SAMPLE CHAPTER

Solr IN ACTION

Trey Grainger
Timothy Potter

FOREWORD BY Yonik Seeley

MANNING

Solr in Action

by Trey Grainger
Timothy Potter

Chapter 3

brief contents

3

Key Solr concepts

This chapter covers

- What differentiates Solr from traditional database technologies
- The basic structure of Solr's internal index
- How Solr performs complex queries using terms, phrases, and fuzzy matching
- How Solr calculates scores for matching queries to the most relevant documents
- How to balance returning relevant results versus returning all possible results
- How to model your content into denormalized documents
- How Solr scales across servers to handle billions of documents and queries

Now that we have Solr up and running, it's important to gain a basic understanding of how a search engine operates and why you'd choose to use Solr to store and retrieve your content. Our main goal for this chapter is to provide the theoretical underpinnings so you can understand and maximize your use of Solr.

If you have a solid background in search and information retrieval, then you may wish to skip some or all of this chapter, but if not, it will help you understand more advanced topics later in this book and maximize the quality of your users' search experience.

Although the content in this chapter is generally applicable to most search engines, we'll be specifically focusing on Solr's implementation of each of the concepts. By the end of this chapter, you should have a solid understanding of how Solr's internal index works, how to perform complex Boolean and fuzzy queries with Solr, how Solr's default relevancy scoring model works, and how Solr's architecture enables queries to remain fast as it scales to handle billions of documents across many servers.

Let's begin with a discussion of the core concepts behind search in Solr, including how the search index works, how a search engine matches queries and documents, and how Solr enables powerful query capabilities to make finding content a problem of the past.

3.1 Searching, matching, and finding content

Many different kinds of systems exist to help us solve challenging data storage and retrieval problems: relational databases, key-value stores, map-reduce engines operating upon files on disk, and graph databases, among many others. Search engines, and Solr in particular, help to solve a specific class of problem quite well—problems requiring the ability to search across large amounts of unstructured text and pull back the most relevant results.

In this section, we'll describe the core features of a modern search engine, including an explanation of a search "document," an overview of the inverted search index at the core of Solr's fast full-text searching capabilities, and a broad overview of how this inverted search index enables arbitrarily complex term, phrase, and partial-matching queries.

3.1.1 What is a document?

We posted some documents to Solr in chapter 2 and then ran example searches against Solr, so this is not the first time we have mentioned documents. It is important, however, that we have a solid understanding of the kind of information we can put into Solr to be searched upon (a document) and how that information is structured.

Solr is a document storage and retrieval engine. Every piece of data submitted to Solr for processing is a *document*. A document could be a newspaper article, a resume or social profile, or, in an extreme case, an entire book.

Each document contains one or more fields, each of which is modeled as a particular field type: string, tokenized text, Boolean, date/time, lat/long, etc. The number of potential field types is infinite because a field type is composed of zero or more analysis steps that change how the data in the field is processed and mapped into the Solr index. Each field is defined in Solr's schema (discussed in chapter 5) as a particular field type, which allows Solr to know how to handle the content as it's received. Listing 3.1 shows an example document, with the values defined for each field.

Listing 3.1 Example Solr document

```
<doc>
  <field name="id">company123</field>
  <field name="companycity">Atlanta</field>
  <field name="companystate">Georgia</field>
  <field name="companyname">Code Monkeys R Us, LLC</field>
  <field name="companydescription">we write lots of code</field>
  <field name="lastmodified">2013-06-01T15:26:37Z</field>
</doc>
```

When we run a query against Solr, we can search on one or more of these fields (or even fields not contained in this particular document), and Solr will return documents that contain content in those fields matching the query.

It's worth noting that although Solr has a flexible schema for each document, it's not "schema-less." All field types must be defined, and all field names (or dynamic field-naming patterns) should be specified in Solr's *schema.xml*, as we'll discuss further in chapter 5. This does not mean that every document must contain every field, only that all possible fields must be mappable to a particular field type should they appear in a document and need to be processed. Solr *does* contain an ability to automatically guess the field type for previously unseen field names when it first receives a document with a new field name. This is accomplished by inspecting the type of data in the field and automatically adding the field to Solr's schema. Since Solr could potentially guess the wrong field type if the input is confusing, it's a better practice to predefine the field.

A document, then, is a collection of fields that map to particular field types defined in a schema. Each field in a document has its content analyzed according to its field type, and the results of that analysis are saved into a search index in order to later retrieve the document by sending in a related query. The primary search results returned from a Solr query are documents containing one or more fields.

3.1.2 The fundamental search problem

Before we dive into an overview of how search works in Solr, it's helpful to understand what fundamental problem search engines are solving.

Let's say you were tasked with creating search functionality that helps users search for books. Your initial prototype might look something like figure 3.1.

Now imagine that a customer wants to find a book on purchasing a new home and searches for *buying a home*. Some potentially relevant book titles you may want to return for this search are listed in table 3.1.

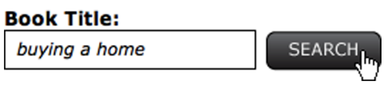


Figure 3.1 Example search interface, as would be seen on a typical website, demonstrating how a user would submit a query to your application

Table 3.1 Books relevant to the query "buying a home"

Potentially relevant books
The Beginner's Guide to Buying a House
How to Buy Your First House
Purchasing a Home
Becoming a New Home Owner
Buying a New Home
Decorating Your Home

All other book titles, as listed in table 3.2, would not be considered relevant for customers interested in purchasing a new home.

Table 3.2 Books not relevant to the query "buying a home"

Irrelevant books
A Fun Guide to Cooking
How to Raise a Child
Buying a New Car

A naïve approach to implementing this search using a traditional SQL database would be to query for the exact text that users enter:

```
SELECT * FROM Books
WHERE Name = 'buying a new home';
```

The problem with this approach is that none of the book titles in your book catalog will match the text that customers type in exactly, which means they will not find any results for this query. In addition, customers will only see results for future queries if the query matches the full book title exactly.

Perhaps a more forgiving approach would be to search for each single word within a customer's query:

```
SELECT * FROM Books
WHERE Name LIKE '%buying%'
  AND Name LIKE '%a%'
  AND Name LIKE '%home%';
```

The previous query, although relatively expensive for a traditional database to handle because it can't use available database indexes, would at least produce one match for the customer that contains all desired words, as shown in table 3.3.

Table 3.3 Results from database `LIKE` query requiring a fuzzy match for every term

Matching books	Nonmatching books
Buying a *New* ***Home***	*The Beginner's Guide to* ***Buying a*** *House*
	How to Buy Your First House
	Purchasing ***a Home***
	Becoming ***a*** *New* ***Home*** *Owner*
	A *Fun Guide to Cooking*
	*How to R****a****ise* ***a*** *Child*
	Buying a *New Car*
	*Decor****a****ting Your* ***Home***

Of course, you may believe that requiring documents to match all of the words your customers include in their queries is overly restrictive. You could easily make the search experience more flexible by only requiring a single word to exist in any matching book titles, by issuing the following SQL query:

```
SELECT * FROM Books
WHERE Name LIKE '%buying%'
   OR Name LIKE '%a%'
   OR Name LIKE '%home%';
```

The results of this query can be seen in table 3.4. You'll see that this query matched many more book titles than the previous query because this query only required a minimum of one of the keywords to match. Additionally, because this query is performing only partial string matching on each keyword, any book title that contains the letter "a" is also returned. The preceding example, which required all of the terms, also matched on the letter "a", but we did not experience this problem of returning too many results because the other keywords were more restrictive.

Table 3.4 Results from database `LIKE` query only requiring a fuzzy match of at least one term

Matching books	Nonmatching books
A *Fun Guide to Cooking*	*How to Buy Your First House*
*Decor****a****ting Your* ***Home***	
*How to R****a****ise* ***a*** *Child*	
Buying a *New Car*	
Buying a *New* ***Home***	
The Beginner's Guide to ***Buying a*** *House*	
*Purch****a****sing* ***a Home***	
Becoming ***a*** *New* ***Home*** *owner*	

The first query (requiring all words to match) resulted in many relevant books not being found; the second query (requiring only one of the words to match) resulted in many more relevant books being found but resulted in many irrelevant books being found as well.

These examples demonstrate several difficulties with this implementation:

- It only performs substring matching and is unable to distinguish between words.
- It doesn't understand linguistic variations, such as "buying" versus "buy."
- It doesn't understand synonyms of words such as "buying" and "purchasing" or "home" and "house."
- Unimportant words such as "a" prevent results from matching as expected (either excluding relevant results or including irrelevant results, depending upon whether "all" or "any" of the words must match).
- There's no sense of relevancy ordering in the results; books that match only one of the queried words often show up higher than books matching multiple or all of the words in the customer's query.

These queries will become slow as the size of the book catalog grows or the number of customer queries grows, because the query must scan through every book's title to find partial matches instead of using an index to look up the words.

