. Because the typical use-case involves users searching for a restaurant name or a cuisine type, these fields receive the highest boost. Recall that `cuisine_lofi` content is contributed directly by the user, so you trust it less than the curated `cuisine_hifi` field. Because of this, you give it a lower weight. The `menu` field is important for people searching for a particular dish, but you weigh it lower because you don't want cuisine types that happen to be in the menu trumping matches in the cuisine fields. Finally, the `description` field is given the lowest boost because it's intended to merely bolster the search content.

Test the content query by ensuring that it matches the expected documents. And test the scoring by making sure that the documents are returned in the appropriate order. This type of testing tends to be subjective. In a moment, you'll learn about test-driven relevance—a technique for making this type of testing as objective as possible.

Next, the user preference signal is the easiest portion of your search. When your users select a price point (number of dollar signs) or a rating (number of stars),



they're asking to filter out all other results. Therefore, no scoring is involved; you only need a filter, as shown in the following listing.

**Listing 9.4 User-preference filter to limit to matches according to price or rating**

```
{  "query": {
    "bool": {
      "filter": [
        { "match": {
            price: "D" } },
        { "match": {
            rating: "S" } } ]
    }
  }
```

Here you're using `D` to represent a single dollar sign and `S` to represent a star. Testing this filter is easy; just make sure that the search results contain only restaurants that match both the specified price and rating.

Finally, you consider the business needs. The goal is to boost restaurants that are promoted members of your service, restaurants that advertise a discount through your service, or restaurants that are engaged (they're providing helpful information).

**Listing 9.5 Business concerns query to promote paying and engaged customers**

```
{  "query": {
    "function_score": {
      "functions": [ {
        "filter": {
          "bool": {
            "should": [
              { "term": { has_discount: True } },
              { "term": { promoted: True } },
              { "term": { engaged: True } } ]
          }
        },
        "script_score" : {
          "script": """
            0.5*doc["promoted"].value +
            0.3*doc["has_discount"].value +
            0.2*doc["engaged"].value
          """
        }
      } ]
    }
  }
```

For the sake of simplicity, you encode the scoring logic in a script score to give promoted restaurants the highest boost, followed by restaurants that have available discounts, followed by the nonpaying but highly engaged restaurants. Script scoring can be computationally expensive, so in order to limit the number of restaurants that have to be processed, you filter the data to include only restaurants that have at least one of promoted, `has_discount`, or engaged. Note that this *won't* filter out any results; it just filters the documents that will be processed by the `script_score`. Testing this query is simple: create a set of documents containing every combination of promoted, `has_discount`, or engaged and make sure that each gets the expected score.

**COMBINING SUBQUERIES**

At this point, you've constructed subqueries corresponding to each of the dominant signals in your search application. As you built each subquery, you also tested it to ensure that the correct documents were matched and the results were scored so that they reflected your relevance goals. In this section, you'll go up one level of abstraction by building the overarching query that backs Yowl's search. Let's first look at the composite query and then discuss the choices you'll make in its construction.

**Listing 9.6** Composite query that incorporates all of the major signals

```
{
  "filter": {
    "bool": {
      "filter": [
        {
          "geo_bounding_box": {
            location: { <user's bounding box> }},
        {
          "match": {
            "price": <user's price preference>}},
        {
          "match": {
            "rating": <user's rating preference>}}]
      }
    }
  },
  "query": {
    "function_score": {
      "score_mode": "sum",
      "query": {
        "multi_match": {
          "query": <user's search terms>,
          "fields": [
            "name^10",
            "cuisine_hifi^10",
            "cuisine_lofi^4",
            "menu^2",
            "description^1" ],
          "tie_breaker": 0.3}},
        "functions": [
          {
            "weight": 1.2,
            "filter": {
              "bool": {
                "should": [
                  { "term": { has_discount: True }},
                  { "term": { promoted: True }},
                  { "term": { engaged: True }}}]
              }
            },
            "script_score": {
              "script": """
                0.3*doc["has_discount"].value +
                0.5*doc["promoted"].value +
                0.2*doc["engaged"].value
              """
            }
          },
          {
            "weight": 2.1,
            "gauss": {
              "location": { <user's bounding box> }},
            "weight": 1.0
          }
        ]
      }
    }
  }
}
```

**Location filter**

**User preference**

**1 Sums the relevance scores**

**Content**

**Business concerns**

**Location**

```

    "location": {
      "origin": { <user's location> },
      "offset": "0km",
      "scale": "4km", }},
  {
    "weight": 1.0}}]]}}}

```

Content weight  
(mysterious!) ←

This query has two top-level sections, filters and queries. The filter limits the result set to include restaurants only near the user's current location and (if specified) match the user's price and rating requirements. The query section is more complex. The first section is the content query. In the context of the entire query, the content query serves to further filter the restaurants to only those that contain the user's search terms. But more important, this query provides a base relevance score for each document. The other signals (location and business concerns) are intended to serve only as boosts—subtle nudges—on top of the base content score. The remainder of the query section contains functions (within a `function_score` query) that provide these boosts. Each function contains a weight parameter that allows you to balance the signals. You have three weights in total: a weight for the location, the business concerns, and the content. The user preferences don't have a weight because they merely serve to filter the content.

At this point, you can clearly see the importance of first creating subqueries for each high-level signal. The preceding query has only three knobs to turn in order to control relevance behavior—the three weights. If you hadn't started by building subqueries, you'd have to consider the interactions of all 11 fields simultaneously, and debugging a relevance problem would be nearly impossible.

### UNDERSTANDING THE BEHAVIOR OF SIGNAL WEIGHTS

You might notice something strange about listing 9.6: the content subquery is separated from the associated content weight. Let's take a closer look at the structure of the query, in order to better understand the behavior of each weight. And at the end of this section, we take a step back and generalize the lessons learned so that you can apply the same reasoning to your own search projects.

In Elasticsearch, a `function_score` query has a query section and a function section. There's nothing special about the query; it matches and scores documents. The interesting part is how the score from the query interacts with the numerical values produced by the functions. By default, the values generated from each function are multiplied together, and the resulting value is then multiplied with the score of the main query. In equation form, this looks something like this:

$$\text{TotalScore} = (\text{BusinessScore} \times \text{LocationScore}) \times \text{ContentScore}$$

But by including `'score_mode': 'sum'` (❶ in listing 9.6), the behavior is modified so that the function values are summed together before finally being multiplied with the query score, as shown in the following listing.

**Listing 9.7** Equation representing the calculation of the overall score

```
TotalScore = (wB × BusinessScore + wL × LocationScore + wC) × ContentScore
```

Here you include weights ( $w_B$ ,  $w_L$ , and  $w_C$ ) corresponding to the weights in listing 9.6. These correspond to the business, location, and content weights, respectively. At this level of abstraction, you no longer directly control the values of `BusinessScore`, `LocationScore`, and `ContentScore`. Those values are determined by the subqueries you put together earlier. But by adjusting the weights  $w_B$ ,  $w_L$ , and  $w_C$ , you can carefully balance `BusinessScore`, `LocationScore`, and `ContentScore`, and thereby control the search relevance.

Notice that the signals aren't symmetric in this equation. The `ContentScore` is multiplied across the `BusinessScore` and `LocationScore`. In a sense, this gives the content score an advantage over the other signals; if content score is low, it will be difficult for the total score to be high. Say, on the other hand, that you make the equation more symmetric by adding in the content score:

```
TotalScore = wB × BusinessScore + wL × LocationScore + wC × ContentScore
```

Then it would be possible to have a high-scoring document that has a content score of *zero*! This would obviously lead to a horrible user experience. A user could search for “sushi,” and the Yowl app would recommend the burger joint next door! To avoid this dynamic, multiplying the content score by the other factors works best.

Now what about that mysterious content weight from listing 9.6? Why is it necessary? To finally answer this, let's consider how search would behave if the content weight wasn't present. In terms of an equation, this would be just like the equation from listing 9.7, but with  $w_C$  removed:

```
TotalScore = (wB × BusinessScore + wL × LocationScore) × ContentScore
```

With this equation in mind, let's think about a degenerate case. Consider a restaurant that's a perfect match for the user's query, but is of no business value (not promoted, engaged, or offering any discounts). Additionally, the restaurant is located at the edge of the geo-bounding box and its location score is therefore quite low. If you use the equation without the content weight, then despite being a perfect match for content, the overall score of this restaurant will be low. Effectively, the business score and the location score are “teaming up” against the content score. *But*—if you add the content weight back in as shown in listing 9.7, you control the extent to which the business score and location score are allowed to team up against the content score. Therefore, when  $w_B$ ,  $w_L$ , and  $w_C$  are properly tuned, the search results will retain an appropriate focus on the quality of the content and won't be overrun by the less important location or business signals.

Let's step back and generalize this discussion. As you develop your own search applications, your queries will likely be structured similarly to the Yowl query of listing 9.6.

They'll be composed of a `function_score` query in which the query clause matches and scores documents based on content and the function takes care of additional boosting logic. But even if you have more exotic queries, it's important to understand their structure in the same way that you now understand the Yowl query. Be able to write out an equation that explains how the high-level signals are combined to create the overall score. Consider extreme cases to guard users from poor behavior caused by unimportant signals overpowering the more important signals.

### TUNING AND TESTING THE OVERALL SEARCH

With a well-structured query in hand, you have one final task: tuning the weights so that content, user, and business signals are properly balanced. But as simple as this task may sound, it's often the most frustrating and time-consuming job in all of relevance work. In this section, we present an organized approach for tuning query parameters.

You'll start by focusing on content-based relevance and ignoring other concerns such as user preferences or business requirements. After you've optimized content-based search, you layer in user preference. Finally, after the user experience has reached its peak performance, you introduce the business requirements into the query. As a result of this progression, you arrive at a search experience that prioritizes the user without neglecting the practical needs of the business. Again, you'll use the Yowl query, but the principles presented here can be applied to your own search projects.

Before you start tuning query parameters, you need to index a realistic set of documents to test against—and the larger the data set, the better. Ideally, you can take a snapshot of the production index and use that when tuning your query parameters. You'll also need a representative set of search requests. For an existing application, you can scan search logs to get an idea of typical requests.

Here you're building out Yowl search for the first time, so you'll have to be creative and infer the types of questions that users will ask. The important part is to have a small set of sample requests that exercise every typical use case. Because you've already created user personas, it might be a good idea to refer to their questions and make sure that you're at least covering all the use cases that you anticipate seeing.

The first thing to tune is the content subsearch. Remember this guy?

```
{ "query": {
  "multi_match": {
    "query": <user query>,
    "fields": [
      "name^10",
      "cuisine_hifi^10",
      "cuisine_lofi^4",
      "menu^2",
      "description^1"],
    "tie_breaker": 0.3}}}
```

Here you have six parameters to tune, one for each content field and one for the `tie_breaker` parameter. You tune these parameters by running through the exemplary

search requests, looking through the results from each request, and modifying parameters to correct any problems you run into. As we've mentioned before, this is typically a subjective process. In the next section, we introduce *test-driven relevance*, which can make this process more organized and objective than the typical approach of "just trying a lot of queries and seeing how they look."

While tuning text queries like this, you'll inevitably run across results that are hard to explain. When this happens, remember to lean heavily on Elasticsearch's built-in query explain feature (activated by placing `'explain': true` in the query). As discussed in chapter 3, the explain details are a little difficult to read at first but they're useful for understanding why documents match a particular query and how they're scored.

When you're happy with the content subquery, zoom back out to the full Yowl query presented in listing 9.6. (Naturally, you'll update the parameters of the content subquery to match the values you just arrived at.) From here on out, all you have to do is find appropriate values for the three weights: the content weight  $w_C$ , the location weight  $w_L$ , and the business weight  $w_B$ . Let's begin by setting  $w_C$  to 1.0 and  $w_L$  and  $w_B$  to 0.0. This serves as a baseline in which documents are scored solely on their content, and neither location nor business concerns affect the overall score. (Referring to the equation in listing 9.7, can you see why this is true?)

Because you're dealing with the overall query, you again have access to location, price, and rating filters. As a next step, you'll make this example search more realistic by including a location filter representative of your typical search requests. Immediately you'll see a drop in content-based relevance. Can you think of why this would be? The drop in relevance is caused by a decreased number of available documents. When you were looking at the entire index, there were plenty of restaurants to match. But as you narrow the focus to a particular locale, there might not be as many relevant results for a given query. Don't fret. You at least know that you're getting more realistic with your search. If the results are terrible at this point, no amount of query tuning will help; you simply need more restaurants in the index—and that's a business problem!

Next let's layer in the location. To do this, you gradually increase the location weight  $w_L$  while keeping an eye on the quality of the search results. At 0.0, the location query has no impact on search results. As you increase the value  $w_L$ , you observe the documents rearrange so that nearby restaurants are preferred above distant ones. And if you increase  $w_L$  too much, you'll begin to see less-relevant results pulling toward the top.

After you find an appropriate value for  $w_L$ , the content and location signals will be in balance. At this point, you've built as close to optimal search app for your users. Unfortunately, you're not yet meeting any of your own business needs! Therefore, it's time to layer in the business signal. The direct approach would be to slowly increase the business weight  $w_B$  from 0.0 until you start to see relevance drop. But you've just increased the location weight so high that any further influence from the business signal will likely cause the relevance to drop off quickly.

You need a different approach, one that treats the location signal and the business signal symmetrically. To achieve this effect, you start by temporarily turning off the location signal by setting  $w_L$  to 0.0. And then, using the same technique as before, you increase  $w_B$  until just before relevance starts to be noticeably affected. After finding an appropriate value for  $w_B$ , you restore the location weight  $w_L$  to the previously determined value. Because you've handled the location signal and the business signal symmetrically, they should now be reasonably well balanced with one another. But with influence now coming from both business *and* location, it's likely that the content signal is being overpowered. Fortunately, this isn't a problem, though you have one more knob to turn. By gradually increasing the content weight  $w_C$  until relevance is restored, you can reach an optimal balance between content, location, and business concerns.

Unfortunately, in real search applications, parameter tuning can get a lot messier than this idealized example. But it's important to at least aim for these ideals in order to avoid unnecessary complexity (for instance, attempting to tune all fields independently). Also, even when tuning gets messy, the preceding thought process still applies: you have to understand how signals interact so that you change the parameters appropriately.

#### **YEAH ... BUT HOW DO YOU TUNE RELEVANCE PARAMETERS?**

You've probably noticed that in all of these steps you're doing a lot of fiddling around with parameter values. And at each step, we talk about making sure the search results "look good" before moving on to the next step. But we haven't talked about *how* to make sure your parameters are moving in the right direction.

For starters, our optimality criterion, "looks good," is vaguely defined. In order for the search results to be optimal, you have to understand what your users think is relevant. But this is difficult. One user searching for "taco" is looking for a restaurant that has *taco* in the title (perhaps the user forgot the name), whereas another user searching for "taco" seeks restaurants that serve tacos. Therefore, our notion of optimal relevance must be aware of the differing information needs of our users. Furthermore, "taco" is just one query; you have to make sure that relevance is maintained across any queries that users might make. If every user searched for "taco," you could simply tune the parameters until the *taco* search looked perfect. Instead, there are 100,000 other searches that need to look good too; and concentrating too heavily on any one of them will certainly be detrimental to the others.

So what should you do to nudge those parameters in the right direction? First you need a good set of typical queries to work with. This set should include popular queries, it should include queries from every imaginable use case, and it should even include some random queries—ideally, things you've plucked from a query log file.

Now, given a typical set of queries, you need to understand what "looks good" means for each of these queries. This can be as vague as "I know it when I see it," but ideally it should be more concretely defined. At an extreme, the field of information retrieval uses so-called *relevance judgments* to define "looks good." Here, given a fixed

set of queries and a fixed set of documents, each query-document pair is graded on a 1-to-5 scale according to how well the document matches the corresponding query. Collectively, these graded query-document pairs are called the *judgments*.

When you have a set of queries and a reasonably good understanding of what “good” looks like, you’re finally ready to start tuning the parameters. This process involves issuing *several* queries, looking through the results of *each* query simultaneously, and then identifying and diagnosing relevance issues as they arise. If you have a good understanding of how the query works, you should also have a good idea of how to modify search parameters. In the preceding Yowl example, if the first page of search results contains only nearby restaurants that are weak matches for the query, then the location signal should be given less weight.

Needless to say, gathering judgments is a labor-intensive process. If you find yourself revisiting relevance settings often, it may be worth spending time to automate this process. You could build a relatively simple application that allows a content curator to issue queries, view results, and mark results as either appropriate or inappropriate for the given search. When enough queries and documents have been annotated (when you have enough relevance judgments), you can automate relevance testing so that each time you modify search parameters, all test queries will be reissued and the new settings will be scored based on how good the results are across *all* test queries.

We call this automated approach *test-driven relevance*. With test-driven relevance, you no longer have to issue searches by hand, and more important, you no longer have to mentally keep track of the search behavior across all searches. Tools like Quepid (<http://quepid.com>) help simplify the process.

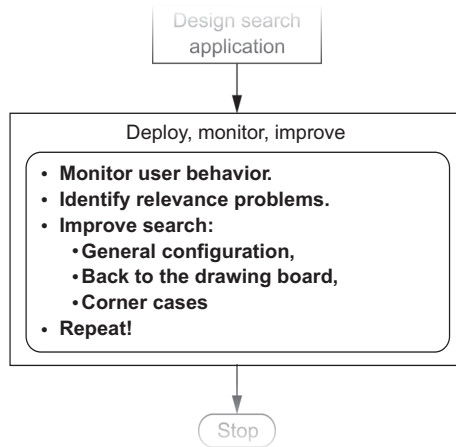
Ultimately, the ideal solution for tuning parameters is to push automation one step further. Many techniques automatically gather relevance judgments given sufficient user traffic. Armed with these judgments, a program can automatically adjust query parameters on your behalf. In the field of information retrieval, this highest level of automation is known as *learning to rank*. Learning to rank turns out to be a tricky problem to solve, but it has become a focus in information retrieval. So be on the lookout for improvements and breakthroughs in the near future.

We’ll have more to say on learning to rank and test-driven relevance in the next chapter.

## 9.4 *Deploying, monitoring, and improving*

At long last, with your query parameters finely tuned, you’re ready to go live with the Yowl restaurant search application! You’ve gone through a lot to get to this point in terms of planning, designing, and implementing the search application. You can breathe a sigh of relief as some of the hardest work is behind you. But, in many ways, deploying the search application means that you’re just at the beginning of your *real* work. Now you enter into a cycle of monitoring, updating, and iteratively improving your search application, as shown in figure 9.9.





**Figure 9.9** Deploying the search application is the start of a cycle of monitoring and iteratively improving search.

### 9.4.1 Monitor

In designing a search application, many of the choices are subjective—from the look-and-feel of the UI to the parameter values for your query, you rely on intuition to guide your decisions. But as soon as the application is launched, you have access to a new stream of information that reflects the quality of your search application. What’s this new stream of information? User behavior.

If a user understands, enjoys, and gains value from Yowl, the behavior of your users will reflect this. And if users encounter problems with the application, you can analyze user behavior in order to track down and correct these problems. Similarly, if you make a change that introduces a new problem, you should be able to see a corresponding shift in user behavior. Therefore, from the moment the application is deployed, it’s important to gather information from your users. There are many potential sources of behavioral information, but here we list just a few to get you thinking about the possibilities:

- *Time on page*—If users glance at search results and quickly leave, they aren’t finding what they’re looking for. Conversely, your relevance feedback may be so good users find answers instantly and leave.
- *Click-through rate*—You can tell that a user has found something interesting when they click through to find more details.
- *Conversion rate*—If a user buys a product (or in the case of Yowl, grabs a restaurant discount), search is working. This is *the bottom line*.
- *Retention*—If users come back to the app regularly, they find it useful.
- *Deep paging*—If users regularly need to go to the second or third page of search results, this may be a symptom of a relevance problem. Why aren’t the results on page 1?
- *Pogo-sticking*—Users click a result but then quickly find out it’s not what they want and return to the results listing. This reveals a problem with relevance

feedback. Ideally, the user should get a good sense of whether the document is worth looking at *before* clicking through to the details page.

- *Thrashing or reformatting*—Users change their search several times in a row because they’re finding nothing that meets their information needs. This may indicate users have a complex information need and your relevance feedback is pointing them in the correct direction.

Whatever metrics you choose to track, the absolute value is often less important than the change in value over time. For instance, deep paging on an e-commerce site may indicate that a user isn’t finding the item needed; but for legal or patent search, deep paging might indicate that the user has uncovered a rich vein of useful documents. Because we’re often looking for telltale *changes* in behavior, it’s important to gather and track baseline values for user-behavior metrics. Then, if you notice any sudden changes or degradations, you can investigate the problem further.

Search logs are another great source of information worth keeping an eye on. By reviewing the searches that your users are making, you can start to build a clearer understanding of users and how they’re using your application.

#### 9.4.2 *Identify problems and fix them!*

When tracking search history and user behavior, it’s easier to spot search problems as they creep up. The easiest problem to spot is the zero-results search. You can almost always do better than “sorry—got nothin’ for ya.” And finding zero-results searches is as simple as grepping through the appropriate search log.

More generally, when you see an unexpected degradation in metrics, try to find the underlying pattern. Usually the search problems that arise from well-tuned search applications represent some sort of relevance corner-case. Can you find a pattern in the user strategies? Maybe Yowl users searching by cuisine find their restaurant, but users searching for a specific dish tend to get lost.

Maybe the underlying problem has to do with a particular topic or set of query keywords that are performing poorly. For instance, as users begin using Yowl, you notice that those seeking French cuisine appear to have a particularly bad experience. Among users who search for “French food,” “French cooking,” and the like, you see a good deal of thrashing, low click-through rate, and poor retention. By looking through the search log and trying some of the problem queries yourself, the reason for the bad user experience becomes clear: every result for a query containing the keyword *french* returns nothing but fast-food restaurants! You dig into the ranking with explain set to true and soon realize that *French fries* are the source of the troubles. Having isolated the root cause of the relevance problem, you can make the appropriate corrections.

As you isolate relevance problems, you may be required to fix them at any level of the search application. Here are some examples from each layer:

- *Configuration*—Indexes are typically sharded across several servers, so that each server holds just a portion of the documents. Because the number of documents

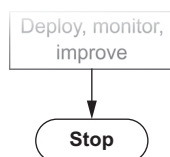
will be different for each shard, relevance can sometimes be negatively impacted. A change to the index configurations can address this.

- *Text analysis*—In the preceding example with french fries, you might need to add an entry in the synonyms file to ensure that french fries aren't conflated with French cuisine. Another common fix is to add certain terms to a *protected words* file so that they don't get inappropriately stemmed.
- *Signal modeling*—Certain signals may be unnecessary or detrimental to search relevance. For instance, as you watch Yowl users, you might find that the description field is too noisy to be useful. After testing in production, you may choose to reduce the description field boost or to remove the field altogether. Alternatively, important signals are sometimes missing and need to be incorporated into search.
- *Query*—As the needs of users and the business change, you'll need to add or remove pieces of the query in order to affect the desired change. If Yowl decides to have tiered services, you may offer an extra boost to your premier members. This will involve dropping in a new `function_score` function and rebalancing the query.
- *Relevance feedback in the UI*—You need to watch your users interact with your application and make sure that they understand what they're seeing. They should be able to easily find what they seek and understand the results that they're provided. If they don't, you should address this by improving the relevance feedback that you provide. For instance, Yowl might provide users with keyword autocompletion in order to guide them toward fruitful searches.

Whenever you successfully address one tough relevance issue, be aware that many more are waiting to be found. Search application maintenance is a process of continual vigilance. Besides the problems yet to be discovered, each new document in the corpus, each new query, and each new business need holds the potential for introducing new relevance problems. But don't fret! A principled approach of monitoring behavior metrics, finding and fixing problems, and testing solutions will keep your search application on track and your users satisfied.

## 9.5 Knowing when good is good enough

Almost as important as building a great search app is to know *when to stop building* (see figure 9.10)! When building a search application, you're effectively building up a model for natural language. But natural languages are dirty, dirty things. English, for example, is full of words that can be used in different settings to mean different things. Some



**Figure 9.10** It's important to build a highly relevant search application, but it's also important to know when to stop.

words in English even serve as *their own* antonyms! For example, consider *cleave*, which means *to split apart* and also means *to adhere to strongly*. See! English is dirty!

Therefore, be aware of this stark truth:

When building and refining a search application, you'll eventually be met by the law of diminishing returns.

