Solr IN ACTION

Trey Grainger
Timothy Potter

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Solr in Action
version 6

For more information on this and other Manning titles go to
www.manning.com

brief contents

1 Introduction to Solr

This chapter covers

- Characteristics of the types of data handled by search engines
- Common search engine use cases
- Key components of Solr
- Reasons to choose Solr
- Feature overview

With fast-growing technologies like social media, cloud computing, mobile applications, and big data, these are exciting times to be in computing. One of the main challenges facing software architects is the need to handle massive volumes of data consumed and produced by a huge global user base. In addition, users expect online applications to always be available and responsive. To address the scalability and availability needs of modern web applications, we've seen a growing interest in specialized, non-relational data storage and processing technologies, collectively known as NoSQL (Not only SQL). These systems share a common design pattern of matching the storage and processing engine to specific types of data rather than forcing all data into the once de facto standard relational model. In other words, NoSQL technologies are optimized to solve a specific class of problems for specific types of data. The need to scale has led to hybrid architectures composed of a variety of NoSQL and relational databases; gone are the days of the one-size-fits-all data processing solution.

This book is about a specific NoSQL technology, Apache Solr, which, like its non-relational brethren, is optimized for a unique class of problems. Specifically, Solr is a scalable, ready-to-deploy enterprise search engine that's optimized to search large volumes of text-centric data and return results sorted by relevance. That was a bit of a mouthful, so let's break the previous statement down into its basic parts:

- **Scalable**—Solr scales by distributing work (indexing and query processing) to multiple servers in a cluster
- **Ready to deploy**—Solr is open-source, is easy to install and configure, and provides a preconfigured example to help you get started
- **Optimized for search**—Solr is fast and can execute complex queries in subsecond speed, often only 10's of milliseconds
- **Large volumes of documents**—Solr is designed to deal with indexes containing millions of documents
- **Text-centric**—Solr is optimized for searching natural language text, like emails, web pages, resumes, PDF documents, and social messages like tweets or blogs
- **Results sorted by relevance**—Solr returns documents in ranked order based on how relevant each document is to the user's query

In this book, you'll learn how to use Solr to design and implement scalable search solutions. We'll begin our journey by learning about the types of data and uses cases Solr supports. This will help you understand where Solr fits into the big picture of modern application architectures and which problems Solr is designed to solve.

1.1 Why do I need a search engine?

We suspect that because you're looking at this book, you already have an idea about why you need a search engine. Therefore, rather than speculate on why you're considering Solr, we'll get right down to the hard questions you need to answer about your data and use cases in order to decide if a search engine is right for you. In the end, it comes down to understanding your data and users and then picking a technology that works for both. Let's start by looking at the properties of data that a search engine is optimized to handle.

1.1.1 Managing text-centric data

A hallmark of modern application architectures is matching the storage and processing engine to your data. If you're a programmer, then you know to select the best data structure based on how you use the data in an algorithm, that is, you don't use a linked list when you need fast random lookups. The same principle applies with search engines. There are four main characteristics of data search engines like Solr are optimized to handle.

1. Text-centric
2. Read-dominant
3. Document-oriented
4. Flexible schema

A possible fifth characteristic is having a large volume of data to deal with, that is, "big data," but our focus is on what makes a search engine special among other NoSQL technologies. It goes without saying that Solr can deal with large volumes of data.

Although these are the four main characteristics of data that search engines like Solr handle efficiently, you should think of them as rough guidelines and not strict rules. Let's dig into each of these characteristics to see why they're important for search. For now, we'll focus on the high-level concepts and get into the "how" in later chapters.

TEXT CENTRIC

You'll undoubtedly encounter the term "unstructured" to describe the type of data that's handled by a search engine. We think "unstructured" is a little ambiguous because any text document based on human language has implicit structure. You can think of the term "unstructured" as being from the perspective of a computer, which sees text as a stream of characters. The character stream must be parsed using language-specific rules to extract the structure and make it searchable, which is exactly what search engines do.

We think the label "text-centric" is more appropriate for describing the type of data Solr handles because a search engine is specifically designed to extract the implicit structure of text into its index to improve searching. Text-centric data implies that the text of a document contains "information" that users are interested in finding. Of course, a search engine also supports non-text data like dates and numbers, but its primary strength is handling text data based on natural language.

Also, the "centric" part is important because if users aren't interested in the information in the text, then a search engine may not be the best solution for your problem. For example, consider an application where employees create travel expense reports. Each report contains a number of structured data fields like date, expense type, currency, and amount. In addition, each expense may include a notes field where employees can provide a brief description of the expense. This would be an example of data that contains text but isn't "text-centric" in that it's unlikely that the accounting department needs to search the notes field when generating monthly expense reports. Put simply, despite data containing text fields doesn't mean it's a natural fit for a search engine.

Take a moment and think about whether your data is "text-centric," The main consideration for us is whether or not the text fields in your data contain information that users will want to query. If yes, then a search engine is probably a good choice. We'll see how to unlock the structure in text using Solr's text analysis in chapters 5 and 6.

READ DOMINANT

Another key aspect of data that search engines handle efficiently is that it's read-dominant, as compared to writes. First, though, let's be clear that Solr does allow you to update existing documents in your index. You can think of read-dominant as meaning that documents are read much more often then they're created or updated. But don't take this to mean that you can't write a lot of data or that you have limits on how frequently you can write new data. In fact, one of the key features in Solr 4 is near-real-time search, which allows you to index thousands of documents per second and have them be searchable almost immediately.

The key point behind read-dominant data is that when you do write data to Solr, it's intended to be read and reread many, many times over its lifetime. You can think of a search engine as being optimized for executing queries (a read operation), for example, as opposed to storing data (a write operation). Also, if you have to update existing data in a search engine often, then that could be an indication that a search engine might not be the best solution for your needs. Another NoSQL technology, like Cassandra, might be a better choice when you need fast random-writes to existing data.

DOCUMENT ORIENTED

To this point, we've used the generic label "data" but in reality, search engines work with documents. In a search engine, a document is a self-contained collection of fields where each field only holds data and doesn't contain nested fields. In other words, a document in a search engine like Solr has a flat structure and doesn't depend on other documents. The "flat" concept is slightly relaxed in Solr in that a field can have multiple values but fields don't contain sub-fields. That is, you can store multiple values in a single field but you can't nest fields inside of other fields.

The flat, document-oriented approach in Solr works well with data that's already in document format, such as a web page, blog, or PDF document, but what about modeling normalized data stored in a relational database? In this case, you need to denormalize data spread across multiple tables into a flat, self-contained document structure. We'll learn how to approach problems like this in chapter 3.

You also want to consider which fields in your documents must be stored in Solr and which should be stored in another system, such as a database. Put simply, a search engine isn't the place to store data unless it's useful for search or displaying results. For example, if you have a search index for online videos, then you don't want to store the binary video files in Solr. Rather, large binary fields should be stored in another system, such as a content distribution network (CDN). In general, you should store the minimal set of information for each document needed to satisfy search requirements. This is a clear example of not treating Solr as a general data storage technology; Solr's job is to find videos of interest and not to manage large binary files.

FLEXIBLE SCHEMA

The last main characteristic of search engine data is that it has a flexible schema. This means that documents in a search index don't need to have a uniform structure. In a relational database, every row in a table has the same structure. In Solr, documents can have different fields. Of course, there should be some overlap between the fields in documents in the same index but they don't have to be identical.

For example, imagine a search application for finding homes for rent or sale. Listings will obviously share fields like location, number of bedrooms, and number of bathrooms, but they'll also have different fields based on the listing type. A home for sale would have fields for listing price and annual property taxes, whereas a home for rent would have a field for monthly rent and pet policy.

To summarize, search engines in general and Solr in particular are optimized to handle data having four specific characteristics: text-centric, read-dominant, document-oriented, and flexible schema. Overall, this implies that Solr isn't a general-purpose data storage and processing technology, which is one of the main differentiating factors of NoSQL (not only SQL) technologies.

The whole point of having a wide variety of options for storing and processing data is that you don't have to find a one-size-fits-all technology. Search engines are good at specific things and quite horrible at others. This means in most cases, you're going to find Solr complements relational databases and other NoSQL technologies more than it replaces them.

Now that we've talked about the type of data Solr is optimized to handle, let's think about the primary use cases a search engine like Solr is designed to handle. These use cases are intended to help you understand how a search engine is different than other data processing technologies.

1.1.2 Common search engine use cases

In this section, we look at some of the things you can do with a search engine like Solr. As with our discussion of the types of data in section 1.1.1, use these as guidelines and not strict rules. Before we get into specifics though, you should keep in mind that the bar for excellence in search is high. Modern users are accustomed to web search engines like Google and Bing being fast and effective at serving modern web information needs. Moreover, most popular websites have powerful search solutions to help people find information quickly. When you're evaluating a search engine like Solr and designing your search solution, make sure you put user experience as a high priority.

BASIC KEYWORD SEARCH

It almost seems obvious to point out that a search engine supports keyword search, as that's its main purpose. But it's worth mentioning because keyword search is the most typical way users will begin working with your search solution. It'd be rare for a user to want to fill out a complex search form initially. Given that basic keyword search will be the most common way users will interact with your search engine, then it stands to reason that this feature must provide a great user experience.

In general, users want to type in a few simple keywords and get back great results. This may sound like a simple task of matching query terms to documents but consider a few of the issues that must be addressed to provide a great user experience:

- Relevant results must be returned quickly, within a second or less in most cases
- Spell correction in case the user misspells some of the query terms
- Auto-suggestions to save users some typing, particularly for mobile applications
- Handling synonyms of query terms
- Matching documents containing linguistic variations of query terms

- Phrase handling, that is, does the user want documents matching all words or any of the words in a phrase
- Proper handling of queries with common words like “a,” “an,” “of,” and “the”
- Giving the user a way to see more results if the top results aren’t satisfactory

As you can see, a number of issues exist that make a seemingly simple feature hard to implement without a specialized approach. But with a search engine like Solr, these features come out of the box and are easy to implement. Once you give users a powerful tool to execute keyword searches, you need to consider how to display the results. This brings us to our next use case of ranking results based on their relevance to the user’s query.

RANKED RETRIEVAL

A search engine stands alone as a way to return “top” documents for a query. In a SQL query to a relational database, a row either matches a query or not and results are sorted based on one of the columns. On the other hand, a search engine returns documents sorted in descending order by a score that indicates the strength of the match of the document to the query. How strength of match is calculated depends on a number of factors but in general a higher score means the document is more relevant to the query.

Ranking documents by relevancy is important for a couple of reasons. First, modern search engines typically store a large volume of documents, often millions or billions of documents. Without ranking documents by relevance to the query, users can become overloaded with results without any clear way to navigate them. Second, users are more comfortable and accustomed to getting results from other search engines using only a few keywords. Users are impatient and expect the search engine to “do what I mean, not what I say.” This is true of search solutions backing mobile applications where users on the go will enter short queries with potential misspellings and expect it to “just work.”

To influence ranking, you can assign more weight or “boost” certain documents, fields, or specific terms. For example, you can boost results by their age to help push newer documents towards the top of search results. We’ll learn about ranking documents in chapter 3.

BEYOND KEYWORD SEARCH

With a search engine like Solr, users can type in a few keywords and get back some results. For many users, though, this is only the first step in a more interactive session where the search results give them the ability to keep exploring. One of the primary use cases of a search engine is to drive an information discovery session. Frequently, your users won’t know exactly what they’re looking for and typically don’t have any idea what information is contained in your system. A good search engine helps users narrow in on their information needs.

The central idea here is to return some documents from an initial query, as well as some tools to help users refine their search. In other words, in addition to returning matching documents, you also return tools that give your users an idea of what to do next. For

example, you can categorize search results using document features to allow users to narrow down their results. This is known as faceted-search and is one of the main strengths of Solr. We'll see an example of faceted search for a real estate search in section 1.2. Facets are covered in-depth in chapter 8.

DON'T USE A SEARCH ENGINE TO DO …

Lastly, let's consider a few use cases where a search engine wouldn't be useful. First, search engines are designed to return a small set of documents per query, usually 10 to 100. More documents for the same query can be retrieved using Solr's built-in paging support. Consider a query that matches a million documents—if you request all of those documents back at once, you should be prepared to wait a long time. The query itself will likely execute quickly but reconstructing a million documents from the underlying index structure will be extremely slow, as engines like Solr store fields on disk in a format from which it's easy to create a few documents, but takes a long time to reconstruct many documents when generating results.

Another use case where you shouldn't use a search engine is for deep analytics tasks that require access to a large subset of the index. Even if you avoid the previous issue by paging through results, the underlying data structure of a search index isn't designed for retrieving large portions of the index at once.

We've touched on this previously, but we'll reiterate that search engines aren't the place for querying across relationships between documents. Solr does support querying using a parent-child relationship, but doesn't provide support for navigating complex relational structures. In chapter 3, you'll learn some techniques to adapt relational data to work with Solr's flat document structure.

Lastly, there's no direct support in most search engines for document-level security, at least not in Solr. If you need fine-grained permissions on documents, then you'll have to handle that outside of the search engine.

Now that we've seen the types of data and use cases where a search engine is the right solution, it's time to dig into what Solr does and how it does it from a high level. In the next section, you'll learn what Solr does and how it approaches important software design principles like integration with external systems, scalability, and high-availability.

1.2 What is Solr?

In this section, we introduce the key components of Solr by designing a search application from the ground up. This will help you understand what specific features Solr provides and the motivation for their existence. But, before we get into the specifics of what Solr is, let's make sure you know what Solr isn't:

1) Solr isn't a web search engine like Google or Bing
2) Solr has nothing to do with search engine optimization (SEO) for a website

Now, imagine we need to design a real estate search web application for potential homebuyers. The central use case for this application will be searching for homes to buy across the United States using a web browser. Figure 1.1 depicts a screen shot from this fictitious web application. Don't focus too much on the layout or design of the user interface,

it's only a quick mock-up to give us some visual context. What's important is the type of experience that Solr can support.

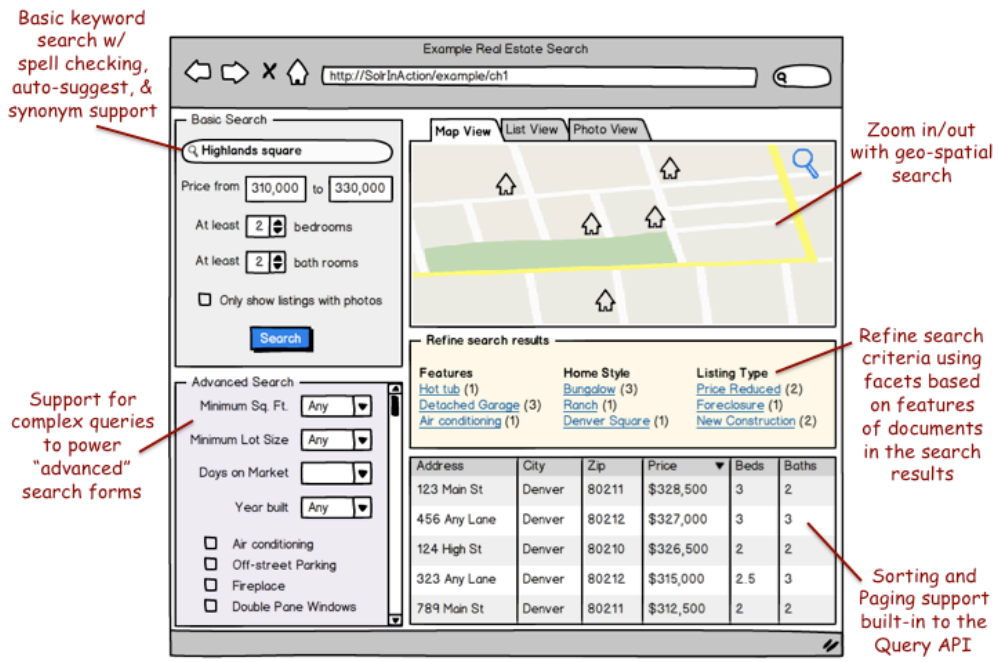


Figure 1.1 Mock-up screen shot of fictitious search application to depict Solr features

Let's take a quick tour of the screen shot in figure 1.1 to illustrate some of Solr's key features. First, starting at the top-left corner, working clock-wise, Solr provides powerful features to support a keyword search box. As we discussed in section 1.1.2, providing a great user experience with basic keyword search requires complex infrastructure that Solr provides out-of-the-box. Specifically, Solr provides spell checking, auto-suggesting as the user types, synonym handling, phrase queries, and text-analysis tools to deal with linguistic variations in query terms, such as "buying a house" or "purchase a home."

Solr also provides a powerful solution for implementing geo-spatial queries. In figure 1.1, matching home listings are displayed on a map based on their distance from the latitude / longitude of the center of some fictitious neighborhood. With Solr's geo-spatial support, you can sort documents by geo-distance or even rank documents by geo-distance. It's also important that geo-spatial searches are fast and efficient to support a user interface that allows users to zoom in and out and move around on a map.

Once the user performs a query, the results can be further categorized using Solr's faceting support to show features of the documents in the result set. Facets are a way to

categorize the documents in a result set in order to drive discovery and query refinement. In figure 1.1, search results are categorized into three facets for features, home style, and listing type.

Now that we have a basic idea of the type of functionality we need to support our real estate search application, let's see how we'd implement these features with Solr. To begin, we need to know how Solr matches home listings in the index to queries entered by users, as this is the basis for all search applications.

1.2.1 Information retrieval engine

Solr is built on Apache Lucene, a popular Java-based open source information retrieval library. We'll save a detailed discussion of what information retrieval is for chapter 3. For now, we'll touch on the key concepts behind information retrieval starting with the formal definition taken from one of the prominent academic texts on modern search concepts:

> Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).
>
> *Introduction to Information Retrieval*, Manning, et. al., 2008, pp. 1

In the case of our example real estate application, the user's primary information need is finding a home to purchase based on location, home style, features, and price. Our search index will contain home listings from across the U.S., which definitely qualifies as a "large collection." In a nutshell, Solr uses Lucene to provide the core infrastructure for indexing documents and executing searches to find documents.

Under the covers, Lucene is a Java-based library for building and managing an inverted index, which is a specialized data structure for matching query terms to text-based documents. Figure 1.2 provides a simplified depiction of a Lucene inverted index for our example real estate search application.

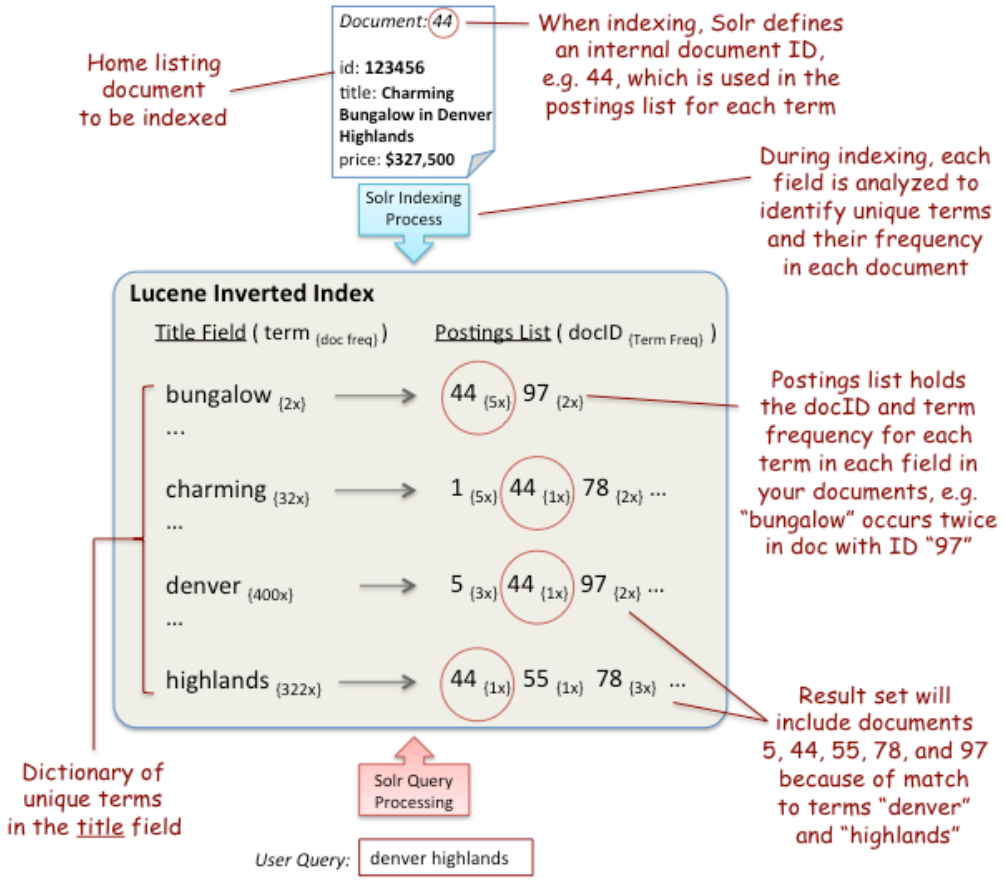


Figure 1.2 The key data structure supporting information retrieval is the inverted index

You'll learn all about how an inverted index works in chapter 3. For now, it's sufficient to review figure 1.2 to get a feel for what happens when a new document (#44 in the diagram) is added to the index and how documents are matched to query terms using the inverted index.

You might be thinking that a relational database could easily return the same results using a SQL query, which is true for this simple example. But one key difference between a Lucene query and a database query is that in Lucene, results are ranked by their relevance to a query and database results can only be sorted by one of the table columns. In other words, ranking documents by relevance is a key aspect of information retrieval and helps differentiate it from other types of search.

Building a web-scale inverted index

It might surprise you that search engines like Google also use an inverted index for searching the web. In fact, the need to build a web-scale inverted index led to the invention of MapReduce.

MapReduce is a programming model that distributes large-scale data processing operations across a cluster of commodity servers by formulating an algorithm into two phases: map and reduce. With roots in functional programming, MapReduce was adapted by Google for building its massive inverted index to power web search. Using MapReduce, the map phase produces unique term and document ID where the term occurs. In the reduce phase, terms are sorted so that all term / docID pairs are sent to the same reducer process for each unique term. The reducer sums up all term frequencies for each term to generate the inverted index.

Apache Hadoop provides an open source implementation of MapReduce and is used by the Apache Nutch open source project to build a Lucene inverted index for web-scale search. A thorough discussion of Hadoop and Nutch are beyond the scope of this book, but we encourage you to investigate these projects if you need to build a large-scale search index.

Now that we know that Lucene provides the core infrastructure to support search, let's look at what value Solr adds on top of Lucene, starting with how you define how your index is structured using Solr's flexible **schema.xml** configuration document.