Search engines like Solr shine in solving such problems. Solr is able to perform text analysis on content and on search queries to determine textually similar words, understand and match on synonyms, remove unimportant words like "a," "the," and "of," and score each result based upon how well it matches the incoming query to ensure that the best results are returned first and that your customers do not have to page through countless less-relevant results to find the content they were expecting. Solr accomplishes all of this by using an index that maps content to documents instead of mapping documents to content as in a traditional database model. This inverted index is at the heart of how search engines work.

3.1.3 The inverted index

Solr uses Lucene's inverted index to power its fast searching capabilities, as well as many of the additional bells and whistles it provides at query time. While we'll not get into many of the internal Lucene data structures in this book, it's important to understand the high-level structure of the inverted index. (We recommend *Lucene in Action*, Second Edition, by Michael McCandless, Erik Hatcher, and Otis Gospodnetić [Manning, 2010] if you want a deeper dive.)

Recalling our previous book-searching example, we can get a feel for what an index mapping each term to each document would look like from table 3.5.

While a traditional database representation of multiple documents would contain a document's ID mapped to one or more content fields containing all of the words/terms in that document, an inverted index inverts this model and maps each word/term in the corpus to all of the documents in which it appears. You can tell from looking at

Table 3.5 Mapping of text from multiple documents into an inverted index. The right table contains an inverted search index showing each of the terms, along with its position, within the original documents from the left table.

Original documents	
Doc #	**Content field**
1	A Fun Guide to Cooking
2	Decorating Your Home
3	How to Raise a Child
4	Buying a New Car
5	Buying a New Home
6	The Beginner's Guide to Buying a House
7	Purchasing a Home
8	Becoming a New Home Owner
9	How to Buy Your First House

Lucene's inverted index			
Term	**Doc #**	**(Continued)…**	
a	1,3,4,5,6,7,8	…	…
becoming	8	guide	1,6
beginner's	6	home	2,5,7,8
buy	9	house	6,9
buying	4,5,6	how	3,9
car	4	new	4,5,8
child	3	owner	8
cooking	1	purchasing	7
decorating	2	raise	3
first	9	the	6
fun	1	to	1,6,9
…	…	your	2,9

table 3.5 that the original input text was split on spaces and that each term was transformed into lowercase text before being inserted into the inverted index, but everything else remained the same. It is worth noting that many additional text transformations are possible, not only these simple ones; terms can be modified, added, or removed during the content-analysis process, which will be covered in detail in chapter 6.

Two final important details should be noted about the inverted index:

- All terms in the index map to one or more documents.
- Terms in the inverted index are sorted in ascending lexicographical order.

This view of the inverted index is greatly simplified; we'll see in section 3.1.6 that additional information can also be stored in the index to improve Solr's querying and scoring capabilities.

As you'll see in the next section, the structure of Lucene's inverted index allows for many powerful query capabilities that maximize both the speed and the flexibility of keyword-based searching.

3.1.4 Terms, phrases, and Boolean logic

Now that we've seen what content looks like in Lucene's inverted index, let's jump into the mechanics of how a query is able to make use of this index to find matching documents. In this section, we'll go over the basics of looking up terms and phrases in an inverted search index and utilizing Boolean logic and fuzzy queries to enhance these lookup capabilities. Referring back to the book-searching example, let's look at a simple query for *new house*, as portrayed in figure 3.2.

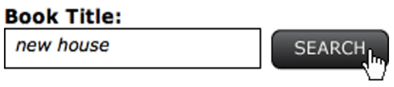


Figure 3.2 Simple search to demonstrate nuances of query interpretation

You saw in the last section that all of the text in the content field was broken up into individual terms when inserted into the Lucene index. Now that there's an incoming query, you need to select from among several options for querying the index:

- Search for two different terms, `new` and `house`, requiring both to match
- Search for two different terms, `new` and `house`, requiring only one to match
- Search for the exact phrase `"new house"`

All of these options are perfectly valid approaches depending upon your use case, and thanks to Solr's powerful querying capabilities, built using Lucene, they're easy to accomplish using Boolean logic.

Required terms

Let's examine the first option, breaking the query into multiple terms and requiring them all to match. There are two identical ways to write this query using the default query parser in Solr:

- `+new +house`
- `new AND house`

These two are logically identical, and, in fact, the second example gets parsed and ultimately reduced down to the first example. The `+` symbol is a unary operator that means that the part of the query immediately following it is required to exist in any documents matched; the `AND` keyword is a binary operator that means that the part of the query immediately preceding and the part of the query immediately following it are both required.

Optional terms

In contrast to the `AND` operator, Solr also supports the `OR` binary operator, which means that either the part of the query preceding or the part of the query following it is required to exist in any documents matched. By default, Solr is also configured to treat any part of the query without an explicit operator as an optional parameter, making the following identical:

- `new house`
- `new OR house`

Negated terms

In addition to making parts of a query optional or required, it's also possible to require that they not exist in any matched documents through either of the following equivalent queries:

- `new house -rental`
- `new house NOT rental`

In those queries, no document that contains the word `rental` will be returned, only documents matching `new` or `house`.

Solr's default operator

While the default configuration in Solr assumes that a term or phrase by itself is an optional term, this is configurable on a per-query basis using the `q.op` URL parameter with many of Solr's query handlers.

`/select/?q=new house&q.op=OR` versus `/select?q=new house&q.op=AND`

Note that if you change the default operator from `OR` to `AND`, this will switch to requiring all terms specified without an explicit Boolean operator. If the default operator is `OR` for the query `new house`, then only one of the terms is required. If the default operator is `AND` for the same query, then both the terms `new` and `house` are required. You can also explicitly specify the operator between the terms (such as `new AND home` or `new OR home`) to override the default operator.

PHRASES

Solr does not only support searching single terms; it can also search for phrases, ensuring that multiple terms appear together in order:

- `"new home" OR "new house"`
- `"3 bedrooms" AND "walk in closet" AND "granite countertops"`

GROUPED EXPRESSIONS

In addition to the preceding query expressions, one final basic Boolean construct that Solr supports is the grouping of terms, phrases, and other query expressions. The Solr query syntax can represent arbitrarily complex queries through grouping terms using parentheses, as in the following examples:

- `New AND (house OR (home NOT improvement NOT depot NOT grown))`
- `(+(buying purchasing -renting) +(home house residence -(+property -bedroom)))`

The use of required terms, optional terms, negated terms, and grouped expressions provides a powerful and flexible set of query capabilities that allow arbitrarily complex lookup operations against the search index, as we'll see in the following section.

3.1.5 *Finding sets of documents*

With a basic understanding of terms, phrases, and Boolean queries in place, we can now dive into exactly how Solr is able to use the internal Lucene inverted index to find matching documents. Recall the index of books from table 3.5, a part of which is reproduced in table 3.6.

Table 3.6 Inverted index of terms from a collection of book titles

Term	Document	(Continued)...	
a	1,3,4,5,6,7,8	…	…
becoming	8	guide	1,6

Table 3.6 Inverted index of terms from a collection of book titles *(continued)*

Term	Document	(Continued)…	
beginner's	6	home	2,5,7,8
buy	9	house	6,9
buying	4,5,6	how	3,9
car	4	new	4,5,8
child	3	owner	8
cooking	1	purchasing	7
decorating	2	raise	3
first	9	the	6
fun	1	to	1,6,9
…	…	your	2,9

If a customer now passes in a query of `new home`, how exactly is Solr able to find documents matching that query, given the preceding inverted index?

The answer is that the query `new home` is a two-term query (there is a default operator between `new` and `home`, remember?). As such, both terms must be looked up separately in the Lucene index:

Term	Document
home	2,5,7,8
new	4,5,8

Once the list of matching documents is found for each term, Lucene will perform set operations to arrive at an appropriate final result set that matches the query. Assuming the default operator is an `OR`, this query would result in a union of the result sets for both terms, as pictured in the Venn diagram in figure 3.3.

Likewise, if the query had been `new AND home` or if the default operator had been set to `AND`, then the intersection of the results for both terms would have been calculated to return a result set of only document 5 and document 8, as shown in figure 3.4.

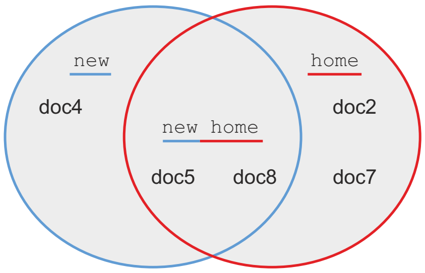


Figure 3.3 Results returned from a union query using the `OR` operator

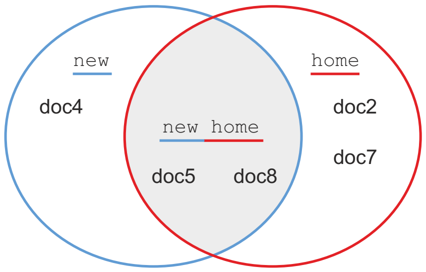


Figure 3.4 Results returned from an intersection query using the AND operator

In addition to union and intersection queries, negating particular terms is also common. Figure 3.5 demonstrates a breakdown of the results expected for many of the result set permutations of this two-term search query (assuming a default OR operator).