In developing the basic Yowl search, you were probably 50% of the way toward “perfect” search when you turned on the search engine and dumped in the restaurant data. By more carefully considering the characteristics of individual fields, and by building and tuning your queries, you jumped to 85%. After deploying search and identifying and resolving the obvious relevance issues, you moved to 93%. Anything beyond this takes considerable work and provides only incremental benefit. And furthermore, the more “perfectly” you configure search at any point in time, then implicitly the more brittle and resistant to change search becomes. You'll eventually arrive at a point where you're effectively overfitting the search application according to the narrow understanding that you have at a particular point in time. Any new documents or business requirements that contradict this understanding will cause new relevance problems to pop up.

Prevent overfitting your search by stepping back and asking yourself, “Is it worth it?” If you can make a change guaranteed to fix an important corner case, go for it. But if your search is already good and you wish to eke out just a little more performance, be cautious—you might be inadvertently making search brittle and resistant to change.

## 9.6 **Summary**

- Typical steps when developing a search application include modeling your users with personas, identifying users' typical needs, and designing an application that will meet those needs.
- Integrating new sources of information supports user and business relevance requirements.
- Transforming and cleaning data sources through transformation and analysis allow you to implement low-level user and business matching rules.
- Low-level signals can be grouped into higher-level signals that balance and combine different user and business ranking concerns.
- A search application, once deployed, should be monitored based on user interactions to identify new problems and unexpected use cases.
- Remember the nature of diminishing returns when resolving relevance corner cases. This should help guard against overfitting search according to the narrow understanding of requirements at a given point in time.

# 10

## *The relevance-centered enterprise*

---

### ***This chapter covers***

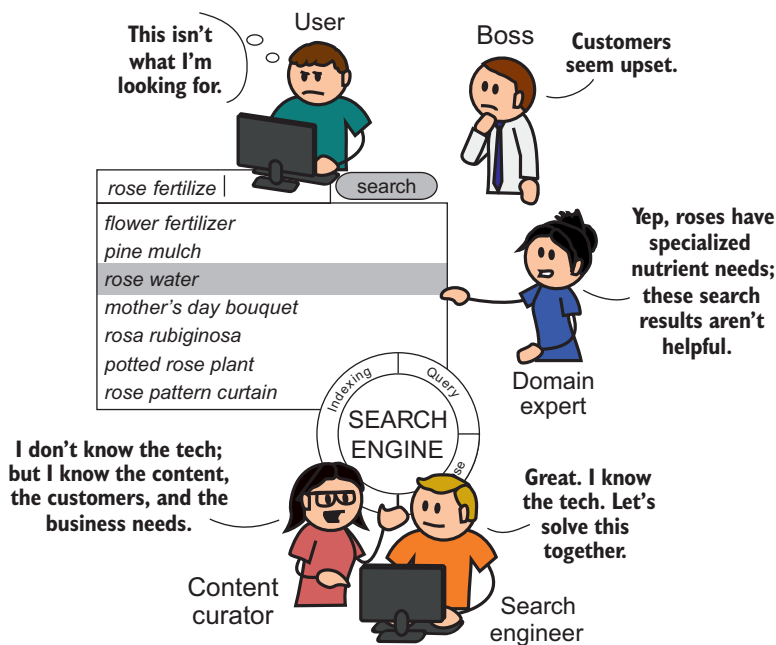
- Improving how your organization works on relevance problems
- Interpreting user behavioral data to evaluate search relevance
- Making the broader culture aware of search relevance and its value
- Incorporating domain experts, marketing, and other colleagues
- Adding content curators and information scientists to the search team
- Measuring correctness with test-driven relevancy

Our conversation up to this point has been about *how* to implement the technical requirements of search relevance. We've yet to discuss the fundamental *why* behind implementing these requirements. If you're a software developer, you know it's easy to toil for months on end on a feature or product. Only finally when you release it do you discover that the market has little interest in your work. Your careful craftsmanship is squandered as users' feedback renders a harsh verdict on what you've delivered.

Search relevance is no different. You might waste months of skilled work, carefully implementing supposedly correct ranking requirements. Only upon release do you realize that users aren't using search as you expect. Perhaps shoppers on e-commerce sites expect to search within product reviews, yet you've implemented only regular product search. Or perhaps a hospital system's search returns pages about diseases when users expect to find doctors who treat those diseases. Despite all our hard work, users surprise us all the time with what they expect from search, not neatly fitting into how we'd like them to behave.

Building a robust, cross-functional culture to discover what users want out of search is the central focus of this chapter. How do you build a business culture with accurate and fast feedback systems to correct your search application's requirements? How can your work be continuously informed or corrected to line up with users' expectations?

Much like chess, relevance is easy to get into—delivering a basic untuned solution is often simple. But discovering subtle user expectations accurately, and quickly responding to them with technical finesse, turns relevance into a frenetic game of speed chess. Luckily, unlike speed chess, relevance is a team sport. Skilled, key positions throughout the enterprise can help you understand what the user needs and respond appropriately. Figure 10.1 details a cast of characters: the roles (relevance engineer, content curator, boss, domain expert) and solutions (analytics, tests) that can deliver feedback to relevance work.



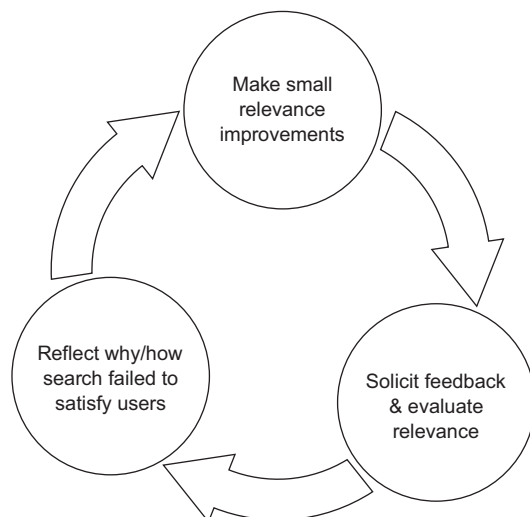
**Figure 10.1** Forms of feedback to the relevance engineer, starting from most removed from the problem (the user, boss/manager types realizing search is bad) to those involved constantly (the content curator role and relevance testing)

## 10.1 Feedback: the bedrock of the relevance-centered enterprise

To figure out what users want out of search, you need to admit something that might feel uncomfortable. Despite holding the title *relevance* engineer, you likely don't know what users deem relevant. You probably have no idea how an eight-year-old will approach a searchable, kid-friendly database of dinosaurs. You almost certainly don't know how a doctor might use a medical search application to diagnose a patient. Or how mechanics might search for car parts. The reasons that users search are endless. Your capacity to understand users and their domain is limited. You operate in the dark, and failing to appreciate this might be your undoing.

Not knowing what users deem relevant is made worse by the inviting nature of search to tweak and prod. Most development work falls in the category of small alterations such as boosting, changing the types of queries, or modifying analyzer settings. Because these little changes alter everything about your search ranking, they have extremely broad ramifications. In your ignorance of your users' relevance needs, you can easily throw search completely out of whack. You might guess right and do something great! Or you might ruin existing, working use cases with the simplest of changes. It's like you're flying a giant Boeing 787 jet in the dark with a giant dashboard of easy-to-flip switches in front of you. Flipping one switch might help you, but when flying without feedback, you're just as likely to fly into turbulence or worse!

To account for your lack of understanding of what *relevant* means, we advocate strongly for a certain style of work. The best way to improve relevance is to deliver a solution, observe how it fails to some degree, and adjust accordingly. We refer to this style of work as *iterative* and *fail fast*. It's iterative in the sense that it focuses on delivering a real, interactive search solution quickly for evaluation. It's fail fast because it anticipates immediate, small failures as the best mechanism to adjust course and avoid more-catastrophic failures. Figure 10.2 illustrates the ideal iterative cycle: bite-sized



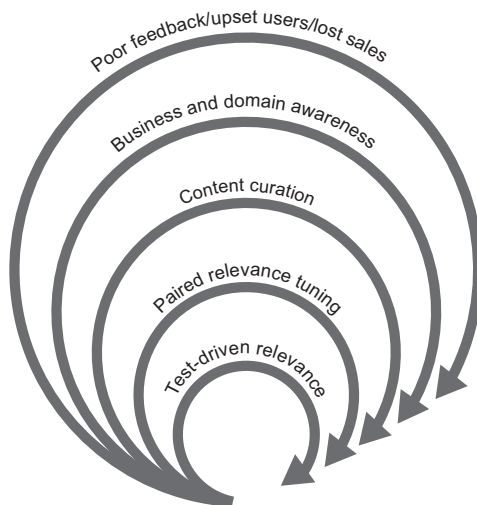
**Figure 10.2** Ideal search relevance iteration: trying simple adjustments, soliciting feedback, and failing fast in order to continuously improve

changes are delivered for evaluation, feedback is observed, and the course is adjusted over and over.

Feedback is the most important ingredient. Obtaining and understanding search feedback needs to be your obsession. It ought to become your whole team's obsession. This chapter advocates a *user-focused culture*—one that bends over backward to bring relevance engineers accurate and precise user feedback to guide their work.

A user-focused culture recognizes several sources of feedback. First, it recognizes that domain experts in your organization can help you correct the direction of relevance work. Yet corrective feedback goes beyond these experts and becomes a full-time job for relevance engineers. You'll see that one role, which we call the *content curator*, becomes the high priest of search feedback. The content curator takes command, examining user behavioral data and the broader business, to understand search correctness. Ultimately, you'll see how you can speed up feedback in the form of test-driven relevancy. As discussed in the previous chapter, this form of feedback continuously evaluates search changes, highlighting where search is taking a slide for the worse.

Figure 10.3 outlines these increasingly more potent forms of user-focused culture. You'll move through this progression through the remainder of this chapter. Each loop in this diagram is a feedback loop—a style of iterative feedback associated with increasingly user-focused organizations. The organization starts on the outermost feedback loop: complete ignorance of the users' search needs. We discuss this phenomenon in section 10.2. From there, you move inward, evolving to correct the course of relevance development. As you move to more-evolved forms of user-focused culture, feedback arrives to the relevance engineer faster and more accurately.



**Figure 10.3** Various key feedback loops enable the relevance engineer to refine search relevance.



## 10.2 Why user-focused culture before data-driven culture?

Before working through the relevance feedback loops, let's discuss one common misperception you might have at this point. As an engineer, you might be thinking, given the problem of gathering relevance feedback, why not simply gather tons of data on what users think of search and just work with that?

In today's world of high-end user analytics and machine learning, it's tempting to sidestep cultural issues. Many discuss building a data-driven culture that bases user-experience decisions solely on user behavioral data. As you may recall, we discussed many such metrics ourselves. In the previous chapter, we introduced click tracking, deep paging, thrashing, and others. With search metrics like these, why bother consulting marketing, domain experts, and others in the organization? You can discern search relevance correctness directly from the data—right?

We believe user-focused culture must come *before* data-driven culture. For search, good user data is often unavailable or hard to interpret. When data is available, it often takes more than a relevance engineer to derive insight about whether a user is satisfied. Making scientifically valid claims from data, frankly, is hard. Making scientifically valid claims about how users think and feel can be doubly so. Let's consider this difficulty, because it speaks directly to why we place cultural issues so highly.

First, why would data be unavailable or not useful for search feedback? For most applications, the majority of individual search queries are special snowflakes. They're too rare to derive meaningful insights. Let's think about why this is. Consider the distribution of all user searches in your application. You probably have a handful of extremely common searches and an extremely large set of rare, special snowflake searches.

Let's think through an example for a garden supply e-commerce search application. The most common searches are for things such as hoses, flowers, and pots. These occur many times each minute. But after this handful of common searches, other searches become increasingly obscure. For instance, a search for “*Rosa rubiginosa*” (the scientific name for a type of rose) might occur twice a year. It's possible that these rare searches compose the bulk of the search traffic; each search item is rare, but there are just so darned many rare searches!

This search phenomenon has come to be called *long-tailed search*. As shown in figure 10.4, *long-tail* refers to the shape of the distribution of possible search items.

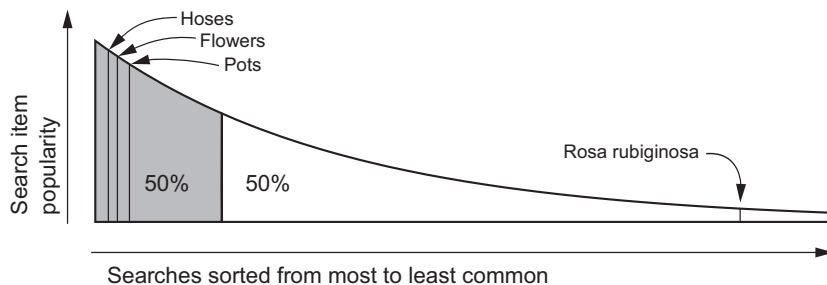


Figure 10.4 For long-tailed search, a majority of traffic is associated with rare search items.

Notice that a relatively small set of items (on the left side) compose 50% of the search traffic in this example. The remaining 50% of search traffic is attributed to a vastly larger number of relatively rare searches (on the right side). These searches form the long tail of search.

Search's long tail can make purely data-driven relevance feedback challenging. If your application has a long tail, you may need to rely on more indirect qualitative measures for relevance feedback. In our gardening example, user data for popular searches such as "Hoses," "Flowers," and "Pots" could be useful. But for most other searches ("Rosa rubiginosa", "Rosa rubiginosa Fertilizer"), user data may be a vague rubric. The long-tail data might tell you what use cases are important to users (such as searching on scientific names), but not much else.

The second problem you'll encounter is deriving information about user satisfaction from behavior data. How will you interpret when users are dissatisfied, for instance? You might assume that visiting page 2 (and beyond) of the search results is clearly a negative behavior. But your user population may want to exhaustively research all available results. It's common in patent or legal search to leave no stone unturned as every match is carefully examined.

The same problem exists when trying to identify results that users perceive as relevant. Users might appear to be satisfied by a search result because they purchased the found product or read through an article. But this conclusion is far from straightforward. Users may decide to purchase at a mediocre, barely relevant result if more-promising results never come to the top. If your users search for "Rosa rubiginosa", and you show them only daisies, the fact that they sometimes purchase daisies doesn't mean a certain kind of daisy is the most relevant result. Less absurdly, they may purchase a mediocre sweet briar rose product and decide that's all your gardening shop has to offer. Meanwhile, a stellar product is buried deep in the data and would result in more purchases if only it surfaced.

The point is, even with a pile of user behavioral data, the relevance engineer remains ill prepared to read the tea leaves. Search quality feedback isn't simply a matter of digesting use data. You need to understand user context, domain expertise, and business requirements to assess a random user's satisfaction with search through data. Just as with relevance, you don't hold the answers here. Your colleagues who better understand the domain, business, and users must help you interpret your application's usage data.

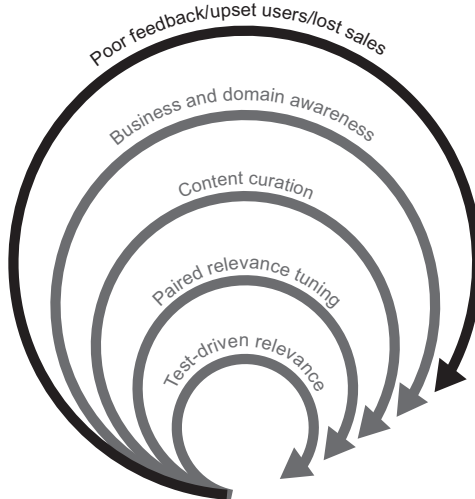
Good feedback goes beyond data. It's based on a broader, collaborative organization that can interpret both search correctness and user behavioral data together. Only then can data be a factor in providing you good information on how users perceive search. Instead of a data-driven culture being the starting point, it's in fact the most mature form of a collaborative, user-focused culture. By building a user-focused culture, you'll end up at a place where data plays its appropriate role.

### 10.3 Flying relevance-blind

You often start with a more mundane problem: your organization isn't aware of search relevance. Stakeholders don't see relevance as foundational to their success. Instead of iterations that fail fast, the only iteration you undergo is the painful process of developing a complete search solution, shipping it to users, and then painfully learning upon delivery how disappointed they are.

Unfortunately, many of your colleagues may assume that the search engine has some kind of built-in Google-like intelligence. They may think the search engine knows best, and isn't meant to be questioned, configured, or programmed. For many, the idea that users have specific relevance expectations unique to your application is a truly novel concept.

Lacking awareness of relevance, in what we call the *relevance-blind enterprise*, can be a dangerous place for your organization to be. Being here exposes you to the most extreme, outer feedback loop: sales losses and customer complaints, as shown in figure 10.5. In this phase, users stop performing desirable actions as a result of their searches (they stop purchasing things, for example). It's not uncommon in this state, shortly after releasing or updating the product, for the organization to come upon a sudden and unpleasant awareness of the importance of search relevance. Unfortunately, this awareness comes only after a sudden downturn in user engagement and product sales.



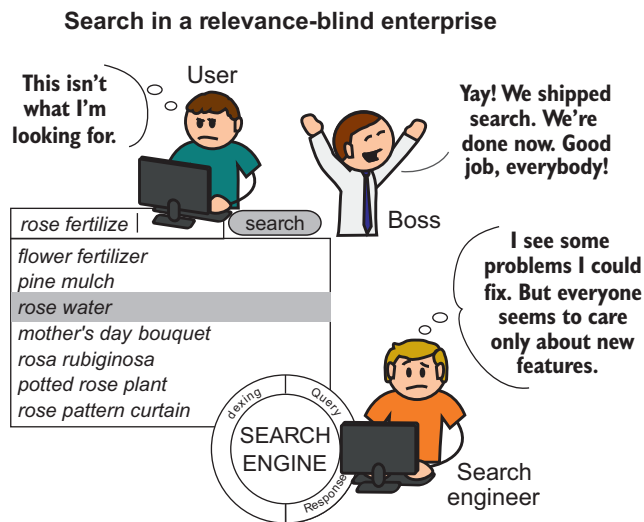
**Figure 10.5** The organization lacks awareness of relevance as an issue. Your team may perceive poor search only through the outermost feedback loop: lost sales and disappointed customers.

How do you know your organization is in such a state? Generally, these relevance-blind organizations share several characteristics. The predominant pattern you'll notice is that search work involves no relevance work. Instead, the focus is on features, performance, and scaling. Instead of a *relevance engineer*, you may be seen as a back-end *search*

*engineer* with no official relevance-tuning role. This is problematic when search is a core component of the user experience. Instead of helping create the user experience, you're seen as another back-end developer divorced from users and their concerns. Additionally, the organization may, to its peril, consciously keep you *away* from conversations that discuss user goals. Instead, they hope you'll make the search 10 milliseconds faster or figure out how to integrate another set of documents.

It's easy to see how this happens. Search engines appear to many as another data-store. Working with search engines shares many characteristics of working with something like a database: both must consistently store and retrieve data. They must remain available and high-performing, storing the data needed for our applications. Starting out, many may not want to question or think too hard about that weird, seemingly mystical ranking component inside the search engine. It doesn't fit how we think about other applications, and many would prefer to think of it in terms of the databases they're already familiar with.

Because of this ignorance, relevance ranking becomes a blind spot when working on the application. If you take a screenshot of a typical search application, you'll find search results presented smack-dab in the middle of the page. But in a relevance-blind organization, as shown in figure 10.6, the front-end developers and designers fine-tune components of the search application other than search relevance ranking. Most features and bugs revolve around items in the application typical of most development: logging in/out, look-and-feel, and other custom features. When focused on search, your team would rather improve easier-to-understand components of the application: faceting, filtering, and other browse capabilities. Or perhaps improve the presentation of the search results themselves—but not how they're ranked.



**Figure 10.6** In a relevance-blind organization, nobody seems to see that searches return poor results. Instead the team is busy on other, more comfortable tasks.

These organizations often have a rude awakening when the app is released. Users type in their first searches and have a hard time finding what they want. In our consulting work, we've witnessed the aftermath of the relevance-blind enterprise over and over. The result is never pretty. Customers are disappointed, and users flock to competitors. Despite the hard work of application developers and designers, the overriding message that "search stinks" comes straight from users.

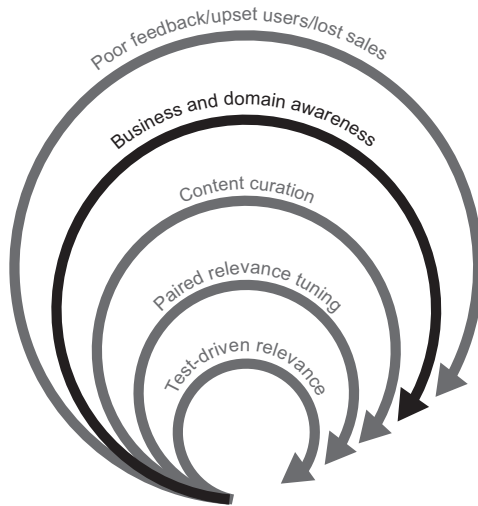
At this point, with the organization backed into a corner, the team turns to you, the search engineer, wondering what to do. Suddenly you aren't simply a back-end developer. You must take a central role in crafting the user experience! The problem we discussed earlier, enabling relevance engineers to get more-immediate feedback for their work, becomes crucial to the success of the application. Now begins your hunt for ways to fail fast. You'll need to identify ways to get accurate feedback on relevance before your organization suffers another embarrassing blunder.

#### **10.4 Relevance feedback awakenings: domain experts and expert users**

Being suddenly tasked with determining what users want out of search ranking can be daunting. As a relevance engineer, you won't be intimately familiar with what every group of users expects. New to relevance work, you might feel more comfortable in the realm of databases and code than user-experience concerns. Making matters worse, the users may be from a specialized domain, such as medicine or law. Searches like "Myocardial Infarction" might make little sense to you. Finally, it may turn out that the business itself has its own ranking expectations, hoping to steer users one way or another through search—perhaps toward more-profitable products or more-informative web pages. You need to seek feedback from the broader organization to adjust course.

To discover what *relevant* means for this application, you must become a champion for the cause. You'll need to get organizational buy-in to create the first relevance feedback loop. And you'll quickly realize that the information you need is scattered throughout the enterprise. Depending on the kind of organization, the definition of "what users want" may be spread among marketing, sales, QA, legal or medical analysts, management, and any number of groups. You must become a cross-functional champion of collaboration. Instead of gleefully hacking away on fun code, you must become a highly social mover. To figure out what the user expects from the search experience, you must break down organizational barriers and find the answers you seek.

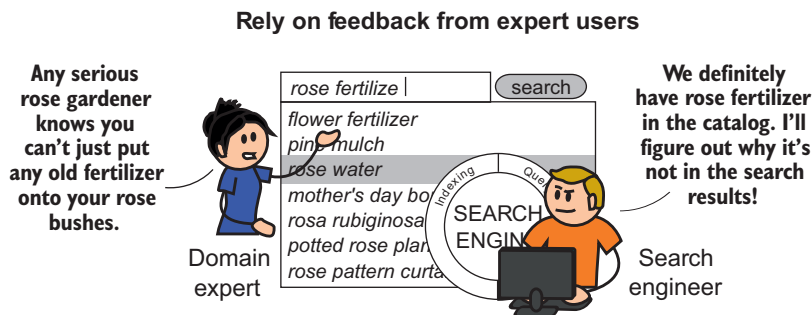
What you're doing is constructing the next feedback loop, as highlighted in figure 10.7. This feedback loop is the organization's first attempt to validate relevance and adjust course before bad effects are seen by users.



**Figure 10.7** First feedback loop internal to the enterprise: relevance feedback is acquired ad hoc from QA, BA, and other specialized siloed teams.

Out of the various groups that study and engage users, you should first look to an expert user. An expert user was once a member of the user population and now works at the organization. If you're building search for a gardening site that caters to professional horticulturalists, then perhaps marketing, QA, or other groups in your organization staff horticulturalists. Expert users are a gold mine for search feedback. They can directly simulate the appropriate negative reaction that users will experience with irrelevant search. Yet unlike most users, they're deeply invested in your success. They're much more likely to give you detailed, constructive feedback. Figure 10.8 shows an expert user guiding a relevance engineer to better results.

Your organization might also perform usability testing. With *usability testing*, groups of target users are studied as they interact with the application. They may have their



**Figure 10.8** The relevance engineer receives feedback on the correctness of the relevance solution from domain experts.

behavior tracked to great depth—including eye tracking and expression monitoring. These users may also provide qualitative feedback about the application, answering questions about what they thought of it. You might be able to piggyback on such testing to get feedback about the application's search relevance. You may even be able to interact with such users, ascertaining what they found helpful with the relevance ranking and more important, what they didn't—getting the same rich feedback you obtained from the expert users.