1.2.2 Flexible schema management

Although Lucene provides the infrastructure for indexing documents and executing queries, what's missing from Lucene is an easy way to configure how you want your index to be structured. With Lucene you need to write Java code to define fields and how to analyze those fields. Solr adds a simple, declarative way to define the structure of your index and how you want fields to be represented and analyzed using an XML configuration document named **schema.xml**. Under the covers, Solr translates the **schema.xml** document into a Lucene index. This saves you programming and makes your index structure easier to understand and communicate to others. On the other hand, a Solr built index is 100% compatible with a programmatically built Lucene index.

Solr also adds a few nice constructs on top of the core Lucene indexing functionality. Specifically, Solr provides copy fields and dynamic fields. Copy fields provide a way to take the raw text contents of one or more fields and have them applied to a different field. Dynamic fields allow you to apply the same field type to many different fields without explicitly declaring them in **schema.xml**. This is useful for modeling documents that have many fields. We cover **schema.xml** in depth in chapters 5 and 6.

In terms of our example real estate application, it might surprise you that we can use the Solr example server out-of-the-box without making any changes to the **schema.xml**. This shows how flexible Solr's schema support is, because the example Solr server is designed to support product search but works fine for our real estate search example.

At this point, we know that Lucene provides a powerful library for indexing documents, executing queries, and ranking results. And, with **schema.xml**, you have a flexible way to define the index structure using an XML configuration document instead of having to program to the Lucene API. Now you need a way access these services from the web. In the next section, we learn how Solr runs as a Java web application and integrates with other technologies using proven standards such as XML, JSON, and HTTP.

1.2.3 Java web application

Solr is a Java web application that runs in any modern Java Servlet engine like Jetty or Tomcat, or a full J2EE application server like JBoss or Oracle AS. Figure 1.3 depicts major software components of a Solr server.

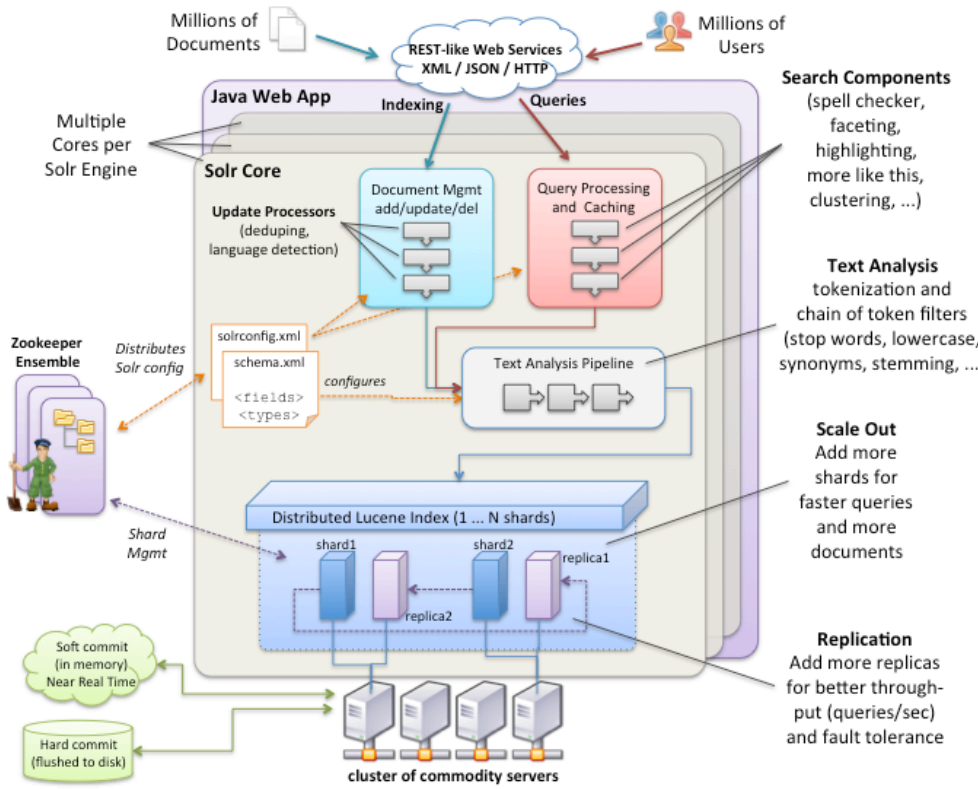


Figure 1.3 Diagram of the main components of Solr 4

Admittedly, figure 1.3 is a little overwhelming at first glance. Take a moment to scan over the diagram to get a feel for some of the terminology. Don't worry if you're not familiar with all of the terms and concepts represented in the diagram--after reading this book, you should have a strong understanding of the all concepts presented in figure 1.3.

As we mentioned in the introduction to this chapter, the Solr designers recognized that Solr fits better as a complementary technology that works within existing architectures. In fact, you'll be hard-pressed to find an environment where Solr doesn't drop right in. As we'll see in chapter 2, you can start the example Solr server in a couple of minutes after you finish the download.

To achieve the goal of easy integration, Solr's core services need to be accessible from many different applications and languages. Solr provides simple REST-like services based on proven standards of XML, JSON, and HTTP. As a brief aside, we avoid the "RESTful" label for Solr's HTTP-based API as it doesn't strictly adhere to all REST (representational state transfer) principles. For instance, in Solr you use HTTP POST to delete documents instead of HTTP DELETE.

A REST-like interface is nice as a foundation, but often times, developers like to have access to a client library in their language of choice to abstract away some of the boilerplate machinery of invoking a web service and processing the response. The good news here is that most popular languages have a Solr client library including Python, PHP, Java, .NET, and Ruby.

1.2.4 Multiple indexes in one server

One hallmark of modern application architectures is the need for flexibility in the face of rapidly changing requirements. One of the ways Solr helps this situation is that you don't have to do all things in Solr with one index, because Solr supports running multiple cores in a single engine. In figure 1.3, we've depicted multiple cores as separate layers all running in the same Java web application environment.

Think of each core as a separate index and configuration and there can be many cores in a single Solr instance. This allows you to manage multiple cores from one server so that you can share server resources and administration tasks like monitoring and maintenance. Solr provides an API for creating and managing multiple cores.

One use of Solr's multicore support is data partitioning, such as having one core for recent documents and another core for older documents, known as chronological sharding. Another use of Solr's multicore support is to support multitenant applications.

In our real estate application, we might use multiple cores to manage different types of listings that are different enough to justify having different indexes for each. Consider real estate listings for rural land instead of homes. Buying rural land is a different process than buying a home in a city, so it stands to reason that we might want to manage our land listings in a separate core.

1.2.5 Extendable (plug-ins)

Figure 1.3 depicts three main sub-systems in Solr: document management, query processing, and text analysis. Of course, these are high-level abstractions for complex sub-systems in Solr; we'll learn about each later in the book. Each of these systems is composed of a modular "pipeline" that allows you to plug-in new functionality. This means that instead of overriding the entire query-processing engine in Solr, you plug-in a new search component into an existing pipeline. This makes the core Solr functionality easy to extend and customize to meet your specific application needs.

1.2.6 Scalable

Lucene is an extremely fast search library and Solr takes full advantage of Lucene's speed. But regardless of how fast Lucene is, a single server will reach its limits in terms of how many concurrent queries from different users it can handle due to CPU and I/O constraints.

As a first pass to scalability, Solr provides flexible cache management features that help your server avoid recomputing expensive operations. Specifically, Solr comes preconfigured with a number of caches to save expensive recomputations, such as caching the results of a query filter. We'll learn about Solr's cache management features in chapter 4.

Caching only gets you so far so at some point, you're going to need to scale out your capacity to handle more documents and higher query throughput by adding more servers. For now let's focus on the two most common dimensions of scalability in Solr. First is query throughput, which is the number of queries your engine can support per second. Even though Lucene can execute each query quickly, it's limited in terms of how many concurrent requests a single server can handle. For higher query throughput, you add replicas of your index so that more servers can handle more requests. This means if your index is replicated across three servers, then you can handle roughly three times the number of queries per second because each server handles one-third of the query traffic. In practice, it's rare to achieve perfect linear scalability so adding three servers may only allow you to handle two and one half times the query volume of one server.

The other dimension of scalability is the number of documents indexed. If you're dealing with large volumes of documents, then you'll likely reach a point where you have too many documents in a single instance and query performance will suffer. To handle more documents, you split the index into smaller chunks called "shards" and then distribute the searches across the shards.

Scaling-out with virtualized commodity hardware

One trend in modern computing is building software architectures that can scale horizontally using virtualized commodity hardware. Put simply, add more commodity servers to handle more traffic. Fueling this trend towards using virtualized commodity hardware are cloud-computing providers such as Amazon EC2. Although Solr will run on virtualized hardware, you should be aware that search is I/O and memory intensive.

Therefore, if search performance is a top priority for your organization, then you should consider deploying Solr on higher end hardware with high-performance disks, ideally solid-state drives (SSD). Hardware considerations for deploying Solr are discussed in chapter 13.

Scalability is important, but ability to survive failures is also important for a modern system. In the next section, we discuss how Solr handles software and hardware failures.

1.2.7 Fault tolerant

Beyond scalability, you need to consider what happens if one or more of your servers fails, particularly if you're planning to deploy Solr on virtualized hardware or commodity hardware. The bottom line is that you must plan for failures. Even the best architectures and high-end hardware will experience failures.

Let's assume you have four shards for your index and the server hosting shard #2 loses power. At this point, Solr can't continue indexing documents and can't service queries, so your search engine is effectively "down." To avoid this situation, you can add replicas of each shard. In this case, when shard #2 fails, Solr reroutes indexing and query traffic to the replica and your Solr cluster remains online. The impact of this failure is that indexing and queries can still be processed but may not be as fast because you've one less server to handle requests. We'll discuss failover scenarios in chapter 16.

At this point, you've seen that Solr has a modern, well-designed architecture that's scalable and fault-tolerant. Although these are important aspects to consider if you've already decided to use Solr, you still might not be convinced that Solr is the right choice for your needs. In the next section, we describe the benefits of Solr from the perspective of different stakeholders, such as the software architect, system administrator, and CEO.

1.3 Why Solr?

In this section, we hope to provide you with some key information to help you decide if Solr is the right technology for your organization. Let's begin by addressing why Solr is attractive to software architects.

1.3.1 Solr for the software architect

When evaluating new technology, software architects must consider a number of factors, but chief among those are stability, scalability, and fault-tolerance. Solr scores high marks in all three categories.

In terms of stability, Solr is a mature technology supported by a vibrant community and seasoned committers. One thing that shocks new users to Solr and Lucene is that it isn't unheard of to deploy from source code pulled directly from the trunk rather than waiting for an official release. We won't advise you either way on whether this is acceptable for your organization. We only point this out because it's a testament to the depth and breadth of

automated testing in Lucene and Solr. Put simply, if you have a nightly build off trunk where all the automated tests pass, then you can be sure the core functionality is solid.

We've touched on Solr's approach to scalability and fault-tolerance in sections 1.2.6 and 1.2.7. As an architect, you're probably most curious about the limitations of Solr's approach to scalability and fault-tolerance. First, you should realize that the sharding and replication features in Solr have been rewritten in Solr 4 to be robust and easier to manage. The new approach to scaling is called SolrCloud. Under the covers, SolrCloud uses Apache Zookeeper to distribute configuration across a cluster of Solr nodes and to keep track of cluster state. Here are some highlights of the new SolrCloud features in Solr:

- Centralized configuration
- Distributed indexing with no Single Point of Failure (SPoF)
- Automated fail-over to a new shard leader
- Queries can be sent to any node in a cluster to trigger a full distributed search across all shards with fail-over and load-balancing support built-in

But this isn't to say that Solr scaling doesn't have room for improvement. SolrCloud is lacking in two areas. First, not all features work in distributed mode, such as joins. Second, the number of shards for an index is a fixed value that can't be changed without reindexing all of the documents. We'll get into all of the specifics of SolrCloud in chapter 16, but we want to make sure architects are aware that Solr scaling has come a long way in the past couple of years yet still has some more work to do.

1.3.2 Solr for the system administrator

As a system administrator, high among your priorities in adopting a new technology like Solr is whether it fits into your existing infrastructure. The easy answer is yes it does. As Solr is Java based, it runs on any OS platform that has a J2SE 6.x/7.x JVM. Out of the box, Solr embeds Jetty, the open source Java servlet engine provided by Oracle. Otherwise, Solr is a standard Java web application that deploys easily to any Java web application server like JBoss and Oracle AS.

All access to Solr can be done via HTTP and Solr is designed to work with caching HTTP reverse proxies like Squid and Varnish. Solr also works with JMX so you can hook it up to your favorite monitoring application, such as Nagios.

Lastly, Solr provides a nice administration console for checking configuration settings, statistics, issuing test queries, and monitoring the health of SolrCloud. Figure 1.4 provides a screen shot of the Solr 4 administration console. We'll learn more about the administration console in chapter 2.

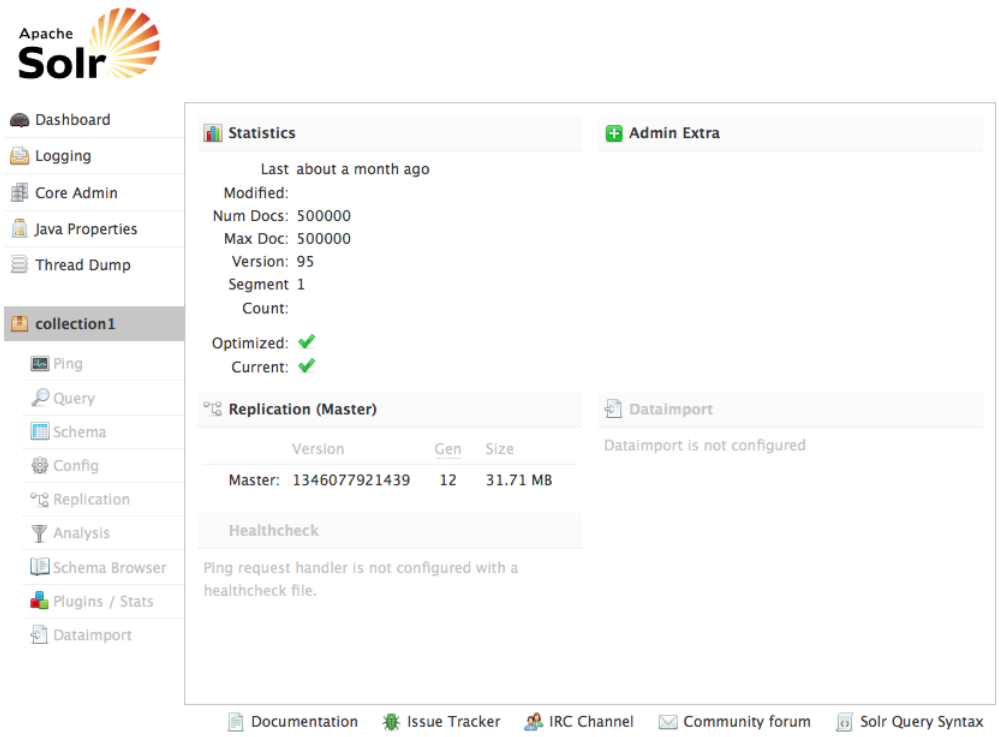


Figure 1.4 Screen shot of Solr 4 administration console where you can send test queries, ping the server, view configuration settings, and see how your shards and replicas are distributed in a cluster.

1.3.3 Solr for the CEO

Although it's unlikely that a CEO will be reading this book, here are some key talking points about Solr in case your CEO stops you in the hall. First, executive types like to know an investment in a technology today is going to payoff in the long term. With Solr, you can emphasize that many companies are still running on Solr 1.4, which was released in 2009, which means Solr has a successful track record and is constantly being improved.

Also, CEO's like technologies that are predictable. As you'll see in the next chapter, Solr "just works" and you can have it up and running in minutes. Another concern is what happens if the Solr guy walks out the door—will business come to a halt? It's true that Solr is complex technology but having a vibrant community behind it means you have help when you need it. And, you have access to the source code, which means if something is broken and you need a fix, you can do it yourself. Many commercial service providers also can help you plan, implement, and maintain your Solr installation; many of which offer training courses for Solr.

This may be one for the CFO, but Solr doesn't require much initial investment to get started. Without knowing the size and scale of your environment, we're confident in saying that you can start up a Solr server in a few minutes and be indexing documents quickly. A modest server running in the cloud can handle millions of documents and many queries with sub-second response times.

1.4 Feature overview

Finally, let's do a quick run-down of Solr's main features organized around the following categories:

- User experience
- Data modeling
- New features in Solr 4

Providing a great user experience with your search solution will be a common theme throughout this book so let's start by seeing how Solr helps make your users happy.

1.4.1 User experience features

Solr provides a number of important features that help you deliver a search solution that's easy to use, intuitive, and powerful. But you should note that Solr only exposes a REST-like HTTP API and doesn't provide search-related UI components in any language or framework. You'll have to roll up your sleeves and develop your own search UI components that take advantage of some of the following user experience features:

- Pagination and sorting
- Faceting
- Auto-suggest
- Spell checking
- Hit highlighting
- Geo-spatial search

PAGINATION AND SORTING

Rather than returning all matching documents, Solr is optimized to serve paginated requests where only the top N documents are returned on the first page. If users don't find what they're looking for on the first page, then you can request subsequent pages using simple API request parameters. Pagination helps with two key outcomes: 1) results are returned more quickly, because each request only returns a small subset of the entire search results, 2) helps you track how many queries result in requests for more pages, which may be an indication of a problem in relevance scoring. You'll learn about paging and sorting in chapter 7.

FACETING

Faceting provides users with tools to refine their search criteria and discover more information by categorizing search results into sub-groups using facets. In our real estate example (figure 1.1) we saw how search results from a basic keyword search were organized into three facets: Features, Home Style, and Listing Type. Solr faceting is one of the more popular and powerful features available in Solr; we cover faceting in depth in chapter 8.

AUTO-SUGGEST

Most users will expect your search application to "do the right thing" even if they provide incomplete information. Auto-suggest helps users by allowing them to see a list of suggested terms and phrases based on documents in your index. Solr's auto-suggest features allows user to start typing a few characters and receive a list of suggested queries as they type. This reduces the number of incorrect queries, particularly because many users may be searching from a mobile device with small keyboards.

Auto-suggest gives users examples of terms and phrases available in the index. Referring back to our real estate example, as a user types "hig…" Solr's auto-suggestion feature can return suggestions like "highlands neighborhood" or "highlands ranch." We cover auto-suggest in chapter 10.

SPELL CHECKER

In the age of mobile devices and people on the go, spell-correction support is essential. Again, users expect to be able to type misspelled words into the search box and the search engine should handle it gracefully. Solr's spell checker supports two basic modes:

Auto-correct—Solr can make the spell correction automatically based on whether the misspelled term exists in the index.

Did you mean that Solr can return a suggested query that might produce better results so that you can display a hint to your users, such as "Did you mean *highlands*?" if you user typed in "hilands."

Spell correction was revamped in Solr 4 to be easier to manage and maintain; we'll see how this works in chapter 10.

HIT HIGHLIGHTING

When searching documents that have a significant amount of text, you can display specific sections of each document using Solr's hit highlighting feature. Most useful for longer format documents, so you can help users find relevant information in longer documents by quickly scanning highlighted sections in search results. We cover hit highlighting in chapter 9.

GEO-SPATIAL

Geographical location is a first-class concept in Solr 4 in that it has built-in support for indexing latitude and longitude values as well as sorting or ranking documents by geographical distance. Solr can find and sort documents by distance from a geo-location (latitude and longitude). In the real estate example, matching listings are displayed on an interactive map where users can zoom in/out and move the map center point to find near-by listings using geo-spatial search.

Another exciting addition to Solr 4 is that you can index geographical shapes such as polygons, which allows you to find documents that intersect geographical regions. This might be useful for finding home listings in specific neighborhoods using a precise geographical representation of a neighborhood. We cover Solr's geo-spatial search features in chapter 14.

1.4.2 Data modeling features

As we discussed in section 1.1, Solr is optimized to work with specific types of data. In this section, we provide an overview of key features that help you model data for search, including:

- Field collapsing / grouping
- Flexible query support
- Joins
- Clustering
- Importing rich document formats like PDF and Word
- Importing data from relational databases
- Multilingual support

FIELD COLLAPSING / GROUPING

Although Solr requires a flat, denormalized document, Solr allows you to treat multiple documents as a group based on some common property shared by all documents in a group. Field grouping, also known as field collapsing, allows you to return unique groups instead of individual documents in the results.

The classic example of field collapsing is threaded email discussions where emails matching a specific query could be grouped under the original email message that started the conversation. You'll learn about field grouping in chapter 11.

POWERFUL AND FLEXIBLE QUERY SUPPORT

Solr provides a number of powerful query features including:

- Conditional logic using and, or, not
- Wildcard matching
- Range queries for dates and numbers
- Phrase queries with slop to allow for some distance between terms
- Fuzzy string matching
- Regular expression matching
- Function queries

Don't worry if you don't know what all of these terms mean as we'll cover all of them in depth in chapter 7.

JOINS

In SQL, you use a JOIN to create a relation by pulling data from two or more tables together using a common property such as a foreign key. But in Solr, joins are more like sub-queries in SQL in that you don't build documents by joining data from other documents. For example, with Solr joins, you can return child documents of parents that match your search criteria. One example where Solr joins are useful would be grouping all retweets of a Twitter message into a single group. We discuss joins in chapter 14.

DOCUMENT CLUSTERING

Document clustering allows you to identify groups of documents that are similar, based on the terms present in each document. This is helpful to avoid returning many documents containing the same information in search results. For example, if your search engine is based on news articles pulled from multiple RSS feeds, then it's likely that you'll have many documents for the same news story. Rather than returning multiple results for the same story, you can use clustering to pick a single representative story. Clustering techniques are discussed briefly in chapter 17.

IMPORT COMMON DOCUMENT FORMATS LIKE PDF AND WORD

In some cases, you may want to take a bunch of existing documents in common formats like PDF and Microsoft Word and make them searchable. With Solr this is easy because it integrates with Apache Tika project that supports most popular document formats. Importing rich format documents is covered in chapter 12.

IMPORT DATA FROM RELATIONAL DATABASES

If the data you want to search with Solr is in a relational database, then you can configure Solr to create documents using a SQL query. We cover Solr's data import handler (DIH) in chapter 12.

MULTILINGUAL SUPPORT

Solr and Lucene have a long history of working with multiple languages. Solr has language detection built-in and provides language-specific text analysis solutions for many languages. We'll see Solr's language detection in action in chapter 6.

1.4.3 New features in Solr 4

Before we wrap-up this chapter, let's look at a few of the exciting new features in Solr 4. In general, version 4 is a huge milestone for the Apache Solr community as it addresses many of the major pain-points discovered by real users over the past several years. We selected a few of the main features to highlight here but we'll also point out new features in Solr 4 throughout the book.

- Near-real-time search
- Atomic updates with optimistic concurrency
- Real-time get
- Write durability using a transaction log

- Easy sharding and replication using Zookeeper

NEAR-REAL-TIME SEARCH

Solr's Near-Real-Time (NRT) search feature supports applications that have a high velocity of documents that need to be searchable within seconds of being added to the index. With NRT, you can use Solr to search rapidly changing content sources such as breaking news and social networks. We cover NRT in chapter 13.