As you can see, the ability to search for required terms, optional terms, negated terms, and grouped terms provides a powerful mechanism for looking up single

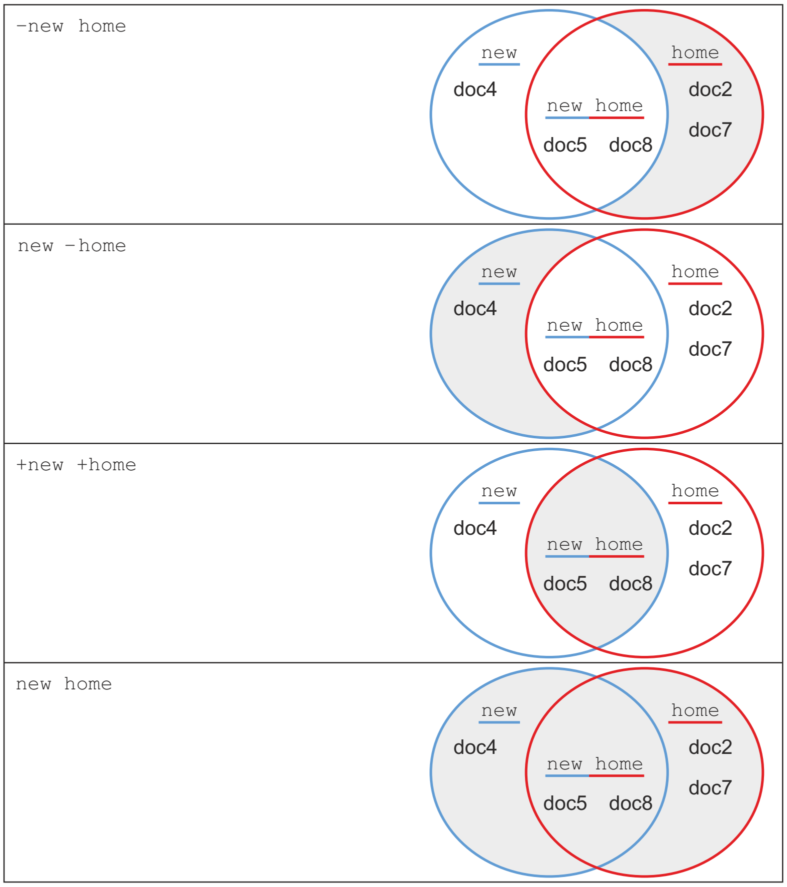


Figure 3.5 Graphical representation of using common Boolean query operators

keywords. As we'll see in the following section, Solr also provides the ability to query for multiterm phrases.

3.1.6 *Phrase queries and term positions*

We saw earlier that, in addition to querying for terms in our Lucene index, it's possible to query Solr for phrases. Recalling that the index contains only individual terms, however, you may be wondering how we can search for full phrases.

The short answer is that each term in a phrase query is still looked up in the Lucene index individually, as if the query `new home` had been submitted instead of `"new home"`. Once the overlapping document set is found, however, a feature of the index that we conveniently left out of our initial inverted index discussion is used. This feature, called *term positions*, is the optional recording of the relative position of terms within a document. Table 3.7 demonstrates how documents (on the left side of the table) map into an inverted index containing term positions (on the right side of the table).

Table 3.7 Inverted index with term positions

Original documents	
Document #	**Content field**
1	A Fun Guide to Cooking
2	Decorating Your Home
3	How to Raise a Child
4	Buying a New Car
5	Buying a New Home
6	The Beginner's Guide to Buying a House
7	Purchasing a Home
8	Becoming a New Home owner
9	How to Buy Your First House

Lucene's inverted index with term positions		
Term	**Document**	**Term position**
a	1	1
	3	4
	4	2
	…	…
cooking	1	5
decorating	2	1
your	2	2
	9	4
home	2	3
	5	4
	7	3
	8	4
…	…	…
new	4	3
	5	3
	8	3
Car	4	4
The	6	1
Beginner's	6	2
House	6	7
	9	6
Purchasing	7	1
…	…	…

From the inverted index in table 3.7, you can see that a query for `new AND home` would yield a result containing documents 5 and 8. The term position goes one step further, telling us where in the document each term appears. Table 3.8 shows a condensed version of the inverted index focused only upon the intersection of the primary terms under discussion: `new` and `home`.

Table 3.8 **Condensed inverted index with term positions**

Term	Document	Term position
home	5	4
	8	4
new	5	3
	8	3

In this example, the term `new` happens to be in position 3 and the term `home` happens to be in position 4 in both matched documents. This makes sense, as the book titles were *Buying a New Home* and *Becoming a New Home Owner.* By ensuring that the matched terms appear within one position of each other, Solr can ensure that the terms formed a phrase in the original document. You have now seen the power of term positions; they allow you to reconstruct the original positions of indexed terms within their respective documents, making it possible to search for specific phrases at query time.

Searching for specific phrases is not the only benefit provided by term positions. We'll see in the next section another great example of their use to improve our search results quality.

3.1.7 Fuzzy matching

It's not always possible to know up front exactly what will be found in the Solr index for any given search, so Solr provides the ability to perform several types of fuzzy-matching queries. *Fuzzy matching* is defined as the ability to perform inexact matches on terms in the search index. For example, someone may want to search for any words that start with a particular prefix (known as *wildcard searching*), may want to find spelling variations within one or two characters (known as *fuzzy searching* or *edit distance searching*), or may want to match two terms within some maximum distance of each other (known as *proximity searching*). For use cases in which multiple variations of the terms or phrases queried may exist across the documents being searched, these fuzzy-matching capabilities serve as a powerful tool.

In this section, we'll explore multiple fuzzy matching query capabilities in Solr, including wildcard searching, range searching, edit-distance searching, and proximity searching.

WILDCARD SEARCHING

One of the most common forms of fuzzy matching in Solr is the use of wildcards. Suppose you want to find any documents that start with the letters `offic`. One way to do this is to create a query that enumerates all of the possible variations:

- *Query:* `office OR officer OR official OR officiate OR …`

  Requiring that this list of words be turned into a query up front can be an unreasonable expectation for customers, or even for you on behalf of your customers.

Because all of the variations you could match already exist in the Solr index, you can use the asterisk (`*`) wildcard character to perform this same function for you:

- *Query:* `offi*` Matches office, officer, official, and so on

In addition to matching the end of a term, a wildcard character can be used inside of the search term as well, such as if you wanted to match both `officer` and `offer`:

- *Query:* `off*r` Matches offer, officer, officiator, and so on

The asterisk wildcard (`*`) matches zero or more characters in a term. If you want to match only a single character, you can make use of the question mark (`?`) for this purpose:

- *Query:* `off?r` Matches offer, but not officer

Leading wildcards

While the wildcard functionality in Solr is fairly robust, it can be expensive to execute certain wildcard queries. Whenever a wildcard search is executed, all of the terms in the inverted index that match the parts of the term prior to the first wildcard must be found. Then, each of those candidate terms must be inspected to see if they match the wildcard pattern in the query. Because of this, the more characters you specify at the beginning of the term before the wildcard, the faster the query should run. For example, the query `engineer*` will not be expensive (because it matches few terms in the inverted index), but the query `e*` will be expensive, as it matches all terms beginning with the letter *e*.

Executing a leading wildcard query is an expensive operation. If you needed to match all terms ending in *ing* (like caring, liking, and smiling), for example, this could cause major performance issues.

- *Query:* `*ing`

If you need to be able to search using these leading wildcards, a faster solution exists, but it requires additional configuration. The solution is achieved by adding `ReversedWildcardFilterFactory` to your field type's analysis chain (configuring text processing will be discussed in chapter 6).

`ReversedWildcardFilterFactory` works by double-inserting the indexed content in the Solr index (once for the text of each term, and once for the reversed text of each term):

- *Index:*

```
caring      liking       smiling
#gnirac     #gnikil      #gnilims
```

When a query is submitted with the leading wildcard of `*ing`, Solr knows to search on the reversed version, getting around the performance issues associated with leading wildcard searches by turning them into standard wildcard searches on the reversed content.

Note, however, that turning this feature on requires dual-indexing all terms in the Solr index, increasing the size of the index and slowing down overall searches. Turning this capability on is not recommended unless it's needed within your search application.

One last important point to note about wildcard searching is that wildcards are only meant to work on individual search terms, not on phrase searches, as demonstrated by the following example:

- *Works:* `softwar* eng?neering`
- *Does not work:* `"softwar* eng?neering"`

If you need the ability to perform wildcard searches within a phrase, you will have to store the entire phrase in the index as a single term, which you should feel comfortable doing by the end of chapter 6.