Still, at this point, you're picking up relevance feedback through your own sleuthing work. None of these helpful but siloed groups have search as their central function. Other groups have other duties to attend to, and have only limited attention to give you. Thus your testing remains ad hoc, informal, and depends on the availability of members of other teams.

So even with persistence, this remains a low-information and slow feedback loop. The feedback given is sometimes deep and valuable and at other times limited. Although many problems are solved and avoided, others continue to get through. Still, the output of this phase can be crucial. By breaking out of the technical silo, you might be able to build powerful allies on other functional teams. Others begin to see the importance that search plays for users. Even with their spotty attention, you and your allies may begin to realize there's good reason for establishing a full-time role for managing the search quality.

## **10.5 Relevance feedback maturing: content curation**

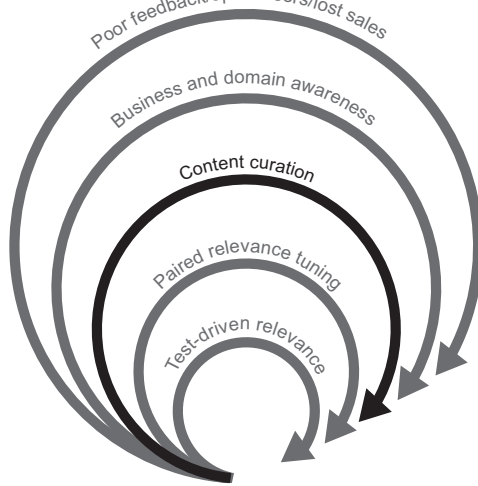
In the previous section, we discussed the relevance engineer as a champion for the cause of relevance feedback: taking down silos, wandering the halls, knocking on doors to identify how the organization defines relevance. This is a crude attempt to build an initial feedback loop to guide your work. Unfortunately, you can't sustain this level of feedback gathering and get your technical work done. Further, your colleagues don't always have time to give you sustained or high-quality search feedback.

More important, the answers you get from colleagues will grow increasingly inconsistent. More and more, finding the "right answer" will require reconciling multiple points of view on what search should do. The marketing department may push you to promote results based on advertisers and supplier relationships, while domain experts may advocate for the needs of high-end experts over lay users. Arriving at the "right answer" may depend on a subtle understanding of the domain, user expectations, office politics, and many other factors.

Instead of driving technical improvements, you may feel increasingly confused and bewildered. You're navigating an incoherent, alien world that requires many skills you may not have: an understanding of business needs, domain expertise, and user familiarity. Value may be gained initially as colleagues help point out what's obviously wrong with the search results, but making sense of how search ought to work beyond

these obvious fixes becomes a job unto itself—a job that distracts you from the important technical work you’re good at.

The organization solves this problem by adding a role responsible for search feedback: the *content curator* (as shown in figure 10.9).



**Figure 10.9** Content curation, the act of curating feedback from the broader enterprise for the relevance engineer

### 10.5.1 *The role of the content curator*

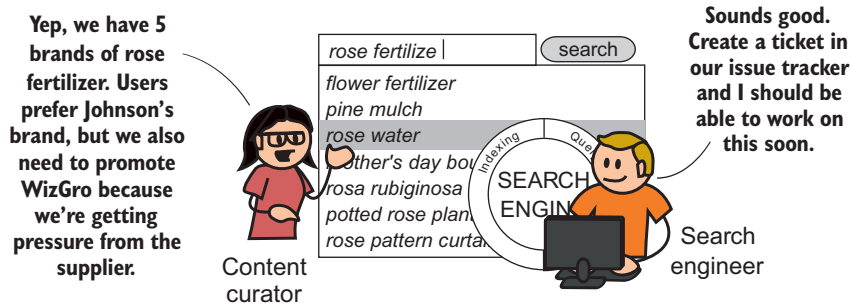
The content curator becomes responsible for accepting all forms of feedback, and determining the “right answer.” The ideal content curator is someone with enough business/domain expertise, user familiarity, and seniority to own search correctness. Content curators think critically about how search ought to behave. They understand deeply how users will interact with the search application and translate that into actionable feedback for the relevance engineer to implement.

In this role, content curators work with the broader business. They also interpret whatever user behavioral data might be available. They understand what the business (not just users) need from search. They capture all of this feedback and expertise into an actionable set of feature enhancements and bugs. They reconcile multiple points of view with political finesse into a coherent road map for search relevance. In a sense, they act as *product owners* for search. In Agile terminology, a *product owner* represents the broader business and stakeholders to the team. They act as the engaged stakeholder, they know how search *should* work, and they provide the technical team—relevance engineer(s)—the broader direction on where search should go, as shown in figure 10.10.

Some content curators might also be somewhat technical, allowing a certain degree of relevance self-service. They may also help organize and manage information. These skills can add a ton of value to search. We discussed how to use synonyms



## Bring content curators onto the team



**Figure 10.10** Content curator defines how search should behave based on feedback/consultation with the broader business and evaluation of user feedback.

and taxonomies in search in chapter 4. These tools can be used to dramatically improve the relevance of search results by relating different terms to each other. Because they have the right domain expertise, content curators can help manage these sorts of conceptual relationships.

A garden search engine that can understand that a “Reel Mower” is a kind of “Lawnmower” or that a “Lawnmower” is synonymous with a “Grass Cutter” can help a great deal in improving the relevance of search results. You don’t likely have the right skills to relate these concepts in a taxonomy of lawn-care products. But a content curator may be able to catalog these relationships in a way that anticipates the jargon and terminology that your users might expect the search to understand.

Thus an ideal content curator also contributes to the relevance solution. This can go as far as letting the content curator control boosts and weights for various ranking factors. Recall in the previous chapter’s Yowl example, you worked to balance the impact of content, restaurant location, price preference, and other factors. After the technical foundation is laid, who better to find the optimal balance between these factors than the content curator?

Adding a content curator to the team is a powerful multiplier for the effectiveness of your relevance efforts. Content curators allow the team to work toward their strengths instead of compensating for their weaknesses. They interface with the broader organization and listen carefully to user feedback. They’re available to provide consistent, informed course correction on search, helping to paint a picture in broad strokes that you can color in with technical finesse. They let you focus on your technical skills instead of being mired in politics and interpreting deep domain requirements.

### 10.5.2 The risk of miscommunication with the content curator

Once a content curator is on the team, you may be tempted to permanently rejoin the other programmers in the server room. You may think you can get back to fun technical problems, and solve relevance problems as they arrive in the issue queue. Yet as

you work, you'll quickly see that having a content curator on the team isn't a panacea. Instead, the most important communication line becomes the one between yourself and the content curator. Neglecting that relationship can create confusion and chaos for tuning efforts.

An unfortunate workflow can be established as the content curator assigns a new task to you: You rarely speak in person. Instead, a slow, lossy feedback channel over email and issue trackers might become ingrained in the team. You might take issues off the front of the work queue and work them until completion, thinking of the issue as relatively self-contained and easy to understand. Unfortunately, solving relevance issues is rarely so simple. Solutions require frequent follow-up communication, and ongoing evaluation by the content curator.

For example, if you're still in the business of building gardening search for horticulturalists, maybe you, as the relevance engineer, pick up a ticket that reads something like this:

*Searches for plants by their scientific name (Rosa rubiginosa) should return the same search results as if the nonscientific name was searched for (sweet briar rose).*

Ah, simple enough! The content curator delivers term equivalences, and you add a few simple synonyms. After you declare the task done, in a few days you get another email from the content curator:

*Hey relevance engineer,*

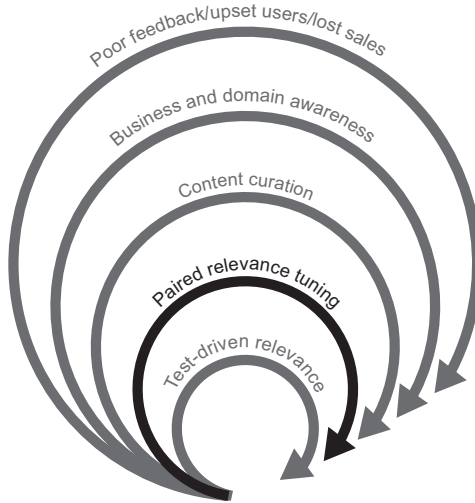
*I'm seeing searches for "Rosa Rubiginosa" returning other rose species such as "Rosa Glauca," not just for a specific type of rose. Also, I sort of expected searches for generic "rosas" to show you a cross-section of different species, and instead they seem to be biased toward one species. Should "Rosa" also return the same results as "Rosas"? Is this possible? Let's discuss...*

There are important distinctions between these species of roses—and an avid gardener and a content curator will immediately pick up on this. Further, the content curator and the broader business have unspoken expectations. You may not understand these distinctions, how users will use these "Rose" scientific terms, or the underlying motivation in the use case. Even if they think you understand the issues, you continue to suffer from a limited ability to evaluate the correctness of the search solution in real time. You may think confidently, "I can do this!" yet learn there are more subtleties to horticulture than you appreciate!

## **10.6 Relevance streamlined: engineer/curator pairing**

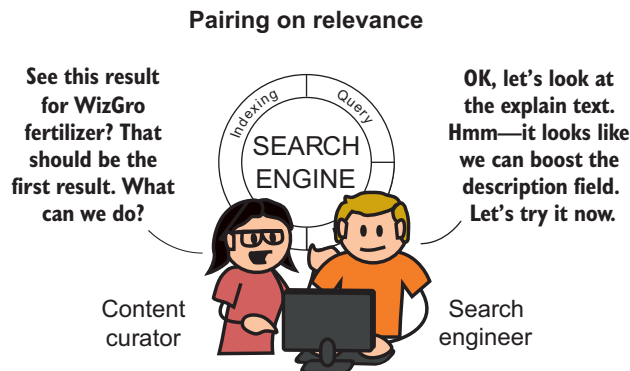
To appropriately tune search relevance, you need to have frequent, in-person conversations with the content curator to know how the search application is expected to behave. These communication lines need to run even deeper than the occasional chat. We advocate that you work side-by-side with the content curator when solving relevance problems.

You may have heard of *pair programming*, whereby two programmers work together to solve a programming problem. Here, the content curator and relevance engineer participate in *pair tuning*, working at the same desk to improve search relevance. This is our next level of feedback, as shown in figure 10.11.



**Figure 10.11** Next feedback loop: content curator and relevance engineer coworking

Through pair tuning, you both sit at the helm of the giant search-relevance jumbo jet! Your fine-tuning can be evaluated immediately by the content curator. Simple mistakes can be seen and fixed with simple course corrections, not after trying to exchange confusing and time-consuming emails that waste your time going far off course. Both sides begin to see tuning as a highly collaborative, iterative cycle requiring the careful application of two distinct forms of expertise. Figure 10.12 shows the content curator and relevance engineer working closely on relevance problems.



**Figure 10.12** The relevance engineer and content curator can quickly solve problems when they work side-by-side.

Most important, when pairing, the duo can discover what's possible instead of just working toward what's required. The content curator might express problems more deeply instead of simply assigning tasks. You may be able to think more carefully about the broader ecosystem of solutions in the search space that can solve broader classes of problems. The communication can also begin to move the other way: you can increasingly let the content curator know which search tasks are simple, and which are more involved projects, quickly identifying the trade-offs and easy wins in certain enhancements.

Looking at the big picture, you've gone from seeking feedback by pestering others for help to now advocating for not only a full-time role, but nearly full-time pairing. Our feedback loops are telescoping down to increasingly tighter and more immediate forms of feedback. This feedback reflects the iterative nature of relevance work. Often the relevance engineer can modify behavior of the search engine quickly, but failing fast and early is important. By providing tighter and more immediate forms of feedback, your work is becoming increasingly poked and prodded toward correctness. Now when flying that jumbo jet, you're able to at least see out of the fog and stay at a safe altitude.

## **10.7 *Relevance accelerated: test-driven relevance***

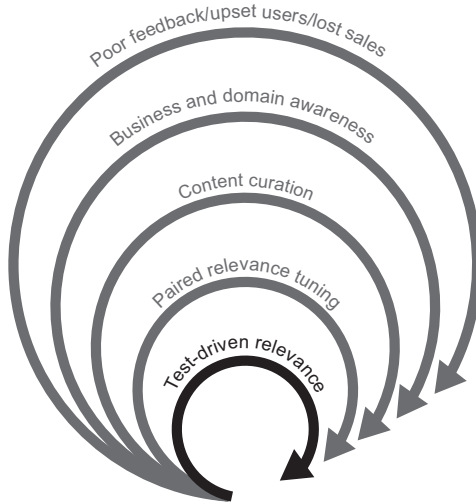
Pairing with the content curator solves important communication problems. Yet, quickly you realize that modifying relevance, even with the best intentions, tends to create new problems. Search tuning begins to feel like a game of Whac-a-Mole. You focus on a spate of problems. Solving those problems then causes existing, working searches to slide backward in quality. In our gardening example, for instance, you might thoroughly fix searches for scientific names ("Rosa rubiginosa"). Unfortunately, without knowing it, you might cause a popular search to return nonsensical results. A search for "Rose" might begin returning daisies! (A rose by any other name indeed!)

### **10.7.1 *Understanding test-driven relevance***

While tuning, you need efficient and real-time course correction for both you and your content curator copilot. What can you do to make feedback broad, instant, and quantitative?

The answer comes from the world of test-driven development in traditional software development. In these methodologies, you construct a set of automated unit tests that run your code. These tests evaluate the output of your code based on your understanding of application correctness. Failed tests then point you to broken functionality. With sufficient test coverage, you can implement new features and fix bugs, all while ensuring that old functionality continues to work.

*Test-driven relevance*, as shown in figure 10.13, solves the Whac-a-Mole problem by evaluating your search solution with automated tests as you tune. You still need the content curator and other colleagues to help define correctness. Ideally, these colleagues



**Figure 10.13** The most immediate form of relevance feedback: test-driven relevance

help you create the automated tests. These tests come in the form of important searches to programmatically run and evaluate for quality and correctness. This effectively puts the expert's brain in a box, letting you evaluate search against their expert feedback automatically.

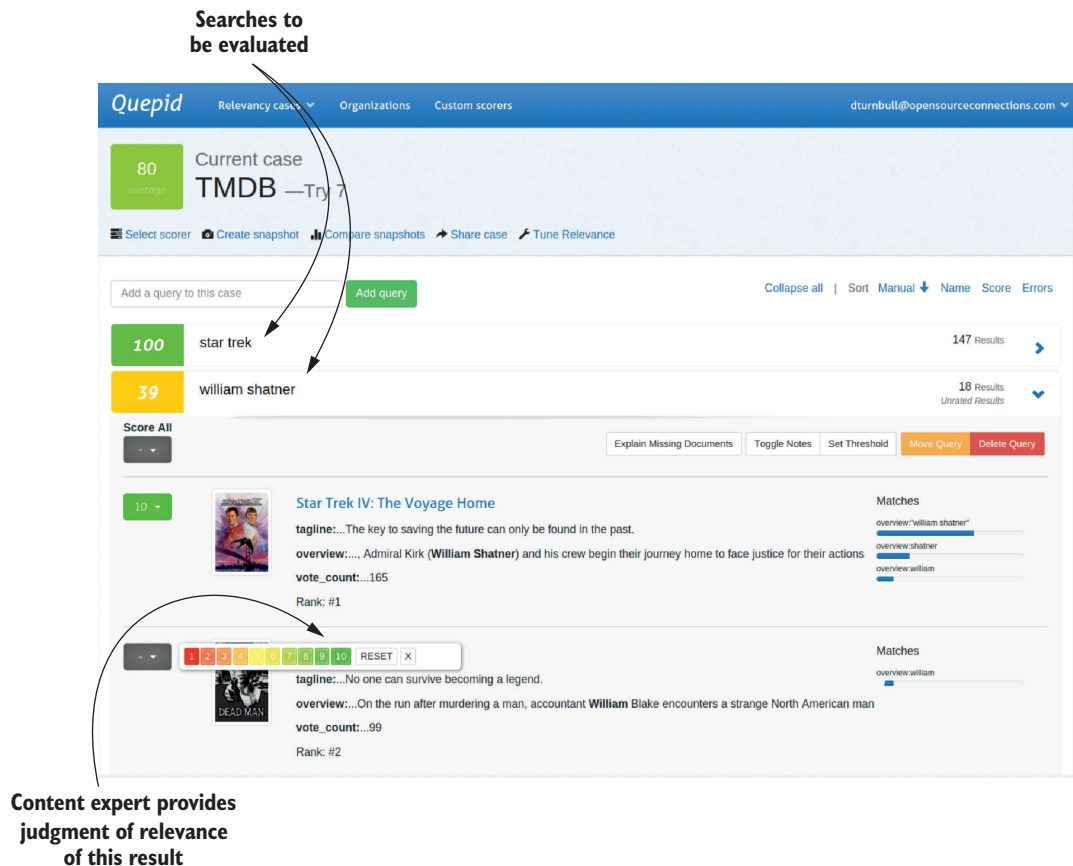
With test-driven relevance, pair tuning takes on a new dimension. Instead of improving relevance followed by manual testing, you and the content curator become responsible for the maintenance of relevance testing. If a fix for “Rosa rubiginosa” breaks the “Rose” query, you’ll see right away, and be able to make informed decisions based on the use cases that matter to your users and business.

What form do the relevance tests tend to take? For search relevance, testing tends to come in two forms:

- *Judgment lists*—For each search query, individual results are given a grade, or judgment. For example, a user search for “Rosa rubiginosa” might have sweet briar rose graded as an exact match. Other roses returned as matches might be graded as “OK” and all other results graded as “poor.”
- *Assertion-based testing*—Instead of gathering explicit judgments, more ad hoc assertions are made in traditional unit tests. For example, assert that when a user searches for “Rosa rubiginosa,” the first result is a sweet briar rose plant.

Figure 10.14 shows gathering relevance judgments using Quepid (disclosure: the Quepid application is developed by one of the authors). A content curator uses Quepid to grade a search result based on the relevance to a query. This results in an overall quality score for the query against the current search solution.

Maintaining these judgments, although ideal, comes with significant costs. As data changes, you must constantly make new judgments. The work becomes cumbersome.



**Figure 10.14** Quepid (<http://quepid.com>), a test-driven relevance product for Solr and Elasticsearch, is seen here using judgments.

Moreover, it's not easily outsourced. Judgments require a basic level of domain expertise. Many organizations staff legions of relevance testers for the sole purpose of maintaining accurate relevance judgments.

In contrast to judgment lists, assertion-based tests allow more ad hoc and black-and-white testing. Traditional unit tests don't require the costly maintenance of accurate judgments. Rather, you simply specify correctness criteria. Many organizations get by on the simpler and easier-to-maintain unit tests. But simple tests can't as easily help you see fine gradations in relevance quality, query to query. There's no way to 75% pass a test, whereas with judgments you value knowing that a query is roughly 75% of the ideal. It could be that 75% is good enough!

From a political standpoint, tests have another important effect. They give content curators and other stakeholders a sense that they're also deeply involved in relevance

efforts. By helping define tests, they gain a sense of control and investment in the relevance process. Further, as they ask for changes, they can instantly witness the impact of their requests. Instead of being an obscure, mystical component that only the relevance engineers understand, search tuning becomes increasingly transparent and collaborative. These stakeholders will feel increasingly empowered and informed to discuss the behavior of search and the associated trade-offs.

You can imagine being asked in a meeting to make a disruptive change to the relevance tuning. With a test-driven relevance tool, you could make the requested change and rerun the required tests right there in the meeting. This gives the team immediate insight into the impact of the changes without constant conversation, indecision, and chance for conflict. Thus despite seeming to be about moving away from collaboration, encoding wisdom in tests enables deeper and more thoughtful collaboration. Search stops feeling like guesswork throughout the organization because it's far easier to understand the consequences of decisions.

### 10.7.2 Using test-driven relevance with user behavioral data

The tests described in the previous section depend heavily on the expertise in your organization. The content curator must use the expertise of the broader organization to try to get inside the heads of users, and to then encode this information into the testing tool via judgment lists or tests. The content curator depends on these colleagues being correct, and not having their judgment clouded with misperceptions, ignorance, or selfish motivations.

Because your colleagues have their own misperceptions, consulting user behavioral data (for example, what users click or purchase) to derive judgment lists becomes increasingly attractive. In the previous chapter, we discussed particular user patterns in search that *might* indicate frustrated users: pogo sticking, thrashing, and so forth. Content curators may be able to work with these patterns to identify frustrated users. They may further be able to identify when users *converge*, or do something valuable (for the user or the business), such as purchase an item or subscribe to a newsletter. These bits of information can then be translated into relevance judgments, either manually or automatically.

This seems like an obvious path, but even with large amounts of data about how users work with your search app, challenges remain. Understanding how behavioral data translates to indications of search relevance or lack thereof continues to require expertise about your users. The same colleagues who contribute to judging what's relevant/not relevant often need to be brought in to determine what indicators are appropriate to determine when users are frustrated or satisfied.

So using behavioral data itself has its own catch-22. You need your human, imperfect colleagues to help get inside the heads of users through behavioral data. At the same time, you need the data to help realign your colleagues' understanding of what users expect from search.

There's no silver bullet! Data-driven feedback comes with plenty of human baggage. Even your well-meaning colleagues sometimes try to find data that fits their personal biases or beliefs. Deriving meaningful information from data means having a smart team willing to be humbled by what they find. You need colleagues who can leverage their expertise to interpret data while simultaneously allowing their expertise to be informed by surprising user antics.

With careful and hard work, you *can* use this data to inform search correctness, up to automating the collection of relevance judgments. Combining user data with your organization's user expertise is the gold standard of relevance feedback. Automated feedback from behavioral data must be constantly reevaluated by a cross-functional team. This lets the relevance engineer test with confidence with user data.

## 10.8 *Beyond test-driven relevance: learning to rank*

With the team working hard to derive information from data, additional automated practices to improve relevance become attractive. Advanced organizations can begin to use machine-learning techniques to predict when a search result will be relevant for a query. The practice of automating relevance is known as *learning to rank*.

With learning to rank, either human or automated relevance judgments are used to learn which of your content's features predict relevance globally. This cutting-edge field is increasingly finding its way out of the halls of information science research and into advanced search shops.<sup>1</sup>

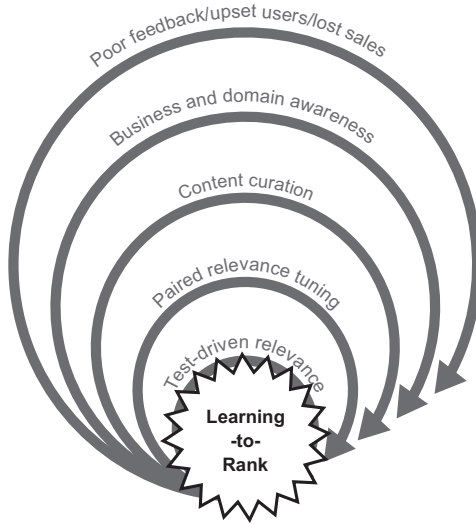
In the feedback loops diagram, shown in figure 10.15, learning to rank becomes the red-hot center inside test-driven relevancy. It builds on every other external feedback loop: it depends on a user-focused organization that knows how to interpret user data to evaluate relevance. This is an important note, as learning to rank is a hot topic these days. It seems to many that you can sidestep the hard work of creating an organization that intimately knows its users. Unfortunately, there's no superhighway that sidesteps the work we discuss in this chapter. You still need an organization deeply focused on users' search needs: a team capable of finding markers in user behavioral data for relevance. Deriving real information from data—information relevant to *your* search user experience—remains a hard cultural and technical problem.

Finally, you need to combine all the insights from this chapter with cutting-edge information retrieval and machine learning. Often simpler relevance gains can be gathered with the straightforward techniques discussed earlier in this book. In our consulting work, we're often hired to implement an advanced solution when a far simpler adjustment can provide more immediate and less risky gains for an organization.

---

<sup>1</sup> Solr readers may be interested in this patch, which proposes to add learning-to-rank features to Solr: <https://issues.apache.org/jira/browse/SOLR-8542>.





**Figure 10.15** Learning to rank is a cutting-edge method for relevance tuning that holds the promise of one day being able to automatically converge the ideal relevance parameters.