ATOMIC UPDATES WITH OPTIMISTIC CONCURRENCY

The atomic update feature allows a client application to add, update, delete, and increment fields on an existing document without having to resend the entire document. For example, if the price of a home in our example real estate application from section 1.2 changes, then we can send an atomic update to Solr to change the price field specifically.

You might be wondering what happens if two different users attempt to change the same document concurrently. In this case, Solr guards against incompatible updates using optimistic concurrency. In a nutshell, Solr uses a special version field named **_version_** to enforce safe update semantics for documents. In the case of two different users trying to update the same document concurrently, the user that submits updates last will have a stale version field so their update will fail. Atomic updates and optimistic concurrency are covered in chapter 12.

REAL-TIME GET

At the beginning of this chapter, we stated that Solr is a NoSQL technology. Solr's real-time get feature definitely fits within the NoSQL approach by allowing you to retrieve the latest version of a document using its unique identifier regardless of whether that document has been committed to the index. This is similar to using a key-value store like Cassandra to retrieve data using a row key.

Prior to Solr 4, a document wasn't retrievable until it was committed to the Lucene index. With the real-time get feature in Solr 4, you can safely decouple the need to retrieve a document by its unique ID from the commit process. This can be useful if you need to update an existing document after it's sent to Solr without having to do a commit first. As we'll learn in chapter 5, commits can be expensive and impact query performance.

DURABLE WRITES

When a document is sent to Solr for indexing, it's written to a transaction log to prevent data loss in the event of server failure. Solr's transaction log sits between the client application and the Lucene index. It also plays a role in servicing real-time get requests as documents are retrievable by their unique identifier regardless of whether they're committed to Lucene.

The transaction log allows Solr to decouple update durability from update visibility. This means that documents can be on durable storage but aren't visible in search results yet. This gives your application control over when to commit documents to make them visible in search results without risking data loss if a server fails before you commit. We'll discuss durable writes and commit strategies in chapter 5.

EASY SHARDING AND REPLICATION WITH ZOOKEEPER

If you're new to Solr, then you may not be aware that scaling previous versions of Solr was a cumbersome process at best. With SolrCloud, scaling is simple and automated because Solr uses Apache Zookeeper to distribute configuration and manage shard leaders and replicas. The Apache website (zookeeper.apache.org) describes Zookeeper as a "centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services."

In Solr, Zookeeper is responsible for assigning shard leaders and replicas and keeps track of which servers are available to service requests. SolrCloud bundles Zookeeper so you don't need to do any additional configuration or setup to get started with SolrCloud. We'll dig into the details of SolrCloud in chapter 16.

1.5 Summary

We hope you now have a good sense for what types of data and use cases Solr supports. As you learned in section 1.1, Solr is optimized to handle data that's text-centric, read-dominant, document-oriented, and has a flexible schema. We also learned that search engines like Solr aren't general-purpose data storage and processing solutions but are intended to power keyword search, ranked retrieval, and information discovery. Using the example of a fictitious real estate search application, we saw how Solr builds upon Lucene to add declarative index configuration and web services based on HTTP, XML, and JSON. Solr 4 can be scaled in two dimensions to support millions of documents and high-query traffic using sharding and replication. Solr 4 has no single points of failure.

We also touched on some key reasons why to choose Solr based on the perspective of key stakeholders. We saw how Solr addresses the concerns of software architects, system administrators, and even the CEO. Lastly, we covered some of Solr's main features and gave you pointers where you can learn more about each feature in this book.

We hope you're excited to continue learning about Solr, so now it's time to download the software and run it on your local system, which is what we'll do in chapter 2.

2

Getting to know Solr

This chapter covers

- Downloading and installing the Apache Solr 4.1 binary distribution
- Starting the example Solr server and indexing example documents
- Basic search features: sorting, paging, and results formatting
- Exploring the Solritas example search UI
- Self-guided tour of the Solr administration console
- Adapting the Solr example server to your specific needs

It's natural to have a sense of uneasiness when you start using an unfamiliar technology. You can put your mind at ease because Solr is designed to be easy to install and set up. In the spirit of being agile, you can start out simple and incrementally add complexity to your Solr configuration. For example, Solr allows you to split a large index into smaller subsets, called shards, as well as add replicas to increase your capacity to serve queries. But you don't need to worry about index sharding or replication until you need them.

By the end of this chapter, you'll have Solr running on your computer, know how to start and stop Solr, know your way around the web-based administration console, and have a basic understanding of key Solr terminology such as solr home, core, and collection.

What's in a name? Solr 4 vs. SolrCloud

You may have heard of SolrCloud and wondered what the difference is between Solr 4 and SolrCloud. Technically, SolrCloud is the code name for a sub-set of features in Solr 4 that makes it easier to configure and run a scalable, fault-tolerant cluster of Solr servers. Think of SolrCloud as a way to configure a distributed installation of Solr 4.

Also, SolrCloud doesn't have anything to do with running Solr in a cloud-computing environment like Amazon EC2, although you can run Solr in a cloud. We presume that the "cloud" part of the name reflects the underlying goal of the SolrCloud feature set to enable elastic scalability, high-availability, and ease of use we've all come to expect from cloud-based services. We cover SolrCloud in-depth in chapter 13.

Let's get started by downloading Solr from the Apache website and installing it on your computer.

2.1 Getting started

Before we can get to know Solr, we have to get it running on your local computer. This starts with downloading the binary distribution of Solr 4.1 from Apache and extracting the downloaded archive. Once installed, we'll show you how to start the example Solr server and verify that it's running by visiting the Solr administration console from your web browser. Throughout this process, we assume you're comfortable executing simple commands from the command line of your chosen operating system. There is no GUI installer for Solr, but you'll soon see that the process is so simple you won't need a GUI-driven installer.

2.1.1 Installing Solr

Installing Solr is a bit of a misnomer in that all you really need to do is download the binary distribution (.zip or .tgz) and extract it. Before you do that, let's make sure you have the necessary prerequisite Java 1.6 or greater (also known as J2SE 6) installed. To verify you have the correct version of Java, open a command-line on your computer and do:

```
java -version
```

You should see output that looks similar to the following:

```
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02, mixed mode)
```

If you don't have Java installed, we recommend you use Oracle's JVM.[1] Also, keep in mind that even though the Solr server requires Java, that doesn't mean you have to use Java in your application to interact with Solr. All client interaction with Solr happens over HTTP so you can use any language that provides an HTTP client library. In addition, a number of open source client libraries are available for Solr for popular languages like .NET, Python, Ruby, PHP, and of course Java.

Assuming you've Java installed, you're now ready to install Solr. Apache provides source and binary distributions of Solr; for now, we'll focus on the installation steps using the binary distribution. We cover how to build Solr from source in the appendix.

In your browser, go to the Solr home page http://lucene.apache.org/solr and click on the **Download** button for **Apache Solr 4.1** on the right; there is also a button for downloading

[1] http://www.oracle.com/technetwork/java/javase/downloads/index.html

Solr 3.6.x so be sure to choose the one for Solr 4. This will direct you to a mirror site for Apache downloads; it's advisable to download from a mirror site to avoid overloading the main Apache site. If you're on Windows, download: **solr-4.1.0.zip**. If you're on Unix, Linux, or Mac OS X, download: **solr-4.1.0.tgz**.

After downloading, move the downloaded file to a permanent location on your computer. For example, on Windows, you could move it to the **C:** root directory, or on Linux, choose a location like **/opt/solr**. For Windows users, we highly recommend that you extract Solr to a directory that doesn't have spaces in the name, i.e. avoid extracting Solr in directories like **C:\Documents and Settings** or **C:\Program Files**. Your mileage may vary on this, but being a Java-based software, you're likely to run into issues with paths that contain a space.

There is no formal installer needed because Solr is self-contained in a single archive file—all you need to do is extract it. When you extract the archive, all files will be created under the **solr-4.1.0** directory. On Windows, you can use the built-in Zip extraction support or a tool like WinZip. On Unix, Linux or Mac, do: `tar zxf solr-4.1.0.tgz`. This will create the directory structure shown in figure 2.1.

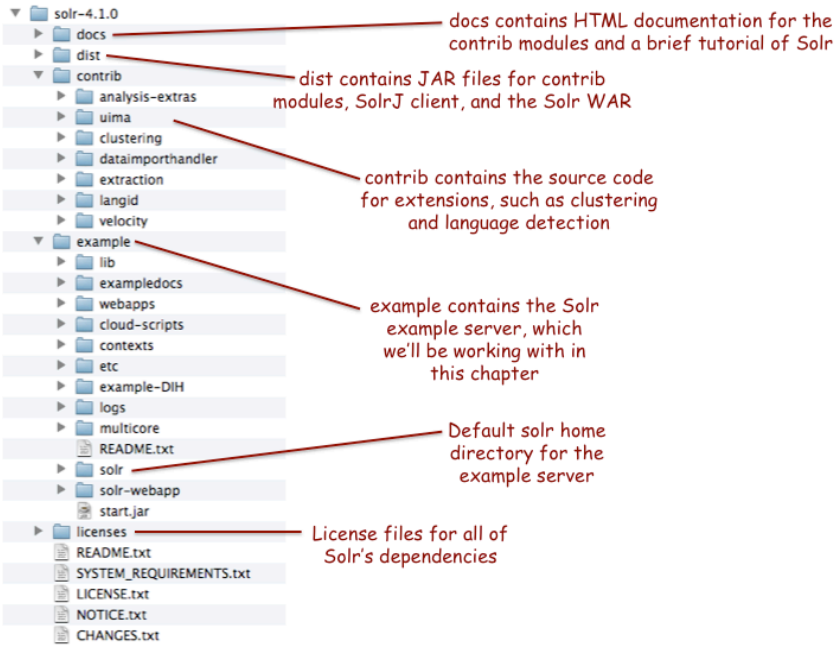


Figure 2.1 Directory listing of the solr-4.1.0 installation after extracting the downloaded archive on your computer. We'll refer to the top-level directory as $SOLR_INSTALL in the rest of this chapter.

We refer to the location where you extracted the Solr archive (.zip or .tgz) as **$SOLR_INSTALL** in the rest of this chapter. We use this name because as you'll see shortly,

Solr home will be a different path, so we didn't want to use $SOLR_HOME as the alias for the top-level directory where you extracted Solr. So now that Solr is installed, you're ready to start it up.

2.1.2 Starting the Solr example server

To start Solr, open a command line and do the following:

```
cd $SOLR_INSTALL/example
java -jar start.jar
```

Remember that **$SOLR_INSTALL** is the alias we're using to represent the directory where you extracted the Solr download archive, such as **C:\solr-4.1.0** on Windows. That's all there is to starting Solr.

During initialization, you'll see some log messages printed to the console. If all goes well, you should see the following log message at the bottom:

```
... :INFO:oejs.AbstractConnector:Started  SocketConnector@0.0.0.0:8983
```

WHAT HAPPENED?

That was so easy, you might be wondering what was actually accomplished. To be clear, you now have a running version of Solr 4.1 on your computer. You can verify that Solr started correctly by directing your web browser to the Solr administration page at: **http://localhost:8983/solr**. Figure 2.2 provides a screen shot of the Solr administration console; please take a minute to get acquainted with the layout and navigational tools in the console.

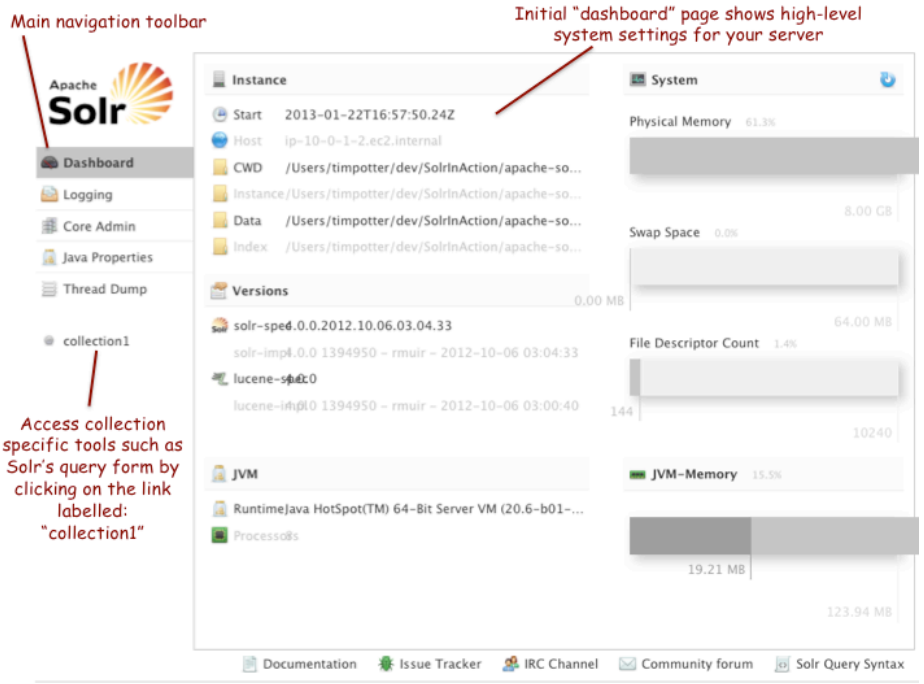


Figure 2.2 The Solr 4 administration console, which provides a wealth of tools for working with your new Solr instance. Click on the link labeled "collection1" to access more tools such as the query form.

Behind the scenes, start.jar launched a Java web server named Jetty, listening on port 8983. Solr is a web application running in Jetty. Figure 2.3 illustrates what is now running on your computer.

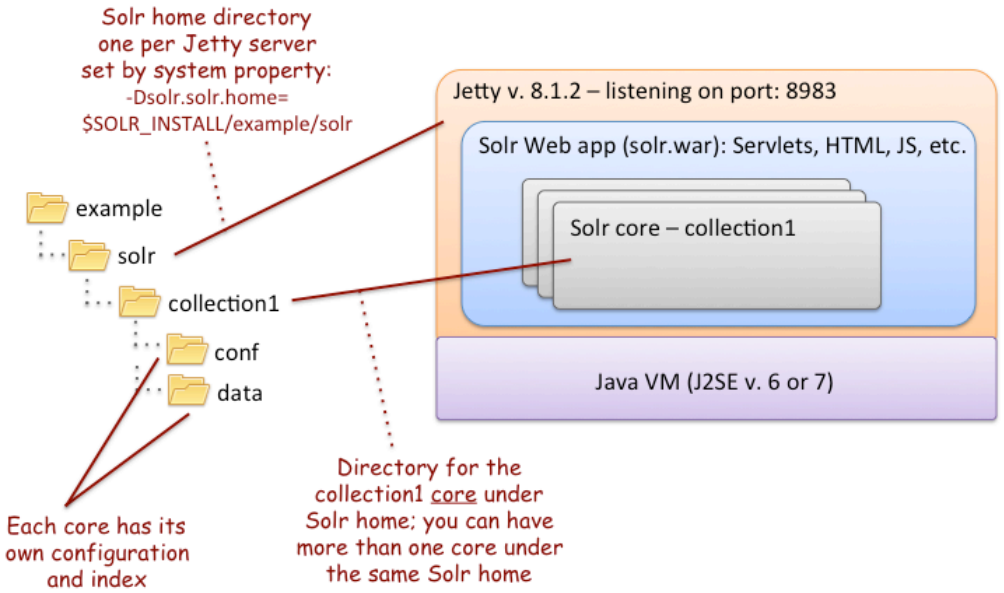


Figure 2.3 Solr from a systems perspective showing the Solr web application (solr.war) running in Jetty on top of Java. There is one Solr home directory set per Jetty server using Java system property solr.solr.home. Solr can host multiple cores per server and each core has a separate directory containing core-specific configuration and index (data) under Solr home, e.g. collection1.

TROUBLESHOOTING

There's not much that can go wrong when starting the example server. The most common issue if the server doesn't start correctly is the default port 8983 is already in use by another process. If this is the case, you'll see an error that looks like: `java.net.BindException: Address already in use`. This is easy to resolve by changing the port Solr binds to by changing your start command to specify a different port for Jetty to bind to using: `java -Djetty.port=8080 -jar start.jar`. Using this command, Jetty will bind to port 8080 instead of 8983.

Jetty versus Tomcat

We recommend just staying with Jetty when first learning Solr. If your organization already uses Tomcat or some other Java web application server, such as Resin, then you can deploy the Solr WAR file. Since we're just getting to know Solr in this chapter, we'll refer you to the Appendix B to learn how to deploy the Solr WAR.

Solr uses Jetty to make the initial setup and configuration process a no-brainer. However, this doesn't mean that Jetty is a bad choice for production deployment. If your

organization already has a standard Java web application platform, then Solr will work with it. However, if you have some choice, then we recommend you try out Jetty. It's fast, stable, mature, and easy to administer and customize. In fact, Google uses Jetty for their AppEngine, see http://www.infoq.com/news/2009/08/google-chose-jetty/, which gives great credibility to Jetty as a solid platform for running Solr in even the most demanding environments!

STOPPING SOLR

For local operation, you can just kill the Solr server by doing Ctrl-C in the console window where you started Solr. Typically, this is safe enough for development and testing. Jetty does provide a safer mechanism for stopping the server, which will be discussed in chapter 12.

Now we have a running server, let's take a minute to understand where Solr gets its configuration information and where it manages its Lucene index. Understanding how the example server you just started is configured will help you when you're ready to start configuring a Solr server for your application.

2.1.3 Understanding Solr Home

In Solr, a “core” is composed of a set of configuration files, Lucene index files, and Solr's transaction log. One Solr server running in Jetty can host multiple cores. Recall in chapter 1, we designed a real estate search application that had multiple cores, one for houses and a separate core for land listings. We used two separate cores because the indexed data was different enough to justify having two different index structures. The Solr example server you started in section 2.1.2 has a single core named "collection1".

As a brief aside, Solr also uses the term "collection", which really only has meaning in the context of a Solr cluster where a single index is distributed across multiple servers. Consequently, we feel it's easier to focus on understanding what a Solr core is for now. We'll return to the distinction between core and collection in chapter 13 when we discuss SolrCloud.

Solr home is a directory structure that encapsulates one or more cores, which are configured by `solr.xml`. Solr also provides a core admin API that allows you to create, update, and delete cores programmatically from your application. Behind the scenes the core admin API makes changes to the `solr.xml` configuration file. Listing 2.1 shows the default solr.xml for the example server. Out of the box, you don’t need to make any changes to this file.

Listing 2.1 Default solr.xml for example server defining the collection1 core

```
<solr persistent="true"> #A
  <logging enabled="true">
    <watcher size="100" threshold="INFO" />
  </logging>

  <cores adminPath="/admin/cores"  #B
```

```
          defaultCoreName="collection1"
          host="${host:}" hostPort="${jetty.port:}"
          hostContext="${hostContext:}"
          zkClientTimeout="${zkClientTimeout:15000}">
      <core name="collection1" instanceDir="collection1" /> #C
    </cores>
  </solr>
```

#A Persistent attribute controls whether or not changes made from the core admin API are persisted to this file
#B Define one or more cores under the <cores> element
#C The collection1 core configuration and index files are in the collection1 directory under solr home

Each Solr process has one and only one solr home directory, which is set by a global Java system property: `solr.solr.home`. Figure 2.4 shows a directory listing of the default Solr home "solr" for the example server.

Figure 2.4 Directory listing of the default Solr home directory for the Solr examples. It contains a single core named "collection1," which is configured in solr.xml. The collection1 directory corresponds to the core named "collection1" and contains core-specific configuration files, Lucene index, and transaction log.

We'll learn more about the main Solr configuration file for a core, named **solrconfig.xml**, in chapter 4. Also, **schema.xml** is the main configuration file that governs index structure and text analysis for documents and queries; you'll learn all about schema.xml in chapter 5. For now, just take a moment to scan figure 2.3 so that you have a sense for the basic structure.

The example directory contains two other solr home directories for exploring advanced functionality. Specifically, the `example/example-DIH` directory provides a Solr core for learning about the data import handler (DIH) feature in Solr. Also, the `example/multicore` directory provides an example of a multi-core configuration. We'll learn more about these features later in the book. For now, let's continue with the simple example by adding some documents to the index, which you'll need to work through the examples in section 2.2 below.

2.1.4 Indexing the example documents

When you first start Solr, there are no documents in the index. It's just an empty server waiting to be filled with data to search. We cover indexing in more detail in chapter 5. For now, we'll gloss over the details in order to get some example data in Solr index so that we can try out some queries. Open a new command-line interface and do the following:

```
cd $SOLR_INSTALL/example/exampledocs
java -jar post.jar *.xml
```

You should see output that looks like:

```
SimplePostTool version 1.5
Posting files to base url http://localhost:8983/solr/update using content-
type application/xml..
POSTing file gb18030-example.xml
POSTing file hd.xml
POSTing file ipod_other.xml
POSTing file ipod_video.xml
POSTing file manufacturers.xml
POSTing file mem.xml
POSTing file money.xml
POSTing file monitor.xml
POSTing file monitor2.xml
POSTing file mp500.xml
POSTing file sd500.xml
POSTing file solr.xml
POSTing file utf8-example.xml
POSTing file vidcard.xml
14 files indexed.
COMMITting Solr index changes to http://localhost:8983/solr/update..
```

The post.jar file sends XML documents to Solr using HTTP POST. After all the documents are sent to Solr, the post.jar application issues a commit, which makes the example documents findable in Solr. To verify that the example documents were added successfully, go to the Query page in the Solr administration console (**http://localhost:8983/solr**) and execute the "find all documents" query (`*:*`). You need to click on the "collection1" link on

the left to access the Query page. Figure 2.5 shows what you should see after executing the find all documents query.

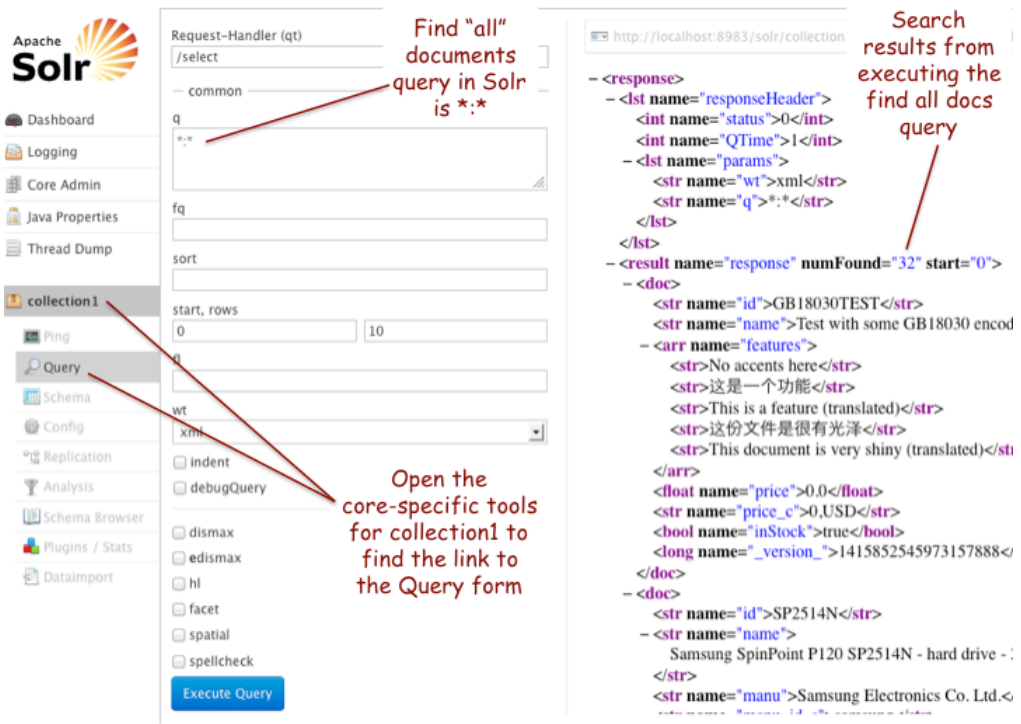


Figure 2.5 Screenshot of the Query form on the Solr administration console. You can verify that the example documents were indexed correctly by executing the find all documents query *:*

At this point, we have a running Solr instance with some example documents loaded.