RANGE SEARCHING

Solr also provides the ability to search for terms that fall between known values. This can be useful when you want to search for a particular subset of documents falling within a range. For example, if you only wanted to match documents created in the six months between February 2, 2012, and August 2, 2012, you could perform the following search:

- *Query:* `created:[2012-02-01T00:00.0Z TO 2012-08-02T00:00.0Z]`

This range `query` format also works on other field types:

- *Query:* `yearsOld:[18 TO 21]` Matches 18, 19, 20, 21
- *Query:* `title:[boat TO boulder]` Matches boat, boil, book, boulder, etc.
- *Query:* `price:[12.99 TO 14.99]` Matches 12.99, 13.000009, 14.99, etc.

Each of these range queries surrounds the range with square brackets, which is the "inclusive" range syntax. Solr also supports exclusive range searching through the use of curly braces:

- *Query:* `yearsOld:{18 TO 21}` Matches 19 and 20 but not 18 or 21

Though it may look odd syntactically, Solr also provides the ability to mix and match inclusive and exclusive bounds:

- Query: `yearsOld:[18 TO 21}` Matches 18, 19, 20, but not 21

While range searches perform more slowly than searches on a single term, they provide tremendous flexibility for finding documents matching dynamically defined groups of values that lie within a particular range within the Solr index. It's important to note that the ordering of terms for range queries is exactly that: the order in which they are found in the Solr index, which is a lexicographically sorted order. If you were to create a text field containing integers, those integers would be found in the following order: 1, 11, 111, 12, 120, 13, etc. Numeric types in Solr, at least the ones we'll recommend in the coming chapters, compensate for this by indexing the incoming content in a special way, but it's important to understand that the sort order within the Solr index is dependent upon how the data within the field is processed when it's written to the Solr index. We'll dive much deeper into this kind of content analysis in chapters 5 and 6.

Fuzzy/edit-distance searching

For many search applications, it's important not only to match a customer's text exactly, but also to allow flexibility for handling spelling errors or even slight variations in correct spellings. Solr provides the ability to handle character variations using edit-distance measurements based upon Damerau-Levenshtein distances, which account for more than 80% of all human misspellings.[1]

Solr achieves these fuzzy edit-distance searches through the use of the tilde (~) character as follows:

- *Query:* `administrator~` Matches: adminstrator, administrater, administratior, and so forth

This query matches both the original term (`administrator`) and any other terms within two edit distances of the original term. An *edit distance* is defined as an insertion, a deletion, a substitution, or a transposition of characters. The term `adminstrator` (missing the "i" in the sixth position) is one edit distance away from administrator because it has one character deletion. Likewise the term `sadministrator` would be one edit distance away because it has one insertion (the "s" that was prepended), and the term `administratro` would also be one edit distance away because it has transposed the last two characters ("or" became "ro").

It's also possible to modify the strictness of edit-distance searches to allow matching of terms with any edit distance:

- *Query:* `administrator~1` Matches within one edit distance.
- *Query:* `administrator~2` Matches within two edit distances. (This is the default if no edit distance is provided.)
- *Query:* `administrator~N` Matches within `N` edit distances.

Please note that any edit distances requested above two will become increasingly slower and will be more likely to match unexpected terms. Term searches with edit distances of one or two are performed using an efficient Levenshtein automaton, but will fall back to a slower edit-distance implementation for edit distances above two.

Proximity searching

In the previous section, we saw that edit distances could be used to find terms that were close to the original term, but not exactly the same. This edit-distance principle is applicable beyond searching for alternate characters within a term; it can also be applied between terms for variations of phrases.

Let's say that you want to search across a Solr index of employee profiles for executives within your company. One way to do this would be to enumerate each of the possible executive titles within your company:

- *Query:* `"chief executive officer" OR "chief financial officer" OR "chief marketing officer" OR "chief technology officer" OR …`

[1] Fred J. Damerau, "A Technique for Computer Detection and Correction of Spelling Errors," *Communications of the ACM*, 7(3):171-176 (1964).

Of course, this assumes you know all of the possible titles, which may be unrealistic if you're searching across other companies with which you're poorly acquainted or if you have a more challenging use case. Another possible strategy is to search for each term independently:

- *Query:* `chief AND officer`

This should match all of the possible use cases, but it will also match any document that contains both of those words anywhere in the document. One problematic example would be a document containing the text: `One chief concern arising from the incident was the safety of the police officer on duty`. This document is clearly a poor match for our use case, but it and similar bad matches would be returned given the preceding query.

Thankfully, Solr provides a basic solution to this problem: proximity searching. In the previous example, a good strategy would be to ask Solr to bring back all documents that contain the term `chief` near the term `officer`. This can be accomplished through the following example queries:

- *Query:* `"chief officer"~1`
  - *Meaning:* `chief` and `officer` must be a maximum of one position away.
  - *Examples:* `"chief executive officer"`, `"chief financial officer"`
- *Query:* `"chief officer"~2`
  - *Meaning:* `chief` and `officer` must be a maximum of two edit distances away.
  - *Examples:* `"chief business development officer"`, `"officer chief"`
- *Query:* `"chief officer"~N`
  - *Meaning:* Finds `chief` within `N` positions of `officer`.

The preceding proximity searches can be seen as "sloppy" versions of traditional phrase searches. In fact, an exact phrase search of `"chief development officer"` could easily be rewritten as `"chief development officer"~0`. These queries will yield the same results, because an edit distance of zero is the definition of an exact phrase search. Both mechanisms make use of the term positions stored in the Solr index (which we discussed in section 3.1.6) to calculate the edit distances. It should also be noted that Solr's proximity searching is not a true use of edit distance because it requires all specified terms to exist, whereas a true edit distance would also allow for substitutions and deletions (as you saw with fuzzy searching on a single term).

The general principle of an edit distance still applies to term proximity queries with regard to term insertions and transpositions, however. Along this line, you may have also noticed that it required a phrase slop of `2` to be specified (`"chief officer"~2`) in order to match the text `officer chief`. This is because the first edit is to move the terms `chief` and `officer` into the same position; the second edit is to move `chief` one more position to come before `officer`. This again underscores the fact that proximity searching does not use a true edit distance (where a transposition may only count as one edit), but is instead asking: "How many positions can be collectively added to a document's text in order to form the exact phrase specified for the proximity search?"

3.1.8 Quick recap

At this point, you should have a basic grasp of how Solr stores information in its inverted index and queries that index to find matching documents. This includes looking up terms, using Boolean logic to create arbitrarily complex queries, and getting results back as a result of the set operations using each of the term lookups. We also discussed how Solr stores term positions and is able to use those to find exact phrases and even fuzzy phrase matches through the use of proximity queries and positional calculations. For fuzzy searching within single terms, we examined the use of wildcards and edit-distance searching to find misspellings or similar words. While Solr's query capabilities will be expanded upon in chapter 7, these key operations serve as the foundation for generating most Solr queries. They also prepare us nicely with the needed background for our discussion of Solr's keyword relevancy scoring model in the next section.

3.2 Relevancy

Finding matching documents is the first critical step in creating a great search experience, but it's only the first step. Most customers aren't willing to wade through page after page of search results to find the documents they're seeking. In our general experience, only 10% of customers are willing to go beyond the first page of any given search on most websites, and only 1% are willing to navigate to the third page.

Solr does a good job out of the box of ensuring that the ordering of search results brings back the best matches at the top of the results list. It does this by calculating a relevancy score for each document and then sorting the search results from the highest score to the lowest. This section will provide an overview of how these relevancy scores are calculated and what factors influence them. We'll dig into both the theory behind Solr's default relevancy calculation and the specific calculations used to compute the relevancy scores, providing intuitive examples along the way to ensure you leave this section with a solid understanding of what, to many, can be the most elusive aspect of working with Solr. We'll start by discussing the `Similarity` class, which is responsible for most aspects of a query's relevancy score calculation.

3.2.1 Default similarity

Solr's relevancy scores are based upon the `Similarity` class, which can be defined on a per-field basis in Solr's *schema.xml* (discussed further in chapter 5). `Similarity` is a Java class that defines how a relevancy score is calculated based upon the results of a query. While you can choose from multiple `Similarity` classes, or even write your own, it is important to understand Solr's default `Similarity` implementation and the theory behind why it works so well.