You don't need data scientists to provide a simple tweak to an analyzer or query strategy that gains you a significant—and with test-driven relevancy—measurable improvement to search's bottom line.

Nevertheless, with the right expertise and data in place, learning to rank can be extremely powerful in helping push beyond the “diminishing returns” of relevance tuning. It iterates on search instantaneously based on user interactions that signal relevance or lack thereof. When you have the right pieces in place, you can begin to think about this exciting technique.

## 10.9 Summary

- Search relevance work is highly iterative, and works best when you seek feedback by failing fast.
- User behavioral data (such as clicks and purchases) isn't a panacea, and a user-focused culture should come before a data-driven culture.
- Relevance-blind organizations lack awareness of search relevance and are often surprised when poor relevance leads to lost sales and complaining users.
- The relevance engineer begins to correct relevance issues by seeking feedback from colleagues who understand the user in general: domain experts, QA, and sales.
- A content curator can help optimize search by bringing feedback to the relevance engineer. Ideally, the relevance engineer and content curator work together on relevance.

- Test-driven techniques can help automate relevance judgments, but these techniques still depend on content curators and other user-focused professionals.
- Incorporating search usage data requires careful interpretation, but this data can inform and correct the organization's understanding of how users use search.
- Learning-to-rank techniques automate parts of the feedback process, but continue to depend on maintaining and interpreting behavioral data.

# 11

## *Semantic and personalized search*

---

### ***This chapter covers***

- Making search personalized for individual users
- Matching documents based on meaning rather than just words
- Implementing recommendation as a generalization of search

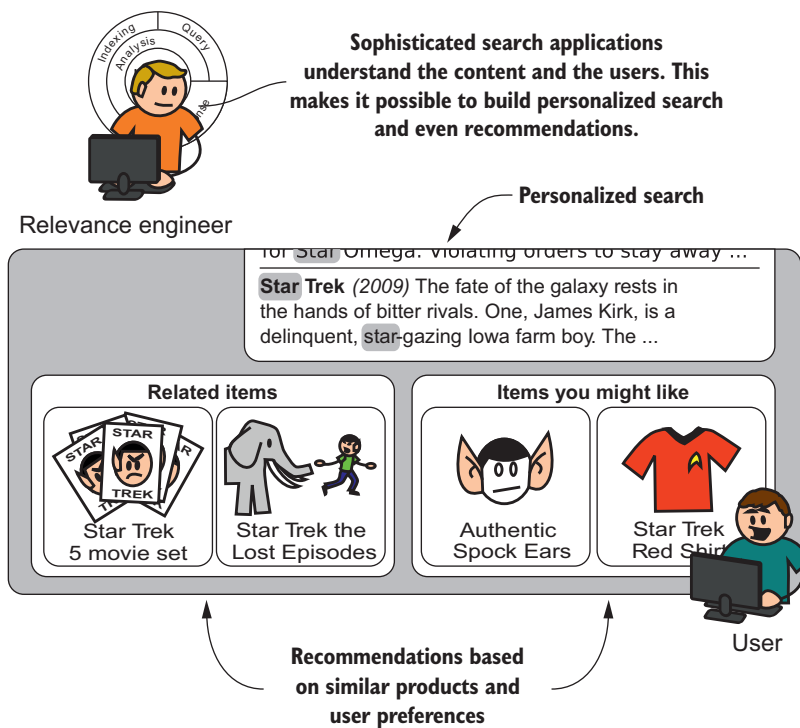
You're at the end of a long journey. You've learned to use search technology to build relevant search applications. But you're still just scratching the surface. In this final chapter, we look toward the horizon to explore some of the more novel—and experimental—ways to improve your users' search experience. In particular, we cover two related techniques that can provide better relevance:

- *Personalized search* provides search results customized to a user's particular tastes using knowledge about that user. User information can be gleaned from users' previous interactions as well as anything they tell us directly.
- *Concept search* ranks documents based on concepts extracted from text, not just words. Concept search relies on deep knowledge of the search domain, including jargon and the relations between concepts in that domain.

When used in tandem, the search solution understands users personal needs as well as the ideas latent in the content.

Building a good personalized search or concept search requires a considerable amount of work. You should treat the methods in this chapter as a conceptual starting point. But please realize that these methods bear some technical risks; they can be difficult to implement, and you often won't know how these methods will perform until after they've been implemented. Nevertheless, for established search applications, these methods are worth careful investigation because of the profound benefits that they may offer. In the discussion that follows, we present several ideas for implementing personalized and concept search. In both cases, we start with relatively simple methods and then outline more sophisticated approaches using machine learning.

In the process of laying out personalized search, we introduce *recommendations*. You can provide users with personalized content recommendations even before they've made a search. In addition, you'll see that a search engine can be a powerful platform for building a recommendation system. Figure 11.1 shows recommendations side-by-side with search, implemented by a relevance engineer.



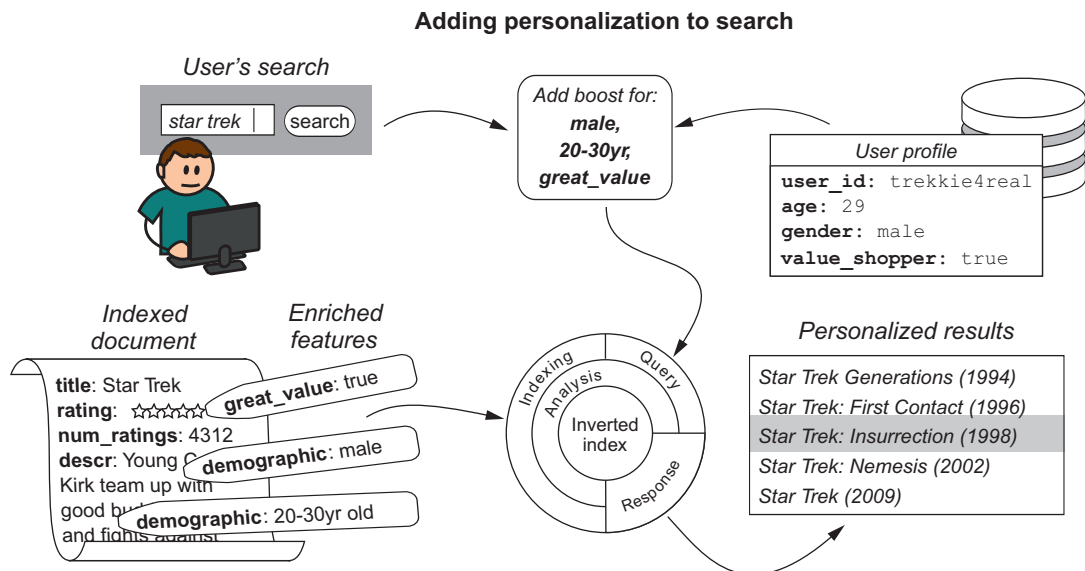
**Figure 11.1** By incorporating knowledge about the content and the user, search can be extended to tasks such as personalized search and recommendations.

## 11.1 Personalizing search based on user profiles

Until now, we've defined *relevance* in terms of how well a search result matches a user's immediate information need. But over time, as you get to know users better, you should be able to incorporate their tastes and preferences into the search application itself. This is known as *personalized search*.

Throughout this book, we've emphasized that, at its core, a search engine is a sophisticated token-matching and document-ranking system. We discussed techniques to ensure that search matches and ranks documents to reflect your notion of relevance. As we move on to personalized search, the fundamental nature of a search engine doesn't change. No special magic or hidden feature makes personalization possible. Search is still about crafting useful signals, and modeling them with the search engine through queries and analysis. The main difference is that instead of drawing information exclusively from documents, you'll look to the users themselves as a new source of information.

With this in mind, we turn our attention to the first method of building personalized search: *profile-based personalization*. With this method, you track knowledge of individual users with *profiles*. At query time, you refer to the user profile and use its information to boost documents that correspond to the user's tastes. Figure 11.2 demonstrates profile-based personalization using our previous Star Trek examples.



**Figure 11.2** Adding personalization with user profile data, including demographics and preferences

### **11.1.1 Gathering user profile information**

But how do you gather information for your user profiles? Well, if you're fortunate enough to have an engaged user base, you can create a profile page and wait for users to tell you about themselves. Be sure to provide incentives for users to fill out their profiles. For socially oriented sites, make the profiles public so your users can project their personality through the profile. Allow users to describe themselves in free text. Let them tag their profiles with categories that interest them. For private profiles, help users understand how creating a profile can provide a more personalized experience. For instance, you can directly ask users about their preferences and indicate that this will influence the behavior of the application. You can incentivize profile building with functionality; for example, by letting users bookmark items that they like, or share items with friends.

If you lack a profile page, you can still gather profile information from user interactions. By observing search behavior, underlying themes will reveal themselves over time. Perhaps a user has historically preferred certain brands. Maybe a user's choices indicate an interest in a particular domain such as photography or video games. By watching a user's interactions, you might identify demographics such as age, gender, and income level. The way that users filter searches often reveals *how* they make purchasing decisions. For instance, if users narrow search results by product reviews or price, you've learned something about that user's priorities. All of this information can be used to tune a user's search experience.

### **11.1.2 Tying profile information back to the search index**

As you gather user profile information, consider how this can be used in a search solution. In some cases, the connection is easy to find. For instance, if the user shows an affinity toward a particular brand, you can subtly boost that brand. And if a user often looks at reviews, boost items with several positive reviews.

But be careful how you do this boosting, because you can create an unexpected feedback loop that can damage search relevance. For instance, if the user buys Acme Co. products more than once, you should probably boost Acme's presence in that user's search results. But if that boost is overwhelming, the search page might be flooded with only Acme products—and less relevant Acme products at that. In future searches, users will show an increased interaction with Acme products, not because they like them, but because your boost makes Acme products so much more prevalent than other products. To make the situation worse, these interactions may look like an increased preference for Acme products and drive further boosting. With Acme products everywhere, you might soon find that customers quit using your search application altogether. Therefore, it's best to alter the boosting for particular product categories only when you have definitive evidence of a preference for that category—such as a purchase rather than just a product page view.

Sometimes you'll need to add new signals to the index in order to match information from the user profile. If you know that a user prefers less expensive, "higher

value” products, you can’t simply boost all items below \$20. A \$20 blender is a great value, whereas a \$20 can of beans is quite pricey. Instead, you should associate documents and users with some sort of general *value* rating scale (“cheap” to “boutique”).

Demographic information such as age and gender can be another good set of information to pull into search. Let’s say a profile indicates that the user is a young adult and male. If you know which products sell better in this demographic, give them a boost in the search results. To accomplish this, include a field in the indexed documents listing demographic groups with a high affinity to this item. The task of annotating this field with demographic information likely falls to the content curator. The information itself will probably come from marketing research.

With sufficiently heavy traffic, another source for demographic data is your search logs. Count the number of sales that occur within various demographics. The next time you reindex, add this information to the demographics field. Once this data is in the index, personalizing search is as easy as boosting using the current user’s demographic data.

## 11.2 Personalizing search based on user behavior

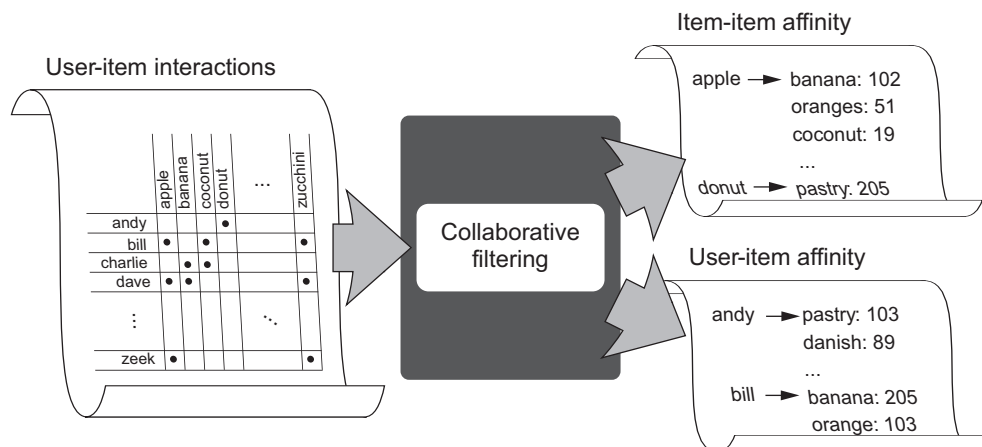
In the previous section, we showed that you can learn about users by observing their behavior in the application. In this section, we take this notion to an extreme with *collaborative filtering*. This technique uses historical information about user-item interactions (views, ratings, purchases, and so forth) to find items that naturally clump together. For instance, collaborative filtering provides an algorithmic way to state that “users who purchase *Barbie dolls* will likely also be interested in *girls’ dresses*.” You can incorporate this information into search for an even more personalized search experience. We call this *behavior-based personalization*. In this section, you’ll walk through a basic collaborative filtering example and see how to incorporate it into search.

### 11.2.1 Introducing collaborative filtering

For behavior-based personalization, you narrow your focus. Rather than considering user demographics, search histories, and profiles, you focus solely on user-item interactions. In this section, you’ll look specifically at user-item purchases. In principle, interactions can be anything: item views, saves, ratings, shares, and so forth. Given the data set of user-item interactions, you’ll use collaborative filtering to reveal hidden relationships among users and the items.

Collaborative filtering comes in many forms, from simple counting-based methods (which we introduce in a moment) to highly sophisticated matrix decomposition techniques (which are outside of the scope of this book). But no matter the technique, the input to collaborative filtering and the output from collaborative filtering follow the same pattern.

As shown in figure 11.3, the input is a matrix representing the users’ interactions with items in the index. Each row corresponds to a user, and each column corresponds



**Figure 11.3** No matter the method used, collaborative filtering typically takes a user-to-item matrix and returns a model to quickly find user-to-item or item-to-item affinity.

to an item. The values in the matrix represent user interactions. For the simplest case, the values of the matrix represent whether an interaction has taken place. For instance, the user has viewed or purchased a particular item. In the more general case, the values of this matrix can represent how positive or negative the user-item interactions are. The values can represent a user's ratings of products purchased in the past, for example.

Collaborative filtering outputs a model that can find which items are most closely associated to a given user or item. So, given a source item such as apple, the collaborative filtering model might return a list of items, such as banana, orange, or grape, for which apple has a high affinity. Additionally, each item includes an affinity score. Consider the output banana:132, orange:32, grape:11. Here banana has a relatively high affinity for apple, grape a low affinity.

### 11.2.2 Basic collaborative filtering using co-occurrence counting

To better understand how collaborative filtering works, let's look at a simple example using a co-occurrence counting approach. The following algorithm is a bit naïve; we intend it to be introductory and don't recommend that you implement it in a production system. Nevertheless, it builds up a basic understanding of collaborative filtering, and it removes the feeling that collaborative filtering is somehow magic. As you'll see, many machine-learning algorithms are based on simple ideas, such as counting the number of times that items are purchased together.

Jumping into the example, let's say that you work for an e-commerce website and you have a log of all items purchased across all users. Table 11.1 shows a sample.



**Table 11.1** Log tabulating users' purchases

Date	User	Item
2015-01-24 15:01:29	Allison	Tunisia Sadie dress
2015-01-26 05:13:58	Christina	Gordon Monk stiletto
2015-02-18 10:28:37	David	Ravelli aluminum tripod
2015-03-17 14:29:23	Frank	Nikon digital camera
2015-03-26 18:11:01	Christina	Georgette blouse
2015-04-06 21:50:18	David	Canon 24 mm lens
2015-04-15 10:21:44	Frank	Canon 24 mm lens
2015-04-15 21:53:25	Brenda	Tunisia Sadie dress
2015-07-26 08:08:25	Elise	Nikon digital camera
2015-08-25 20:29:44	Elise	Georgette blouse
2015-09-18 06:40:11	Allison	Georgette blouse
2015-10-15 17:29:32	Brenda	Gordon Monk stiletto
2015-12-15 18:51:19	David	Nikon digital camera
2015-12-20 22:07:16	Elise	Ravelli aluminum tripod

The first thing you must do is group all purchases according to user, as shown in table 11.2.

**Table 11.2** The first step for determining item co-occurrence is grouping items by user. A dot (•) indicates a purchase.

	Tunisia Sadie dress	Gordon Monk stiletto	Georgette blouse	Nikon digital camera	Canon 24 mm lens	Ravelli aluminum tripod
Allison	•		•			
Brenda	•	•				
Christina		•	•			
David				•	•	•
Elise			•	•		•
Frank				•	•	

It's in this next step where all the "magic" happens. For any given item A, you count the number of times that the purchase of item A co-occurs with a purchase of any

other item B by the same user. (Here, the term *co-occurs* doesn't imply that the purchases were made at the same time, but that they were made by the same user.) You perform this calculation for every pair of items encountered in the purchase history. After collecting all the co-occurrence counts, you have a measure of the *affinity* between any two items A and B.

As a specific example based on the information in table 11.2, consider the relationship between the Canon 24 mm lens and other items in the index. You can see that only one individual, David, has purchased both the Canon lens and the Ravelli tripod; therefore, these items receive a co-occurrence count of 1. But two individuals, David and Frank, purchased both the Canon lens and the Nikon camera. The co-occurrence count for this pair of items is 2. And finally, no user has purchased both the Canon lens and the Tunisia Sadie dress. Therefore, this co-occurrence count is 0. After performing this calculation for every pair of items in the index, you arrive at the matrix of results displayed in table 11.3.

**Table 11.3** Item co-occurrence counts for every item in the purchase history

	Tunisia Sadie dress	Gordon Monk stiletto	Georgette blouse	Nikon digital camera	Canon 24 mm lens	Ravelli aluminum tripod
Tunisia Sadie dress	-	1	1	0	0	0
Gordon Monk stiletto	1	-	1	0	0	0
Georgette blouse	1	1	-	1	0	1
Nikon digital camera	0	0	1	-	2	2
Canon 24 mm lens	0	0	0	2	-	1
Ravelli aluminum tripod	0	0	1	2	1	-

These values indicate the strength of associations between every pair of items. Notice that in this example, as expected, fashion items co-occur more highly with other fashion items. Similarly, photography items co-occur more highly with other photography items. In some instances, fashion items and photography items co-occur. This also is to be expected, because a few users are interested in both fashion and photography at the same time.

Item-to-item affinities can be directly used for item-based recommendations. The data shown in table 11.3 can be saved to a key-value store. Then, when a user visits the details page for the Ravelli aluminum tripod, you look up this item in the key-value store, pull back an ordered set of the corresponding high-affinity items (the Nikon digital camera and the Canon 24 mm lens) and present these items to the user as recommendations. As shown in figure 11.4, this is what Amazon does when it shows you its version of related item recommendations.

## Customers Who Bought This Item Also Bought

Page 16 of 25



Hamilton Beach 58149  
Blender and Chopper  
★★★★☆ 2,741  
\$26.49 ✓Prime



Crock-Pot SCCPVL600S  
Cook 'N Carry 6-Quart Oval  
Manual Portable Slow  
Cooker, Stainless Steel  
★★★★☆ 1,409  
\$24.00 ✓Prime



Kitchen MissionTM  
Stainless Steel Mixing  
Bowls 1.5, 3, 4, and 5 quart.  
Plus Measuring Cup and...  
★★★★☆ 146  
\$21.99 ✓Prime



Cuisinart CTG-00-CCR7  
Curve Crock with Tools,  
Set of 7  
★★★★☆ 136  
\$25.49 ✓Prime

**Figure 11.4** Item-to-item affinities can be used to make “related item” recommendations. When the user lands on the page for a Frigidaire microwave, you can display items with high affinity to the microwave in a panel similar to these recommendations from Amazon.

Taking the analysis one step further, you can find the affinity between users and the products in your catalog. To do this, refer to the user-item purchases in table 11.2 and, for every purchase made by that user, collect the corresponding item-to-item affinity rows and add them together. For instance, Allison bought the Tunisia Sadie dress and the Georgette blouse. Table 11.4 shows the corresponding rows from the co-occurrence matrix along with the sum of those rows.

**Table 11.4** User-to-item affinities can be generated by adding together rows of the item-to-item matrix that correspond to a user's purchases.

	Tunisia Sadie dress	Gordon Monk stiletto	Georgette blouse	Nikon digital camera	Canon 24 mm lens	Ravelli aluminum tripod
Allison purchases Tunisia Sadie dress	-	1	1	0	0	0
Allison purchases Georgette blouse	1	1	-	1	0	1
Summation	1	2	1	1	0	1

After you perform this summation for every user you're interested in, you end up with a matrix like that shown in table 11.5. The values represent each user's affinity to every item in the catalog.

**Table 11.5** Complete user-to-item affinity matrix

	Tunisia Sadie dress	Gordon Monk stiletto	Georgette blouse	Nikon digital camera	Canon 24 mm lens	Ravelli aluminum tripod
Allison	1	2	1	1	0	1
Brenda	1	1	2	0	0	0
Christina	2	1	1	1	0	1
David	0	0	2	4	3	3
Elise	1	1	2	3	3	3
Frank	0	0	1	2	2	3

Because you started with item purchases, it isn't usually meaningful to track user affinities toward items that they've already purchased. It also isn't meaningful to keep track of products that users have 0 affinity toward; why would you recommend users something that you don't think they care about? So let's remove these values and have another look at the remaining user-item affinity data, shown in table 11.6.

**Table 11.6** User-to-item affinity matrix with purchased items and zero-affinity items removed

	Tunisia Sadie dress	Gordon Monk stiletto	Georgette blouse	Nikon digital camera	Canon 24 mm lens	Ravelli aluminum tripod
Allison	-	2	-	1	-	1
Brenda	-	-	2	-	-	-
Christina	2	-	-	1	-	1
David	-	-	2	-	-	-
Elise	1	1	-	-	3	-
Frank	-	-	1	-	-	3

With the clutter removed, it's easy to see that collaborative filtering works well. Just as in the previous item-to-item case, this information can be used directly for personalized recommendations. *If only there was some way to incorporate this into your search application!* Don't worry; you'll get there soon.

Looking at the data, you can see that fashion shoppers have highest affinity toward fashion items, and that photography shoppers have highest affinity toward photography items. But because one of the users, Elise, has interests in both photography and fashion, crossover recommendations exist between fashion and photography. Because of this, David will probably be confused when he gets a recommendation for the

Georgette blouse. Fortunately, as the input data becomes richer (more items and more purchases per item), crossover recommendations such as this will become less prominent, and the user-item affinities will be dominated by the statistically significant co-occurrences.

Furthermore, in richer data sets, when unusual crossovers like this *do* exist, they're often fortuitous because they point out a latent relationship among the catalog items. For instance, about the only thing that Mentos has in common with Diet Coke is that they're both food (sort of). But toward the end of 2005, when the Mentos + Diet Coke experiment became viral on the internet, it became highly likely that these two items would show a spike in purchasing co-occurrence. This highlights the fact that collaborative filtering can identify connections that wouldn't be obvious by looking only at the textual content of the documents.

As alluded to earlier, finding affinities in this way is a fairly naïve approach. For example, you haven't normalized products that are extremely popular. Consider socks. No matter whether you're interested in fashion, photography, or any other field you can think of, you still regularly purchase socks. Therefore, the co-occurrence count between socks and every item in the index will be very large; everybody will be recommended socks. To resolve this issue, you'd need to divide the co-occurrence values by a notion of popularity for each pair of items.

Co-occurrence-based collaborative filtering isn't the only option for generating item-to-item or user-to-item affinities. If you're considering building your own recommendations, make sure to review the various matrix-based collaborative-filtering methods such as truncated singular-value decomposition, non-negative matrix factorization, and alternating least squares (made famous in the Netflix movie recommendation challenge). These methods are less intuitive than the simple co-occurrence counting method presented here, and they tend to be more challenging to implement. But they often provide better results, because they employ a more holistic understanding of item-user relationships. To dive deeper into recommendation systems, we recommend *Practical Recommender Systems* by Kim Falk (Manning, 2016). And no matter the method you choose, keep in mind that the end result is a model that lets you quickly find the item-to-item or user-to-item affinities. This understanding is important as we explain how collaborative filtering results can be used in the context of search.

### 11.2.3 Tying user behavior information back to the search index

In the previous section, we demonstrated how to build a simple recommendation system. But we're supposed to be talking about personalized search! In this section, we return to search and explain how the output of collaborative filtering can be used to build a more personalized search experience. We also point out some pitfalls to be aware of.

You can pull collaborative filtering information into search in several ways. The three strategies demonstrated here are related in that they take a standard, text-only

search and incorporate collaborative filtering as a multiplicative boost. Here's how it works: Consider an example base query in which you take the user's query "Summer Dress" and search across two fields, title and description, as shown in the following listing.