2.2 Searching is what it's all about

Now it's time to see Solr shine. Without a doubt, Solr's main strength is powerful query processing. Think about it this way, who cares how scalable or fast a search engine is if the results it returns aren't useful or accurate? In this section, you'll see Solr query processing in action, which we think will help you see why Solr is such a powerful search technology.

Throughout this section, pay close attention to the link between each query we execute and the documents that Solr returns, especially the order of the documents in the results. This will help you start to think like a search engine, which will come in handy in chapter 3 when we cover core search concepts.

2.2.1 Exploring Solr's query form

You've already used Solr's query form to execute the "find all documents" query (*:*). Let's take a quick tour of the other features on this form so that you get a sense for the types of queries Solr supports. Figure 2.6 provides some annotations of key sections of this form. Take a minute to read through each annotation in the diagram.

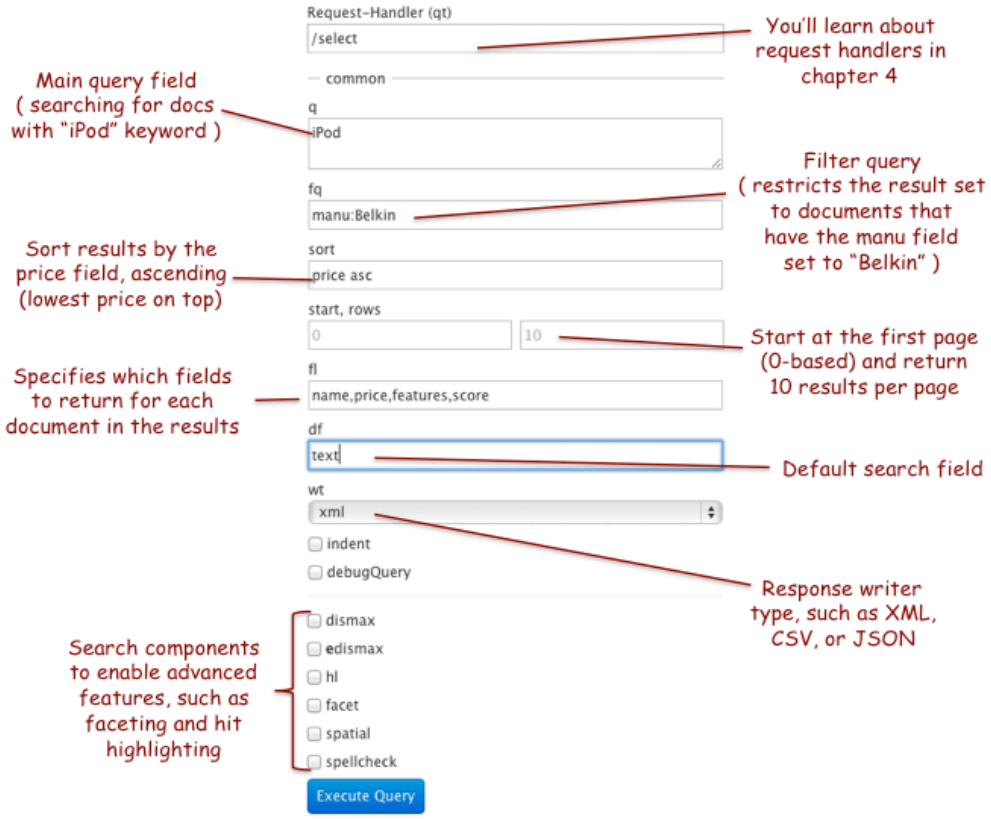


Figure 2.6 Annotated screen shot of Solr's query form to illustrate the main features of Solr query processing, such as filters, results format, sorting, paging, and search components.

In figure 2.6, we formulate a query that returns two of the example documents we added in section 2.1.4 above. Take a moment to fill-out the form and execute the query in your own environment. Do the two documents that Solr returned make sense? Table 2.1 provides an overview of the form fields we're using for this example.

Table 2.1 Overview of query parameters from figure 2.6

Form field	Value	Description
q	iPod	Main query parameter; documents are scored by their similarity to terms in this parameter.
fq	manu:Belkin	Filter query; restricts the result set to documents matching this filter but doesn't affect scoring. In this example, we filter results that have manufacturer field "manu" equal to "Belkin"
sort	price asc	Specifies the sort field and sort order; in this case we want results sorted by the price field in ascending order (asc) so that documents with the lowest price are listed first.
start	0	Specifies the starting page for results; since this is our first request, we want the first page using 0-based indexing. Start should be incremented by the page size to advance to the next page.
rows	10	Page size; restricts the number of results returned per page, in this case 10. The next page index will be 10 and not 1.
fl	name,price, features,score	List of fields to return for each document in the result set. The "score" field is a built-in field that holds each document's relevancy score for the query. You have to request the score field explicitly as is done in this example.
df	text	Default search field for any query terms that don't specify which field to search on; text is the catch all field for the example server.
wt	xml	Response writer type; governs the format of the response

As we discussed in chapter 1, section 1.2.3, all interaction with Solr's core services, such as query processing, is performed with HTTP requests. When you fill out the query form, an HTTP GET request is created and sent to Solr. The form field names shown in table 2.1 correspond to parameters passed to Solr in the HTTP GET request. Listing 2.1 shows the actual HTTP GET request sent to Solr when you execute the query depicted in figure 2.6; note that the actual request doesn't include line breaks between the parameters, which we've included here to make it easier to see the separate parameters.

Listing 2.1 Breakdown of the HTTP GET request sent by the Query form

```
http://localhost:8983/solr/collection1/select?    #A
q=iPod&                                           #B
fq=manu%3ABelkin&                                 #C
sort=price+asc&                                   #D
fl=name%2Cprice%2Cfeatures%2Cscore&                #E
df=text&                                          #F
wt=xml&                                           #G
```

```
start=0&rows=10                                                    #H
```

#A Invokes the "select" request handler for the "collection1" core
#B Main query component looking for documents containing "iPod"
#C Filter documents that have manu field equal to "Belkin"
#D Sort results by price in ascending order (smallest to largest)
#E Return the name, price, features, and score fields in results
#F Default search field is "text"
#G Return results in XML format
#H Start at page 0 and return up to 10 results

Looking for more example queries?

We cover queries in more depth in chapter 7. But, if you don't want to wait that long and want to see more queries in action, then we recommend looking at the Solr tutorial provided with Solr. Open **$SOLR_INSTALL/docs/tutorial.html** in your web browser and you'll find some additional queries for the example documents you loaded in the previous section 2.1.4.

Lastly, we probably don't have to tell you that this form isn't designed for end users; the Solr team built the query form so that developers and administrators have a way to send queries without having to formulate HTTP requests manually or develop a client application just to send a query to Solr. But, let's be clear that with Solr, you're responsible for developing the user interface to Solr. As we'll see in section 2.2.5 below, Solr provides an example search UI, called Solritas, to help you get started building your own awesome search application.

2.2.2 What comes back from Solr when you search

We've seen what gets sent to Solr, so now let's learn about what comes back from Solr in the results. The key point in this section is that Solr returns documents that match the query as well as additional information that can be processed by your Solr client to deliver a quality search experience. The operative phrase being "by your Solr client"! Solr returns the raw materials that you need to translate into a quality search experience for your users.

Figure 2.7 shows what comes back from the example query we used in section 2.2.1. As you can see, the results are in XML format and are sorted by lowest to highest price. Each document contains the term "iPod". Paging doesn't really come into play with this result set because there are only two results total.

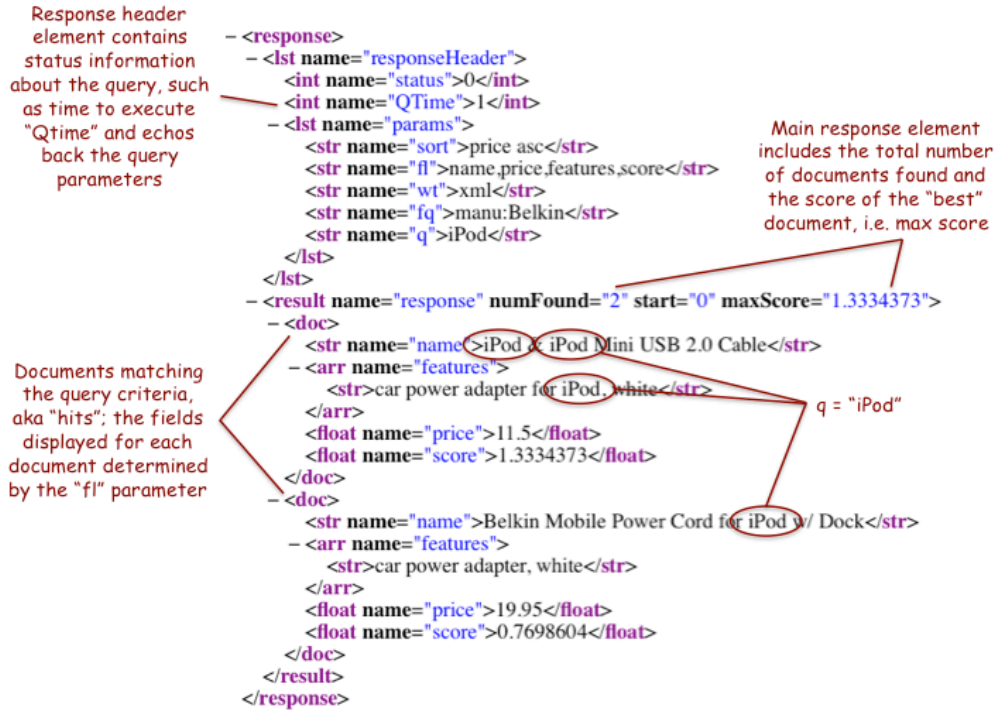


Figure 2.7 Solr response in XML format from our sample query from figure 2.6

So far, we've only seen results returned as XML, but Solr also supports other formats such as CSV, JSON, and language specific formats for popular languages. For instance, Solr can return a Python specific format that allows the response to be safely parsed into a Python object tree using the `eval` function.

2.2.3 Ranked retrieval

As we touched upon on chapter 1, the key differentiator between Solr's query processing and that of a database or other NoSQL data store is ranked retrieval—the process of sorting documents by their relevance to a query, where the most relevant documents are listed first.

Let's see ranked retrieval at work with some of the example documents you indexed in section 2.1.4. To begin, enter "`iPod`" in the **q** text box, "`name,features,score`" in the **fl** text field, and push the Execute button. This should return three documents sorted in descending order by score. Take a moment to scan the results and decide if you agree with the ranking for this simple query.

Intuitively, the ordering makes sense because the query term "iPod" occurs three times in the first document listed, twice in the name and once in the features; it only occurs once in

the other documents. The actual numeric value of the score field isn't as important as it's used internally by Lucene to do the ranking. The key take-away is that every document that matches a query is assigned a relevance score for that specific query and results are returned in descending order by score; the higher the score, the more relevant the document is to the query.

Next, change your query to be "`iPod power`" and you'll see that the same three documents are returned and are in the same order. This is because all three documents contain both query terms in either their name or features field. However, the scores of the top two documents are much closer: 1.521 and 1.398 for the second query versus 1.333 and 0.770 (rounded) for the first query. This makes sense because "power" occurs twice in the second document so its relevance to the "`iPod power`" query is much higher than its relevance to the "`iPod`" query.

Lastly, change your query to be "`iPod power^2`" which boosts the "power" query term by 2. In a nutshell, this means that the "power" term is twice as important to this query as the "iPod" term, which has an implicit boost of 1. Again, the same three documents are returned but in a different order. Now the top document in the results is "Belkin Mobile Power Cord for iPod w/ Dock" because it contains the term "power" in the name and features field and we told Solr that "power" is twice as important as "iPod" for this query.

Now you have a taste of what ranked retrieval looks like. You'll learn more about ranked retrieval and boosting in chapters 3 and 7. Let's move on and see some other features of query processing, starting with how to work with queries that return more than three documents using paging and sorting.

2.2.4 Paging and sorting

Our example Solr index only contains 32 documents, but a production Solr instance typically has millions of documents. You can imagine that in a Solr instance for an electronics super-store, a query for "iPod" would probably match thousands of products and accessories. To ensure results are returned quickly, especially on bandwidth constrained mobile devices, you don't want to return thousands of results at once, even if the most relevant are listed first.

PAGING

The solution, of course, is to return a small sub-set of results called a "page" along with navigational tools to allow the user to request more pages if needed. Paging is a first-class concept in Solr query processing in that every query includes parameters that control the page size (rows) and starting position (start). By default, Solr uses a page size of 10, but you can control that using the "rows" parameter in the query request. To request the "next" page in the results, you increment the start parameter by the page size. For example, if you're on the first page of results (start=0), then to get the next page, you increment start parameter by the page size, i.e. start=10.

It's important to use as small a page size as possible to satisfy your requirements because the underlying Lucene index isn't optimized for returning many documents at once. Rather, Lucene is optimized for query processing so the underlying data structures are

designed to maximize matching and scoring documents. Once the search results are identified, Solr must re-construct each document, in most cases by reading data off disk. It uses intelligent caching to be as efficient as possible, but in comparison to query processing, results construction is a slow process, especially for large page sizes. Consequently, you'll get much better performance from Solr using small page sizes.

SORTING

As we learned in section 2.2.2, results are sorted by relevance score, in descending order (highest to lowest score). However, you can request Solr to sort results by other fields in your documents. You've already seen an example of this in section 2.2.1 where we sorted results by the **price** field in ascending order, which produces the lowest priced products at the top.

Sorting and paging go hand-in-hand because the sort order determines the page position for results. To help get you thinking about sorting and paging, consider the question of whether Solr will return deterministic results when paging without specifying a sort order? On the surface, this seems obvious because the results are descending sorted by score if you don't specify a sort parameter. But, what if all documents in a query have the same score? For example, if your query is "`inStock:true`" then all matching documents will have the same score; be sure to verify this yourself using the Query form.

It turns out that Solr will indeed return all documents when you page through the results even though the score is the same. This works because Solr finds all documents that match a query and then applies the sorting and paging offsets to the entire set of documents. In other words, Solr keeps track of the entire set of documents that match a query independently of the sorting and paging offsets.

2.2.5 Enabling search components

The query form contains a list of check boxes that enable advanced functionality during query processing; in Solr speak these additional features are called "search components". As shown in figure 2.6, the form contains check boxes that reveal additional form fields to activate the following search components:

- dismax – Disjunction Max query processor (chapter 7)
- edismax – Extended Disjunction Max query processor (chapter 7)
- hl – Hit highlighting (chapter 9)
- facet – Faceting (chapter 8)
- spatial – Geo-spatial search, such as sorting by geo-distance (chapter 14)
- spellchecking – Spell checking on query terms (chapter 10)

If you click on any of these checkboxes, you'll see that it's not clear what to do when you look at the form. That's because using these search components from the query form requires some additional knowledge that we can't cover quickly in this getting started chapter. Rest assured that we cover each of these components in-depth later in the book.

For now, though, we can see some of these search components in action using Solr's example search interface, called "Solritas", available in your local Solr instance at: **http://localhost:8983/solr/collection1/browse**. Navigate to this URL in your web browser and you'll see a screen that looks like figure 2.8.

Figure 2.8 The Solritas **Simple** example, which illustrates how to use various search components, such as faceting, More Like This, hit highlighting, and spatial, to provide a rich search experience for your users.

As shown at the top of figure 2.8, Solr provides three examples to choose from: **Simple**, **Spatial**, and **Group By**. We'll briefly cover the key aspects of the **Simple** example here and encourage you to explore the other two examples on your own.

Take a moment to scan over figure 2.8 to identify the various search components at work. One of the more interesting search components in this example is the facet search component shown on the left side of the page, starting with the header **Field Facets**. The facet component categorizes field values in the search results into useful sub-sets to help the user refine their query and discover new information. For instance, when we search for "video", Solr returns three example documents and the faceting component categorizes the

"cat" field of these documents into three sub-sets: electronics (3), graphics card (2), and music (1). Click on the music facet link and you'll see the results are filtered from three documents down to only one. The idea here is that in addition to search results, you can help users refine their search criteria by categorizing the results into different filters. Take a few minutes to explore the various facets shown in the example. We cover facets in detail in chapter 8.

Next, let's take a look at another search component that isn't immediately obvious from figure 2.8, namely the spell check component. To see how spell checking works, type "vydeoh" in the search box instead of "video". Of course no results are found, as shown in figure 2.9, but Solr does return a link that effectively asks the user if they meant "video" and if so, they can re-search using the link.

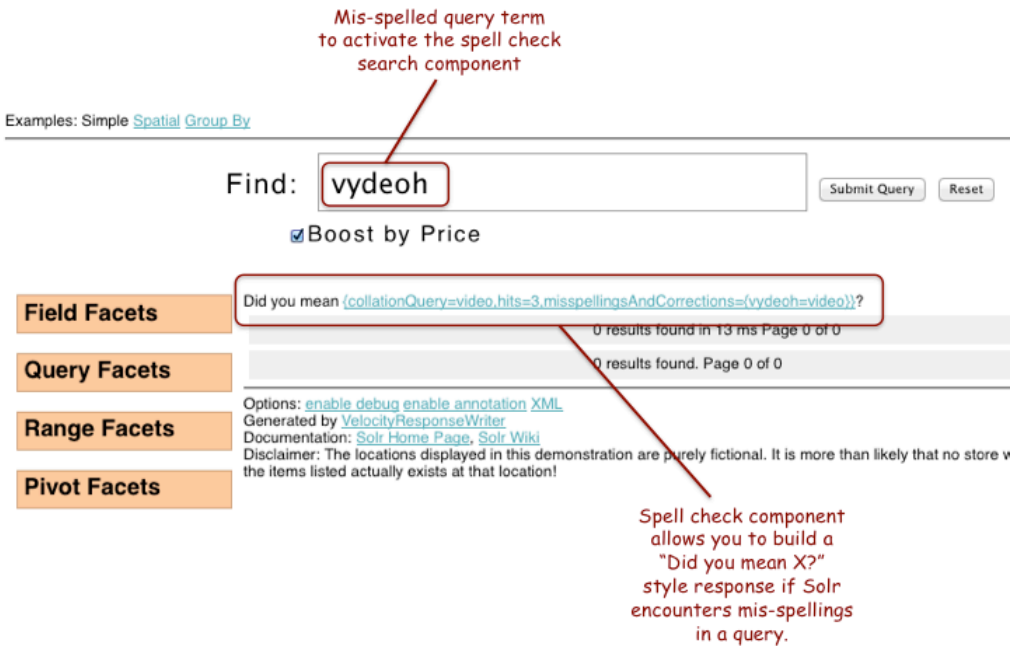


Figure 2.9 Example of how the spell check component allows your search UI to prompt the user to re-search using the correct spelling of a query term, in this case, Solr found "video" as the closest match for "vydeoh".

There's a lot of powerful functionality packed into the three Solritas examples and we encourage you to spend a little time with each. For now, let's move on and tour the rest of the administration console.

2.3 Tour the Solr administration console

At this point, you should have a good feel for the query form, so let's take a quick tour of the rest of the administration console shown in figure 2.10.

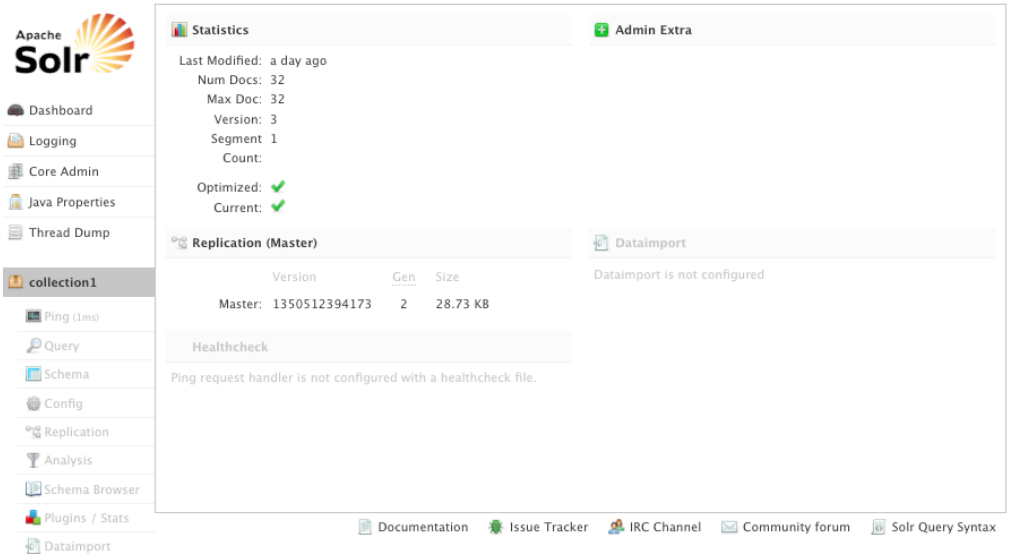


Figure 2.10 The Solr administration console; explore each page using the toolbar on the left.