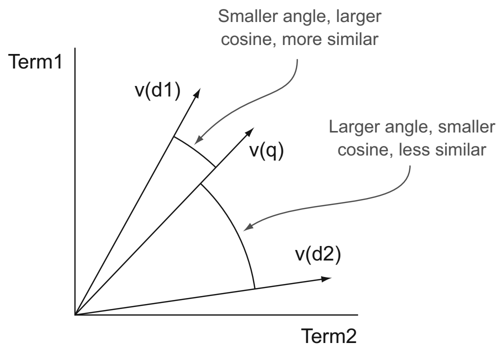

Smaller angle, larger cosine, more similar
Term1
v(d1)
Larger angle, smaller cosine, less similar
v(q)
v(d2)
Term2

Figure 3.6 Cosine similarity of term vectors. The query term vector, v(q), is closer to the document 1 term vector v(d1) than the document 2 term vector v(d2), as measured by the cosine of the angle between each document's term vector and the query vector. The smaller the angle between the query term vector and a document's term vector, the more similar the query and the document are considered to be.

By default, Solr uses Lucene's (appropriately named) `DefaultSimilarity` class, which uses a two-pass model to calculate similarity. First, it makes use of a Boolean model (described in section 3.1) to filter out any documents that do not match the customer's query. Then it uses a vector space model for scoring and drawing the query as a vector, as well as an additional vector for each document. The similarity score for each document is based upon the cosine between the query vector and that document's vector, as depicted in figure 3.6.

In this vector space scoring model, a term vector is calculated for each document and is compared with the corresponding term vector for the query. The similarity of two vectors can be found by calculating a cosine between them, with a cosine of 1 being a perfect match and a cosine of 0 representing no similarity. More intuitively, the closer the two vectors appear together, as in figure 3.6, the more similar they are. The smaller the angle between vectors or the larger the cosine, the closer the match.

Of course, the most challenging part of this whole process is coming up with reasonable vectors that represent the important features of the query and of each document for comparison. Let's look at the entire, complicated relevancy formula for the `DefaultSimilarity` class. We'll then go line by line to explain intuitively what each component of the relevancy formula is attempting to accomplish.

Given a query (`q`) and a document (`d`), the similarity score for the document to the query can be calculated as shown in figure 3.7.

Wow! That equation can be quite overwhelming, particularly at first glance. Fortunately, it's much more intuitive when each of the pieces is broken down. The math is presented for reference, but you will likely never need to dig into the full equation unless you decide to overwrite the `Similarity` class for your search application.

The important concepts in the relevancy calculation are demonstrated as pieces of the high-level formula represented in figure 3.7: term frequency (`tf`), inverse document frequency (`idf`), term boosts (`t.getBoost`), field normalization (`norm`), coordination factor (`coord`), and query normalization (`queryNorm`). Let's dive into the purpose of each of these.

Score(q,d) =

$$\sum_{t \text{ in } q} \left(\textbf{tf}(t \text{ in } d) \cdot \textbf{idf}(t)^2 \cdot \textbf{t.getBoost}() \cdot \textbf{norm}(t,d) \right) \cdot \textbf{coord}(q,d) \cdot \textbf{queryNorm}(q)$$

Where:

t = term; **d** = document; **q** = query; **f** = field

tf(t in d) = $\text{numTermOccurrencesInDocument}^{1/2}$

idf(t) = 1 + log (numDocs / (docFreq +1))

coord(q,d) = numTermsInDocumentFromQuery / numTermsInQuery

queryNorm(q) = $1 / (\text{sumOfSquaredWeights}^{1/2})$

$$\textbf{sumOfSquaredWeights} = \text{q.getBoost}()^2 \cdot \sum_{t \text{ in } q} (\, \text{idf}(t) \cdot \text{t.getBoost}() \,)^2$$

norm(t,d) = d.getBoost() • lengthNorm(f) • f.getBoost()

Figure 3.7 `DefaultSimilarity` scoring algorithm. Each component in this formula will be explained in detail in the following sections.

3.2.2 Term frequency

Term frequency (tf) is a measure of how often a particular term appears in a matching document, and it's an indication of how "well" a document matches the term.

If you were searching through a search index filled with newspaper articles for an article on the President of the United States, would you prefer to find articles that only mention the president once, or articles that discuss the president consistently throughout the article? What if an article happens to contain the phrases `President` and `United States` each one time (perhaps out of context); should it be considered as relevant as an article that contains these phrases multiple times?

In table 3.9, clearly the second article discussed is more relevant than the first, and the identification of the phrases `President` and `United States` multiple times throughout the article provides a strong indication that the content of the second article is more closely related to this query.

Table 3.9 Documents mentioning `President` and `United States`

Article 1 (less relevant)	Article 2 (more relevant)
Dr. Kohrt is the interim `president` of Furman University, one of the top liberal arts universities in the southern `United States`. In 2011, Furman was ranked the 2nd most rigorous college in the country by Newsweek magazine, behind St. John's College (NM). Furman also consistently ranks among the most beautiful campuses to visit and ranks among the top 50 liberal arts colleges nation-wide each year.	Today, international leaders met with the `President` of the `United States` to discuss options for dealing with growing instability in global financial markets. `President` Obama indicated that the `United States` is cautiously optimistic about the potential for significant improvements in several struggling world economies pending the results of upcoming elections. The `President` indicated that the `United States` will take whatever actions necessary to promote continued stability in the global financial markets.

In general, a document is considered to be more relevant for a particular topic (or query term) if the topic appears multiple times.

This is the basic premise behind the `tf` component of the default Solr relevancy formula. The more times the search term appears within a document, the more relevant that document is considered. It's not likely to be the case that 10 appearances of a term make the document 10 times more relevant, however, so tf is calculated using the square root of the number of times the search term appears within the document, in order to diminish the additional contribution to the relevancy score for each subsequent appearance of the search term.

3.2.3 Inverse document frequency

Not all search terms are created equal. Imagine if someone were to search a library catalog for *The Cat in the Hat* by Dr. Seuss, and the top results returned were those that included a high term frequency for the words `the` and `in` instead of `cat` and `hat`. Common sense would indicate that words that are more rare across all documents are likely to be better matches for a query than terms that are more common.

Inverse document frequency (idf), a measure of how "rare" a search term is, is calculated by finding the document frequency (how many total documents the search term appears within), and calculating its inverse (see the full formula in figure 3.7 for the calculation).

Because idf appears for the term in both the query and the document, it's squared in the relevancy formula.

Figure 3.8 shows a visual example of the "rareness" of each of the words in the title *The Cat in the Hat* (relative to a generic collection of library books), with a higher idf being represented as a larger term.

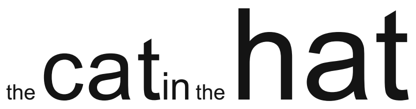


Figure 3.8 Visual depiction of the relative significance of terms as measured by idf. The terms that are rarer are depicted as larger, indicating a larger inverse document frequency.

Likewise, if someone were searching for a profile for `an experienced Solr development team lead` across a large collection of resumes, we wouldn't expect documents to rank higher that best match the words `an`, `team`, or `experienced`. Instead, we would expect the important terms to resemble the largest terms in figure 3.9.

Clearly the user is looking for someone who knows Solr and can be a team lead, so these terms stand out with considerably more weight when found in any document.

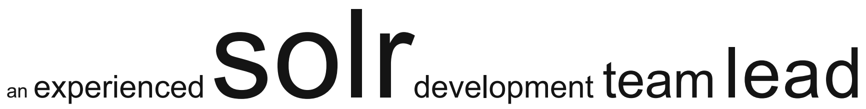


Figure 3.9 Another demonstration of relative score of terms derived from idf. Once again, a higher idf indicates a rarer and more relevant term, depicted here using larger text.

Term frequency and inverse document frequency, when multiplied together in the relevancy calculation, provide a nice counterbalance. The term frequency elevates terms that appear multiple times within a document, whereas the inverse document frequency penalizes those terms that appear commonly across many documents. Therefore, common words in the English language such as `the`, `an`, and `of` ultimately yield low scores, even though they may appear many times in any given document.

3.2.4 Boosting

It is not necessary to leave all aspects of your relevancy calculations up to Solr. If you have domain knowledge about your content—you know that certain fields or terms are more (or less) important than others—you can supply boosts at either indexing time or query time to ensure that the weights of those fields or terms are adjusted accordingly.

Query-time boosting, the most flexible and easiest to understand form of boosting, uses the following syntax:

- *Query:* `title:(solr in action)^2.5 description:(solr in action)`

This example provides a boost of 2.5 to the search phrase in the title field, while providing the default boost of 1.0 to the description field. Unless otherwise specified, all terms receive a default boost of 1.0 (which means multiplying the calculated score by 1, or leaving it as originally calculated).

Query boosts can also be used to penalize certain terms if a boost of less than 1.0 is used:

- *Query:* `title:(solr in action) description:(solr in action)^0.2`

Note that a boost of less than 1 is still a positive boost. It doesn't penalize the document in absolute terms; it boosts the term less than the normal boost of 1 that it otherwise would have received.

These query-time boosts can be applied to any part of the query:

- *Query:* `title:(solr^2 in^.01 action^1.5)^3 OR "solr in action"^2.5`

Certain query parsers even allow boosts to be applied to an entire field by default, which we'll cover further in chapter 7.