#### Listing 11.1 Base query

```
{ "query": {
  "multi_match": {
    "query": "summer dress",
    "fields": ["title^3", "description"]}}
```

Given this base query, you incorporate collaborative filtering by applying a multiplicative boost using a `function_score` query, as shown next.

#### Listing 11.2 A multiplicative boost can be used to incorporate collaborative filtering

```
{ "query": {
  "function_score": {
    "query": {
      "multi_match": {
        "query": "summer dress",
        "fields": ["title^3", "description"]}},
    "functions": [{
      "filter": { COLLAB_FILTER },
      "weight": 1.1}]}}
```

In this simple implementation, the documents that get the collaborative filtering boost are determined by the contents of your `COLLAB_FILTER` filter (which we discuss in a moment). Notice that this filter doesn't filter out any documents from the result set. Instead, documents matching this filter are given a multiplicative boost of 1.1, as indicated by the `weight` parameter. The query of listing 11.2 returns the same documents as the query of listing 11.1, but any documents also matching the `COLLAB_FILTER` get a 10% boost over the score of the base query. This subtly affects the ordering of the search results so that users making the query will see results that are more aligned with their previous behavior. *This is the goal of personalized search.*

#### QUERY-TIME PERSONALIZATION

With the basic structure in place, we can discuss the three strategies for incorporating collaborative filtering, each of which corresponds to a different `COLLAB_FILTER` and indexing strategy. For now, assume that the output of our collaborative filtering process is a set of user-to-item affinities—things like “Elise likes Tunisia Sadie dresses, Gordon Monk stilettos, and Canon 24 mm lenses.” But because we're talking to machines here, Elise is `user381634`, the Tunisia Sadie dress is `item4816`, the Gordon Monk stiletto is `item3326`, and the Canon 24 mm lens is `item9432`. Further, assume that you have similar information for all users and all the products in your catalog.

Given this data set, the most straightforward approach for incorporating collaborative filtering is as follows: Start by storing collaborative filtering data in a key-value store (outside the search engine). Let's say that Elise, `user381634`, has high-affinity items: `item4816`, `item3326`, and `item9432`. Now, next time Elise uses the search engine, the first step is to retrieve her high-affinity items from the data store, and then referring again to listing 11.2, replace `COLLAB_FILTER` with a filter to directly boost her high-affinity items by ID:

```
COLLAB_FILTER = {
  "terms": {
    "id": ["item4816", "item3326", "item9432"]
  }
}
```

Then any item matching Elise's high-affinity items will be driven further toward the top of the search results.

Although this is the most obvious approach, it might become computationally expensive at query time. In the preceding example, Elise has only three high-affinity items. In a more realistic implementation, a user could have hundreds or potentially thousands of high-affinity items. And at some point, having too many terms ORed together like this makes for slow queries. But you might be surprised with the extent to which this approach *will* scale. For instance, consider that Lucene does quite well with geo search. But as we discussed in chapter 4, under the hood geo search is implemented by ORing together many, possibly hundreds, of terms that represent a geographic area. Besides, in many personalized search applications, you won't need thousands of high-affinity items anyway; a user's searches will often tend toward a relatively small domain of interest. A few hundred high-affinity items in this domain will likely provide users with a noticeably personalized search experience.

#### INDEX-TIME PERSONALIZATION

If your application can't afford the performance hit at query time, the next approach places the burden on the index size. A benefit of this technique is that there's no need for an external key-value store, because you'll save collaborative filtering information directly to the index.

To do this, you add a new field to the documents being indexed named `users_who_might_like`. As the name indicates, this field contains a list of all users who might like a given item. For example, when you index the Gordon Monk stiletto, you include all the typical information that you need for search: title, description, price, and so forth. But this time you also include a `users_who_might_like` field, which is a list of all users showing a high affinity to this item. Referring to table 11.6, you can see that both Allison (`user121212`) and Elise (`user989898`) have a high affinity to this item. In this case, the `users_who_might_like` field will be the list `user121212, user989898`.

After all documents are indexed with their corresponding `users_who_might_like` field, the rest is easy. At query time, when Allison (`user121212`) makes a search, you issue her query along with a simple boosting filter:

```
COLLAB_FILTER = {
  "term": {
    "users_who_might_like": "user121212"
  }
}
```

Again, this returns the same results as the base query, but any document that includes Allison as a “user who might like” gets a 10% boost—pushing it toward the top of the search results. You can see that this query is much easier on the search engine at query time, because there’s only one extra term. But with this method, you must be watchful of the index size. As a frame of reference, in a modest Lucene index of 500,000 one-page English text documents, you might expect something like 1 million unique terms in the index. With this approach, each unique user ID represents another term in the index. So you should expect this approach to scale well for hundreds of thousands of users. But if you have millions of users, your index may outgrow its servers. Fortunately, this method scales well horizontally. You can create shards that represent a portion of your customers. If you have millions of users, you probably have the resources for this.

#### INDEX- AND QUERY-TIME PERSONALIZATION

Our final approach splits the difference between the two preceding approaches. Previously, you used user-to-item affinities, but for this last approach you’ll assume that the output of the collaborative filtering is a set of item-to-item affinities like those shown in table 11.3. In this new approach, the search engine calculates user-to-item affinity implicitly at the time of the query.

The setup for this approach is more involved than in the other approaches. You’ll again require a key-value store to look up user-related information. But this time rather than storing user-to-item affinities, you store the users’ most recent purchases. You also add a new field, `related_items`, to the index. As the name suggests, this field will contain a list of IDs for high-affinity items. At query time when Frank makes a search, you first pull his recent purchases—a Nikon digital camera (`item1234`) and a Canon 24 mm lens (`item9432`)—and then you issue his query along with the following boosting filter:

```
COLLAB_FILTER = {
  "terms": {
    "related_items": ["item1234", "item9432"]
  }
}
```

You query using a list of IDs just as in the first method, but it’s a much shorter list. And as in the second method, you’re required to index an extra field with a list of IDs, but



unless you have millions of items, this will also be much less information than in the previous method.

As mentioned at the opening of this section, the preceding methods are just a few of the many ways that collaborative filtering can be incorporated into search. And you can improve these methods in many ways. For instance, you probably noticed that we didn't mention the affinity values in any of these solutions. Instead we lumped items into two groups: high-affinity items (matching the `COLLAB_FILTER` filter) and lower-affinity items. You can modify these methods to take the individual affinity values into account, but this requires creativity involving payloads and scripting or possibly even a custom search-engine plugin.

### 11.3 Basic methods for building concept search

Personalized search is just one of the many possible directions to explore outside the more standard approaches presented in the previous chapters. Another interesting extension of search is *concept search*. Before reading this book, you probably thought of search as the process of finding documents that match user-supplied keywords and filters. Hopefully, you've come to realize that a good search application works to infer the user's intent and provide documents that carry the *information* that the user seeks. Concept search takes this notion to an extreme.

The goal of concept search is to augment a search application so that it in some sense *understands the meaning* of the user's query. And because of this understanding, the documents returned from a concept search may not match *any* of the user's search keywords, but will nevertheless contain meaningful information that the user is looking for. To borrow a phrase coined by Google, the goal of concept search is to allow users to "search for *things*, not strings."

Perhaps an example will help bring home the need for concept search. Consider a search application for medical journals. Using a typical string-based approach, a search for "Heart Attack" would fall short of the ideal. Why? Because medical literature uses various words for *heart attack*, such as *myocardial infarction*, *cardiac arrest*, *coronary thrombosis*, and many more. Plenty of articles about heart attacks won't mention *heart attack* at all. Concept search provides the user with an augmented search experience by bringing back documents that talk about heart attack even if they happen to not contain that specific phrase.

Still—and we can't emphasize this enough—a search engine at its core is a sophisticated token-matching and document-scoring system. The crux of concept search isn't magic; it involves augmenting queries and documents to take advantage of new relevance signals that increase search recall. By carefully balancing these new *concept* signals, you can ensure that search results retain a high level of precision. In this section, we cover several human-driven methods for augmenting your search application to take on a more *conceptual* nature.

### 11.3.1 Building concept signals

Initially, you may reach toward human-powered document tagging to implement concept search. You can create a field that will serve as a dumping ground for terms and phrases that answer the question “What is this document about?” This field will be the home of your new concept signal.

With this approach, when users of the medical journal application search for “Heart Attack” but miss an important article, you add the phrase *heart attack* to your concept field. Thereafter the document will be a match. This approach also helps fine-tune a document’s score. For instance, let’s say you have an important article about heart attacks. It may even contain the phrase *heart attack*. Unfortunately, it shows up on the second page of search results. Rather than attempt to solve the problem globally, add the phrase *heart attack* to the concept field (maybe even add it multiple times). This nudges the score for that document just a little bit higher whenever the user searches for “Heart Attack.”

But be forewarned that human curation can be challenging and resource consuming. Accurate tagging requires extensive domain expertise and rigorous consistency. For example, should the *heart attack* query also be tagged with *heart*? In the domain of your users, does *acute heart attack* differ from *heart attack*? Should a document receive both tags? Only trained, domain-aware content curators can make these fine-grained distinctions. Tagging also takes a lot of human effort. It may require deep reading of the content, which may not scale to cover realistic data sets.

One way to reduce the curation workload is by looking to your users as a possible way to crowd-source the concept signal. Do you recall the conversation about thrashing behavior in chapter 9? When thrashing, an unsatisfied search user quickly moves from one search to another, indicating that the search results don’t match their intent. Imagine that a user searches for “Myocardial Infarction,” spends about 20 seconds on the results page, and then makes a new search for “Cardiac Arrest.” It’s obvious that this user isn’t finding what they are looking for.

But often these users do finally find a relevant search result. Once there, it’s as if the user tells you, “Hey, remember all that other stuff I was searching for? *This* is what I meant!” Imagine that in our example, the user still doesn’t find anything interesting in the *cardiac arrest* results and submits a third query—this time for “Heart Attack.” Upon seeing the results, the user clicks the second document in the result set and then doesn’t return to search. This user implicitly tells us that the phrases *heart attack*, *cardiac arrest*, and *myocardial infarction* are somehow related. Therefore, why not take the thrashing search terms and add them to the concept field for the document that finally satisfies the user’s information need? This way, the next time someone follows the same path as our thrashing user, they will more likely find what they need in their first set of search results.

Again, the main goal of the new concept field is to increase search recall. But you should be careful to not ignore the impact made upon precision. In the preceding example, if the user initially searches for “Cardiac Arrest” but then changes to “Gall

Bladder,” your concept signal may become noisy. Make sure to properly balance the concept signal with the existing signals. If the concept field is human curated, it’s more likely to be high quality than a user-generated field and should be more strongly represented in the relevance score.

### 11.3.2 Augmenting content with synonyms

Synonym analysis is another useful way to inject deeper conceptual understanding into search. The first time you open the synonym file and add an entry such as the following, you’re building out concept search:

```
TV, T.V., television
```

When the user asks the search application for a “TV,” the search application answers back, “I have documents with TV—but I bet you’re also interested in these other documents that have words like T.V. and television, right?”

Initially, synonym augmentation of the documents takes place somewhat manually. Content curators may hand-generate an extensive synonym list customized to the jargon of a field. In larger domains—medicine is again a good example—it may be possible to repurpose a publically available taxonomy such as the Medical Subject Headings (MeSH) for use during synonym analysis.

One thing to think about when using synonyms is whether to encode hierarchical structuring with the synonyms. For instance, in the preceding simple case with *television*, all the synonym entries are semantically on the same level; they truly are synonyms. But, as discussed in chapter 4, it’s common to use synonyms to encode a notion of specificity into the indexed terms. For instance, consider the following synonym entries:

```
marigold => yellow, bright_color  
canary => yellow, bright_color  
yellow => bright_color
```

This encodes a hierarchy for yellow things (and you can imagine what a much larger synonym set for all colors would look like). When used correctly, synonyms like this will serve to expand a user’s narrow query, for instance “Canary,” to a broader notion: yellow. This improves recall, allowing the user to find items that use more-general terminology.

As always, recall and precision must be balanced. Fortunately, the natural  $TF \times IDF$  scoring works in this case. Namely, after synonym analysis has been applied, specific terms such as *marigold* will occur much less often in the index than terms like *yellow* and *bright\_color*. Therefore, when a user searches for a specific term such as “Marigold,” both *marigold* and *yellow* documents will be returned, but *marigold* documents will be scored above the more general *yellow* documents.

As a final note, synonym augmentation and concept fields are complementary approaches. Synonym analysis usually dumps the synonyms back in the same field as the source text that was analyzed. But if you’re concerned that your synonyms are

noisy, it might be a good idea to stick them into a separate field so that they can be given a lower weight. Another good combination of concept fields and synonym augmentation occurs in document tagging. In this scenario, you can greatly reduce the burden placed on the people doing the tagging by having them apply only the most specific tags. Then hierarchically structured synonyms can be used to automatically augment the documents with less specific tags.

## 11.4 Building concept search using machine learning

In the earlier sections, we introduced personalized search using simple methods and then moved on to more sophisticated, machine-learning approaches. Here we follow this same route, moving from human-powered concept management to a machine-learning approach that we call *content augmentation*.

Just as before, the goal is to include a new content signal into the documents to improve search recall. In particular, you'll use machine learning to automatically generate *pseudo-content* to be added back into the indexed document. This content won't be new paragraphs of human-readable text. Rather, the pseudo-content will be a dump of terms that aren't in the original document but that, in some statistically justifiable sense, should be present because they pertain to the concepts in that document.

To generate the new pseudo-content, you algorithmically model the statistical relationship between words based on the documents that contain them. For instance, consider a medical journal article that contains the word *cardiac*. There's a high probability that the same article will also contain words like *heart*, *arrest*, *surgery*, and *circulatory*; these words are related to the *cardiac* topic. But it's unlikely that the same article will contain the words *clown*, *banana*, *pajamas*, and *Spock*; these words have little in common with the *cardiac* topic. By looking at the co-occurrence of words, you can begin to understand how they're interrelated. And once you have a good model of these relationships, you can take any given document and generate a set of words that in some sense should be in that document.

Let's look at an extremely simplified example. Consider the small set of documents displayed in listing 11.3. Each document is a sentence ostensibly about dogs or cats. If you put these documents through analysis, you can split out the tokens, normalize them (by lowercasing and stemming), and filter out the common stop words. The end result can be represented in matrix form, as shown in table 11.7. Here a dot (•) indicates that the term (represented in that column) has occurred one or more times in the document (represented in that row).

### Listing 11.3 Simple documents illustrating term co-occurrence

```
doc1: The dog is happy.  
doc2: A friendly dog is a happy dog.  
doc3: He is a dog.  
doc4: Cats are sly.  
doc5: Fluffy cats are friendly.  
doc6: The sly cat is sly.
```

**Table 11.7** Matrix representation of the words and the documents that contain them. (Stop words have been removed and plural words have been stemmed. Additionally, the columns are arranged to draw attention to statistically clustered words.)

	dog	happy	friendly	cat	fluffy	sly
doc1	•	•				
doc2	•	•	•			
doc3	•					
doc4				•		•
doc5			•	•	•	
doc6				•		•

You may notice that this term-document matrix resembles table 11.2’s user-items matrix. This is no coincidence. The problem of identifying related items based on user interactions is nearly identical to the problem of identifying related terms based on document co-occurrence. Whereas the earlier personalization example revealed natural clustering among fashion items and photography items, table 11.7 reveals a clustering among dog terms and another clustering among cat terms. In principle, you could even use the same co-occurrence counting method to identify closely related terms. But in practice, more-sophisticated methods are used such as latent semantic analysis, latent Dirichlet allocation, or the recently popular Word2vec algorithm. Though these methods are well beyond the scope of this book, the models generated from these algorithms allow you to “recommend” pseudo-content for documents in much the same way that section 11.2.2 showed how to recommend products based on user-to-item and item-to-item affinity.

After automatically generated pseudo-content is indexed with each document, queries can match documents that didn’t originally contain the user’s search terms. Nevertheless, these documents *will* be strongly associated with the keywords based on statistical term co-occurrence. In the preceding example, a search for “Cat” may return a document that talks about a sly fluffy animal even if that document doesn’t contain the word *cat*.

#### 11.4.1 The importance of phrases in concept search

We haven’t discussed an important component of concept search: phrases. Often phrases carry meaning that’s more specific than the terms that compose the phrase. Case in point: *software developer*. A software developer isn’t software, but a person who develops software. Additionally, there are many types of developers, and a *land developer*; for example, has nothing to do with software development.

Therefore, prior to *content augmentation*, it’s useful to first identify statistically significant phrases within the text. These phrases can be added to the columns of the

term-document matrix so that during content augmentation these conceptually precise phrases will be included in the newly generated content.

*Collocation extraction* is a common technique for identifying statistically significant phrases. The text of the documents is split into n-grams (commonly bigrams). Then statistical analysis is used to determine which n-grams occur frequently enough to be considered statistically significant. As is often the case, this analysis is a glorified counting algorithm. For instance, if your document set contains 1,000 occurrences of the term *developer*, and if 25% of the time the term *developer* is preceded by the term *software*, the bigram *software developer* should probably be marked as a significant phrase.

### 11.5 The personalized search—concept search connection

As we earlier indicated, a strong relationship exists between personalization and concept search. Both rely on crafting new signals to improve precision and recall. Both use similar machine-learning approaches. But the relationship goes even deeper because these methods can be used *together* to improve relevance even further.

Consider the *cold-start* problem. Let's say that you're trying to build personalized search based on collaborative filtering. What happens when a new item is introduced to your catalog? Recall that collaborative filtering methods depend on user interactions. Because no one has ever interacted with the new item, *no personalization* happens. You can't recommend a new item based on behavioral patterns that don't exist; this is known as the *cold-start problem*.

This begs an important question, though. You do at least have *some* information for the item: its textual content. Can this be used to generate personalized recommendations? Yes! To do this, you incorporate aspects of concept search into your personalization strategy. With concept search, you augment documents with a broader, conceptual understanding of the text. When pulling concept search into personalization, you must instead augment your user profiles to track the *concepts* that they're interested in. And in this case, just as in the preceding section, *concepts* are the important words and phrases that a user has shown high affinity toward.

There are plenty of ways to determine content that holds high affinity to your users. One way is to turn back to machine learning and somehow infer user-to-term affinities based on the user's interactions with documents and the text of those documents. But there's no need to get overly sophisticated; users are constantly feeding us high-affinity terminology in the searches that they make. And if you're fortunate enough to have highly engaged users, you may even be able to directly ask what types of content they're interested in.

Now, reversing the perspective, consider how the behavioral information used with collaborative filtering can also be used to augment concept search. In section 11.2, you saw how collaborative filtering could establish a relationship among camera equipment and a different relationship among fashion items. These relationships were established based solely on user behavior; the items' textual content played no role.

Nevertheless, as this example demonstrates, behaviorally related items are often conceptually related as well. Therefore, when the textual content associated with an item is weak, behavioral information can be employed to help users find what they're looking for.

## 11.6 Recommendation as a generalization of search

Throughout this book, we've covered the ins and outs of search. We've pulled open the search engine, explained the inner workings, and built techniques for producing a highly relevant search application. We discussed business concerns, describing how to shape a culture that makes search relevance a central issue. In this chapter, we've pointed to ways to imbue search with an almost spooky ability to understand the user's intent.

But here, at the end of this last chapter, we expose a new challenge to everything we've written to this point:

Maybe *search* is not the application you should be building. Maybe you should be building *recommendations*.

Consider what happens if there's no explicit search in a personalized, concept-based search application. What if the search box is left empty and filters left unchecked? Can the application still be put to use? Yes! Even without immediate input from the user, the application has a significant amount of context that can be used to make rich recommendations. For instance, if the user looks at an item detail page, then the application can use methods discussed in this chapter to recommend related items. If the user interacted with the application in the past, the recommendations can incorporate the user's behavioral and conceptual information so that the recommendations will be personalized. So you see, *recommendation* is something that can still exist without an explicit query from the user. As we'll show in the following paragraphs, it may even be useful to think of search as a subset of recommendation.

Let's dig farther. Think about the analogues to search and recommendation that exist in real life. When viewed in its worst possible light, a basic search application can be like a gum-chewing, disinterested, teenage store clerk. You say, "I need a shirt," and the clerk points to a wall of shirts. There are hundreds of shirts—a mixture of all styles, sizes, and prices imaginable. It's too much to process, so you attempt to filter the search: "Yeah, but do you have anything in size M?" The clerk (still smacking gum) glances up and points to the bottom rack. There are still a lot of shirts to choose from—a mixture of various styles and prices—but you need a new shirt, so you walk over and start looking through the shirts in your size.

Even though we've couched this story in terms of a search for a new shirt, the store clerk is effectively making recommendations to the customer. They're just not particularly good recommendations. The clerk ignores the other personalization and conceptual context clues that could help direct customers to just the item they're looking for.



Continuing with the analogy, let's replace the gum-chewing store clerk with your own personal fashion consultant. This time you walk into the store and say to the fashion consultant, "I need a shirt," and the fashion consultant takes you directly to the shelf with shirts that match your size. The fashion consultant is a well-studied expert, keenly aware of the types of clothing in style and how to pair clothing items to make a good outfit. She's also keenly aware of your personal style. The fashion consultant busies herself looking through the rack, pulling out the shirts that are a good match, and arranging them for you to look through yourself. Then she looks up and asks, "Oh, what price range are we looking in today?"—extra context. Upon hearing your response, she plucks out a couple of the overpriced shirts and hangs them back on the rack. Finally, she helps you look through the remaining items.

Now you're getting somewhere. The attentive and highly educated fashion consultant doesn't leave you wandering aimlessly to search for a shirt by yourself, but works with you to provide recommendations that take into account information about both you and the fashion domain. And you know what? After helping you pick out that shirt, the fashion consultant takes you over to the hat rack beside the register and says, "Check out this hat. This is a perfect match for you and would look great when you're wearing that new shirt." And she's right; it's a cool hat! This is the epitome of recommendation, because, even without making an explicit search, the fashion consultant is ready to provide feedback based on whatever information is at hand.

### 11.6.1 *Replacing search with recommendation*

As the preceding story illustrates, *recommendation* could be seen as an overarching and unifying concept—which happens to include the notion of search. Here's a formal definition:

*Recommendation* is the ability to provide users with the best items available based on the best information at hand.

The most interesting part of this definition is the word *information*. Here, information comes in three flavors: information about the users, about the items in the catalog, and about the current context of recommendation:

- *User information*—As users interact with the application, you can identify patterns in their behavior and learn about their interests and tastes. Particularly engaged users might even be willing to directly tell us about their interests.
- *Item information*—To make good recommendations, it's important to be familiar with the items in the catalog. At a minimum, the items need to have useful textual content to match on. Items also need good metadata for boosting and filtering. In more advanced recommendation systems, you should also take advantage of the overall user behavior that gives you new information about how items in the catalog are interrelated.



- *Recommendation context*—To provide users with the best recommendations possible, you must consider their current context. Are they looking at an item details page? Then you should make recommendations for related items in case they aren't sold on this one. Is the user getting ready to check out? Then let's recommend popular, low-cost items. Are you sending out an email newsletter? Then let's show the users some highly personalized recommendations and see if you can bring them back on the site.

You might notice that search is barely mentioned in this discussion. What gives? Is it just ... gone? Quite the opposite! Search is still present; it's just another one of the possible contexts for recommendation. And as a matter of fact, search is the most important context, because it represents users telling you exactly what they're looking for *right now*. When a user makes a search, you have access to the richest information possible and should therefore be able to make better-informed recommendations. To pick back up with the fashion consultant example, search is the point where you tell the consultant, "You know what? Today I'm looking for an Hawaiian shirt." And the consultant *recommends* a shirt that not only matches the current search context (it's Hawaiian), but also matches your established personal preferences.

## 11.7 Best wishes on your search relevance journey

We've finally come to the close of this book. Before you leave, consider how far you've come:

- *Chapter 1* helped familiarize you with the problem space and the challenges that you'll likely encounter as you work to improve your own search relevance issues.
- *Chapter 2* laid the foundation for technical discussions in the book by explaining how search technology works—inside and out.
- *Chapter 3* introduced debugging tools useful for isolating a wide range of relevance problems.
- *Chapter 4* described how text is processed in order to extract the features most useful in search.
- *Chapters 5 and 6* discussed how textual features are used to build higher-level relevance signals and the various ways that these signals can be combined.
- *Chapter 7* explained how functions and boosting are used to further tune relevance and to shape search results to achieve business goals.
- *Chapter 8* revealed that relevance is more than just tuning parameters; it's also about helping users understand and refine the information being made available to them.
- *Chapter 9* provided an end-to-end relevance case study that combines the lessons of the previous chapters and outlines a systematic approach to designing relevant search applications.