Rather than spending your time reading about the administration panel, we think it's better to just start clicking through some of the pages yourself. Thus, we leave it as an exercise for you to visit all the links in the administration console to get a sense for what is available on each page. To get you started, here are some highlights of what the administration console provides:

- See how your Solr is configured from **Dashboard**
- View recent log messages from **Logging**
- Temporarily change log verbosity settings from **Level** under Logging
- Add and manage multiple cores from **Core Admin**
- View Java system properties from **Java Properties**
- Get a dump of all active threads in your JVM from **Thread Dump**

In addition to the main pages described above, there are a number of core specific pages for each core in your server. Recall that the example server we've been working with has only one core named "collection1". The core-specific pages allow you to do the following:

- View core specific properties such as the number of Lucene segments from the main core page, e.g. **collection1**
- Send a quick request to a core to make sure it's alive using **Ping**
- Execute queries against the core's index using **Query**
- View the currently active schema.xml for the core from **Schema**; you'll learn all about schema.xml in chapters 5 and 6
- View the currently active solconfig.xml for the core from **Config**; you'll learn more about solrconfig.xml in chapter 4
- See how your index is replicated to other servers from **Replication**
- Analyze text from **Analysis**; you'll learn all about text analysis in chapter 6, including how to use the Analysis form
- Determine how fields in your documents are analyzed from **Schema Browser**
- Get information about the top terms for a field using **Load Term Info** on the Schema Browser
- View the status and configuration for Plug-ins from **PlugIns / Stats**; you'll learn all about PlugIns in chapter 4
- View statistics about core Solr cache regions, such as how many documents are in the documentCache from **PlugIns / Stats**
- Manage the data import handler from **Dataimport**; this isn't enabled in the example server

We'll dig into the details for most of these pages in various places throughout the book, when it's more appropriate. For instance, you'll learn all about the Analysis page in chapter 6 when we cover text analysis. For now, take a few moments to explore these pages on your own. To give your self-guided tour some direction, see if you can answer the following questions about your Solr server.

1. What's the value of the **lucene-spec** version property for your Solr server?
2. What's the log level of the **org.apache.solr.core.SolrConfig** class?
3. What's the value of the **maxDoc** property for the collection1 core?
4. What's the value of the **java.vm.vendor** Java system property?
5. What's the **segment count** for the collection1 core?
6. What's the response time of **pinging** your server?
7. What's the top term for the manu field? (hint: select the "manu" field in the schema browser and click on the **Load Term Info** button)
8. What's the current size of your documentCache? (hint: think stats)

9. What's the analyzed value of the name "Belkin Mobile Power Cord for iPod w/ Dock"? (hint: select the name field on the **Analyzed** page)

Let's now turn our attention to what needs to be done to start customizing Solr for your specific needs.

2.4 Adapting the example to your needs

So now that you've had a chance to work with the example server, you might be wondering what's the best way to proceed with adapting it to your specific needs. You have a couple of choices here. First, you could just use the example directory as-is and start making changes to it to meet your needs. However, we think it's better to keep a copy of example around and make your application specific changes in a clone of example. This allows you to refer back to example in case you break something when working on your own application.

If you choose the latter approach, then you need to select a name for the directory that is more appropriate for your application than "example". For example, if we were building the real estate search application described in chapter 1, then we might name the directory: realestate. Once you've settled on a name, do the following steps to create a clone of the example directory in Solr:

1. Create a deep copy of the example directory; e.g. `cp -R example realestate`
2. Clean-up the cloned directory to remove un-used solr home directories, such as example-DIH and multicore; they'll be in example if you need to refer back to them.
3. Under the solr home directory, rename collection1 to something more intuitive for your application.
4. Update solr.xml to point to the name of your new collection by replacing "collection1" with the name of your core from step 3.

Note that you don't need to make any changes to the Solr configuration files, such solrconfig.xml or schema.xml, at this time. These files are designed to provide a good experience out-of-the-box and let you adapt them to your needs iteratively without having to swallow such a big pill at once.

Cleaning up your index

There may come a time when you want to start with a fresh index. After stopping Solr, you can remove all documents by deleting the contents of the data directory for your core, such as solr/collection1/data/*. When you restart Solr, you'll have a fresh index with 0 documents.

Restart Solr from your new directory using the same process from section 2.1.2. For example, to restart our clone for the **realestate** application, we'd do:

```
cd $SOLR_INSTALL/realestate
java -jar start.jar
```

Lastly, you might be wondering about setting JVM options, configuring backups, monitoring, setting Solr up as a service, and so on? We feel these are important concerns when you're ready to go to production so we cover these questions in chapter 12, when we discuss taking Solr to production.

2.5 Summary

To recap, we started by installing Solr 4.1 from the binary distribution Apache provided. In reality, the installation process was only a matter of choosing an appropriate directory where to extract the downloaded archive (.zip or .tgz) and then doing the extraction. Next, we started the example Solr server and added some example documents using the post.jar command-line application.

After adding documents, we introduced you to Solr's query form, where you learned the basic components of a Solr query. Specifically, you learned how to construct queries containing a main query parameter "q" as well as an optional filter "fq." You saw how to control which fields are returned using the "fl" parameter and how to control the ordering of results using "sort." We also touched on ranked retrieval concepts where results are ordered by relevancy score. You'll learn more about queries in chapter 7.

We also introduced you to search components and provided some insights into how they work in Solr using the Solritas example user interface. Specifically, you saw an example of how the facet component allows users to refine their search criteria using dynamically generated filters called facets. We also touched on how the spell check component allows you to prompt users "did you mean X? " when their query contains a misspelled term.

Next, we gave you some tips on what other tools are available in the Solr administration console. You'll find many great tools and statistics available about Solr, so we hope you were able to answer the questions we posed as you walked through the administration console in your browser. Lastly, we presented the steps to clone the example directory to begin customizing it for your own application. We think this is a good way to start so that you always have a working example to refer to as you customize Solr for your specific needs.

So now that you have a running Solr instance, it's time to learn about core Solr concepts. In chapter 3 you'll gain a better understanding of core search concepts that will help you throughout the rest of your Solr journey.

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 most relevant documents
- How to balance returning relevant results versus ensuring they're all returned
- How to model your content into a denormalized document
- How search 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 as opposed to a traditional database. In this chapter, we'll provide a solid understanding of how a search engine stores documents in its internal index, how it calculates a relevancy score to ensure only the "best" results are returned for display, and how Solr is able to scale to handle billions of documents as it still maintains lightning-fast search response times.

Our main goal for this chapter is to provide you with the theoretical underpinnings necessary to understand and maximize your use of Solr. If you have a solid background in search technology or information retrieval already, 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 upon 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 today 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 thousands of others. Search engines, and Solr in particular, help to solve a particular class of problem very well – those 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 a 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 some 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 which 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 even 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 is received. Listing 3.1 shows an example document, defining the values 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 search 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 specified in the search.

It is worth noting that, unlike Lucene, Solr is not schema-less. All field types must be defined, and all field names (or dynamic field naming patterns) must 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 mapped to a particular field type should they appear in a document and need to be processed.

A document, then, is a collection of fields that map to particular field types defined in a schema. Each field in a document is analyzed according to its field type and stored in 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 is helpful to understand what fundamental problem search engines are solving.

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

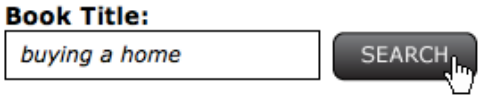


Figure 3.1 Example search interface, as would be seen on a typical website, demonstrating how a user would submit a query to your application

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

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 (Structured Query Language) database would be to simply query for the exact text that users enter:

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

The problem with this approach, of course, 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 ever 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, while 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 which 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	**Non-matching Books**
Buying a** New **Home	*The Beginner's Guide to **Buying a** House*
	How to Buy Your First House
	*Purch**a**sing **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 the above 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 merely 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 at least one term

Matching books	Non-matching 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*	

We have just seen that the first query (requiring all words to match) resulted in many relevant books not being found, while 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.

The above examples demonstrate several difficulties with this implementation:

- It does not understand linguistic variations of words such as "buying" vs "buy"
- It does not understand synonyms of words such as "buying" and "purchasing"
- 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 much match).

- There is no sense of relevancy ordering in the results – books which only match 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, as 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 really shine in solving problems like the ones listed above. 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 utilizing an index that maps content to documents instead of mapping documents to content like a traditional database model. This “inverted index” is at the heart of how search engines work.

3.1.3 The inverted index

Solr utilizes 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 (I recommend picking up a copy of *Lucene in Action* if you want a deeper dive), it is important to understand the high-level structure of the inverted index.

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.

Table 3.5 Mapping of text from multiple documents into an inverted index. The left table contains nine original documents with their content, while the right table represents an inverted search index containing each of the terms from the original content mapped back to the documents in which they can be found.

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 #
a	1,3,4,5, 6,7,8
becoming	8
beginner's	6
buy	9
buying	4,5,6
car	4
child	3
cooking	1
decorating	2
first	9
fun	1
…	…

(Continued)

…	…
guide	1,6
home	2,5,7,8
house	6,9
how	3,9
new	4,5,8
owner	8
purchasing	7
raise	3
the	6
to	1,6,9
your	2,9

Table 3.5 demonstrates the process of mapping the content from documents into an inverted index. 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 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 any additional text transformations are possible, not just these simple ones – terms can be modified, added, or even removed during the content analysis process, which will be covered in detail in chapter 5.

A few 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 alphanumeric 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 will see in the next section, the structure of Lucene's inverted index allows many powerful query capabilities that maximize both the speed and flexibility of keyword based searching.

3.1.4 Terms, phrases, & 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 this inverted search index and utilizing Boolean logic and fuzzy queries to enhance these lookup capabilities. Referring back to our book-searching example, let's take a look at a simple query of "new house", as portrayed in figure 3.2.

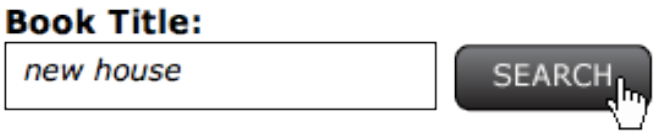


Figure 3.2 Simple search to demonstrate nuances of query interpretation

We saw in the last section that all of the text in our content field was broken up into individual terms when inserted into the Lucene index. Now that we have an incoming query, we 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, are very 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 standard 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 which means that part of the query immediately following it is required to exist in any documents matched, whereas the "AND" keyword is a binary operator which 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*

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** vs /select?q=new house&**q.op=AND**

Do note, however, that if you change the default operator from OR to AND that now parts of the query without an operator will always be required unless you explicitly place an OR between them to override the default AND operator.

NEGATED TERMS

In addition to making parts of a query optional and required, it is 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 the queries above, no document which contains the word "rental" will be returned, only documents matching "new" or "house."

PHRASES

Solr does not just support searching single terms, however; it can also search for phrases, which ensures 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 above query expressions, one final basic Boolean construct which Solr supports is the grouping of terms, phrases, and other query expressions. The Solr query syntax can represent arbitrarily complex queries through grouping terms together using parenthesis like 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 very 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 utilize the internal Lucene inverted index to find matching documents. Let us recall our index of books from earlier, re-produced 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
beginner's	6	home	2,5,7,8
buy	9	house	6,9
buying	4,5,6	how	3,9

car	4
child	3
cooking	1
decorating	2
first	9
fun	1
…	…

new	4,5,8
owner	8
purchasing	7
raise	3
the	6
to	1,6,9
your	2,9

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

The answer is that the query `new home` is actually 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		**Documents**
New	=>	4,5,8
Home	=>	2,5,7,8

Once the list of matching documents is found for each term, Lucene will perform set operations to arrive at an appropriate final result set which matches the query. Assuming our 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.

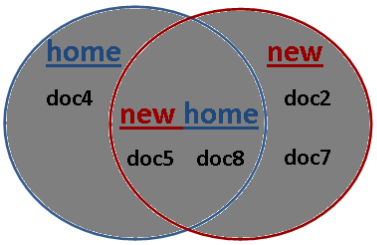


Figure 3.3 Results returned from an Union query using the “OR” operator

Likewise, if our 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.

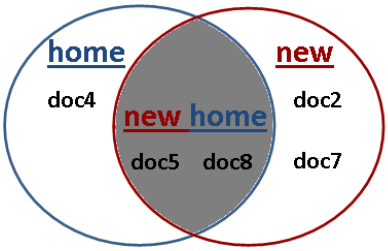


Figure 3.4 Results returned from an Intersection query using the "AND" operator

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

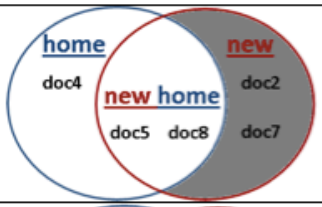
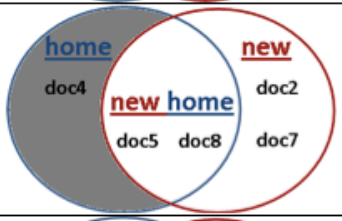
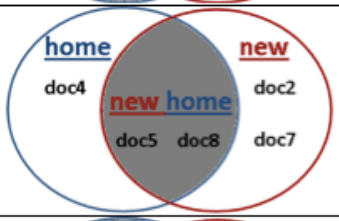
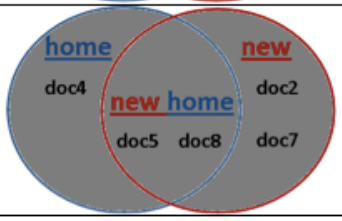
new –home	
-new home	
+new +home	
new home	

Figure 3.5 Graphical representation of using common boolean query operators

As you can see, the ability to search for required terms, optional terms, negated terms, and grouped terms provides a very powerful mechanism for looking up single keywords. As we'll see in the following section, Solr also provides the ability to query for multi-term phrases.

3.1.6 Phrase Queries and Term Positions

We saw earlier that, in addition to querying for terms in our Lucene Index, that it is also possible to query Solr for phrases. Recalling that the index contains only individual terms, however, you may be wondering exactly 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, just 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 utilized. 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 actually 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 upon only 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 particular example, the term new happens to be in position 3, and the term home happens to be in position 4 in both of the matched documents. This makes sense, as the original book titles for these books were “Buying a New Home” and “Becoming a new Home Owner.” We have thus seen the power of term positions – they allow us to reconstruct the original positions of our 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, though. 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 or edit distance searching), or may even 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 very 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 generally the use of wildcards. Suppose you wanted to find any documents which started with the letters “offic". One way to do this would be to create a query which 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. Since 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, etc.
```

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 off**e**r, off**icer**, off**iciato**r, etc.
```

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

```
Query:    off?r  → Matches off**e**r, but NOT off**ice**r
```

Leading Wildcards

While the wild card functionality in Solr is fairly robust, it is only possible, by default, to use wildcards inside or at the end of a term. If you needed to match all terms ending in "ing" (like caring, liking, and smiling), for example, you would receive an exception if you tried running this search:

```
Query:    *ing
```

The reason for this is that Solr searches through the inverted index for terms which begin with the characters provided before the wildcard, and without some initial characters to begin that lookup, it is too expensive to walk the entire index to find matching terms.

If you really need to be able to search using these leading wildcards, a solution to this problem does exist, but it will require you to perform some additional configuration. The solution is achieved by adding the `ReversedWildcardFilterFactory` to your field type's analysis chain (configuring text processing will be discussed in chapter 5).

The `ReversedWildcardFilterFactory` works by double storing 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, thus getting around Solr's inability to perform leading wildcard searches by turning them into standard wildcard searches on the reversed content.

Do note, however, that turning this feature on requires dual-storing all terms in the Solr index, increasing the size of the index and thus slowing overall searches down somewhat. Turning this capability on is thus not recommended unless it is 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 5.

RANGE SEARCHING

Solr also provides the ability to search for values which 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 2nd, 2012 and August 2nd, 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 the above range queries surrounds the range with square brackets, which is the "inclusive" range syntax. Solr also supports exclusive range searching through use of curly braces:

```
Query: yearsOld:{18 TO 21}        //Matches 19 and 20 but NOT 18 or 21
```

Thought 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 to find documents matching dynamically defined groups of values which lie within a particular range within the Solr index. It is important to note that the ordering of terms of range queries is exactly that – the order in which they are found in the Solr index, which is an alphanumeric sorted order. Thus if you were to create a text field containing integers, those integers would actually 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 is important to understand that the sort order within the Solr index is dependent upon how the data within the field is processed when it is written to the Solr index. We'll dive much deeper into this kind of content analysis in chapter 5.

FUZZY/EDIT DISTANCE SEARCHING

For many search application, it is important not only to match a customer's text exactly, but also to allow some 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 a 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, etc.

The above query matches both the original term (administrator) and any other terms within 2 "edit distances" of the original term. An edit distance here is defined as an insertion, a deletion, a substitution, or a transposition of characters. The term adminstrator (missing the "i" in the 6th position) is one edit distance away from administrator because it has one character deletion. Likewise the term sadministrator is one edit distance away because it has one insertion (the "s" that was added to the beginning), and the term administratro is one edit distance away because it has transposed the last two characters ("or" became "ro").

It is also possible to modify the strictness of edit distance searches to allow matching of terms with an edit distance of greater than one:

```
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 2 will become increasingly slower and will be more likely to match unexpected terms. Term searches with edit distances of 1 or 2 are performed using a very efficient Levenshtein Automaton, but will fall back to slower edit distance implementation for edit distances above 2.

PROXIMITY SEARCHING

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

Let's say that 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 …`

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:

---

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

Query: `chief AND officer`

This should match all of the possible use cases, but it will also match any document which 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 above query.

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

Query: `"chief officer"~1`
chief and officer must be a maximum of one edit distance away.
Examples: "chief executive officer", "officer chief"

Query: `"chief officer"~2`
Examples: "chief business development officer", "officer, police chief"

Query: `"chief officer"~N`
Finds chief within N edit distances of officer.

The edit distances above can be seen as nothing more than sloppy 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 exact same results, because an edit distance of zero is the very 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.

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 intersections of 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 edit distance calculations. For fuzzy searching within single terms, we examined the use of wildcards and edit distance searching to find misspellings or very similar words. While Solr's query capabilities will be expanded upon in chapter 6, 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, which we'll discuss in the next section.

3.2 Relevancy

Finding matching documents is the first critical step in creating a great search experience, but it is only the first step. No customer is willing to wade through page after page of search results to find the document he or she is seeking. In my 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 very good job out of the box at ensuring 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 top 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 also into the specific calculations used to calculate 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 eluding 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 a Similarity class which can be defined on a per-field basis in Solr's Schema.xml (discussed later in chapter 5). The Similarity class 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' default Similarity implementation and the theory behind why it works so well.

By default, Solr uses Lucene's (appropriately named) DefaultSimilarity class. This class utilizes a two-pass model to calculate similarity. First, it makes use of a Boolean model (described in the last section) to filter out any documents that do not match the customer's query. Then, it utilizes a Vector Space model for scoring, 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

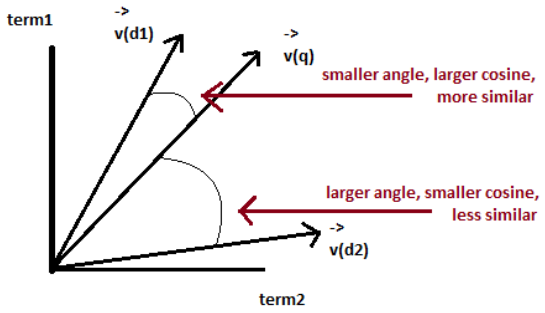


Figure 3.6 Cosine Similarity of Term Vectors. The query term vector, v(q), is closer to the document 1 term vector v(d1) than the document 2 term vector (d2), as measured by the cosine of the angle between each document term vector and the query vector. The smaller the angle between the query term vector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

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 zero representing no similarity. More intuitively, the closer the two vectors appear together, as in the image above, the more similar they are. Thus, 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 actually coming up with reasonable vectors which represent the important features of the query and of each document for comparison. Let's take a 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.

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

Where:

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

$\textbf{tf}(t \text{ in } d) = \left(\sum_{t \text{ in } d} 1 \right)^{½}$

$\textbf{idf}(t) = 1 + \log (\text{numDocs} / (\text{docFreq} + 1))$

$\textbf{coord}(q,d) = \text{numTermOccurrencesInQuery} / \text{numTermsInQuery}$

$\textbf{queryNorm}(q) = 1 / (\text{SumOfSquaredWeights}^{½})$

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

$\textbf{norm}(t,d) = d.\text{getBoost}() \cdot \text{lengthNorm}(\text{field}) \cdot f.\text{getBoost}()$

Figure 3.7 DefaultSimilarity Scoring Algorithm. Each component in this formula will be explained in detail in the following sections.

Wow – that equation can be quite overwhelming, especially at first glance. Fortunately, it is much more intuitive when each of the pieces are broken down. The math is presented above for reference, but you will likely never need to really dig into the full equation unless you decide to overwrite the similarity for your search application.

At a high level, the important concepts are demonstrated by the high-level formula – namely, Term Frequency (tf), Inverse Document Frequency (idf), Term Boosts (t.getBoost), the Field Normalization (norm), the Coordination Factor (coord), and the Query Normalization (queryNorm). Let's dive into the purpose of each of these.

3.2.2 Term Frequency (tf)

Term Frequency is a measure of how often a particular term appears in a matching document, and is 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 which only mention the president once, or articles which discuss the president consistently throughout the article? What if an article just 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?

Take the example from table 3.9. Clearly the second article above 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. Smolla is the **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 (Term Frequency) 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 is unlikely the case that 10 appearances of a term makes the document 10 times more relevant, however, so TF is actually 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 (idf)

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

Inverse Document Frequency (IDF) is a measure of how "rare" a search term is, and it is calculated by finding the document frequency (how many total documents the search term appears within), and calculating it's inverse (see the full formula in section 3.2.1 for the actual calculation).

Because the Inverse Document Frequency appears for the term in both the query and the document, it is squared in the relevancy formula.

Figure 3.8 shows an visual example of the "rare-ness" of each of the words in the title "*The Cat in the Hat*," with a higher IDF being represented as a larger term.

Figure 3.8 Visual depiction of the relative significance of terms as measured by Inverse Document Frequency. The terms which are more rare 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," we wouldn't expect documents to rank higher which best match the words "an", "team", or "experienced". Instead, we would expect the important terms to resemble the largest terms in figure 3.9.

Figure 3.9 Another demonstration of relative score of terms derived from Inverse Document Frequency. Once again, a higher Inverse Document Frequency indicates a more rare and more relevant term, which is depicted here using larger text.

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

Term Frequency and Inverse Document Frequency, when multiplied together in the relevancy calculation, provide a nice counter-balance to each other. The term frequency elevates terms which appear multiple times within a document, while the inverse document frequency penalizes those terms which appear commonly across many documents. Thus common words such as "the", "an", and "of" in the English language ultimately yield very low scores, even though they may appear many times in any given document.

3.2.4 Boosting (t.getBoost)

It is not necessary to leave all aspects of your relevancy calculations of up to Solr. If you have domain knowledge about your content – 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 is the most flexible and easiest to understand form of boosting, utilizing the following syntax:

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

The above 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. It is important to note that, unless otherwise specified, all terms receive a default boost of 1.0 (which means multiply the calculated score by 1, or leave it as 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`

Do note, however, that a boost of less than 1 is still a positive boost – it does not penalize the document in absolute terms, it simply boosts the term less than 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 6.

In addition to query time boosting, it is also 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 (NORM)

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

$$\mathbf{norm}(t,d) = d.\text{getBoost}() \cdot \text{lengthNorm}(\text{field}) \cdot f.\text{getBoost}()$$

Figure 3.10 Field Norms Calculation. Field Norms factor in the matching documents boost, the matching field's boost, and a length normalization factor which 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 together into this single "field norms" variable.

The d.getBoost() component represents the boost applied to the document when it is sent to Solr, and the f.getBoost() component represents the boost applied to the field for which the norm is being calculated. It is worth mentioning that Solr actually allows the same field to be added to a document multiple times (performing some magic under the covers to actually 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, the f.getBoost() actually 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 3 times, for example, once with a boost of 3, once with a boost of 1, and once with a boost of .5, then the f.getBoost() for each of the three fields (or one underlying field) would be:

Boost: `(3)· (1) · (.5) = 1.5`

In addition to the index time boosts, a parameter called the Length Norm is also 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.

The purpose of the Length Norm is to adjust for documents of varying lengths, such that longer documents do not maintain an unfair advantage simply by having a larger likelihood of containing any particular term a given number of times.

For example, let's say that you perform a search for the keyword "`Beijing`". Would you prefer for a news article to come up mentions `Beijing` 5 times, or would you rather have an obscure 300 page book come back which also happens to mention Beijing only 5 times. Common sense would indicate that 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 which is 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 is usually only big differences which matter given the variance in all other relevancy criteria.

QUERY NORMS (QUERYNORM)

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 utilizes 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 (COORD)

One final normalization factor taken into account in the default Solr relevancy calculation is the Coord factor. The Coord factor's role is to measure how much of the query each document matches. For example, 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" many times 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, then it is 1/4.

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

We have now discussed all of the major components of the default relevancy algorithm in Solr. We discussed term frequency, inverse document frequency, the two most key components of the relevancy score calculation. We then went through boosting and normalization factors, which refine the scores calculated by term frequency and inverse document frequency 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 very simple to explain, but are also very 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 which match a query) is a measurement answering attempting to answer the question "Were the documents which 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 earlier example from section 3.1 about searching for a book on the topic of buying a new home. We've determined that by our internal company measurements that the books in table 3.10 would be considered good matches for such a query.

Table 3.10 Relevant List of 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 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 Irrelevant List of 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 which 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 results came back, the precision would only be .5, since half of the results which 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 which came back were correct. As you can see, Precision is a measure of how "good" each of the results of a query are, but it pays no attention to how thorough they are – a query which 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 essentially answering the question of "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 using the Recall calculation, our example showing relevant books and irrelevant books from the last section has been recreated in table 3.12 for reference purposes.

Table 3.12 List of relevant and irrelevant books

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

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

If all six documents were returned for a search query, the Recall would actually be 1 since all correct matches were found and there were no missed matches (whereas we saw the Precision earlier would be .5).

Likewise, if only document 1 were returned, the Recall would only be 1/3, since 2 of the documents that should have been returned/recalled were missing.

This underlies the critical difference between Precision and Recall – Precision is high when the results returned are "correct", whereas Recall is high when the correct results are "present." Recall does not care that *all* of the results are correct, whereas 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 case from the previous example where the query only returns documents 1, 2, and 3, the Precision and Recall are actually 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 just about 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 very 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 actually 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 actually went to the last page of the search results.

This is only one way to approach the problem, however. Since many websites, for example, want to appear to have as much content as possible, and since those sites know that visitors will never actually go beyond the first few pages, they can actually show very precise results on the first few pages which yielding a very high Recall value across the entire result set since they are increasing the chances of pulling back more content by being very lenient in which keywords are able to match the initial query.

The decision on how to best balance Precision and Recall is ultimately dependent upon your use case. In scenarios like legal discovery, there is a very heavy emphasis placed on Recall, as there are legal ramifications if any documents are missed. For other use cases, the requirement may simply be to find a few really 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 generally no one right answer. Regardless, understanding the concepts of Precision and Recall and why changes you make swing you more towards one of these two conceptual goals (and likely away from the other) is critical to effectively improving the quality of your search results. We have an entire chapter dedicated to relevancy tuning, chapter 16, 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 it's 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. Chapter 12 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 completely 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 a "city", "state", and "postalCode" field, even though in most cases the "city" and "state" field will be exactly the same across all documents for each unique "postalCode" value. This is in contrast to a normalized document where 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 big "X" over it to make it obvious that this is not the kind of data-modeling strategy you will use with Solr.

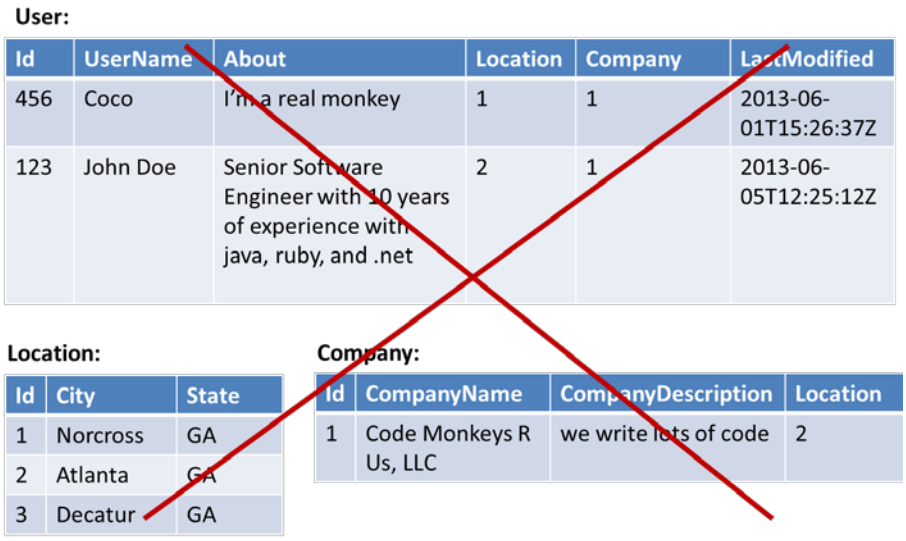


Figure 3.11 Solr documents do not follow the traditional normalized model of a relational database. This figure demonstrates how NOT to think of Solr Documents. Instead of thinking in terms of multiple entities with relationship to each other, a Solr Document is modeled as a flat, denormalized data structure,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

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.

Listing 3.2 Two Denormalized User Documents