In addition to query-time boosting, it's possible to boost documents or fields within documents at index time. These boosts are factored into the field norm, which is covered in the following section.

3.2.5 Normalization factors

The default Solr relevancy formula calculates three kinds of normalization factors (norms): field norms, query norms, and the coord factor.

Field norms

The *field normalization factor* (field norm) is a combination of factors describing the importance of a particular field on a per-document basis. Field norms are calculated at index time and are represented as an additional byte per field in the Solr index.

norm(t,d) = d.getBoost() • lengthNorm(f) • f.getBoost()

Figure 3.10 Field norms calculation. Field norms combine the matching document's boost, the matching field's boost, and a length-normalization factor that penalizes longer documents. These three fairly separate pieces of data are stored as a single byte in the Solr index, which is the only reason they are combined into this single `field norms` variable.

This byte packs a lot of information: the boost set on the document when indexed, the boost set on the field when indexed, and a length normalization factor that penalizes longer documents and helps shorter documents (under the assumption that finding any given keyword in a longer document is more likely and therefore less relevant). The field norms are calculated using the formula in figure 3.10.

The `d.getBoost()` component represents the boost applied to the document when it's sent to Solr, and the `f.getBoost()` component represents the boost applied to the field for which the norm is being calculated. It's worth mentioning that Solr allows the same field to be added to a document multiple times (performing some magic under the covers to map each separate entry for the field into the same underlying Lucene field). Because duplicate fields are ultimately mapped to the same underlying field, if multiple copies of the field exist, `f.getBoost()` becomes the product of the field boost for each of the multiple fields with the same name.

If the `title` field were added to a document three times, for example, once with a boost of 3, once with a boost of 1, and once with a boost of 0.5, `f.getBoost()` for each of the three fields (or the one underlying field) would be

- *Boost:* $(3) \cdot (1) \cdot (0.5) = 1.5$

In addition to the index-time boosts, a parameter called the length norm is factored into the field norm. The length norm is computed by taking the square root of the number of terms in the field for which it is calculated.

It is also worth mentioning that document boosts are internally implemented as a boost on every field of the document. In other words, there is no difference between applying a boost to a document versus applying the same boost individually to every field, as all document boosts are ultimately stored per field for each document inside the field norm.

The purpose of the length norm is to adjust for documents of varying lengths, such that longer documents don't maintain an unfair advantage by having a larger likelihood of containing any particular term a given number of times.

Let's say that you perform a search for the keyword `Beijing`. Would you prefer for a news article to come up that mentions `Beijing` five times, or would you rather have an obscure, 300-page book come back that also happens to mention `Beijing` only five times. Common sense would indicate that a document in which `Beijing` is proportionally more prevalent is probably a better match, everything else being equal. This is what the length norm attempts to take into account.

The overall field norm, calculated from the product of the document boost, the field boost, and the length norm, is encoded into a single byte that's stored in the Solr index. Because the amount of information being encoded from this product is larger than a single byte can store, some precision loss does occur during this encoding. In reality, this loss of fidelity generally has negligible effects on overall relevancy, as it's usually only big differences that matter given the variance in all other relevancy criteria.

QUERY NORMS

The *query norm* is one of the least interesting factors in the default Solr relevancy calculation. It does not affect the overall relevancy ordering, as the same `queryNorm` is applied to all documents. It merely serves as a normalization factor to attempt to make scores between queries comparable. It uses the sum of the squared weights for each of the query terms to generate this factor, which is multiplied with the rest of the relevancy score to normalize it. The query norm should not affect the relative weighting of each document that matches a given query.

THE COORD FACTOR

One final normalization factor taken into account in the default Solr relevancy calculation is the *coord factor*. Its role is to measure how much of the query each document matches. Let's say you perform the following search:

- *Query:* `Accountant AND ("San Francisco" OR "New York" OR "Paris")`

You may prefer to find an accountant with offices in each of the cities you mentioned as opposed to an accountant who has happened to mention "New York" over and over again.

If all four of these terms match, the coord factor is 4/4. If three match, the coord factor is 3/4, and if only one matches, it's 1/4.

The idea behind the coord factor is that, all things being equal, documents that contain more of the terms in the query should score higher than documents that only match a few.

We have now discussed all of the major components of the default relevancy algorithm in Solr. We discussed tf and idf, the two most key components of the relevancy score calculation. We then went through boosting and normalization factors, which refine the scores calculated by tf and idf alone. With a solid conceptual understanding and a detailed overview of the specific components of the relevancy scoring formula, we're now set to discuss Precision and Recall, two important aspects for measuring the overall quality of the result sets returned from any search system.

3.3 Precision and Recall

The information retrieval concepts of *Precision* (a measure of accuracy) and *Recall* (a measure of thoroughness) are simple to explain, but are also important to understand when building any search application or understanding why the results being returned are not meeting your business requirements. We'll provide a brief summary here of each of these key concepts.

3.3.1 Precision

The Precision of a search results set (the documents that match a query) is a measurement attempting to answer the question, "Were the documents that came back the ones I was looking for?"

More technically, Precision is defined as (between 0.0 and 1.0)

```
# Correct Matches / # Total Results Returned
```

Let's return to our example from section 3.1 about searching for a book on the topic of buying a new home. We've determined by our internal company measurements that the books in table 3.10 would be considered good matches for such a query.

Table 3.10 List of relevant books

	Relevant books
1	*The Beginner's Guide to Buying a House*
2	*How to Buy Your First House*
3	*Purchasing a Home*

All other book titles, for the purposes of this example, would not be considered relevant for someone interested in purchasing a new home. A few examples are listed in table 3.11.

Table 3.11 List of irrelevant books

	Irrelevant books
4	*A Fun Guide to Cooking*
5	*How to Raise a Child*
6	*Buying a New Car*

For this example, if all of the documents that were supposed to be returned (documents 1, 2, and 3) were returned, and no more, the Precision of this query would be 1.0 (`3 Correct Matches / 3 Total Matches`), which would be perfect.

If, however, all six results came back, the Precision would only be 0.5, because half of the results that were returned were not correct; that is, they were not precise.

Likewise, if only one result came back from the relevant list (number 2, for example), the Precision would still be 1.0, because all of the results that came back were correct. As you can see, Precision is a measure of how "good" each of the results of a query is, but it pays no attention to how thorough it is; a query that returns one single correct document out of a million other correct documents is still considered perfectly precise.

Because Precision only considers the overall accuracy of the results that come back and not the comprehensiveness of the result set, we need to counterbalance the Precision measurement with one that takes thoroughness into account: Recall.

3.3.2 Recall

Whereas Precision measures how correct each of the results being returned is, Recall is a measure of how thorough the search results are. Recall is answering the question: "How many of the correct documents were returned?"

More technically, Recall is defined as

```
# Correct Matches / (# Correct Matches + # Missed Matches)
```

To demonstrate an example of the Recall calculation, the example showing relevant books and irrelevant books from the last section has been recreated in table 3.12 for reference.

Table 3.12 List of relevant and irrelevant books

Relevant books		Irrelevant books	
1	*The Beginner's Guide to Buying a House*	4	*A Fun Guide to Cooking*
2	*How to Buy Your First House*	5	*How to Raise a Child*
3	*Purchasing a Home*	6	*Buying a New Car*

If all six documents were returned for a search query, the Recall would be 1 because all correct matches were found and there were no missed matches (whereas we saw earlier that the Precision would be 0.5).

Likewise, if only document 1 were returned, the Recall would only be 1/3, because two of the documents that should have been returned/recalled were missing.

This highlights the critical difference between Precision and Recall: Precision is high when the results returned are correct; Recall is high when the correct results are present. Recall does not care that *all* of the results are correct. Precision does not care that *all* of the results are present.

In the next section, we'll talk about strategies for striking an appropriate balance between Precision and Recall.

3.3.3 Striking the right balance

Though there is clearly tension between the two, Precision and Recall are not mutually exclusive. In the previous example in which the query only returns documents 1, 2, and 3, the Precision and Recall are both 1.0, because all of the results were correct and all of the correct results were found.

Maximizing for full Precision and full Recall is the ultimate goal of most every search-relevancy-tuning endeavor. With a contrived example (or a hand-tuned set of results), this seems easy, but in reality, this is a challenging problem.

Many techniques can be undertaken within Solr to improve either Precision or Recall, though most are geared more toward increasing Recall in terms of the full document set being returned. Aggressive textual analysis (to find multiple variations of words) is a great example of trying to find more matches, though these additional matches may hurt overall Precision if the textual analysis is so aggressive that it matches incorrect word variations.

One common way to approach the Precision versus Recall problem in Solr is to attempt to solve for both: measuring for Recall across the entire result set and measuring for Precision only within the first page (or few pages) of search results. Following this model, better matches will be boosted to the top of the search results based upon how well you tune your use of Solr's relevancy scoring calculations, but you will also find that many poorer matches appear at the bottom of your search results list if you go to the last page of the search results.