- *Chapter 10* described how to shift the organizational culture to focus on relevance.
- *Chapter 11*, this chapter, broadened the horizons of search to include personalized search, concept search, and recommendations.

After covering all these topics, we're sure that you'll find—and are probably already finding—that each search application has its own set of relevance challenges. But after reading through this book, you should find yourself much better equipped to meet and overcome these challenges.

## 11.8 Summary

- Personalized search provides a customized experience for individual users and allows high-affinity items to be boosted toward the top of search results.
- Build basic search personalization by creating user profiles that track preference and demographic information. Make sure that this information is represented in the search index.
- Use collaborative filtering to create personalized search based on the user behavior.
- Concept search provides users not only with documents that match their search terms but also with documents that match the *meaning* of their search.
- Concept search requires adding new content to the index or to the users' queries in order to improve recall. It's important to carefully balance the new concept signals in order to retain precision.
- Use personalized search and concept search together to complement and augment one another.
- Personalized and concept search without an explicit user query is effectively the same thing as a recommendation.
- Consider recommendation as a generalization of search.

# *appendix A*

## *Indexing directly from TMDB*

---

Starting in chapter 3, we used examples from TMDB, The Movie Database. We pre-packaged a `tmdb.json` file for ease of use with the examples. But we're strong believers in showing your work. In this appendix, we show you how to use the examples with an up-to-date version of TMDB's data.

We warn you that we have no control over TMDB, so TMDB may make changes to its API or policies that over time make this appendix obsolete. That being said, we also want to add that we're grateful to TMDB for permission to use its data for this book. We encourage you to visit TMDB (<http://themoviedb.org>) and make a donation for all their hard work!

This appendix walks you through the following steps:

- 1 Obtaining an API key from TMDB
- 2 Setting up Python code to talk to TMDB
- 3 Pulling back a list of movies, by crawling TMDB's top-rated movies
- 4 Pulling back extended data on each movie

You're going to set up a process that crawls TMDB for movie IDs and then visits several API endpoints such as `/movie/<id>` and `/movie/<id>/cast` to extract extended details for each movie. The code for this can be found in the appendix A IPython notebook examples in the GitHub repo (<http://github.com/o19s/relevant-search>).

### **A.1 *Setting the TMDB key and loading the IPython notebook***

The first steps are to obtain an API key from TMDB. This key gives you authorization to use the API. You pass the key to TMDB to identify yourself when accessing

the API. Follow the instructions at this website to get an API key: [www.themoviedb.org/documentation/api](http://www.themoviedb.org/documentation/api).

Armed with your API key, open up whatever command prompt you plan to use for these Python examples. Before accessing TMDB using the Python examples in this appendix, you'll need to set the environment variable `TMDB_API_KEY`.

In a Mac/Linux system, set this with the following:

```
export TMDB_API_KEY=<API KEY from above>
```

In Windows, type this at the command prompt:

```
set TMDB_API_KEY=<API KEY from above>
```

## A.2 **Setting up for the TMDB API**

Next, you'll perform the setup you need to interact with TMDB. You'll install a library, import a few system libraries into your Python code, and set up Python to talk over HTTP.

The only additional Python library used here is `requests`. Be sure to install the `requests` library using Python's package installer, `pip`. If you don't have Python's package installer, you should still have Python's built-in `easy_install` utility:

```
easy_install pip  
pip install requests
```

With an API key and the right libraries, you're ready to begin pulling movies down from TMDB. But first, let's make sure you have all the setup code needed to support the functions that interact with the API. This code, shown in the following listing, imports the needed Python modules, fetches the `TMDB_API_KEY` environment variable, and creates a dedicated HTTP session for communicating with TMDB's API.

### Listing A.1 How boilerplate is done

```
import requests  
import json  
import os  
import time  
  
tmdb_api_key = os.environ["TMDB_API_KEY"]  
  
tmdb_api = requests.Session()  
tmdb_api.params={'api_key': tmdb_api_key}
```

← Be sure you installed the Python "requests" library

← Reads the `TMDB_API_KEY` environment variable

← An HTTP session you'll use to interact with TMDB

The most important part of this code is the last two lines. Here you ask the Python `requests` library to create a session. The returned session is configured with an `api_key` argument, using your `tmdb_api_key`. Now that you have boilerplate code out of the way, let's get cracking on using the TMDB API directly.

### A.3 Crawling the TMDB API

Next, you'll get to the fun work of extracting movies from TMDB! As we stated, this requires two components:

- Code to pull back a list of movie IDs
- Code that, for each movie ID, fetches additional details

First, you'll construct the `movieList` function used to pull back a list of movie identifiers. The TMDB API is organized around a series of endpoints that list movies, such as `/movies/popular` (list of movies by popularity) or `/movies/top_rated/` (list of movies that are top-rated). As you access each endpoint, you'll have an outer loop for passing a page URL query parameter. Because each request returns 20 movies at a time, you'll need to continue to request additional pages to have enough movies to search.

To start, you'll pull down movies you'd like to place in the search engine in the `movieList` function. To do this, the function snags the IDs of popular movies by paging through `top_rated`, collecting each ID in a `movieIds` list. Then, after each ID is collected, `movieList` returns a list of TMDB movie IDs.

**Listing A.2** Crawling movies from TMDB—`movieList`

```
def movieList(maxMovies=10000):
    url = 'https://api.themoviedb.org/3/movie/top_rated'
    movieIds = []
    numPages = maxMovies / 20
    for page in range(1, numPages + 1):
        httpResp = tmdb_api.get(url, params={'page': page})
        try:
            if int(httpResp.headers['x-ratelimit-remaining']) < 10:
                time.sleep(3)
        except Exception as e:
            print e
        jsonResponse = json.loads(httpResp.text)
        movies = jsonResponse['results']
        for movie in movies:
            movieIds.append(movie['id'])
    return movieIds
```

**Accesses the top\_rated endpoint for this page** ①

**Parses the JSON response** ②

**For each page of top\_rated movies ...**

**... throttles the access to the API (TMDB API throttling rules subject to change) ...**

**... iterates the page of movies, storing the movie's ID ...**

**... returns the accumulated movie IDs.**

In `movieList`, you issue a GET request to the TMDB API ①. Next, you parse the HTTP text body, loading the JSON body into a Python dictionary ②. The entry `results` contains the meat of the response, holding each movie for you to work with. You needn't concern yourself much about the structure of the response, as you simply extract the ID of each movie returned, append it to a list, and discard the rest. This process is repeated as you page through additional responses from TMDB.

Now to pull back extended information on each movie. In the book we used the `extract` function to pull movies out of the `tmdb.json` file. In this version of `extract`,

in listing A.3, you pass in the list of movie IDs returned from `movieList`. The function `extract` pulls back even deeper information about movies, by accessing each movie's detailed information individually. This function also accesses additional details needed in chapter 5 and later regarding the cast and crew. Finally, `extract` returns the accumulated details in the form of a dictionary that maps a movie ID to movie details.

We show this code in reverse order for context. First, `extract` pulls back movies one at a time, calling a `getCastAndCrew` function. Farther down, you see `getCastAndCrew` implemented by accessing each movie's `/credits` endpoint.

### Listing A.3 Extracting each movie from TMDB—`extract`

```
def extract(movieIds=[], numMovies=10000):
    movieDict = {}
    for idx, movieId in enumerate(movieIds):
        try:
            httpResp = tmdb_api.get("https://api.themoviedb.org/3/movie/%s"
                                     % movieId, verify=False)
            if int(httpResp.headers['x-ratelimit-remaining']) < 10:
                time.sleep(6)
            movie = json.loads(httpResp.text)
            getCastAndCrew(movieId, movie)
            movieDict[movieId] = movie
        except ConnectionError as e:
            print e
    return movieDict
```

**... accesses the movie/<id> endpoint for additional details**

**For each movie ID in the movies ...**

**... throttles the access to the API (Note: TMDB API throttling rules subject to change)**

**... parses the JSON response, adds entry to movie dictionary**

**... enriches movie with additional cast & crew information**

**... returns a movie dictionary.**

In the next listing, we show you how cast and crew information is accessed. This function accesses TMDB's `/movie/<movieId>/credits` endpoint and adds information about directors and cast details to the movie record.

### Listing A.4 Get cast and crew

```
def getCastAndCrew(movieId, movie):
    httpResp = tmdb_api.get("https://api.themoviedb.org/3/movie/%s/credits"
                             % movieId)
    credits = json.loads(httpResp.text)
    crew = credits['crew']
    directors = []
    for crewMember in crew:
        if crewMember['job'] == 'Director':
            directors.append(crewMember)
    movie['cast'] = credits['cast']
    movie['directors'] = directors
```

**Accesses the TMDB credits endpoint**

**Parses the credits JSON response**

**For each crew member, pulls out the directory and adds the record to the director list**

**Saves the cast and directors to the movie**

## A.4 Indexing TMDb movies to Elasticsearch

With all the pieces in place, you can use the `reindex` function from chapter 3 and index all the movies into Elasticsearch. Recall that `reindex` presumes you have Elasticsearch running at `http://localhost:9200`, the default install location of Elasticsearch. But you can also visit the book's GitHub repo (<http://github.com/o19s/relevant-search-book>) for other options for running Elasticsearch.

The following listing restates the `reindex` function for completeness. There's no need to get into too many details here. The big point to remember is that this function deletes and re-creates the index with the provided analyzer and mapping settings.

**Listing A.5** `reindex` function

```
def reindex(analysisSettings={}, mappingSettings={}, movieDict={}):
    settings = {
        "settings": {
            "number_of_shards": 1,
            "index": {
                "analysis" : analysisSettings,
            }}}
    if mappingSettings:
        settings['mappings'] = mappingSettings

    resp = requests.delete("http://localhost:9200/tmdb")
    resp = requests.put("http://localhost:9200/tmdb",
                        data=json.dumps(settings))

    bulkMovies = ""
    for id, movie in movieDict.iteritems():
        addCmd = {"index": {"_index": "tmdb",
                            "_type": "movie",
                            "_id": movie["id"]}}
        bulkMovies += json.dumps(addCmd) + "\n" + json.dumps(movie) + "\n"
    resp = requests.post("http://localhost:9200/_bulk", data=bulkMovies)
```

**Default settings**

**Uses the provided analysis and mapping settings**

**Deletes and re-creates the tmdb index**

**Bulk-indexes the provided movies**

With that function in place, you can pass the `MovieDict` into `reindex`, as shown in the following listing.

**Listing A.6** Index to Elasticsearch

```
movieIds = movieList()
movieDict = extract(movieIds)
reindex(movieDict=movieDict)
```

Further, should you want to overwrite the `tmdb.json` file, encode the `movieDict` as JSON and save it to a file, as shown in the next listing.

**Listing A.7 Create tmdb.json**

```
with open('tmdb.json', 'w') as f:
    f.write(json.dumps(movieDict))
    f.close()
```

That's it! `tmdb.json` is a direct reflection of the source data model from TMDB. It holds the contents of the `/movies/<id>` endpoint enriched with content taken directly from `/movies/<id>/credits`.



## appendix B

### *Solr reader's companion*

---

Welcome, Solr reader! The lessons of *Relevant Search* apply to your work as well. As we noted earlier, both Solr and Elasticsearch provide a friendly interface on top of the underlying core Lucene search library. In this appendix, we point out how your search engine's features fit into the discussion by highlighting, chapter by chapter, where you'll find comparable functionality in Solr. We also point out some of the pros and cons of both search engines when it comes to relevance.

To be clear: our goal is to provide scaffolding to your learning efforts. We don't reimplement the examples one for one. This book is too general for one search engine. Instead we want to provide, as much as possible, a *mapping* of Solr *features* into the book's larger relevance discussion. This discussion assumes that you have general knowledge of Solr and how it's configured. If you hit a topic in this appendix that you'd like to learn more about, we invite you to examine Solr's official reference guide (<https://cwiki.apache.org/confluence/display/solr/Apache+Solr+Reference+Guide>), the *Solr Start* online series ([www.solr-start.com](http://www.solr-start.com)), or great books like Trey Grainger's *Solr in Action* (Manning, 2014).

To structure our discussion and help you follow along with the book, we roughly subdivide this appendix by chapter. Luckily, this tends to correspond to functional components of the search engine. We cover chapters 4–8, as these chapters focus on hands-on interaction with the search engine. We omit chapter 3, as we included some footnotes in that chapter to get you started. We also omit chapter 9, because it uses Elasticsearch features introduced in previous chapters. If you'd prefer to avoid the detailed discussion, you can also get a rough mapping from the table included in each section.

So let's get started turning your Solr search engine into a relevant search powerhouse!

## B.1 Chapter 4: taming Solr's terms

Chapter 4 focuses on analyzers. Let's see how Solr allows you to configure the analyzers found in this chapter. Analyzers translate text like

```
"the doctor's brown fox"
```

into stemmed tokens without stop words:

```
[doctor] [brown] [fox]
```

The big takeaway from chapter 4 is that these tokens model features of your data. Controlling this process is fundamental for managing your relevance. The same lessons hold 100% for Solr; only the implementation nuts-and-bolts differ. In this section, first you'll see how to build a custom analyzer in Solr. Next, you'll see how to map an analyzer to a specific field in an incoming Solr document.

### B.1.1 Summary of Solr analysis and mapping features

Table B.1 maps general analysis and mapping features to their Solr counterparts.

**Table B.1** Solr analysis features

Chapter	Feature	Solr analogue
4	Custom analyzers	Implemented in schema via custom <code>fieldTypes</code>
4	Field mappings (mapping each field to an analyzer)	Done by creating a field of a custom <code>fieldType</code>

### B.1.2 Building custom analyzers in Solr

In chapter 4, you saw how Elasticsearch creates analyzers by using JSON. These JSON settings are part of the configuration for a particular index. Typically, when creating an index, you specify the custom analyzers and mappings to be used in Elasticsearch:

```
{
  "settings": {
    "analysis": {
      "analyzer": {
        "standard_clone": {
          "tokenizer": "standard",
          "filter": [
            "standard",
            "lowercase",
            "stop" ] ] ] ] }
```

Luckily for you, Solr reader, the code in this chapter can be easily translated from Elasticsearch to Solr. You need to understand just one key difference: instead of JSON over HTTP, Solr configures analyzers by using XML within Solr's `schema.xml` configuration file.

Solr allows you to define custom `fieldTypes` that control analysis—for example:

```
<fieldType name="text_standard_clone" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StandardFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" />
  </analyzer>
</fieldType>
```

Beginning to look familiar? This analyzer block corresponds to the analyzer configured previously in Elasticsearch. Once you see the correspondence, you can easily implement chapter 4's examples in Solr. In the preceding snippet, a single `<tokenizer .../>` configures this analyzer with a standard tokenizer (`class="solr.StandardTokenizerFactory"`). This corresponds to the `"tokenizer":` JSON element for the previous Elasticsearch specification. Similarly the `<filter .../>` lines specify a sequence of token filters to run, just as Elasticsearch's analyzer has a `filter` element followed by a JSON list.

Unlike Elasticsearch, in which you'd configure the stopwords filter elsewhere with a stop-words list or file, Solr places the configuration options on the filter itself (notice `words="stopwords.txt"`).

Elasticsearch provides fine-grained control of analysis, allowing you to specify analyzers even at query time. Solr works a bit differently, giving you the capability to customize only at the `fieldType` level. The `fieldType` allows you to define whether the analyzer applies to index or query time. So to analyze differently when querying, you'd use the following structure:

```
<fieldType name="text_standard_clone" class="solr.TextField">
  <analyzer type="query">
    ...<!-- query-time analysis -->
  </analyzer>
  <analyzer type="index">
    ...<!-- index-time analysis -->
  </analyzer>
</fieldType>
```

Great! You have a custom field type with a custom analysis chain. Next you'll see how to associate it with a specific field.

### B.1.3 Using field mappings in Solr

How does Solr associate a field such as `title` with a custom analyzer? In Elasticsearch, analyzers are assigned to fields via mappings. Field mappings are created using JSON syntax such as this:

```
"mappings": {  
  "movie": {  
    "properties": {  
      "title": {  
        "analyzer": "standard_clone"  
      }  
    }  
  }  
}
```

Solr differs slightly, not radically. In Solr, you declare a field in your schema, specifying the `fieldType` to be used with the `type` attribute. Here a field `title` is of type `text_standard_clone`:

```
<field name="title" type="text_standard_clone"/>
```

The search engine will transform text for documents containing a `title` field by using the `text_standard_clone` index analyzer. A search against the `title` field will use `text_standard_clone`'s query analyzer to break apart the query terms.

As you see, with some differences, chapter 4's examples apply straight to Solr. The hands-on work differs, and you'll find the various knobs and dials that control settings in different places. Nevertheless, now you can get to work modeling text as terms with your tools!

## B.2 Chapters 5 and 6: multifield search in Solr

Chapter 5 begins your journey into *signals*. Solr implementations need signals too. Signals map query-time relevance scores to factors meaningful to users. Is the query looking for a title? Is it mentioned prominently in text? Fields become a unit for measuring these factors. In chapter 5, you begin controlling precisely how each field is constructed to measure signals. Chapter 6 continues this discussion by introducing the idea of term-centric search, which considers documents more relevant when more query terms match.

### B.2.1 Summary of query feature mappings

Table B.2 maps the Elasticsearch query strategies to those of Solr.

**Table B.2 Solr query features**

Chapter	Query strategy	Solr analogue
5	best_fields	edismax with tie=0.0 close, but not identical
5	most_fields	edismax with tie=1.0
6	Elasticsearch query_string	edismax query parser
6	Custom all fields using copy_to	Using copyField in your schema to build a large all field
6	cross_fields	No analogue or close feature, but can be implemented in a custom query parser using Lucene's BlendedTermQuery

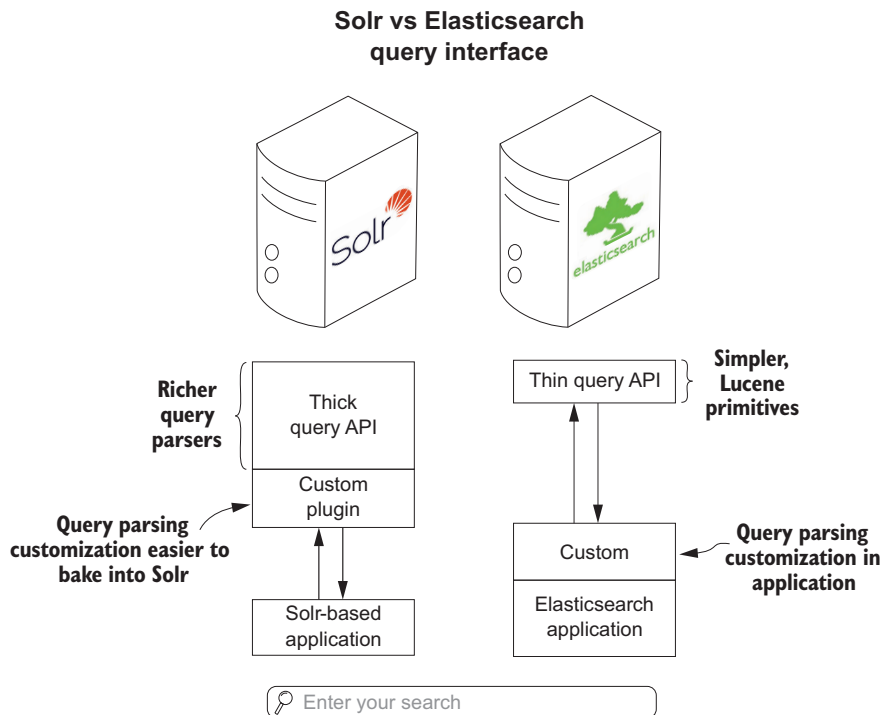
### B.2.2 Understanding query differences between Solr and Elasticsearch

In this overview of query functionality, you'll notice we've combined chapters 5 and 6. This is because a Solr-centric discussion would be organized quite differently across these two chapters. Why?

- The Solr out-of-the-box experience doesn't start on field-centric search, but Elasticsearch does.
- Solr starts you out with term-centric search, using its edismax query parser. Query parsers are less central to Elasticsearch.
- Solr doesn't have cross-field search, but you could easily add it with a query parser.

What accounts for these differences? Solr is a thick server querying system; it does quite a bit for you. Elasticsearch, more of a thin server, layers on relatively little. Instead, Elasticsearch exposes an API closer to the raw Lucene query API. Figure B.1 illustrates this difference.

What do we mean by a thick versus thin query API? When querying, Solr adds a lot of capabilities in its extensive library of meaty query parsers. A query parser lets you interact with search by using its own custom query syntax. It then translates this into underlying Lucene queries such as Boolean queries, term queries, or dismax. These query parsers abstract many of the Lucene details from you. Elasticsearch's strategy is quite the opposite. Elasticsearch works to give you primitives closer to the underlying Lucene queries for you to construct larger, more complex queries outside the search engine. Elasticsearch's query API does relatively little for you. The exceptions to this don't add a lot of magic, and translate one or two basic settings to Lucene queries (for example, `multi_match`). Only `query_string` stands out as a true query parser as Solr would define it.



**Figure B.1** Query customization in Solr and Elasticsearch. Solr lets developers push more customization into its thicker query API. Elasticsearch exposes simpler primitives to work with.

In many ways, this makes Elasticsearch a better choice for teaching you about Lucene search: its API is closer to the underlying Lucene queries. Solr, while often giving you some powerful query parsers, obscures the nuts and bolts. Solr, however, makes it easy to create *your own* query parsers by using underlying Lucene code. For a Solr developer, this would be one of the first plugins you'd implement. There's also a thriving ecosystem of query parsers implementing functionality such as graph search or complex phrase search. For Elasticsearch developers, however, a custom query parser would be one of the last things you'd try to do. The Elasticsearch developer would be more likely to attempt to implement comparable functionality outside the search engine itself by using the query API.

With that context in place, you can begin to see where Solr fits into our discussions in chapters 5 and 6.

### **B.2.3** *Querying Solr: the ergonomics*

As a Solr developer, you know that the ergonomics of querying Solr differ quite a bit from Elasticsearch. The most common way to interact with Solr is through URL

parameters over HTTP. You saw this in chapter 3, when we pointed out how to search Solr. Solr queries come in this form:

```
http://solr.quepid.com/solr/tmdb/select?q=sea biscuit likes to fish!
```

By controlling parameters passed to the URL, many of which depend on knowing the current query parser, you control how the query string is translated into underlying Lucene queries. You control the query parser with the `defType` parameter. Here, enabling the `edismax` query parser also lets you pass in the fields to be searched via the `qf` parameter:

```
http://solr.quepid.com/solr/tmdb/select?q=sea biscuit likes to fish!&defType=edismax&qf=title overview
```

Solr also has a fairly unique syntax called *local params*. This syntax lets you scope parameters to a part of the query string (hence the name *local*). The preceding `edismax` query could be rewritten in local params syntax as follows:

```
http://solr.quepid.com/solr/tmdb/select?q={!defType=edismax qf='title overview'}sea biscuit likes to fish!
```

With these basic ergonomics out of the way, let's take a look at this `edismax` query parser—the starting point for most Solr query solutions. How does it fit into the discussion in chapters 5 and 6?

### **B.2.4 Term-centric and field-centric search with the `edismax` query parser**

Solr often advocates that developers start with the `edismax` query parser, Solr's Swiss Army knife. As a Solr developer, `edismax` often sits at the root of your relevance work. The `edismax` query parser does the following:

- 1 It evaluates whether the query string (`q = ...`) corresponds to a Lucene-style query string or a regular query string.
- 2 If the query string is a Lucene-style query (`q=title:"taco nacho"`), that query is evaluated.
- 3 If the query string is not a Lucene-style query (`q=sea biscuit likes to fish`), `edismax` runs a `dismax`-style term-centric search.
- 4 Then it layers on various boosts.

Here you'll focus on the precise behavior in step 3 to see how it maps into the discussion of chapters 5 and 6. It's worth noting that `edismax` comes with additional parameters for additive and multiplicative boosting; we'll get to these in section B.3.