```
<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>

 <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>
```

Notice that all of the company information is repeated in both the first and second user's documents from listing 3.2, 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 which will join data from multiple tables when resolving a query. While some basic "join" functionality does now exist Solr (which will be discussed in chapter 14), it is only recommended for cases where it is impractical to actually 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 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 (since 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 Federated or Distributed Searching, and will be covered in the following section.

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… it would also be a much more boring world. In reality, sometimes your search servers 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 is 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 will actually be sent to both servers, and the results will be returned 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 shards 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 example above:

- 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 which are found in separate Solr Cores.
- The Solr Core being searched on (box1, core1) is also included in the list of shards – it will not automatically search itself unless explicitly requested as above.
- This distributed search is searching across multiple servers
- There is 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 take-away here is the nature of scaling Solr. It should scale theoretically linearly. The reason for this is that a federated search across multiple Solr Cores is run in parallel on each of those index partitions. Thus, if you divide one Solr index into two Solr indexes with the same 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 at some point). The conceptual formula for determining total query speed after adding an additional index partition (assuming the same total number of documents) is thus as follows:

```
(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. Since 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 last section we introduced the concept of distributed searching to enable scaling to handle large document sets. It is also possible to add multiple essentially 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 essentially defined as multiple servers, working in concert, to perform the same function.

Take the following example, which should look similar to the example from section 3.3.6:

```
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 box 2. When running this distributed search, what happens to queries hitting box 1 if box 2 goes down? Listing 3.3 shows Solr's response under this scenario, which includes the error message from the failed connection to box 2.

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 occured 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 listing 3.3. It is thus important to think in terms of "clusters" of servers instead of single servers when building out Solr solutions which must scale beyond a single box, as those servers are essentially combining to serve as a single computing resource. Solr provides some built-in cluster management capabilities, through the use of Apache Zookeeper, which will be covered 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, several of which will be discusses in the next section.

3.4.4 The limits of solr

Solr is an incredibly powerful document-based NoSQL datastore which 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 essentially linearly across multiple servers to handle additional content or query volumes. What then, are the use cases where Solr is not a good solution – what are the limits of Solr?

One limit, as we have already seen, is that Solr is NOT relational in any way across documents. It is not well suited for joining significant amounts of data across different fields on different documents, and it cannot 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 have also already discussed the denormalized nature of Solr documents – data which is redundant must be repeated across each document for which that data applies. This can be particularly problematic when the data in one field which is shared across many documents changes.

For example, let's say that you were creating a search engine of social networking user profiles, and one of your user's, 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 simple 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 (very 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 very wasteful. Even then, Solr is internally still updating the entire document based upon re-indexing 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 has changes, every single document in the Solr index must be reprocessed in its entirety before the data will be populated for the new field. Many other NoSQL systems suffer from this same problem, but it is worth noting that data updates across the corpus require a non-trivial amount of document management to ensure the updates make it to Solr and in a timely fashion.

Solr is also optimized for a very 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 actual results for display. Solr is not optimized, however, at processing very long (1000's of terms) queries or returning back very 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 very well across servers, it does not yet elastically scale by itself in a fully automatic way. Recent work with Solr Cloud (covered in chaper 13), utilizing Apache Zookeeper for cluster management, is a great first step in this direction, but there are still many features to be worked out, such as automatic content resharding and pluggable sharding strategies which are currently being discussed but are not yet fully implemented.

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. 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 with 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 federated distributed searching to ensure query

execution can be parallelized to maintain or decrease search speed 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 we scale our search architectures, and we ended by discussing 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 the most important search concepts 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.

4

Configuring Solr

This chapter covers

- Solr's main configuration file **solrconfig.xml**
- Query request handling
- Extending query processing with search components
- Managing searchers
- Cache management
- Core admin API

Up to this point, you've taken much of what has been presented on faith without learning how Solr actually works. We'll change that in this chapter and the next by looking under-the-hood to learn how Solr is configured and how configuration settings impact Solr's behavior.

As we learned in chapter 2, Solr works out-of-the-box without making any configuration changes. But, at some point, you're going to need to make some configuration changes to optimize Solr for your specific search application requirements. Broadly speaking, most of the configuration you'll do with Solr focuses around three main XML files:

1. **solr.xml** – Defines one or more cores per Solr server
2. **solrconfig.xml** – Defines the main settings for a specific Solr core
3. **schema.xml** – Defines the structure of your index including fields and field types

In this chapter, we'll focus on **solrconfig.xml** and **solr.xml**. In chapter 5, we'll learn all about **schema.xml**, which drives how your index is structured.

As most of Solr's configuration is specified in XML documents, this chapter contains numerous code listings showing XML snippets from **solrconfig.xml** and **solr.xml**. However, our main focus is on the concepts behind the configuration settings rather than the specific XML syntax, which is mostly self-explanatory.

To begin let's see what happens from a configuration perspective when you start the Solr server. Recall from chapter 2 that Solr runs as a Java Web application in Jetty. The Solr Web application uses a global Java system property (`solr.solr.home`) to identify the root directory from which to look for configuration files. Figure 4.1 depicts how **solr.xml** and **solrconfig.xml** are used during the Solr initialization process.

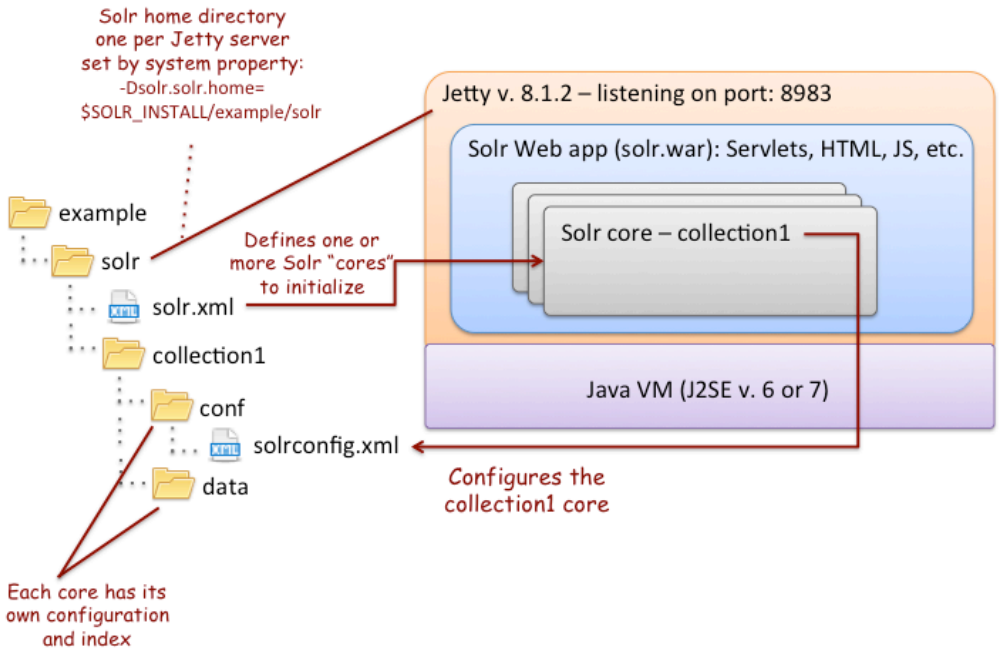


Figure 4.1 Depiction of how solr.xml and solrconfig.xml are used to configure Solr during initialization

During initialization, Solr locates **solr.xml** in the top-level Solr home directory; in the example server this is `$SOLR_INSTALL/example/solr/solr.xml`. The **solr.xml** file identifies one or more cores to initialize. Listing 4.1 shows the **solr.xml** for the example server.

Listing 4.1: Default solr.xml for example server defining the collection1 core

```
<solr persistent="true"> #A
  <logging enabled="true">
    <watcher size="100" threshold="INFO" />
  </logging>

  <cores adminPath="/admin/cores"  #B
         defaultCoreName="collection1"
         host="${host:}" hostPort="${jetty.port:}"
```

```
        hostContext="${hostContext:}"
        zkClientTimeout="${zkClientTimeout:15000}">
    <core name="collection1" instanceDir="collection1" /> #C
  </cores>
</solr>
```

#A persistent attribute controls whether changes made from the core admin API are persisted to this file
#B define one or more cores under the <cores> element
#C the collection1 core configuration and index files are in the collection1 directory under solr home

The initial configuration only has a single core named "collection1", but in general there can be many cores defined in **solr.xml**. For each core, Solr locates the **solrconfig.xml** file, under `$SOLR_HOME/$instanceDir/conf/solrconfig.xml`, where **$instanceDir** is the directory for a specific core as specified in **solr.xml**. Solr uses the **solrconfig.xml** file to initialize the core.

Now that we've seen how Solr identifies configuration files during startup, let's turn our attention to understanding the main sections of the **solrconfig.xml**, as that will give you an idea of what's to come in the rest of this chapter.

4.1 Overview of solrconfig.xml

To illustrate the concepts in **solrconfig.xml**, we'll build upon the work done in chapter 2 by using the pre-configured example server and the Solritas example search UI. To begin, we recommend that you start up the example server we used in chapter 2 using:

```
cd $SOLR_INSTALL/example
java -jar start.jar
```

Once you've started the server, go to the Solr admin console at http://locahost:8983/solr, click on the **collection1** link on the left, and then on the **Config** link. This will display the active **solrconfig.xml** for the **collection1** core running on your computer. Listing 4.2 shows a condensed version of the **solconfig.xml** to give you an idea of the main elements.

Listing 4.2 Condensed version of solrconfig.xml

```
<config>
  <luceneMatchVersion>LUCENE_40</luceneMatchVersion>      #A
  <lib dir="../../../contrib/extraction/lib" regex=".*\.jar" /> #B
  <dataDir>${solr.data.dir:}</dataDir>                    #C
  <directoryFactory name="DirectoryFactory" class="..."/> #C
  <indexConfig> ... </indexConfig>                        #C
  <jmx />  #D
  <updateHandler class="solr.DirectUpdateHandler2"> #E
    <updateLog> ... </updateLog>                    #E
    <autoCommit> ... </autoCommit>                  #E
  </updateHandler>                                  #E
  <query>
    <filterCache ... />              #F
    <queryResultCache ... />         #F
    <documentCache ... />            #F
    <listener event="newSearcher" class="solr.QuerySenderListener">   #G
      <arr name="queries"> ... </arr>                                 #G
```

```
    </listener>                                                          #G
    <listener event="firstSearcher" class="solr.QuerySenderListener"> #G
      <arr name="queries"> ... </arr>                                    #G
    </listener>                                                          #G
  </query>
  <requestDispatcher handleSelect="false" >                              #H
    <requestParsers ... />
    <httpCaching never304="true" />
  </requestDispatcher>
  <requestHandler name="/select" class="solr.SearchHandler">           #I
    <lst name="defaults"> ... </lst>                                     #I
    <lst name="appends"> ... </lst>                                      #I
    <lst name="invariants"> ... </lst>                                   #I
    <arr name="components"> ... </arr>                                   #I
    <arr name="last-components"> ... </arr>                              #I
  </requestHandler>                                                      #I
  <searchComponent name="spellcheck"                                     #J
     class="solr.SpellCheckComponent"> ... </searchComponent>
  <updateRequestProcessorChain name="langid"> ...                        #K
      </updateRequestProcessorChain>
  <queryResponseWriter name="json"                  #L
      class="solr.JSONResponseWriter"> ... </queryResponseWriter>
  <valueSourceParser name="myfunc" ... />   #M
  <transformer name="db"    #N
      class="com.mycompany.LoadFromDatabaseTransformer">
      ... </transformer>
</config>
```

#A Activates version-dependent features in Lucene, see 4.2.1
#B Lib directives indicate where Solr can find JAR files for extensions, see 4.2.1
#C Index management settings covered in chapter 5
#D Enable JMX instrumentation of Solr MBeans, see 4.2.1
#E Update handler for indexing documents, see chapter 5
#F Cache management settings, see section 4.4
#G Register event handlers for searcher events, e.g. queries to execute to warm new searchers, section 4.3
#H Unified request dispatcher, section 4.2
#I Request handler to process queries using a chain of search components, 4.2.4
#J Example search component for doing spell correction on queries
#K Extend indexing behavior using update request processors, such as language detection
#L Format the response as JSON
#M Declare a custom function for boosting, ranking or sorting documents
#N Transforms result documents

As you can see, **solrconfig.xml** has a number of complex sections. The good news is that you don't have to worry about these until you encounter a specific need. On the other hand, we do think it is a good idea to make a mental note of what is in **solrconfig.xml** as it shows how flexible Solr is and what types of behavior you can control and extend.

When organizing this chapter, we chose to present the configuration settings in an order that builds on previous sections rather than following the order of elements in the XML document. For example, we present Solr's request handling framework before we discuss caching even though cache-related settings come before request handler settings in **solrconfig.xml**. We took this approach because you should understand how requests are handled before you worry about optimizing a specific type of request with caching. However,

this means that you will have to jump around the XML document as you work through this chapter.

Index settings deferred to chapter 5

The **solrconfig.xml** contains index management settings. However, we save a discussion of index-related settings for the next chapter when you can learn about them after gaining a basic understanding of the indexing process. Specifically, you can ignore the following elements until chapter 5:

<dataDir> ... </dataDir>

<directoryFactory name="DirectoryFactory" class="..."/>

<indexConfig> ... </indexConfig>

<updateHandler class="solr.DirectUpdateHandler2"> ...

<updateRequestProcessorChain name="langid"> ...

4.1.1 Common XML data structure and type elements

As you work through **solrconfig.xml**, you will encounter common XML elements that Solr uses to represent various data structures and types. Table 4.1 provides a brief description and example of the types of elements Solr uses throughout the **solrconfig.xml** document. You will also encounter these elements in XML search results, so please spend a minute getting familiar with this Solr-specific syntax.

Table 4.1 Solr's XML elements for data structures and typed values

Element	Description	Example

<arr>	Named, ordered array of objects	`<arr name="last-components">`
		`  <str>spellcheck</str>`
		`</arr>`

<lst>	Named, ordered list of name/value pairs	`<lst name="defaults">`
		`  <str name="omitHeader">true</str>`
		`  <str name="wt">json</str>`
		`</lst>`

<bool>	Boolean value; true or false	`<bool>true</bool>`

<str>	String value	<str>spellcheck</str> or <str name="wt">json</str>
<int>	Integer value	<int>512</int>
<long>	Long value	<long>1359936000000</long>
<float>	Float value	<float>3.14</float>
<double>	Double value	<double>3.14159265359</double>

The primary difference between `<arr>` and `<lst>` is that every child element of an `<lst>` has a name attribute and child elements of `<arr>` are un-named.

4.1.2 Applying configuration changes

Learning about configuration is not the most exciting of tasks so to help keep you interested, we recommend that you experiment with configuration changes as you work through this chapter. However, your changes won't be applied until you reload the Solr core. In other words, Solr doesn't "watch" for changes to **solrconfig.xml** and apply them automatically; you have to take an explicit action to apply configuration changes. For now, the easiest way to apply configuration changes is to use the **Reload** button from the **Core Admin** page of the administration console as shown in figure 4.2.

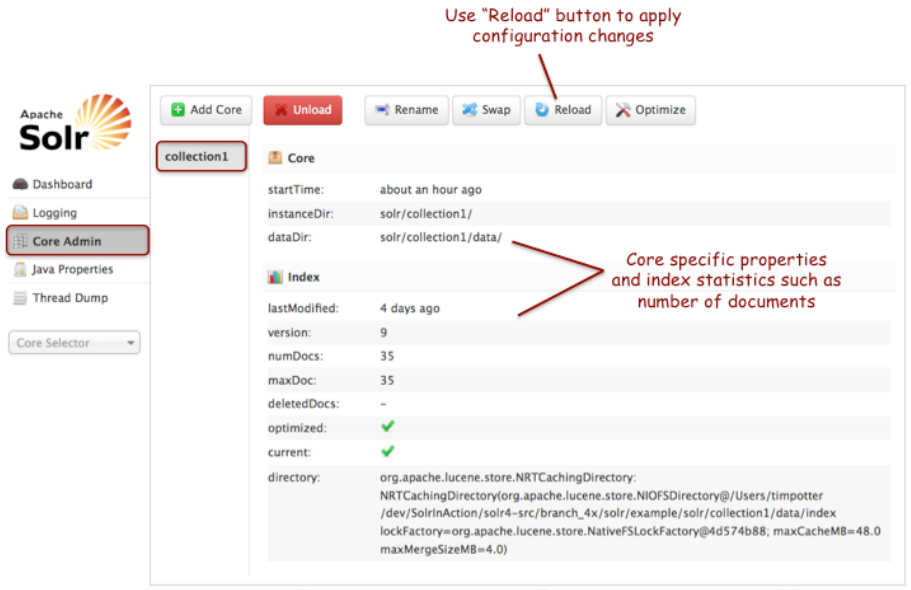


Figure 4.2 Reload a core to apply configuration changes from the **Core Admin** page

If you're running Solr locally, then go ahead and click on the **Reload** button for the **collection1** core just to verify the functionality works. Also, we'll see another way to reload cores programmatically using the Core Admin API at the end of this chapter.

4.1.3 Miscellaneous settings

Now that we've covered some of the configuration background, let's start our tour of **solrconfig.xml** by looking at some miscellaneous settings for the Solr server. Listing 4.3 shows the configuration settings we'll discuss in this section.

Listing 4.3 Global settings near the top of solrconfig.xml

```
<config>
  <luceneMatchVersion>LUCENE_40</luceneMatchVersion>          #A
  <lib dir="../../../contrib/langid/lib/" regex=".*\.jar" />  #B
  <lib dir="../../../dist/" regex="solr-langid-\d.*\.jar" />  #B
  ...
  <jmx />     #C
  ...
</config>
```

#A Lucene version
#B Loading dependency JAR files
#C Enable JMX

LUCENE VERSION

Lucene and Solr take backwards compatibility very serious. The `<luceneMatchVersion>` element controls the version of Lucene your index is based on. If you're just starting out, then use the version that is specified in the example server, such as:

```
<luceneMatchVersion>LUCENE_40</luceneMatchVersion>
```

Now imagine that after running Solr for several months and indexing millions of documents, you decide that you need to upgrade to a later version of Solr. When you start the updated Solr server, it uses the `<luceneMatchVersion>` to understand which version your index is based on and whether to disable any Lucene features that depend on a later version than what is specified.

You will be able to run the upgraded version of Solr against your older index, but at some point you may need to raise the `<luceneMatchVersion>` to take advantage of new features and bug fixes in Lucene. In this case, you can either re-index all your documents or use Lucene's built-in index upgrade tool[1]. As that is a problem for the future, we'll refer you to the JavaDoc for instructions on how to run the upgrade tool.

LOADING DEPENDENCY JAR FILES

The `<lib>` element allows you to add JAR files to Solr's runtime classpath so that it can locate plug-in classes. Let's take a look a few of the `<lib>` elements in the example **solrconfig.xml** to see how `<lib>` elements work.