This is only one way to approach the problem, however. Because many search websites, for example, want to appear to have as much content as possible, and because those sites know that visitors will never go beyond the first few pages, they can show precise results on the first few pages while still including many less precise matches on subsequent pages. This results in a high Recall score across the entire result set by being lenient about which keywords are able to match the initial query. Simultaneously, the Precision of the first page or two of results is still high due to the elevation of the best matches to the top of the long list of search results.

The decision on how to best balance Precision and Recall is ultimately dependent upon your use case. In scenarios like legal discovery, there's a heavy emphasis placed on Recall, as there are legal ramifications if any documents are missed. For other use cases, the requirement may only be to find a few great matches and find nothing that does not exactly match every term within the query.

Most search applications fall somewhere between these two extremes, and striking the right balance between Precision and Recall is a never-ending challenge: mostly because there is often no one right answer. Regardless, understanding the concepts of Precision and Recall and why changes you make swing you more toward one of these two conceptual goals (and likely away from the other) is critical to effectively improving the quality of your search results. Chapter 16 is dedicated to mastering relevancy, so you can be sure you will see this tension between Precision and Recall surface again.

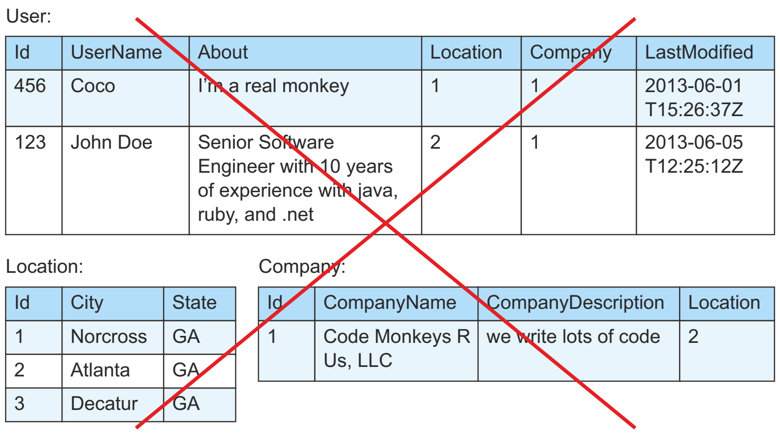
3.4 Searching at scale

One of the most appealing aspects of Solr, beyond its speed, relevancy, and powerful text-searching features, is how well it scales. Solr is able to scale to handle billions of documents and an infinite number of queries by adding servers. Chapters 12 and 13 will provide an in-depth overview of scaling Solr in production, but this section will lay the groundwork for how to think about the necessary characteristics for operating a scalable search engine. Specifically, we'll discuss the nature of Solr documents as denormalized documents and why this enables linear scaling across servers, how distributed searching works, the conceptual shift from thinking about servers to thinking about clusters of servers, and some of the limits of scaling Solr.

3.4.1 The denormalized document

Central to Solr is the concept of all documents being denormalized. A *denormalized document* is one in which all fields are self-contained within the document, even if the values in those fields are duplicated across many documents. This concept of denormalized data is common to many NoSQL technologies. A good example of denormalization is a user-profile document having `city`, `state`, and `postalCode` fields, even though in most cases the `city` and `state` fields will be the same across all documents for each unique `postalCode` value. This is in contrast to a *normalized document* in which relationships between parts of the document may be broken up into multiple smaller documents, the pieces of which can be joined back together at query time. A normalized document would only have a `postalCode` field, and a separate location document would exist for each unique `postalCode` so that the `city` and `state` would not need to be duplicated on each user-profile document. If you have any training whatsoever in building normalized tables for relational databases, please leave that training at the door when thinking about modeling content into Solr. Figure 3.11 demonstrates a traditional normalized database table model, with a big "X" over it to make it obvious that this is not the kind of data-modeling strategy you will use with Solr.

Notice that the information in figure 3.11 represents two users working at a company called "Code Monkeys R Us, LLC." While this figure shows the data nicely normalized into separate tables for the employees' personal information, location, and company, this is not how we would represent these users in a Solr document. Listing 3.2 shows the denormalized representation for each of these employees as mapped to a Solr document.



User:

Id	UserName	About	Location	Company	LastModified
456	Coco	I'm a real monkey	1	1	2013-06-01 T15:26:37Z
123	John Doe	Senior Software Engineer with 10 years of experience with java, ruby, and .net	2	1	2013-06-05 T12:25:12Z

Location:

Id	City	State
1	Norcross	GA
2	Atlanta	GA
3	Decatur	GA

Company:

Id	CompanyName	CompanyDescription	Location
1	Code Monkeys R Us, LLC	we write lots of code	2

Figure 3.11 Solr documents don't follow the traditional normalized model of a relational database. This figure demonstrates how *not* to think of Solr documents. Instead of thinking in terms of multiple entities with relationship to each other, a Solr document is modeled as a flat, denormalized data structure, as shown in listing 3.2.

Listing 3.2 Two denormalized user documents

```
<doc>
  <field name="id">123</field>
  <field name="username">John Doe</field>
  <field name="about">Senior Software Engineer with 10 years of
                      experience with java, ruby, and .net
  </field>
  <field name="usercity">Atlanta</field>
  <field name="userstate">Georgia</field>
  <field name="companyname">Code Monkeys R Us, LLC</field>
  <field name="companydescription">we write lots of code</field>
  <field name="companycity">Decatur</field>
  <field name="companystate">Georgia</field>
  <field name="lastmodified">2013-06-05T12:25:12Z</field>
</doc>

<doc>
  <field name="id">456</field>
  <field name="username">Coco</field>
  <field name="about">I'm a real monkey</field>
  <field name="usercity">Norcross</field>
  <field name="userstate">Georgia</field>
  <field name="companyname">Code Monkeys R Us, LLC</field>
  <field name="companydescription">we write lots of code</field>
  <field name="companycity">Decatur</field>
  <field name="companystate">Georgia</field>
  <field name="lastmodified">2013-06-01T15:26:37Z</field>
</doc>
```

1 Company information for first user.

2 The same company information repeated for the second user.

Notice that all of the company information is repeated in both the first **1** and second **2** user documents, which seems to go against the principles of normalized database design for reducing data redundancy and minimizing data dependency. In a traditional relational database, a query can be constructed that will join data from multiple tables when resolving a query. Although some basic join functionality does now exist in Solr (which will be discussed in chapter 15), it's only recommended for cases in which it's impractical to denormalize content. Solr knows about terms that map to documents but does not natively know about any relationships between documents. That is, if you wanted to search for all users (in the previous example) who work for companies in Decatur, GA, you would need to ensure that the `companycity` and `companystate` fields are populated for all of the users for that lookup to be successful.

While this denormalized document data model may sound limiting, it also provides a sizable advantage: extreme scalability. Because we can make the assumption that each document is self-contained, this means that we can also partition documents across multiple servers without having to keep related documents on the same server (because documents are independent of one another). This fundamental assumption of document independence allows queries to be parallelized across multiple partitions of documents and multiple servers to improve query performance, and this ultimately allows Solr to scale horizontally to handle querying billions of documents. This ability to scale across multiple partitions and servers is called *distributed searching*, and it will be covered next.

3.4.2 Distributed searching

The world would be a much simpler place if every important data operation could be run using a single server. In reality, however, sometimes your search servers may become overloaded by either too many queries at a time or by too much data needing to be searched through for a single server to handle.

In the latter case, it's necessary to break your content into two or more separate Solr indexes, each of which contains separate partitions of your data. Then every time a search is run, it'll be sent to both servers, and the separate results will be processed and aggregated before being returned from the search engine.

Solr includes this kind of distributed searching capability out of the box. We'll discuss how to manually segment your data into multiple partitions in chapter 12 when we talk about scaling Solr for production. Conceptually, each Solr index (called a Solr core) is available through it's own unique URL, and each of those Solr cores can be told to perform an aggregated search across other Solr cores using the following syntax:

```
http://box1:8983/solr/core1/select?q=*:*&shards=box1:8983/solr/core1,
    box2:8983/solr/core2,box2:8983/solr/core3
```

Notice four features about the preceding example:

- The `shards` parameter is used to specify the location of one or more Solr cores. A shard is a partition of your index, so the `shards` parameter on the URL tells Solr to aggregate results from multiple partitions of your data that are found in separate Solr cores.
- The Solr core being searched on (`box1`, `core1`) is also included in the list of shards; it won't automatically search itself unless explicitly requested as shown previously.
- This distributed search is searching across multiple servers.
- There's no requirement that separate Solr cores be located on separate machines. They can be on the same machine, as is the case here with `core2` and `core3` both being located on `box2`.