In chapter 6, we discussed that query parsers, such as `query_string`, must choose a consistent method for tokenizing query text that doesn't involve an analyzer. What

edismax does (and what Solr does on all query parsers) is break up the search query on whitespace prior to analysis. So edismax takes the following query

```
q=sea biscuit likes to fish!&defType=edismax&qf=title overview
```

and turns it into a term-by-term dismax query:

```
(title:sea | overview:sea) (title:biscuit | overview:biscuit)
(title:likes | overview:likes) (title:to | overview:to)
(title:fish! | overview:fish!)
```

Recall that `|` here corresponds to picking the dismax operation—taking the maximum query scores. Pure dismax corresponds to a `best_fields`, “winner takes all” strategy. The fields that score the highest will always win the competition. One way to adjust this equation is by adjusting boosts. The edismax query parser lets you adjust boosts by using the same syntax you saw with Elasticsearch:

```
qf=title^0.1 overview^10
```

Also recall that we discussed the `tie_breaker` parameter, which allows you to layer in scores from the other “losing” fields. Luckily, edismax comes with a `tie` parameter that performs the same functionality:

```
tie=1.0
```

In chapter 5, we discussed how setting `tie_breaker` to 1.0 results in the searched fields being summed. This moves the search from a `best_fields` “winner takes all” multifield search strategy to a `most_fields` “every field gets a vote” strategy in Elasticsearch parlance. You used this strategy to let every field’s score have a say in the ranking function. To replicate this multifield `most_fields` search discussed in chapter 5, you would simply run an edismax query with `tie` set to 1.0.

### B.2.5 *All fields and cross\_fields search*

To round out our discussion from chapters 5 and 6 are two topics: `cross_fields` search and `all fields`. Remember, the goal of these two strategies is to resolve *field discordance*. By default, the scoring component IDF isn’t counted across fields. We pointed out in chapter 6 that query parsers, like edismax or the `query_string` query parser, solve part of the term-centric search problem. But they don’t solve field discordance. We presented two strategies commonly used to resolve field discordance:

- *All fields*—Combining fields into a single field. This combines the document frequency of the source fields.
- *cross\_fields search*—Using Elasticsearch’s `cross_fields` search to account for document frequency across fields at query time, albeit somewhat less accurately.



**ALL FIELDS IN SOLR**

For all fields, you're in luck. Just as Elasticsearch has `copy_to`, Solr's schema comes with `copyField`. Using what Solr calls *copy fields*, you can implement the same behavior. Copy fields sit alongside field declarations in the `schema.xml` file:

```
<copyField source="title" dest="text_all" />
```

Here Solr copies the `title` field into the `text_all` field at index time. Unlike Elasticsearch, there's no built-in `_all` field. But default Solr schemas often define a `text_all` field with several `copy` directives. This field often acts just like the default `_all` field.

**CROSS\_FIELDS SEARCH IN SOLR**

For cross-fields search, there's no comparable functionality in Solr. But as we discussed, it's easy in Solr for you to implement your own query parsers. The underlying intelligence exists in Lucene, in the Java class `BlendedTermQuery`. This query performs the hard work of the `cross_fields` functionality in Elasticsearch. If you're interested in having this functionality in Solr, we recommend using Solr resources linked previously to implement your own query parser.

**B.3 Chapter 7: shaping Solr's ranking function**

Chapter 7 encourages you to see relevance as something you can tightly control. The good news is, Solr brings a ton of power to the table. Recall that chapter 7 focuses heavily on boosting, both additive and multiplicative. We also discuss two modes of boosting: using a function query or a Boolean query.

Solr, largely through the `edismax` query parser, supports all of these capabilities. Solr surpasses Elasticsearch in the power of its capabilities, even if ultimately Solr leaves something to be desired in its ease of use.

**B.3.1 Summary of boosting feature mappings**

Table B.3 maps general types of boosting to the equivalent Solr features.

**Table B.3 Solr boost features**

Chapter	Feature	Solr Analogue
7	Additive boosting, Boolean	<code>edismax</code> with <code>bq</code> parameter
7	Additive boosting, function	<code>edismax</code> with <code>bf</code> parameter
7	Multiplicative boosting, function	<code>edismax</code> with <code>boost</code> parameter

### B.3.2 Solr's Boolean boosting

In chapter 7, we used additive boosting via Boolean queries. This syntax uses the `bool` query primitives that map straight to Lucene's Boolean query, as in this query snippet:

```
"query": {
  "bool": {
    "should": [
      {
        "match": {
          "title": "Rambo"
        }
      }
      ...
    ]
  }
}
```

Additive boost  
Boolean clause

Solr's “Boolean” boosting acts similarly. As with Elasticsearch, you can boost using any query you like. The syntax, however, looks quite different. With Solr, you apply a query boost with the `bq` parameter. Recall that Solr has a local params query syntax that lets you specify the query parser used. An additive boost would leverage local params syntax with a `bq` additive boost like so:

```
bq={!defType=lucene}title:Rambo
```

Here, you tell Solr to boost additively by running the query string `title:Rambo` through the Lucene query parser. The result of that query parser is appended to the base edismax query as a Boolean SHOULD clause. For Solr, the addition of query parsers makes this a rich piece of functionality. Solr's richness here allows you to layer all kinds of custom functionality into the boosting process.

### B.3.3 Solr's function queries

Just like Elasticsearch, Solr lets you run additive or multiplicative function queries. In Elasticsearch, this takes the form of a list of functions specified in JSON. Consider, for example, this simple boost on a TMDb movie's ratings (`vote_average`):

```
"query": {
  "function_score": {
    "query": {
      ...
    },
    "functions": [
      {
        "field_value_factor": {
          "field": "vote_average",
          "modifier": "sqrt"
        }
      },
      ...
    ],
    "boost_mode": "sum"
  }
}
```

Root search  
query

Takes the square root  
of vote\_average

Adds the results of  
these functions and  
the root query score

In Solr, the syntax looks more like a formula from an Excel spreadsheet. Further, Solr comes with an extremely powerful addition that Elasticsearch sorely lacks: the query function. That's right, kids, you can add query functions to your function queries—say that three times fast! The query function lets you run an arbitrary query and inject its TF × IDF relevance score into the function query itself (as if specified in the list of functions in the preceding Elasticsearch snippet).

So what does a Solr function query look like? As we promised, it looks like something straight out of a spreadsheet. The `edismax bf` parameter performs an additive function boost. Here's an additive function query boost by a numeric popularity field:

```
bf=sqrt(popularity)
```

Or to multiply the `sqrt` of popularity by the TF × IDF score of `rambo` in the title, you might use the query function:

```
ramboScore={!defType=lucene}title:Rambo
&bf=product(sqrt(popularity),query($ramboScore))
```

Taken as a full Solr query, with all parameters included, this would be as follows:

```
http://solr.quepid.com/solr/tmdb/select?q=*&defType=edismax
&qf=titleoverview&ramboScore={!defType=lucene}title:Rambo&
bf=product(sqrt(vote_average),query($ramboScore))
```

Here we've broken up the boosts into two components: a query and a function. For the query, you can see that Solr lets you organize components into variables. You see that with the previous `ramboScore` variable.<sup>1</sup> This query runs a simple search for Rambo in the `title` field. The function, `bf`, takes the square root of the movie's popularity and multiplies it by the movie title's TF × IDF score for Rambo. This value is then added to the overall relevance score (an additive boost). The end result is popular Rambo movies are boosted to the top.

As you can see, Solr's function queries are far more concise than Elasticsearch's. But you may also see that given sufficient complexity, they become hard to work with! In this realm, Solr can be like Perl; loyalists have learned to do surprisingly amazing things with its power, but detractors see the result of such a function query and cringe at the poor readability.

Elasticsearch lets you go so far as to write custom scripts as a function in `function_score_query`. Scripts in Elasticsearch are implemented in several programming

---

<sup>1</sup> To learn more about ways of organizing Solr boosts, we recommend this article, written by one of the authors: "Parameterizing and Organizing Solr Boosts" on the OpenSource Connections blog, <http://opensourceconnections.com/blog/2013/11/22/parameterizing-and-organizing-solr-boosts/>. For strategies like those discussed in chapter 7, but heavily targeted at Solr, we recommend "Improve search relevancy by telling Solr exactly what you want" on the OpenSource Connections blog, <http://opensourceconnections.com/blog/2013/07/21/improve-search-relevancy-by-telling-solr-exactly-what-you-want/>.

languages such as Groovy and JavaScript. A true programming environment has the advantage of being far more readable.

### **B.3.4 Multiplicative boosting in Solr**

Elasticsearch multiplicative boosting occurs by changing the `sum` line in the preceding `function_score_query` to `product`. Similarly, applying multiplicative boosting in Solr requires a simple change to the additive boost.

Solr multiplicative boosting means using the `boost` parameter to multiply the base relevance score by the result of a Solr function query. For instance, to bias results toward popularity, you might implement the following multiplicative boost:

```
boost=sqrt(popularity)
```

## **B.4 Chapter 8: relevance feedback**

Chapter 8 discusses the importance of user experience components such as faceting, autocomplete, highlighting, and field collapsing. The chapter implements these by using many relevance features discussed in earlier chapters. But it does discuss some features not introduced in previous chapters. So we'll round out our Solr discussion by mapping the grab bag of features to help you get through the content in chapter 8.

### **B.4.1 Summary of relevance feedback feature mappings**

Table B.4 maps Elasticsearch feedback features to Solr equivalents.

**Table B.4 Relevance feedback features in Solr**

Chapter	Feature	Solr analogue
8	Facets (in Elasticsearch via terms aggregation)	Solr faceting
8	Phrase prefix querying	Solr's <code>complexphrase</code> query parser
8	Grouping (in Elasticsearch via top hits aggregation)	Solr's field-collapsing functionality
8	Suggestions and spell-checking	See documentation for Solr's suggestion and spell-checking features
8	Highlighting	See documentation for Solr's highlighter

### **B.4.2 Solr autocomplete: match phrase prefix**

One strategy used in Elasticsearch to implement an autocomplete solution depends on the `match_phrase_prefix` query. This query runs a phrase query with a prefix on the last term, such as in this snippet:

```
{ "query": {
  "match_phrase_prefix" : {
    "title": "star tr"}}}
```

Solr delivers comparable functionality in its `complexphrase` query parser. This query parser interprets a string with a trailing `*` identically to the `match_phrase_prefix`. So the preceding query becomes `title:star tr*`, as shown here:

```
http://solr.quepid.com/solr/tmdb/select?q=title:star
tr*&defType=complexphrase
```

This Solr query runs a phrase query identical to `match_phrase_prefix`, focusing on the phrase up until the last term. Then the trailing `*` tells the query parser to perform a prefix search on the last term.

### B.4.3 Faceted browsing in Solr

Facets are a crucial component of most search and browsing applications! These are the components on the side of the search application that divide search results into filterable categories. Chapter 8 uses Elasticsearch's terms aggregation to implement faceted search and some autocomplete functionality. Consider the following snippet:

```
"aggregations": {
  "completion": {
    "terms": {
      "field": "title"
    }
  }
}
```

You'll find comparable functionality with Solr's facets. The preceding aggregation can be implemented as a simple facet on the field `title`:

```
facet=true&facet.field=title
```

Similarly, Elasticsearch allows you to include/exclude aggregation results by using the `include` parameter, as in this snippet:

```
"terms": {
  "field": "title",
  "include": "rambo*",
}
```

Solr provides this capability via the `facet.prefix` option:

```
facet=true&facet.field=title&facet.prefix=rambo
```

Taken together, the entire facet query then becomes the following:

```
http://solr.quepid.com/solr/tmdb/select?q=*:*&facet=true&
facet.field=title&facet.prefix=rambo
```

Now you can go forth and facet!

### B.4.4 Field collapsing

Chapter 8 notes that you often want to group search results in a hierarchy. Elasticsearch supports this with its `top_hits` aggregation. This groups search results by a common, shared property. For example, consider this query:

```
"aggs": {
  "original_versions": {
    "terms": {
      "field": "original_id",
      "order": { "top_score": "desc" },
      "aggs": {
        "hits": {
          "top_hits": { "size": 1 },
          "top_score": {
            "max": { "script": "_score" }
          }
        }
      }
    }
  }
}
```

Group by `original_id`

Top hits subaggregation per unique `original_id` term

This groups results by the `original_id`, and then shows scores bucketed by `original_id`. Solr can do this too! Solr has a field-collapsing feature for the same functionality via its `group` parameters:

```
&group=true&group.field=original_id&group.limit=3
```

Results for such a query come back grouped by `original_id`, with three results per unique ID in this case. You can explore additional grouping parameters in Solr's documentation.

### B.4.5 Suggestion and highlighting components

Spell-checking, query suggestions, and highlighting go a long way to support any search application. Chapter 8 demonstrates Elasticsearch's suggestion and highlighting features. As these are topics in place to support the relevance discussion, we won't dive into a detailed comparison of the two search engines' spell-checking and highlighting components.

Instead, we'll just point out that nearly identical functionality exists in Solr. Solr comes prepackaged with a suggestion component *and* a spell-checking component. Solr also comes with an extremely configurable (and pluggable) highlighting component. For the most part, with out-of-the-box settings, you can turn on these features with `suggest=true` and `hl=true`. For example, this result of this query contains a highlighted response:

```
http://solr.quepid.com/solr/tmdb/select?q=sea biscuit likes to fish!&hl=true
```

The number of features available on these components is extensive. We invite you to explore their capabilities in the official documentation.

## **Symbols**

---

- character 34
- ^ symbol 46
- + character 34
- | character 72, 149

## **Numerics**

---

- 2-gram completions 209
- 2D graphs 200
- 3D graphs 200

## **A**

---

- acronyms 90–91
- actionable information 186
- ad hoc searches 148
- add shingling 126
- addCmd 45
- additive boosting, with Boolean queries 176–178
  - combining boost and base query 177–178
  - function queries vs. 174–175
  - optimizing boosts in isolation 176–177
  - Solr 318
- adjusted boosts 72
- affinity 286
- aggregate information 20
- aggregations 36–37
- albino elephant example 140–144
- all fields
  - combining fields into customized 157–161
  - overview 164, 169
  - Solr 317
- alternative results ordering 222–223

- Amazon-style filtering 36
- an \_explanation entry 57
- analysis 20–21, 25–26, 48
  - components of 30–31
  - tokens as search features 29
- analysis plugin 104
- analyze endpoint 51, 81, 94
- analyzers
  - overview 51
  - Solr 310–312
    - analysis and mapping features 310
    - building custom 310–311
    - field mappings 312
- AND operator 32–33
- anticipating
  - user behavior 93
  - user intent 76
- api\_key argument 304
- assertion-based testing 273–274
- asymmetric analysis 96, 100
- asymmetric tokenization 101
- autocomplete keyword 255

## **B**

---

- bag of words model 63, 65
- base query, boosting 173
- base signal 176
- basic highlighter 225
- begin sentinels 188
- behavior-based personalization 283
- behavior, anticipating 93
- best\_fields 244, 313, 316
  - calibrating 129–130
  - controlling field preference in results 124–126
  - more-precise signals 126–129

- bf parameter 317, 319
  - bigram\_filter 127
  - bigrams 126–127
  - black scores 189
  - BlendedTermQuery class 313, 317
  - BM25 69
  - bold matches 144
  - bool query 175–176, 202, 318
  - Boolean boost 190
  - Boolean clauses 50, 148
  - Boolean queries
    - additive boosting with 176–178
      - combining boost and base query 177–178
      - function queries vs. 174–175
      - optimizing boosts in isolation 176–177
    - Solr 318
    - overview 148–149, 154, 175
  - Boolean search 32–33
  - boost parameter 320
  - boosting 46, 57, 171–182
    - additive, with Boolean queries 176–178
      - combining boost and base query 177–178
      - function queries vs. 174–175
      - optimizing boosts in isolation 176–177
    - multiplicative, with function queries 179–180
      - Boolean queries vs. 174–175
      - simple 180–182
    - signals 182, 186–189
    - Solr 317–320
      - additive, with Boolean queries 318
      - boosting feature mappings 317
      - multiplicative, with function queries 318–320
    - user ratings 196
    - vs. filtering 183
  - breadcrumb navigation 221–222
  - browse experience 218
  - browse interface, Yowl 239
  - buckets section 228
  - building signals 144
  - bulk index API 44–45
  - bulkMovies string 45
  - business and domain awareness 265–267
  - business concerns group 242
  - business weight 250
  - business-ranking logic 3
  - BusinessScore 248
- C**
- 
- cast.name field 117–119, 124, 126, 128, 158
  - cast.name scores 124
  - cast.name.bigrammed field 128, 130, 143, 191
  - character filtering 30, 51, 53–54
  - character offsets 24
  - classic similarity 68–69
  - classification features 12
  - cleaning 27
  - click-through rate 253
  - co-occurrence counting 284–289
  - cold-start problem 298
  - COLLAB\_FILTER filter 290, 293
  - collaboration
    - filtering, using co-occurrence counting 284–289
    - search relevance and 12–14
  - collation 217
  - collocation extraction 298
  - combining fields 157
  - committed documents 32
  - common words, removing 31
  - completion field 210, 213
  - completion suggester 213
  - completion\_analyzer 210
  - completion\_prefix variable 211
  - complexphrase query parser 320–321
  - compound queries 60–61, 64–65, 72
  - concept search 279
    - basic methods for building 293–296
      - augmenting content with synonyms 295–296
    - concept signals 294–295
    - building using machine learning 296–298
    - personalized search and 298–299
  - configurations 254
  - conflate tokens 98
  - constant\_score query 167, 195
  - content
    - augmentation 296–297
    - curation 267–270
      - engineer/curator pairing 270–272
      - risk of miscommunication with content curator 269–270
      - role of content curator 268–269
    - exploring 20
    - extracting into documents 26–27
    - providing to search engine 20–21
    - searching 18–19
  - content group 242
  - content weight 247–248, 250–251
  - ContentScore 248
  - control analysis 311
  - controlling field matching 192
  - converge 275
  - conversion rate 253
  - coord (coordinating factor) 64, 89, 121, 133, 150, 154, 177
  - copyField 157, 313, 317
  - copy\_to option 158–159, 313



- cosine similarity 65
- cross\_fields 157, 313, 316
  - searching 164, 173, 177–178, 191
  - Solr 317
  - solving signal discordance with 161–162
- cuisine field 244
- cuisine\_hifi field 241, 244
- cuisine\_lofi field 241
- curation, search relevance and 12–14
- custom all field 158–159
- custom score query 172

## D

---

- data-driven culture 261–262
- debugging 40–73
  - example search application 43–48
    - Elasticsearch 41–42
    - first searches with 46–48
    - The Movie Database 42
    - Python 43
  - matching 50
  - query matching 48–56
    - analysis to solve matching issues 51–53
    - comparing query to inverted index 53–54
    - fixing by changing analyzers 54–56
    - query parsing 50
    - underlying strategy 49
  - ranking 56–71
    - computing weight 67–68
    - explain feature 57–61
    - scoring matches to measure relevance 65–66
    - search term importance 70
    - similarity 68–69
    - vector-space model 61–64
- decay functions 197, 200
- deep paging 253
- default analyzer 188
- defType parameter 315
- delimiters
  - acronyms 90–91
  - modeling specificity 96–100
  - phone numbers 91–93
  - synonyms 93–96
  - tokenizing geographic data 102–103
  - tokenizing integers 101
  - tokenizing melodies 103–106
- deployment, relevance-focused search
  - application 252–255
- description field 94, 241, 244, 255, 290
- descriptive query 47
- directors field 117
- directors.name field 119, 124, 155, 158
- directors.name score 124
- directors.name.bigrammed 143, 146, 191

- disable\_coord option 177
- disabling tokenization 187
- discriminating fields 167
- DisjunctionMaximumQuery 149
- dismax 149, 313
- doc frequency 24, 37
- doc values 25
- document search and retrieval 32–39
  - aggregations 36–37
  - Boolean search 32–33
  - facets 36–37
  - filtering 36–37
  - Lucene-based search 34–35
  - positional and phrase matching 35
  - ranked results 37–39
  - relevance 37–39
  - sorting 37–39
- document-ranking system 86
- documents
  - analysis 28–31
  - enhancement 27
  - enrichment 27
  - extraction 26–27
  - flattening nested 116–118
  - grouping similar 228–230
  - matching 174
  - meaning of 76–77
  - scored 144
  - search completion from documents being
    - searched 209–213
  - tokens as features of 75–77
    - matching process 76
    - meaning of documents 76–77
- dot character 296
- dot product 62, 64–65
- down-boosting title 133
- DSL (domain-specific language) 46

## E

---

- e-commerce search 5, 8
- easy\_install utility 304
- edismax query parser 313, 315–317
- Elasticsearch
  - example search application 41–42
  - overview 12
- end sentinels 188
- engaged field 242
- engaged restaurants 237
- English analyzer
  - overview 82
  - reindexing with 54
- english\_\* filters 94
- english\_bigrams analyzer 127
- english\_keywords filter 83

english\_possessive\_stemmer filter 83  
 english\_stemmer filter 83  
 english\_stop filter 83  
 enrichment 25, 27  
 ETL (extract, transform, load) 25, 45  
 every field gets a vote 122  
 exact matching 185, 187–188, 190, 193  
 expert search 5, 9, 13  
 explanation field 49  
 external sources 27  
 extract function 43–45, 115, 305  
 extracting features 75  
 extraction 25–27

## F

faceted browsing  
   overview 218–221  
   Solr 321  
 facet.prefix option 321  
 facets 20, 36–37, 218  
 fail fast 116, 259, 263, 265  
 fast vector highlighter 225–228  
 feature modeling 75, 83  
 feature selection 11  
 feature space 62  
 features  
   creation of 76  
   overview 11, 21, 29  
 feedback  
   at search box 206–218  
     search completion 207–215  
     search suggestions 215–218  
     search-as-you-type 206–207  
   business and domain awareness 265–267  
   content curation 267–270  
     risk of miscommunication with content  
       curator 269–270  
     role of content curator 268–269  
   in search results listing 223–231  
     grouping similar documents 228–230  
     information presented 224–225  
     snippet highlighting 225–228  
     when there are no results 230–231  
   search relevance and 12–14  
   Solr 320–322  
     faceted browsing 321  
     field collapsing 322  
     match phrase prefix 320–321  
     relevance feedback feature mappings 320  
     suggestion and highlighting components 322  
   while browsing 218–223  
     alternative results ordering 222–223  
     breadcrumb navigation 221–222  
     faceted browsing 219–221

field boosts 155  
 field collapsing  
   overview 228–230  
   Solr 322  
 field discordance 316  
 field mappings 127  
 field normalization 177–178  
 field scores 140, 149  
 field synchronicity, signal modeling and  
   152–153  
 field-by-field dismax 149  
 field-centric methods 146, 161  
 field-centric search, combining term-centric  
   search and 162–169  
   combining greedy search and conservative  
     amplifiers 166–168  
   like fields 163–165  
   precision vs. recall 168  
   Solr 315–316  
 fieldNorms 69, 71, 73  
 fields 18  
 fieldType 310–311  
 field\_value\_factor function 196  
 fieldWeight 66, 69, 71, 73  
 filter clause 183  
 filter element 311  
 filter queries 183  
 filtering 171–172  
   Amazon-style 36  
   collaborative  
     overview 283–284  
     using co-occurrence counting 284–289  
   score shaping 182–183  
   vs. boosting 183  
 finite state transducer 213  
 fire token 52  
 first\_name field 110–112  
 floating-point numbers 100  
 fragment\_size parameter 227  
 fudge factors 179  
 full bulk command 45  
 full search string 139  
 full-text search 21  
 full\_name field 112–113  
 function decay 199  
 function queries, multiplicative boosting  
   with 179–180  
   Boolean queries vs. 174–175  
   combining 200–202  
   high-value tiers scored with 193–194  
   simple 180–182  
   Solr 318–320  
 function\_score query 194–195, 247, 249,  
   290

**G**

garbage features 75  
 Gaussian decay 198  
 generalizing matches 97  
 generate\_word\_parts 91  
 genres aggregation 220  
 genres.name field 219  
 geographic data, tokenizing 102–103  
 geohashing 28  
 geolocation 12, 28  
 getCastAndCrew function 306  
 GitHub repository 42–43  
 granular fields 146  
 grouping fields 163–164

**H**

has\_discount field 242  
 high-quality signals 189  
 highlighted snippets 24  
 highlights 20  
 HTMLStripCharFilter 30  
 HTTP commands 44, 80