```
<lib dir="../../../contrib/langid/lib/" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-langid-\d.*\.jar" />
```

[1] Refer to the Lucene JavaDoc for the **org.apache.lucene.index.IndexUpgrader** class for usage instructions.

Each <lib> element identifies a directory and a regular expression to match files in the directory. Notice that the **dir** attribute uses relative paths, which are evaluated from the core directory root, commonly referred to as the core **instanceDir**. For the collection1 core in the example server, **instanceDir** is $SOLR_INSTALL/example/solr/collection1; emember that $SOLR_INSTALL is our variable name for the directory where you extracted the Solr distribution archive. Consequently, the two example <lib> elements shown above result in the following JAR files being added to Solr's classpath:

- jsonic-1.2.7.jar *(from contrib/langid/lib)*
- langdetect-1.1-20120112.jar *(from contrib/langid/lib)*
- apache-solr-langid-4.0.0.jar *(from dist)*

Note that the version number for the apache-solr-langid JAR file may be different depending on the exact version of Solr 4 you are using. Alternatively, you can use the path attribute to identify a single JAR file, such as:

```
<lib path="../../../dist/solr-langid-4.0.0.jar" />
```

Alternatively, you can also put JAR files for plug-ins in the $SOLR_HOME/lib directory, such as $SOLR_INSTALL/example/solr/lib.

ENABLE JMX

The <jmx> element activates Solr's MBeans to allow system administrators to monitor and manage core Solr components from popular system monitoring tools, such as Nagios. In a nutshell, an MBean is a Java object that exposes configuration properties and statistics using the Java Management Extensions (JMX) API. MBeans can be auto-discovered and introspected by JMX-compliant tools. This allows Solr to be integrated into your existing system administration infrastructure.

However, you don't need an external JMX-enabled monitoring tool to see Solr's MBeans in action. The Solr administration console provides access to all of Solr's MBeans. Figure 4.3 provides a screen shot of the MBean for the collection1 core.

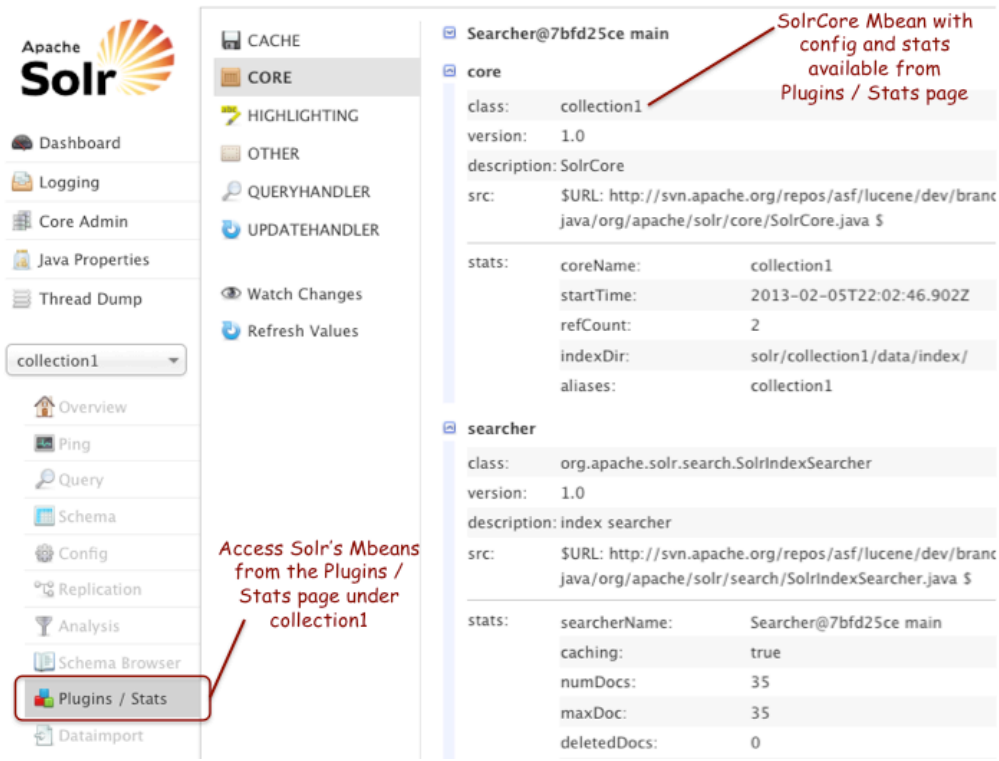


Figure 4.3 Inspecting the SolrCore MBean from the **Plugins / Stats** page for the **collection1** core

We will see a few more examples of inspecting Solr's MBeans from the administration console throughout this chapter. For now, let's move on to learning how Solr processes requests.

4.2 Query request handling

Solr's main purpose is to search, so it follows that handling search requests is one of the most important processes in Solr. In this section, you'll learn how Solr processes search requests and how to customize request handling to better fit your specific search requirements.

4.2.1 Request handling overview

All requests to Solr happen over HTTP. For example, if you want to query Solr, then you send an HTTP GET request. Alternatively, if you want to index a document in Solr, you use an HTTP POST request. Listing 4.4 shows a HTTP GET request to query the example Solr server

(note that the actual request does not include line breaks between the parameters, which we've included here to make it easier to see the separate parameters).

Listing 4.4 HTTP GET request to query the example Solr server

```
http://localhost:8983/solr/collection1/select?    #A
q=iPod&                                           #B
fq=manu%3ABelkin&                                 #C
sort=price+asc&                                   #D
fl=name%2Cprice%2Cfeatures%2Cscore&                #E
df=text&                                          #F
wt=xml&                                           #G
start=0&rows=10                                   #H
```

#A Invokes the "select" request handler for the "collection1" core
#B Main query component looking for documents containing "iPod"
#C Filter documents that have manu field equal to "Belkin"
#D Sort results by price in ascending order (smallest to largest)
#E Return the name, price, features, and score fields in results
#F Default search field is "text"
#G Return results in XML format
#H Start at page 0 and return up to 10 results

You can either input this URL into a Web browser, use a command-line tool like cURL, or our recommended approach is to use the example driver application that comes with the book. To execute this request using the example driver, you simply do:

```
java -jar target/sia-examples.jar http –listing #.#
```

The (`#.#`) parameter should be the number of the listing to execute, such as 4.4. Using the http utility from the book example code is recommended because you don't have to do any copy-and-pasting or extra typing to run a code listing and it works on all Java platforms. The output when running this utility for listing 4.4 is:

```
java -jar target/sia-examples.jar http -listing 4.4

INFO [main] (ExampleDriver.java:92) - Found example class sia.Http for arg
http
 INFO [main] (ExampleDriver.java:125) - Running example Http with args: -
listing 4.4
Feb 13, 2013 6:21:32 PM org.apache.solr.client.solrj.impl.HttpClientUtil
createClient
INFO: Creating new http client,
config:maxConnections=128&maxConnectionsPerHost=32&followRedirects=false


Sending HTTP GET request (listing 4.4):

     http://localhost:8983/solr/collection1/select?q=iPod&
            fq=manu%3ABelkin&
            sort=price+asc&
            fl=name%2Cprice%2Cfeatures%2Cscore&
            df=text&
            wt=xml&
            start=0&
            rows=10
```

```
Solr returned: HTTP/1.1 200 OK

<?xml version="1.0" encoding="UTF-8"?>
<response>
...
</response>
```

The **http** utility provides a couple of other options to allow you to override the address of your Solr server or to change the response type to something other than XML, such as JSON. To see a full list of options, simply do: `java -jar target/sia-examples.jar http -h`

Figure 4.4 shows the sequence of events and main components involved in handling this Solr request.

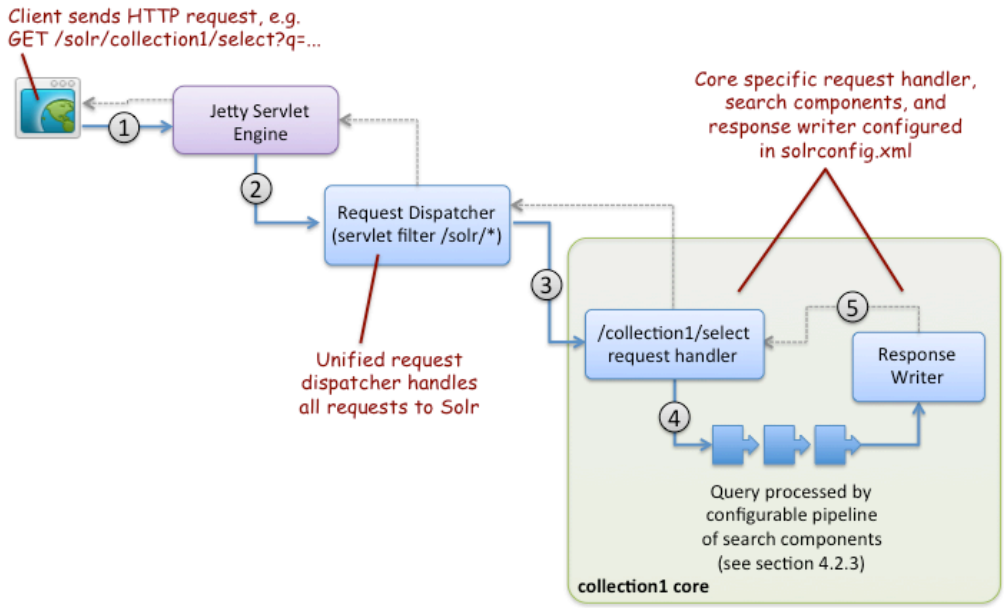


Figure 4.4 Sequence of events to process a request to the **/select** request handler.

Starting at the top-left of figure 4.4:

1. A client application sends an HTTP GET request to http://localhost:8983/solr/collection1/select?q=... Query parameters are passed along in the query string of the GET request.
2. Jetty accepts the request and routes it to Solr's unified request dispatcher using the `/solr` context in the request path. In technical terms, the unified request dispatcher is a Java servlet filter mapped to `/*` for the solr Web application, see `org.apache.solr.servlet.SolrDispatchFilter`.

3. Solr's request dispatcher uses the "collection1" part of the request path to determine the core name. Next, the dispatcher locates the **/select** request handler registered in solrconfig.xml for the **collection1** core.
4. The **/select** request handler processes the request using a pipeline of search components (covered in section 4.2.4 below).
5. After the request is processed, results are formatted by a response writer component and returned to the client application, by default the **/select** handler returns results as XML. Response writers are covered in section 4.5.

The main purpose of the request dispatcher is to locate the correct core to handle the request, such as **collection1**, and then route the request to the appropriate request handler registered in the core, in this case **/select**. In practice, the default configuration for the request dispatcher is sufficient for most applications. On the other hand, it is common to define a custom search request handler or to customize one of the existing handlers, such as **/select**. Let's dig into how the **/select** handler works to gain a better understanding of how to customize a request handler.

4.2.2 Search handler

Listing 4.5 shows the definition of the **/select** request handler from **solrconfig.xml**.

Listing 4.5 Definition of /select request handler from solrconfig.xml

```
<requestHandler name="/select"              #A
                class="solr.SearchHandler"> #B
  <lst name="defaults">                     #C
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>               #D
    <str name="df">text</str>
  </lst>
</requestHandler>
```

#A A specific type of request handler designed to process queries
#B Java class that implements the request handler
#C List of default parameters (name/value pairs)
#D Sets the default page size to 10

Behind the scenes, all request handlers are implemented by a Java class, in this case `solr.SearchHandler`. At runtime, `solr.SearchHandler` resolves to the built-in Solr class: `org.apache.solr.handler.component.SearchHandler`. In general, anytime you see "solr." as a prefix of a class in **solrconfig.xml**, then you know this translates to the fully qualified Java package: `org.apache.solr.handler.component.` This shorthand notation helps reduce clutter in Solr's configuration documents. In Solr, there are two main types of request handlers:

1. **search handler** – query processing
2. **update handler** - indexing

We'll learn more about update handlers in the next chapter when we cover indexing. For now, let's concentrate on how search request handlers process queries, as depicted in figure 4.5.

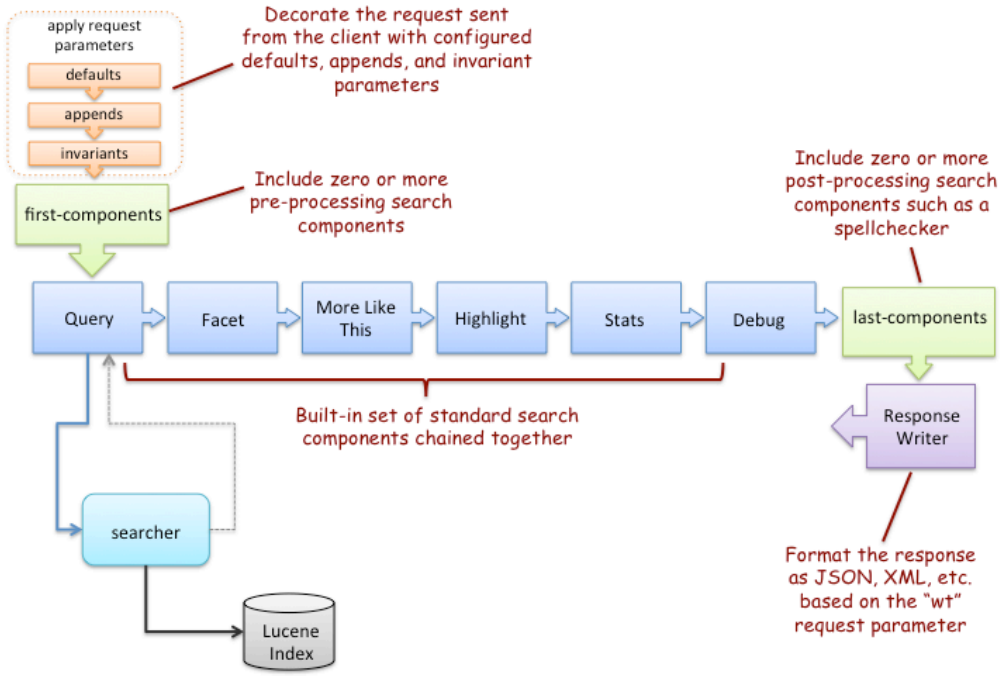


Figure 4.5 Search request handler made up of parameter decoration (defaults, appends, invariants), first-components, components, and last-components

The search handler structure depicted in figure 4.5 is designed to make it easy for you to adapt Solr's query processing pipeline for your application. For example, you can define your own request handler or, more commonly, add a custom search component to an existing request handler, such as **/select**. In general, a search handler is comprised of the following phases, where each phase can be customized in **solrconfig.xml**:

1. request parameter decoration using:
    a. **defaults**: set default parameters on the request if they are not explicitly provided by the client
    b. **invariants**: set parameters to static values, which override values provided by the client
    c. **appends**: additional parameters to be combined with the parameters provided by the client

2. **first-components**: optional chain of search components that are applied first to perform pre-processing tasks
3. **components**: primary chain of search components; must at least include the query component
4. **last-components**: optional chain of search components that are applied last to perform post-processing tasks

A request handler does not need to define all phases in **solrconfig.xml**. As you can see from listing 4.5, the **/select** only defines the **defaults** section. This means that all other phases are inherited from the base `solr.SearchHandler` implementation. In practice, customized request handlers are commonly used to simplify client applications. For instance, the Solritas example we introduced in chapter 2 uses a custom request handler **/browse** to power a feature-rich search experience while keeping the client-side code for Solritas very simple.

4.2.3 Browse request handler for Solritas: an example

Hiding complexity from client code is at the heart of Web services and object-oriented design. Solr adopts this proven design pattern by allowing you to define a custom search request handler for your application, which allows you to hide complexity from your Solr client. For example, rather than requiring every query to send the correct parameters to enable spell correction, you can use a custom request handler that has spell correction enabled by default.

The Solr example server comes pre-configured with a great example of this design pattern at work to support the Solritas example application. Listing 4.6 shows an abbreviated definition of the **/browse** request handler from **solrconfig.xml**.

Listing 4.6 Browse request handler for Solritas

```
<requestHandler name="/browse" class="solr.SearchHandler"> #A
   <lst name="defaults">                                   #B
     <str name="echoParams">explicit</str>
     <str name="wt">velocity</str>                         #C
     <str name="v.template">browse</str>                   #C
     <str name="v.layout">layout</str>                     #C
     <str name="title">Solritas</str>                      #C

     <str name="defType">edismax</str>                     #D

     <str name="qf">text^0.5 features^1.0 ...</str> #E

     <str name="mlt.qf">text^0.5 features^1.0 ...</str> #F

     <str name="facet">on</str>  #G
       ...
     <str name="hl">on</str>   #H
     ...
     <str name="spellcheck">on</str> #I
     ...
```

```
    </lst>
    <arr name="last-components">
     <str>spellcheck</str>    #J
    </arr>
  </requestHandler>
```

#A A SearchHandler invokes query processing pipeline
#B default list of query parameters
#C VelocityResponseWriter settings
#D Use the extended dismax query parser
#E Query settings
#F Enable the MoreLikeThis component
#G Enable the Facet component
#H Enable the Highlight component
#I Enable spell checking
#J Invoke the spell checking component as the last step in the pipeline

We recommend that you take a minute to go through all the sections of the **/browse** request handler in the actual **solrconfig.xml** file. One thing that should stand out to you is that a great deal of effort was put into configuring this handler, in order to demonstrate many of the great features in Solr. When starting out with Solr, you definitely do not need to configure something similar for your application all at once. In other words, you can build up a custom request handler over time as you gain experience with Solr.

Let's see the **/browse** request handler in action using the Solritas example. With the example Solr server running, direct your browser to http://localhost:8983/solr/collection1/browse. Enter "iPod" into the search box as shown in Figure 4.6.

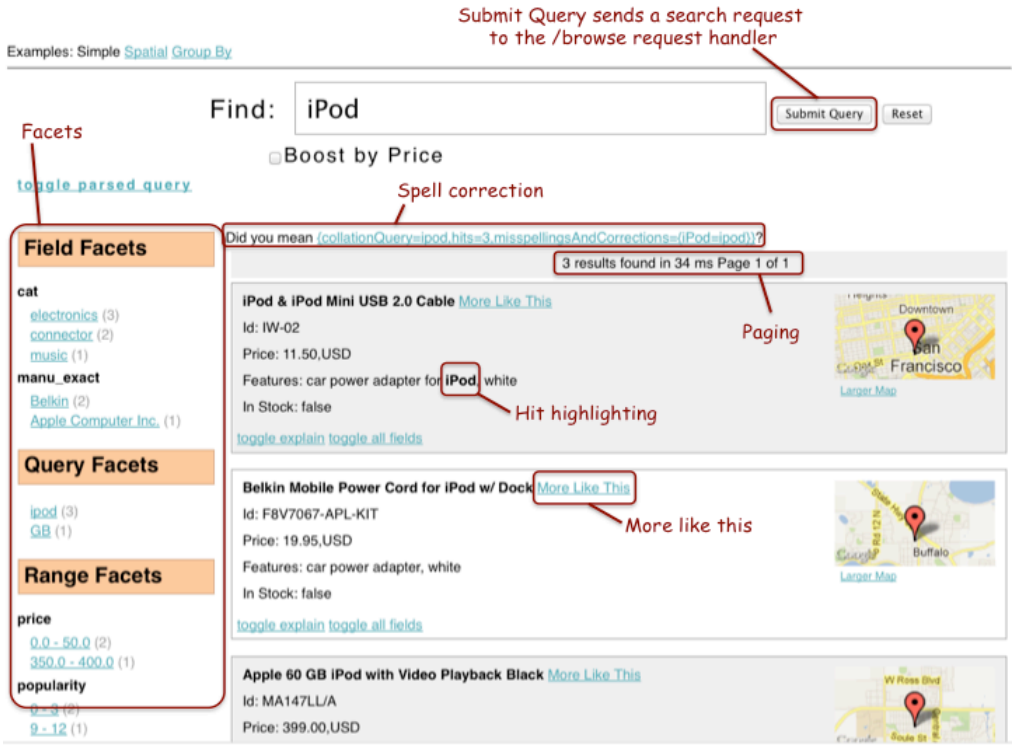


Figure 4.6 Screen shot of Solritas example powered by the **/browse** request handler

Take a moment to scan over figure 4.6 to see all the search features activated for this simple query. Behind the scenes, the Solritas search form submits a query to the **/browse** request handler. In the log, we see:

```
INFO: [collection1] webapp=/solr path=/browse params={q=iPod} hits=3
status=0 QTime=22
```

Notice that the only parameter sent by the search form is `q=iPod`, but the response includes facets, more like this, spell correction, paging, and hit highlighting. That's an impressive list of features for a simple request like `q=iPod`! As you may have guessed, these features are enabled using default parameters in the **/browse** request handler.

The defaults `<lst>` element from listing 4.6 is an ordered list of name/value pairs that provides default values for query parameters if they are not explicitly sent by the client application. For example, the default value for the response writer type parameter "wt" is

"velocity" (`<str name="wt">velocity</str>`). Velocity is an open source templating engine written in Java[2].

From the log message shown above, the only parameter sent by the form was "q", so all other parameters are set by defaults. Let's do a little experiment to see the actual query that gets processed. Instead of using response writer type "velocity", let's set the **wt** parameter to "xml" so we can see the response in raw form without the HTML decoration provided by Velocity. Also, in order to see all the query parameters, we need to set the **echoParams** value to "all". This is a good example of overriding default values by explicitly passing parameters from the client. Listing 4.7 shows the GET URL and a portion of the `<params>` element returned with the response; remember to use the **http** tool provided with the book source code to execute this request. Notice how the number of parameters actually sent to the **/browse** request handler is quite large.

Listing 4.7 Actual list of parameters sent to the /browse request handler for q=iPod

`http://localhost:8983/solr/collection1/`**`browse?q=iPod&wt=xml&echoParams=all`**