The important takeaway here has to do with the nature of scaling Solr. It should scale theoretically linearly because a distributed search across multiple Solr cores is run in parallel on each of those index partitions. Thus, if you split one Solr index into two Solr indexes with the same combined number of documents, the distributed search across the two indexes should be approximately 50% faster, minus any aggregation overhead.

This should also theoretically scale to any other number of servers (in reality, you will eventually hit a limit). The conceptual formula for determining total query speed after adding an additional index partition (assuming the same total number of documents) is

(Query Speed on N+1 indexes) = Aggregation Overhead + (Query Speed on N indexes)/(N+1)

This formula is useful for estimating the benefit you can expect from increasing the number of partitions into which your data is evenly divided. Because Solr scales nearly linearly, you should be able to reduce your query times proportional to the additional number of Solr cores (partitions) you add, assuming you're not constrained by server resources due to heavy load.

3.4.3 Clusters vs. servers

In the previous section we introduced the concept of distributed searching to enable scaling to handle large document sets. It's also possible to add multiple more or less identical servers into your system to balance the load of high query volumes.

Both of these scaling strategies rely on a conceptual shift away from thinking about servers and toward thinking about clusters of servers. A *cluster* of servers is defined as multiple servers, working in concert, to perform a unified function.

Take the following example, which should look similar to the example from section 3.4.2:

```
http://box1:8983/solr/core1/select?q=*:*&shards=box1:8983/solr/core1,
    box2:8983/solr/core2
```

This example performs a distributed search across two Solr cores, `core1` on `box1` and `core2` on `box2`. When running this distributed search, what happens to queries hitting `box1` if `box2` goes down? Listing 3.3 shows Solr's response under this scenario, which includes the error message from the failed connection to `box2`.

Listing 3.3 A failed distributed search (`RemoteServer` down)

```
<response>
    <lst name="responseHeader">
        <int name="status">500</int>
        <int name="QTime">1076</int>
        <lst name="params">
            <str name="shards">
                box1:8983/solr/core1,
                box2:8983/solr/core2
            </str>
        </lst>
    </lst>
    <lst name="error">
        <str name="msg">
            org.apache.solr.client.solrj.SolrServerException: IOException
            occurred when talking to server at: http://box2:8983/solr/core2
        </str>
        <str name="trace">...</str>
        <int name="code">500</int>
    </lst>
</response>
```

Notice that the servers for this use case are mutually dependent. If one becomes unavailable for searching, they all become unavailable for searching and begin failing, as indicated in the exception in the listing. It's therefore important to think in terms of clusters of servers instead of single servers when building out Solr solutions that must scale beyond a single box, as those servers are combining to serve as a single computing resource. Solr provides excellent built-in cluster-management capabilities through the use of Apache ZooKeeper, which will be covered in our discussion of Solr-Cloud in chapter 13.

As we wrap up our discussions of the key concepts behind searching at scale, we should be clear that Solr does have its limitations in this area, several of which will be discussed in the next section.

3.4.4 The limits of Solr

Solr is an incredibly powerful document-based NoSQL datastore that supports full text searching and data analytics. We have already discussed the powerful benefits of Solr's inverted index and complex, keyword-based, Boolean query capabilities. We have also seen how important relevancy is, and we've seen that Solr can scale more-or-less linearly across multiple servers to handle additional content or query volumes. What then are the use cases in which Solr is not a good solution? What are the limits of Solr?

One limit, as we've already seen, is that Solr is *not* relational in any way across documents. It's not well suited for joining significant amounts of data across different fields on different documents, and it can't perform join operations at all across multiple servers. While this is a functional limit of Solr, as compared to relational databases, this assumption of independence of documents is a tradeoff common among many NoSQL technologies, as it enables them to scale well beyond the limits of relational databases.

We've also already discussed the denormalized nature of Solr documents: data that is redundant must be repeated across each document to which that data applies. This can be particularly problematic when the data in one field that is shared across many documents changes.

Let's say that you were creating a search engine of social networking user profiles and one of your users, John Doe, becomes friends with another user named Coco. Now, I not only need to update John's and Coco's profiles, but I also need to update the "second-level connections" field for all of John's and Coco's friends, which could represent hundreds of document updates for one basic operation: two users becoming friends. This harkens back to the notion of Solr not being relational in any way.

An additional limitation of Solr is that it currently serves primarily as a document-storage mechanism; that is, you can insert, delete, and update documents, but not single fields (easily). Solr does currently have some minimal capability to update a single field, but only if the field is attributed in such a way that its original value is stored in the index, which can be wasteful. Even then, Solr is still updating the entire document based upon reindexing all of the stored fields internally. What this means is that whenever a new field is added to Solr or the contents of an existing field have changed, every single document in the Solr index must be reprocessed in its entirety before the data will be populated for the new field in all documents. Many other NoSQL systems suffer from this same problem, but it's worth noting that adding or changing a field in all documents across the corpus currently requires a nontrivial amount of document-update coordination to ensure the updates make it to Solr and do so in a timely fashion.

Solr is also optimized for a specific use case, which is taking search queries with small numbers of search terms and rapidly looking up each of those terms to find matching documents, calculating relevancy scores and ordering them all, and then only returning a few results for display. Solr is not optimized for processing quite long queries (thousands of terms) or returning quite large result sets to users.

One final limitation of Solr worth mentioning is its elastic scalability: the ability to automatically add and remove servers and redistribute content to handle load. While Solr scales well across servers, it doesn't yet elastically scale by itself in a fully automatic way. Recent work with SolrCloud (covered in chapter 13) using Apache ZooKeeper for cluster management is a great first step in this direction, but there are still many features to be worked out. For example, Solr cannot yet automatically reshard its content and increase the number of index replicas to dynamically handle content growth and query load. Many smart people in the community are working toward these long-term goals, however, and Solr continues to move closer and closer with every release.

3.5 Summary

In this chapter, we've discussed the key search concepts that serve as the foundation for most of Solr's search capabilities. We discussed the structure of Solr's inverted index and how it maps terms to documents in a way that allows quick execution of complex Boolean queries. We also discussed how fuzzy queries and phrase queries use position information to match misspellings and variations of terms and phrases in the Solr index.

We took a deep dive into Solr relevancy, laying out the default relevancy formula Solr uses and explaining conceptually how each piece of relevancy scoring works and why it exists.

We then provided a brief overview of the concepts of Precision and Recall, which serve as two opposing forces within the field of information retrieval and provide us with a good conceptual framework from which to judge whether or not our search results are meeting our goals.

Finally, we discussed key concepts for how Solr scales, including discussions of content denormalization within documents and distributed searching to ensure that query execution can be parallelized to maintain or decrease search time, even as content grows beyond what can be reasonably handled by a single machine. We ended with a discussion of thinking in terms of clusters as opposed to servers as you scale your search architecture, and we looked at some of the limitations of Solr and use cases for when Solr may not be a great fit.

At this point, you should have all of the conceptual background necessary to understand the core features of Solr throughout the rest of this book and should have a solid grasp of the most important concepts needed for building a killer search application. In the next chapter, we'll begin digging into Solr's key configuration settings, which will enable more fine-grained control over many of the features discussed in this chapter.

SEARCH

Solr IN ACTION

Grainger • Potter

Whether you're handling big (or small) data, managing documents, or building a website, it is important to be able to quickly search through your content and discover meaning in it. Apache Solr is your tool: a ready-to-deploy, Lucene-based, open source, full-text search engine. Solr can scale across many servers to enable real-time queries and data analytics across billions of documents.

Solr in Action teaches you to implement scalable search using Apache Solr. This easy-to-read guide balances conceptual discussions with practical examples to show you how to implement all of Solr's core capabilities. You'll master topics like text analysis, faceted search, hit highlighting, result grouping, query suggestions, multilingual search, advanced geospatial and data operations, and relevancy tuning.

What's Inside

- How to scale Solr for big data
- Rich real-world examples
- Solr as a NoSQL data store
- Advanced multilingual, data, and relevancy tricks
- Coverage of versions through Solr 4.7

This book assumes basic knowledge of Java and standard database technology. No prior knowledge of Solr or Lucene is required.

Trey Grainger is a director of engineering at CareerBuilder. **Timothy Potter** is a senior member of the engineering team at LucidWorks. The authors work on the scalability and reliability of Solr, as well as on recommendation engine and big data analytics technologies.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SolrinAction

Free eBook
SEE INSERT

"The knowledge and techniques you need."
—From the Foreword by Yonik Seeley, Creator of Solr

"Readable and immediately applicable ... an excellent book."
—John Viviano, InterCorp, Inc.

"The go-to guide for Solr ... a definitive resource for both beginners and experts."
—Scott Anthony
Business Instruments

"A well-dosed combination of deep technical knowledge and real-world experience."
—Alexandre Madurell
Piksel, Inc.

ISBN 13: 978-1-617291-02-9
ISBN 10: 1-61729-102-1
54999
9 781617 291029

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]