**I**

ideal document 135  
 IDF (inverse document frequency)  
   ignoring when ranking 194–195  
   overview 67–68, 89–90  
 inconsistent scoring 155  
 index-time analysis 96, 99  
 index-time personalization 291–293  
 indexing documents 44  
 information and requirements gathering  
   234–237  
   business needs 236  
   required and available information  
     236–237  
   users and information needs 234–236  
 information retrieval, creating relevance  
   solutions through 8–10  
 inner objects 118  
 innermost calculation 149  
 integers, tokenizing 101  
 inventory-related files 99  
 inventory\_dir configuration 99–100  
 inverse document frequency. *See* IDF  
 inverted index data structure 25–32  
   analysis 28–31  
   comparing query to 53–54  
   enrichment 27  
   extraction 26–27  
   indexing 31–32

isolated testing 188  
 item information 300  
 iterative 259–260, 271–272

**J**

JSON standard library 43  
 judgment lists 7–8, 273–275

**K**

keyword tokenizer 92, 105  
 keywords.txt file 83

**L**

last\_name field 110  
 latitude points 100  
 law of diminishing returns 255–256  
 leading vowels 85  
 lexicographical order 23  
 like fields  
   grouping together 163–164  
   limits of 164–165  
 local params 315, 318  
 location field 241–242  
 location weight 250–251  
 LocationScore 248  
 long-tail application 208, 261–262  
 longitude points 100  
 lowercase filter 81, 83, 85  
 Lucene-based search  
   Boolean queries in 34–35  
   explain feature 57–61

**M**

machine learning, building concept search  
   using 296–298  
 malicious websites 4  
 map signals 163  
 mapping fields 158  
 master signal modeling 109  
 match phrase prefix, Solr 320–321  
 matched fields 144  
 matching  
   documents 37  
   multiple terms 32  
 match\_phrase query 176, 178, 206, 211–212,  
   217  
 max\_gram setting 105  
 McCandless, Mike 52  
 melodies, tokenizing 103–106  
 menu field 241  
 MeSH (Medical Subject Headings) 5, 98, 295

- metadata, storing 31
- metrics, capturing general-quality 195–197
- middle\_initial field 110
- min\_gram setting 105
- misspellings 84
- monitoring relevance-focused search
  - application 253–254
- most\_fields
  - boosting in 132–134
  - searching 141, 143, 163
  - when additional matches don't matter 134–135
- movieDict dictionary 44–45
- movieList function 305
- multifield search 107–135
  - The Movie Database 114–118
  - signal modeling 114, 118–135
    - best\_fields 122–130
    - most\_fields 131–135
  - signals 109–114
    - defined 109–110
    - implementing 112–114
    - source data model 110–112
- Solr 312–317
  - all fields 317
  - cross\_fields search 317
  - edismax query parser 315–316
  - ergonomics 314–315
  - query differences between Solr and Elasticsearch 313–314
  - query feature mappings 313
- multifield searches 157
- multi\_match query 46, 57, 110–112, 119, 151, 161, 244
- multiple documents 118
- multiplicative boosting, with function
  - queries 179–180
  - Boolean queries vs. 174–175
  - combining 200–202
  - high-value tiers scored with 193–194
  - simple 180–182
  - Solr 318–320
- multiplying variables 200
- MUST clause 34–35, 50
- MUST\_NOT clause 34–35, 50
- my\_doublemetaphone filter 85

## N

---

- n-gram token filter 104
- n-gramming analyzer 104
- name field 241
- named attributes 18
- negative boosting 172
- nested documents 118

- no\_match\_size parameter 227
- nongreedy clauses 166
- nonwinning fields 155
- normalize acronyms 90
- NOT operator 33
- number\_of\_fragments 227
- numerical attributes 42
- numerical boosts 38
- numerical data 100
- num\_of\_fragments parameter 227

## O

---

- OLAP (online analytical processing) 37
- optimizing signals 185
- OR operator 33
- order parameter 227
- ordering documents 19
- origin variable 197
- original\_id field 230, 322
- overview field 149–150, 226–228

## P

---

- PageRank algorithm 4, 9, 11
- pair tuning 271, 273
- paired relevance tuning 270–272
- parent-child documents 118
- parentheses 34
- parsons analyzer 105
- Parsons code 105
- path\_hierarchy analyzer 99
- path\_hierarchy tokenizer 212
- paths, modeling specificity with 99–100
- pattern\_capture filter 92
- payloads 24, 31
- people.name field 157–160, 162
- persona 234
- personalizing search
  - based on user behavior 283–293
    - collaborative filtering 283–289
    - tying behavior information back to search index 289–293
  - based on user profiles 281–283
    - gathering profile information 282
    - tying profile information back to search index 282–283
  - concept search and 298–299
- phone\_number field 92
- phone\_num\_parts filter 92
- phonetic analyzer 84–85
- phonetic plugin 84
- phonetic tokenization 84, 86, 90
- phrase query 113, 183, 187
- phrase-matching clause 177

- phrases, concept search and 297–298
- pogo-sticking 253
- popularity field 319
- position entry 52
- positional and phrase matching 35
- postings highlighter 225–226, 228
- postings list 22–23, 32
- post\_tags parameter 226
- precision 77–90
  - analysis for 80–84
  - by example 77–80
  - combining field-centric and term-centric search 168
  - multiple search terms and multiple fields 89–90
  - phonetic tokenization 84–86
  - scoring strength of feature in single field 86–89
- premature optimization 116
- pre\_tags parameter 226
- price field 241
- prioritizing documents 172
- product codes 27
- product owner 268
- profile-based personalization 281
- profiles 281
- promoted field 242
- prose text 42
- pseudo-content 296–297
- Python example search application 43

## Q

- quadrants 102
- query behavior, explaining 49
- Query DSL 46–50, 61, 171
- query function 319
- query matching, debugging 48–56
  - analysis to solve matching issues 51–53
  - comparing query to inverted index 53–54
  - fixing by changing analyzers 54–56
  - query parsing 50
  - underlying strategy 49
- query normalization 70
- query parameter 181
- query parsers 148–152, 155, 161–162
- query validation endpoint 49–50, 154
- query-time analysis 81, 96, 99
- query-time boosting 70
- query-time personalization 290–293
- queryNorm 70
- queryWeight 66, 70
- quotes 50

## R

- ranking
  - adding high-value tiers 189–193
  - adding new tier for medium-confidence boosts 191–192
  - tiered relevance layers 193
- debugging 56–71
  - computing weight 67–68
  - explain feature 57–61
  - scoring matches to measure relevance 65–66
  - search term importance 70
  - similarity 68–69
  - vector-space model 61–64
- learning to rank 276–278
- term-centric 148–150
- real-estate search 6
- recall 77–90
  - analysis for 80–84
  - by example 77–80
  - combining field-centric and term-centric search 168
  - improving 78
  - multiple search terms and multiple fields 89–90
  - phonetic tokenization 84–86
  - scoring strength of feature in single field 86–89
- recency
  - achieving users' recency goals 197–200
  - overview 179
- reducing boost weight 178
- reindex function 44–45, 115, 187, 307
- reindexing with English analyzer 54
- related\_items field 292
- relevance engineers
  - duties of 10
  - gaining skills of 2
  - overview 263
- relevance. *See* search relevance
- relevance-blind enterprise 263, 265
- relevance-centered enterprise 257–278
  - business and domain awareness 265–267
  - content curation 267–270
    - risk of miscommunication with content curator 269–270
    - role of content curator 268–269
- feedback 259–260
- learning to rank 276–278
- paired relevance tuning 270–272
- test-driven relevance 272–276
  - using with user behavioral data 275–276
- user-focused culture vs. data-driven culture 261–262

- relevance-focused search application
  - 232–256
  - deploying 252–255
  - designing 238–252
    - combine and balance signals 252
    - combining and balancing signals 241–242
    - defining and modeling signals 241–242
    - user experience 239–241
  - improving 254–255
  - information and requirements gathering 234–237
    - business needs 236
    - required and available information 236–237
    - users and information needs 234–236
  - law of diminishing returns 255–256
  - monitoring 253–254
- requests library 304
- reranking 172
- rescoring 172
- response page 19
- retail\_analyzer filter 94
- retail\_syn\_filter filter 94
- retention 253
- reweighting boosts 178

## S

- salient features 10
- scale variable 197
- scorable units 112
- score boost 38, 56
- score shaping
  - boosting 172–182
    - additive, with Boolean queries 174–178
    - multiplicative, with function queries 174–175, 179–182
    - signals 182
  - defined 171–172
  - filtering 182–183
  - Solr 317–320
  - strategies for 184–203
    - achieving users' recency goals 197–200
    - capturing general-quality metrics 195–197
    - combining function queries 200–202
    - high-value tiers scored with function queries 193–194
    - ignoring TF × IDF 194–195
    - modeling boosting signals 186–189
    - ranking 189–193
- scored documents 144
- scoring tiers 189, 193
- script scoring 172, 245

- search 16–39
  - content
    - exploring 20
    - providing to search engine 20–21
    - searching 18–19
  - document search and retrieval 32–39
    - aggregations 36–37
    - Boolean search 32–33
    - facets 36–37
    - filtering 36–37
    - Lucene-based search 34–35
    - positional and phrase matching 35
    - ranked results 37–39
    - relevance 37–39
    - sorting 37–39
  - documents 18
  - inverted index data structure 22–32
    - analysis 28–31
    - enrichment 27
    - extraction 26–27
    - indexing 31–32
  - search antipattern 112
  - search completion 207–215
    - choosing method for 215
    - from documents being searched 209–213
    - from user input 208–209
    - via specialized search indexes 213–214
  - search engineer 264–265
  - search relevance 1–14
    - collaboration and 12–14
    - curation and 12–14
    - defined 13
    - difficulty of 3–6
      - class of search and 4–5
      - lack of single solution 6
    - feedback and 12–14
    - gaining skills of relevance engineer 2
    - information retrieval 7–10
    - research into 6–10
    - systematic approach for improving 10–12
  - search-as-you-type 206–207
  - searchable data 147
  - semantic expansion 96
  - sentiment analysis 27
  - sentinel tokens 187–188, 192
  - sharding 45
  - short-tail application 208
  - SHOULD clause 34–35, 50, 120, 175, 318
  - signal construction 255
  - signal discordance 157, 160–163
    - avoiding 144–145
    - combining fields into custom all fields 157–161
    - mechanics of 145–147
    - solving with cross\_fields search 161–162
  - signal measuring 146

- signal modeling 118–135
  - best\_fields 122–124
  - calibrating 129–130
  - controlling field preference in results 124–126
  - more-precise signals 126–129
  - field synchronicity and 152–153
  - most\_fields 131–132, 135
  - boosting in 132–134
  - when additional matches don't matter 134–135
- signals 109–114
  - boosting 182, 186–189
  - combining and balancing 242–252
    - behavior of signal weights 247–249
    - building queries for related signals 243–245
    - combining subqueries 246–247
    - tuning and testing overall search 249–251
    - tuning relevance parameters 251–252
  - concept 294–295
  - defined 109–110
  - defining and modeling 241–242
  - implementing 112–114
  - source data model 110–112
- silli token 83
- similarity 68–69
- simple constants 172
- SimpleText data structure 52, 67
- snippet highlighting 225–228
- Solr 309–322
  - analyzers 310–312
    - analysis and mapping features 310
    - building custom 310–311
    - field mappings 312
  - boosting 317–320
    - additive, with Boolean queries 318
    - boosting feature mappings 317
    - multiplicative, with function queries 318–320
  - feedback 320–322
    - faceted browsing 321
    - field collapsing 322
    - match phrase prefix 320–321
    - relevance feedback feature mappings 320
    - suggestion and highlighting components 322
  - multifield search 312–317
    - all fields 317
    - cross\_fields search 317
    - ergonomics 314–315
    - query differences between Solr and Elasticsearch 313–314
    - query feature mappings 313
    - term-centric and field-centric search with edismax query parser 315–316
  - sorting 37–39
  - source data model 144
  - span queries 35
  - specificity, modeling
    - with paths 99–100
    - with synonyms 96–99
  - standard analyzer 51–52, 80–81, 83–84, 87–88
  - standard filter 81, 85
  - standard tokenizer 30, 81, 83, 85, 210
  - standard\_clone analyzer 81
  - stemming 86
  - stop filter 81
  - stop words 31, 54, 56
  - stored fields 24
  - storing metadata 31
  - string types 18
  - subdivided text 114
  - subobjects 116
  - subquadrants 102
  - suggest clause 214
  - suggest endpoint 214, 216
  - suggestion field 216
  - sum\_other\_doc\_count 220
  - synonyms
    - augmenting content with 295–296
    - modeling specificity with 96–99
    - overview 12, 93–96

## T

---

- term dictionary 22–23, 32
- term filter 100
- term frequency. *See* TF
- term offsets 24
- term positions 24
- term query 50
- term specificity 126
- term-centric search 137–169
  - albino elephant example 140–144
  - combining field-centric search and 162–169
    - combining greedy search and conservative amplifiers 166–168
    - like fields 163–165
    - precision vs. recall 168
  - defined 138–140
  - field synchronicity 152–153
  - need for 140–147
  - overview 119, 135
  - query parsers 151–155
  - ranking function 148–150
  - signal discordance 157–162
    - avoiding 144–145
    - combining fields into custom all fields 157–161
    - mechanics of 145–147



- term-centric search (*continued*)
    - query parsers and 153–155
    - solving with cross\_fields search 161–162
  - Solr 315–316
  - tuning 155–157
  - terms aggregation 211–212, 228, 230
  - term\_vector 226
  - test-driven relevance 244, 250, 252, 272–276
  - text analysis 21, 255
  - text field 152
  - text tokenization 28
  - text-relevance scores 172
  - text\_all field 317
  - text\_standard\_clone 312
  - TF (term frequency)
    - ignoring when ranking 194–195
    - overview 67–68, 89–90
  - TF × IDF scoring 87, 177–178, 182, 188, 195
  - thrashing 254
  - tie\_breaker parameter 249, 316
  - time on page 253
  - title phrases 211
  - title score 57
  - title-based completions 210
  - title\_exact\_match 187
  - title:with clause 54
  - TMDB (The Movie Database) 303–307
    - crawling API 305–306
    - example search application 42
    - indexing to Elasticsearch 307
    - multifield search 114–118
    - setting API key and loading IPython notebook 303–304
    - setting up for API 304
  - tmdb index 45–46
  - tmdb\_api\_key 304
  - TMDB\_API\_KEY variable 304
  - tmdb.json file 42–43, 115
  - tokenization 28, 30
  - tokenizers 51, 54
  - tokens 28–29, 74–106
    - as document features 75–77
      - matching process 76
      - meaning of documents 76–77
    - creation of 76
    - delimiters 90–93
      - acronyms 90–91
      - modeling specificity 96–100
      - phone numbers 91–93
      - synonyms 93–96
    - tokenizing geographic data 102–103
    - tokenizing integers 101
    - tokenizing melodies 103–106
  - filtering 30
  - matching 28–29
  - overview 21, 25, 48
  - precision and recall 77–90
    - analysis for 80–84
    - by example 77–80
    - multiple search terms and multiple fields 89–90
    - phonetic tokenization 84–86
    - scoring strength of feature in single field 86–89
  - top\_hits aggregation 322
  - top\_score field 230
  - transform function 187–188
  - trustworthiness score 4
  - tuned recency boost 180
  - tuning term-centric search 156
  - tweaking weights 175
  - two field groupings 165
  - two-word pairs 126
  - two-word subphrases 126
- 
- ## U
- 
- unstemmed 83
  - usability testing 266
  - user behavior
    - anticipating 93
    - personalizing search based on 283–293
      - collaborative filtering 283–289
      - tying behavior information back to search index 289–293
  - user experience, designing 239–241
  - user information 300
  - user intent
    - anticipating 76
    - overview 82
  - user preference group 242
  - user profiles, personalizing search based on 281–283
    - gathering profile information 282
    - tying profile information back to search index 282–283
  - user rating field 194–195, 200
  - user-focused culture 260, 262
  - user's ratings 196
  - user\_input variable 211
  - users\_who\_might\_like field 291–292
  - UTF-8 binary strings 48
- 
- ## V
- 
- value rating scale 283
  - VD vector 64
  - vector-space model 61–64
  - vote\_average field 195, 200
  - VQ vector 64



**W**

---

web search 4–5, 9  
weight  
    behavior of signal weights 247–249  
    computing with TF x IDF 67–68  
whistle encoder 103  
white bars 189  
whitespace tokenization 30  
Williams, Chuck 141  
winner-takes-all search 122  
winning field score 122  
with token 52, 54  
with\_positions\_offsets 226  
word endings 83  
word position 24  
Word2vec algorithm 297  
word\_delimiter filter 91–92  
wrapping queries 60

**X**

---

x-axis 199

**Y**

---

Yowl application example 232–256  
    deploying 252–255  
    designing 238–252  
    improving 254–255  
    information and requirements gathering  
        234–237  
    law of diminishing returns 255–256  
    monitoring 253–254

**Z**

---

Z-encoding 102–103

## MORE TITLES FROM MANNING



### *Solr in Action*

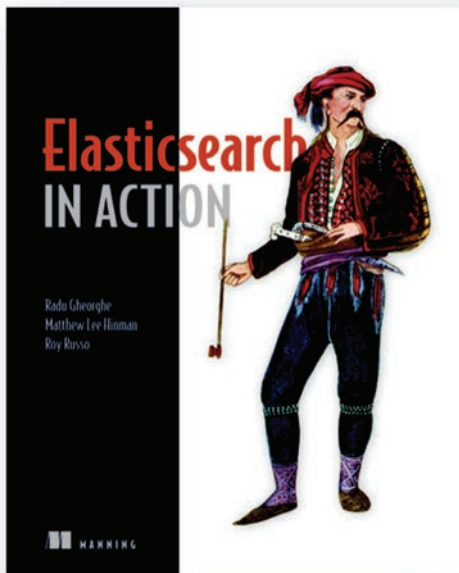
by Trey Grainger and Timothy Potter

ISBN: 9781617291029

644 pages

\$49.99

March 2014



### *Elasticsearch in Action*

by Radu Gheorghe, Matthew Lee Hinman,  
and Roy Russo

ISBN: 9781617291623

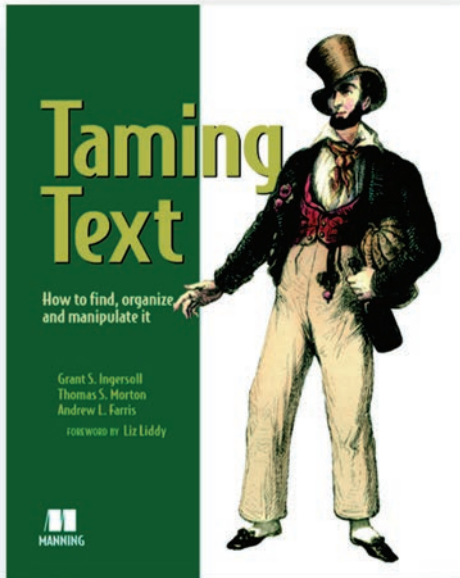
496 pages

\$44.99

November 2015

*For ordering information go to [www.manning.com](http://www.manning.com)*

MORE TITLES FROM MANNING



*Taming Text*  
*How to Find, Organize, and Manipulate It*

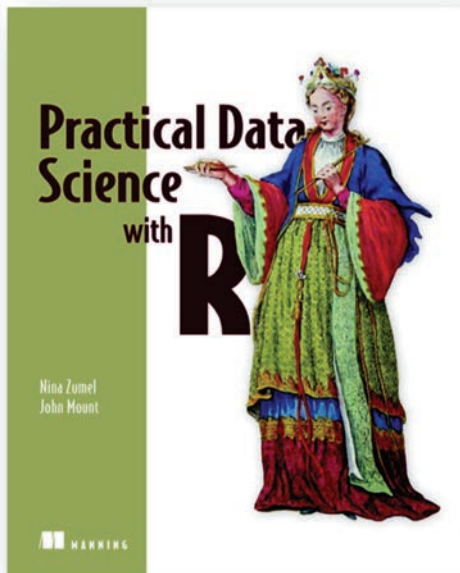
by Grant S. Ingersoll, Thomas S. Morton,  
and Andrew L. Farris

ISBN: 9781933988382

320 pages

\$44.99

December 2012



*Practical Data Science with R*

by Nina Zumel and John Mount

ISBN: 9781617291562

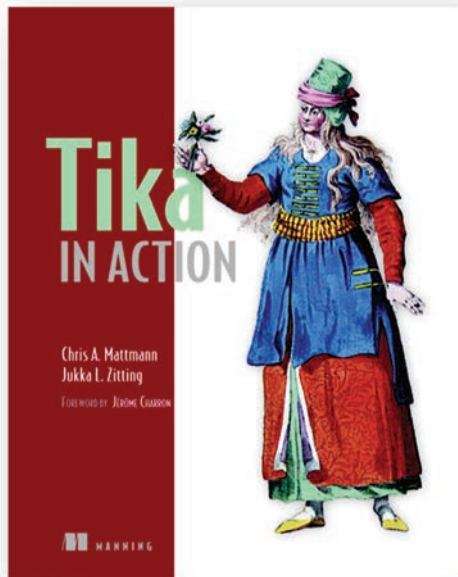
416 pages

\$49.99

March 2014

*For ordering information go to [www.manning.com](http://www.manning.com)*

## MORE TITLES FROM MANNING



### *Tika in Action*

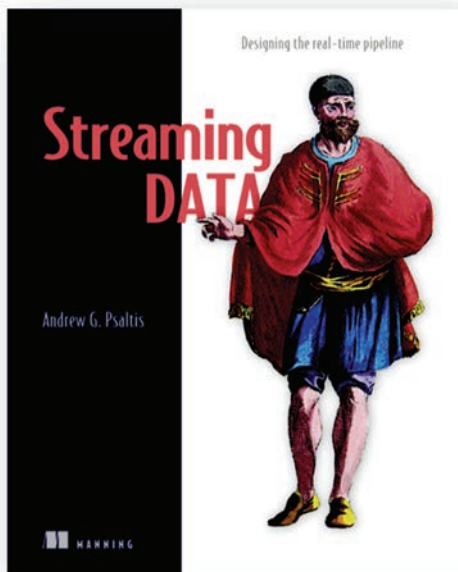
by Chris A. Mattmann  
and Jukka L. Zitting

ISBN: 9781935182856

256 pages

\$44.99

December 2011



### *Streaming Data*

*Designing the real-time pipeline*

by Andrew G. Psaltis

ISBN: 9781617292286

300 pages

\$49.99

February 2017

*For ordering information go to [www.manning.com](http://www.manning.com)*

Licensed to Anant Sahay <[anant.sahay@lexxtechnologies.com](mailto:anant.sahay@lexxtechnologies.com)>

# Relevant SEARCH

Turnbull • Berryman

**U**sers are accustomed to and expect instant, relevant search results. To achieve this, you must master the search engine. Yet for many developers, relevance ranking is mysterious or confusing.

**Relevant Search** demystifies the subject and shows you that a search engine is a programmable relevance framework. Using Elasticsearch and Solr, it teaches you to express your business's ranking rules in this framework. You'll discover how to program relevance and how to incorporate secondary data sources, taxonomies, text analytics, and personalization. In practice, a relevance framework requires softer skills as well, such as collaborating with stakeholders to discover the right relevance requirements for your business. By the end, you'll be able to achieve a virtuous cycle of provable, measurable relevance improvements over a search product's lifetime.

## What's Inside

- Techniques for debugging relevance
- Applying search engine features to real problems
- Using the user interface to guide searchers
- A systematic approach to relevance
- A business culture focused on improving search

For developers trying to build smarter search with Elasticsearch or Solr.

**Doug Turnbull** is lead relevance consultant at OpenSource Connections, where he frequently speaks and blogs.

**John Berryman** is a data engineer at Eventbrite, where he specializes in recommendations and search.

“One of the best and most engaging technical books I’ve ever read.”

—From the Foreword  
by Trey Grainger  
Author of *Solr in Action*

“Will help you solve real-world search relevance problems for Lucene-based search engines.”

—Dimitrios Kouzis-Loukas  
Bloomberg L.P.

“An inspiring book revealing the essence and mechanics of relevant search.”

—Ursin Stauss, Swiss Post

“Arms you with invaluable knowledge to temper the relevancy of search results and harness the powerful features provided by modern search engines.”

—Russ Cam, Elastic



To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit  
[www.manning.com/books/relevant-search](http://www.manning.com/books/relevant-search)

ISBN-13: 978-1-61729-277-4  
ISBN-10: 1-61729-277-X



9 781617 292774