```
    <lst name="params">
       <str name="facet">on</str>
       <str
name="mlt.fl">text,features,name,sku,id,manu,cat,title,description,keywords
,author,resourcename</str>
       <str name="f.manufacturedate_dt.facet.range.gap">+1YEAR</str>
       <str name="f.price.facet.range.gap">50</str>
       <str name="q.alt">*:*</str>
       <str name="f.content.hl.fragsize">200</str>
       <str name="v.layout">layout</str>
       <str name="echoParams">all</str>
       <str name="fl">*,score</str>
       <str name="f.price.facet.range.end">600</str>
       <str name="hl.simple.post">&lt;/b&gt;</str>
       <str name="f.name.hl.fragsize">0</str>
       <arr name="facet.field">
         <str>cat</str>
         <str>manu_exact</str>
         <str>content_type</str>
         <str>author_s</str>
       </arr>
       <str name="hl.encoder">html</str>
       <str name="v.template">browse</str>
       <str name="spellcheck.alternativeTermCount">2</str>
       <str name="f.popularity.facet.range.end">10</str>
       <str name="f.manufacturedate_dt.facet.range.start">NOW/YEAR-
10YEARS</str>
       <str name="spellcheck.extendedResults">false</str>
       <str name="spellcheck.maxCollations">3</str>
       <str name="hl.fl">content features title name</str>
       <str name="f.content.hl.maxAlternateFieldLength">750</str>
```

[2] http://velocity.apache.org/engine/index.html

```
      <str name="spellcheck.collate">true</str>
      <str name="wt">xml</str>
      <str name="defType">edismax</str>
      <str name="rows">10</str>
      <str name="facet.range.other">after</str>
      <str name="f.popularity.facet.range.start">0</str>
      <str name="f.title.hl.alternateField">title</str>
      <str name="facet.pivot">cat,inStock</str>
      <str name="f.title.hl.fragsize">0</str>
      <str name="spellcheck">on</str>
      <str name="spellcheck.maxCollationTries">5</str>
      <arr name="facet.range">
        <str>price</str>
        <str>popularity</str>
        <str>manufacturedate_dt</str>
      </arr>
      <str name="hl.simple.pre">&lt;b&gt;</str>
      <str name="hl">on</str>
      <str name="title">Solritas</str>
      <str name="df">text</str>
      <arr name="facet.query">
        <str>ipod</str>
        <str>GB</str>
      </arr>
      ...      #A
</lst>
```

#A Many more default parameters in this request not shown here

From looking at listing 4.7, it should be clear that parameter decoration for a search request handler is a powerful feature in Solr. Specifically, the **defaults** list provides two main benefits to your application. First, helps simplify client code by establishing sensible defaults for your application in one place. For instance, setting the response writer type "wt" to "velocity" means that client applications do not need to worry about setting this parameter. Moreover, if you ever swap out Velocity for another templating engine, your client code does not need to change!

Second, as you can see from listing 4.7, the actual request includes a number of complex parameters needed to configure search components used by Solritas. For example, there are over twenty different parameters to configure the faceting component for Solritas. By pre-configuring complex components like faceting, you can establish consistent behavior for all queries while keeping your client code simple.

The **/browse** handler serves as a good example of what is possible with Solr query processing, but it's also unlikely that it can be used by your application because the default parameters are tightly coupled to the Solritas data model. For example, range faceting is configured for the **price**, **popularity**, and **manufacturedate_dt** fields. Consequently, you should treat the **/browse** handler as an example and not a 100% reusable solution when designing your own application-specific request handler.

4.2.4 Extending query processing with search components

Beyond a set of defaults, the **/browse** request handler defines an array `<arr>` of search components to be applied after the default set of search components are applied to the request using the `<last-components>` element. From listing 4.6, notice that the **/browse** request handler specifies:

```
<arr name="last-components">
  <str>spellcheck</str>
</arr>
```

This configuration means that the default set of search components is applied and then the **spellcheck** component is applied. This is a very common design pattern for search request handlers. In fact, you'll be hard-pressed to come up with an example of where you need to redefine the `<components>` phase for a search handler. Figure 4.7 shows the chain of six built-in search components that get applied during the `<components>` phase of query processing:

Figure 4.7 Chain of six built-in search components

QUERY COMPONENT

The query component is the core of Solr's query processing pipeline. At a high-level, the query component parses and executes queries using the active searcher, which is discussed in section 4.3 below. The specific query parsing strategy is controlled by the "defType" parameter. For instance, the **/browse** request handler uses the **edismax** query parser (`<str name="defType">edismax</str>`), which will be discussed in chapter 7.

The query component identifies all documents in the index that match the query. The set of matching documents can then be used by other components in the query processing chain, such as the facet component. The query component is always enabled and all other components need to be explicitly enabled using query parameters.

FACET COMPONENT

Given a result set identified by the query component, the facet component, if enabled, calculates field-level facets. We cover faceting in-depth in chapter 8. The key take-away for now is that faceting is built-in to every search request and it just needs to be enabled with query request parameters. For **/browse**, faceting is enabled using default parameter: `<str name="facet">on</str>`.

MORELIKETHIS COMPONENT

Given a result set created by the query component, the **More Like This** component, if enabled, identifies other documents that are similar to the documents in search results. To see an example of the **More Like This** component in action, search for "hard drive" in the

Solritas example. Click on the **More Like This** link for the **Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133** result to see a list of similar documents as shown in figure 4.8 below.

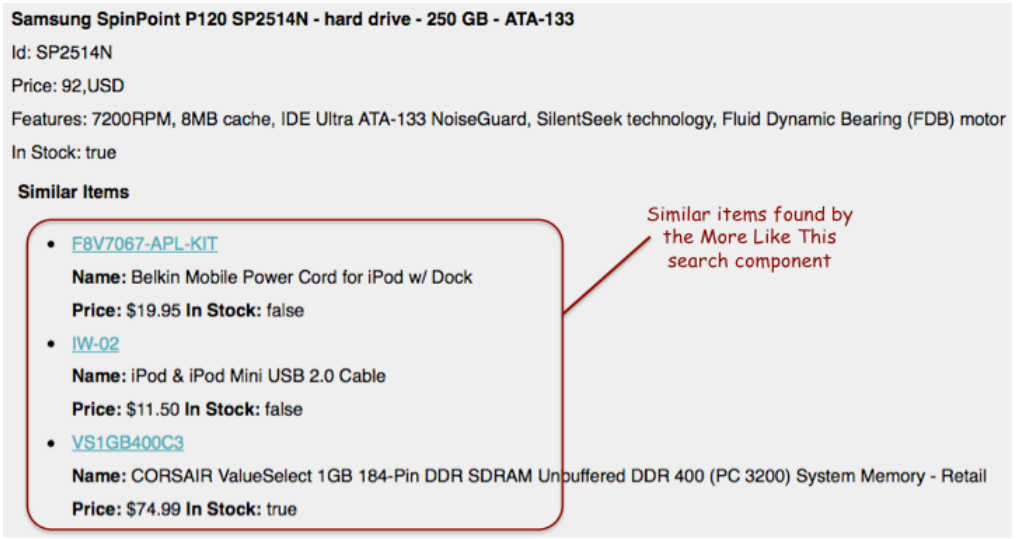


Figure 4.8 Example of similar items found by the **More Like This** search component

We cover the **More Like This** component in chapter 10.

HIGHLIGHT COMPONENT

If enabled, the highlight component highlights sections of text in matching documents to help the user identify highly relevant sections of text in matching documents. Hit highlighting is covered in chapter 9.

STATS COMPONENT

The stats component computes simple statistics like min, max, sum, mean, and standard deviation for numeric fields in matching documents. To see an example of what the stats component produces, execute GET request as shown in listing 4.8:

Listing 4.8 Request summary statistics for the price field using the stats component

```
http://localhost:8983/solr/collection1/select?
  q=*%3A*&
  wt=xml&
  stats=true&
  stats.field=price                #A

<lst name="price">                       #B
  <double name="min">0.0</double>
```

```
    <double name="max">2199.0</double>
    <long name="count">16</long>
    <long name="missing">16</long>
    <double name="sum">5251.270030975342</double>
    <double name="sumOfSquares">6038619.175900028</double>
    <double name="mean">328.20437693595886</double>
    <double name="stddev">536.3536996709846</double>
    <lst name="facets"/>
  </lst>
```

#A Request statistics for the price field
#B Summary statistics returned for the price field

DEBUG COMPONENT

The Debug component returns the parsed query string that was executed and detailed information about how the score was calculated for each document in the result set. The parsed query value is returned to help you track down query formulation issues. The debug component is useful for troubleshooting ranking problems. To see the debug component at work, direct your browser to the following URL:

`http://localhost:8983/solr/collection1/`**`browse?q=iPod&wt=xml&debugQuery=true`**

You should notice that this is the exact same query that we executed from the Solritas form except we changed the response writer type "wt" to XML (instead of Velocity) and enabled the debug component using `debugQuery=true` in the HTTP GET request. Listing 4.9 shows a snippet of the XML output produced by the debug component:

Listing 4.9 Snippet of the XML output produced by the Debug component

```
http://localhost:8983/solr/collection1/browse?
    q=iPod&
    wt=xml&
    debugQuery=true #A

  <lst name="debug">
      <str name="rawquerystring">iPod</str>
      <str name="querystring">iPod</str>
      <str name="parsedquery">(+DisjunctionMaxQuery((id:iPod^10.0 |  #B
  author:ipod^2.0 | title:ipod^10.0 | text:ipod^0.5 | cat:iPod^1.4 |
  keywords:ipod^5.0 | manu:ipod^1.1 | description:ipod^5.0 |
  resourcename:ipod | name:ipod^1.2 | features:ipod |
  sku:ipod^1.5)))/no_coord</str>
      ...
      <lst name="explain">       #C
        <str name="IW-02">
  0.13513829 = (MATCH) max of:
    0.045974977 = (MATCH) weight(text:ipod^0.5 in 4) [DefaultSimilarity],
  result of:
      0.045974977 = score(doc=4,freq=3.0 = termFreq=3.0
  ), ...</str>
    </lst>
```

#A Enable the debug component
#B query produced by the edismax query parser
#C Explanation of score calculation for each document in the request

Notice how a single term query "iPod" entered by the user results in a fairly complex query composed of many different boosts on numerous fields. The more complex query is created by the **edismax** query parser, which is enabled by the **defType** parameter under defaults. The edismax parser is covered in chapter 7.

ADDING SPELLCHECK AS A LAST-COMPONENT

After the six built-in search components process the request, the **/browse** search handler invokes the spellcheck component, which is listed in the `<last-components>` phase. Listing 4.10 shows the definition of the spellcheck component from **solrconfig.xml**:

Listing 4.10 Define a search component to do spell checking

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent"> #A
  <str name="queryAnalyzerFieldType">textSpell</str>   #B
  <lst name="spellchecker">                            #B
    <str name="name">default</str>
    <str name="field">name</str>
    <str name="classname">solr.DirectSolrSpellChecker</str>
    ...
  </lst>
</searchComponent>
```

#A Define a search component named "spellcheck" of type solr.SpellCheckComponent
#B Spell check component specific parameters, see chapter 10

Notice that the name of the component "spellcheck" matches what is listed in the `<last-components>` section of the **/browse** request handler. You'll need some more background on how Solr's spell correction feature works before the settings in listing 4.10 make sense, so we'll return to this configuration element in chapter 9. The key take-away at this point is seeing how a search component is added to the search request-handling pipeline using `<last-components>`.

At this point, you should have a solid understanding of how Solr processes query requests. Before we move on to another configuration topic, you should be aware that the Solr administration console provides access to all active search request handlers under **Plugins / Stats > QUERYHANDLER**. Figure 4.9 shows properties and statistics for the **/browse** search handler, which as you might have guessed is just another MBean.

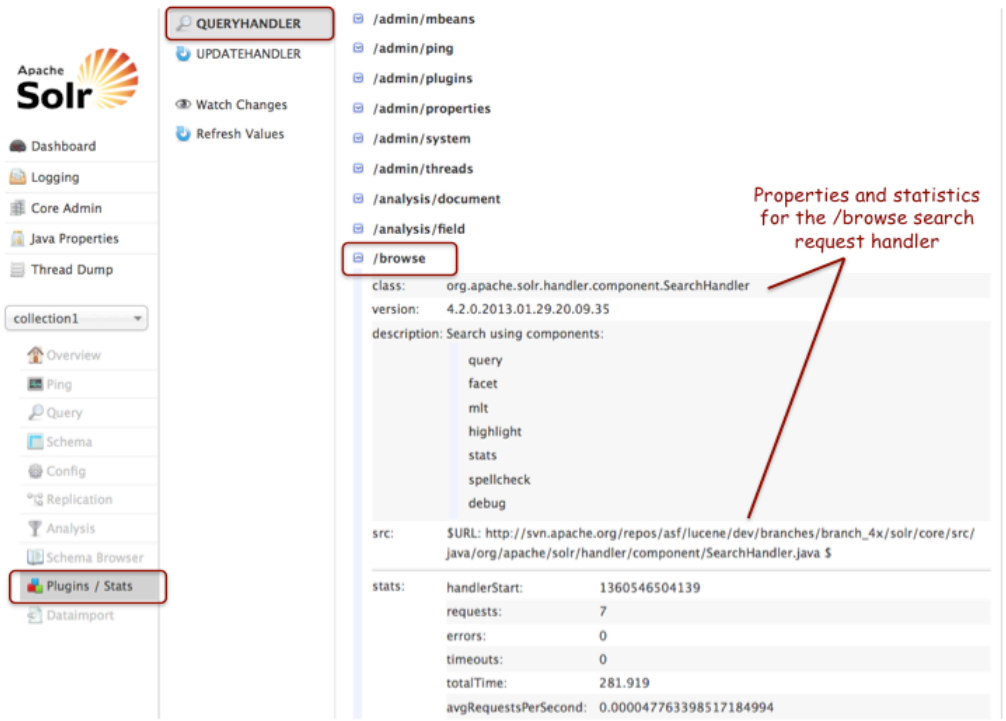


Figure 4.9 Screen shot showing properties and statistics for the **/browse** request handler in the Solr administration console under **Plugins / Stats > QUERYHANDLER**

Now let's turn our attention to configuration settings that help optimize query performance.

4.3 Managing searchers

The <query> element contains settings that allow you to optimize query performance using techniques like caching, lazy field loading, and new searcher warming. It goes without saying that designing for optimal query performance from the start is critical to the success of your search application. In this section, you'll learn about managing searchers, which is one of the most important techniques for optimizing query performance.

4.3.1 New searcher overview

In Solr, queries are processed by a component called a **searcher**. There is only one "active" searcher in Solr at any given time. All query components for all search request handlers execute queries against the active searcher.

The active searcher has a read-only view of a snapshot of the underlying Lucene index. It follows that if you add a new document to Solr, then it is not visible in search results from

the current searcher. This raises the question of how do new documents become visible in search results? The answer, of course, is to close the current searcher and open a new one that has a read-only view of the updated index. This is what it means to commit documents to Solr. The actual commit process in Solr is more complicated but we'll save a thorough discussion of the nuances of commits for the next chapter. For now you can think of a commit as a black-box operation that makes new documents and any updates to your existing index visible in search results by opening a new searcher.

Figure 4.10 shows the active searcher MBean for the **collection1** core in the example server available under the **CORE** section of the **Plugins / Stats** page.

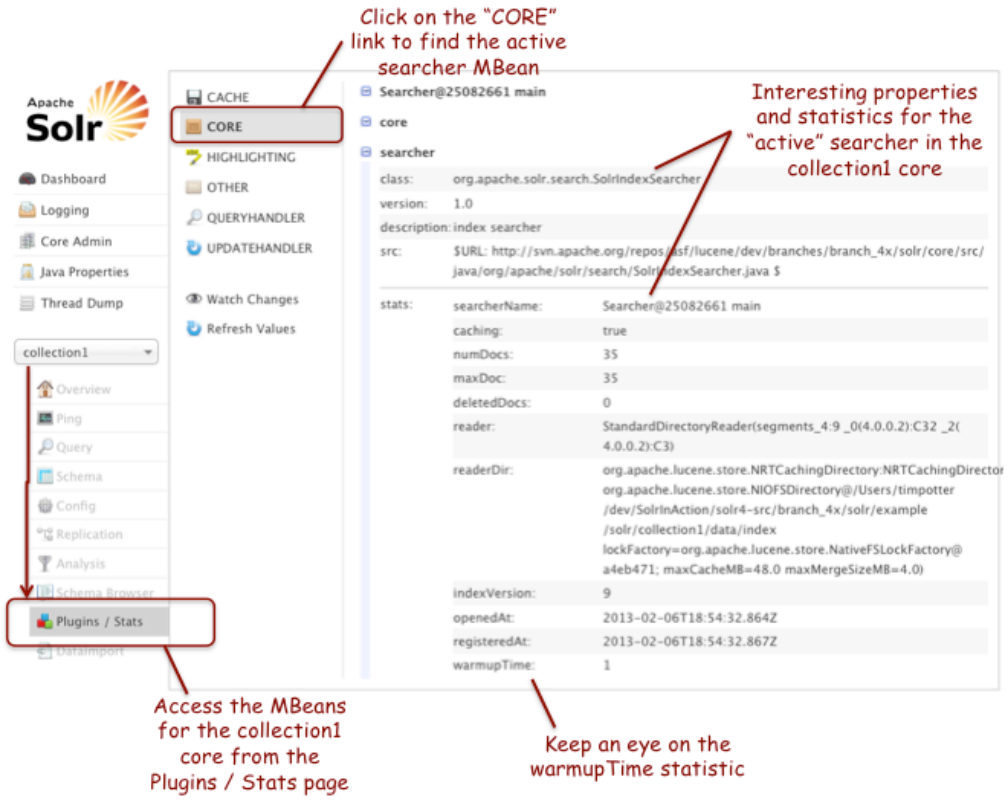


Figure 4.10 Inspecting the active searcher MBean in the collection1 core from the Solr admin console

On the **CORE** page, take note of the **searcherName** property (in the diagram it is "Searcher@25082661 main"). Let's trigger the creation of a new searcher by re-sending all the example documents to your server as we did in section 2.1.4 using:

```
cd $SOLR_INSTALL/example/exampledocs
```

```
java -jar post.jar *.xml
```

Now, refresh the **CORE** page and notice that the **searcherName** property has changed to be a different instance of the searcher. A new searcher was created because the post.jar command sent a commit after adding the example documents.

So now that we know a commit creates a new searcher to make new documents and updates visible, let's think about the implications of creating a new searcher. First, the old searcher must be destroyed. However, there could be queries currently executing against the old searcher so Solr must wait for all in-progress queries to complete.

Also, any cached objects that are based on the current searcher's view of the index must be invalidated. We'll learn more about Solr cache management in the next section. For now, think about a cached result set from a specific query. As some of the documents in the cached results may have been deleted and new documents may now match the query, it should be clear that the cached result set is not valid for the new searcher.

Because pre-computed data, such as a cached query result set, must be invalidated and re-computed, it stands to reason that opening a new searcher on your index is potentially an expensive operation. This can have a direct impact on user experience. For instance, imagine a user paging through search results and a new searcher is opened after they click on page 2 but before they request page 3. When the user requests the next page, all of the previously computed filters and cached documents are no longer valid. Without some care, the user is likely to experience some slowness, especially if their query is complex.

The good news is that Solr has a number of tools to help alleviate this situation. First and foremost, Solr supports the concept of warming a new searcher in the background and keeping the current searcher active until the new one is fully warmed.

4.3.2 Warming a new searcher

Solr takes the approach that it is better to serve stale results for a short period of time rather than allowing query performance to slow down significantly. This means that Solr does not close the current searcher until a new searcher is warmed up and ready to execute queries with optimal performance.

Warming a new searcher is much like a sprinter in track and field. Before a sprinter goes full speed in a race, she makes sure her muscles are warmed up and ready perform at full speed when the gun fires to start the race. Just as a sprinter wouldn't start a race with cold muscles, nor should Solr activate a "cold" searcher.

In general, there are two types of warming activities: 1) auto-warming new caches from the old caches, and 2) executing cache warming queries. We'll learn more about auto-warming caches in the next section when we dig into Solr's cache management features.

A cache-warming query is a pre-configured query (in solrconfig.xml) that gets executed against a new searcher in order to populate the new searcher's caches. Listing 4.11 shows the configuration of cache warming queries for the example server.

Listing 4.11 Define a listener for newSearcher events to execute warming queries

```
<listener event="newSearcher" class="solr.QuerySenderListener"> #A
 <arr name="queries">  #B
   <!--                                                                  #C
   <lst><str name="q">solr</str><str name="sort">price asc</str></lst>   #C
   <lst><str name="q">rocks</str><str name="sort">weight asc</str></lst> #C
   -->                                                                  #C
  </arr>
</listener>
```

#A Define a listener to handle newSearcher events
#B Define a named list of query objects to warm the new searcher
#C Intentionally commented out so that you configure application specific queries for your environment

The configuration settings in listing 4.11 register a named list (`<arr name="queries">`) of queries to execute whenever a **newSearcher** event occurs in Solr, such as after a commit. Also, note that the actual queries are commented out! This is intentional because there is a cost to executing warming queries and the Solr developers wanted to ensure you configure warming queries explicitly for your application. In other words, the cache warming queries are application specific so the out-of-the-box defaults are strictly for example purposes. Put simply, you are responsible for configuring warming queries.

CHOOSING WARMING QUERIES

Having the facility to warm new searchers by executing queries is only a great feature if you can identify queries that will help improve query performance. As a rule of thumb, warming queries should contain query parameters (q, fq, sort, etc.) that are used frequently by your application. Since we haven't covered Solr query syntax yet, we'll table the discussion of creating warming queries in until chapter 7. For now, it's sufficient to make a mental note that you need to revisit this topic once you have a more thorough understanding of Solr query construction.

We should also mention that you do not need to have any warming queries for your application. If query performance begins to suffer after commits, then you'll know it is time to consider using warming queries.

TOO MANY WARMING QUERIES

The old adage of "less is more" applies to warming queries. Each query takes time to execute so having many warming queries configured can lead to long delays in opening new searchers. Thus, it's best to keep the list of warming queries to the minimal set of the most important queries for your search application.

You might be wondering what the problem with a new searcher taking a long time to warm-up is. It turns out that warming too many searchers in your application concurrently can consume too many resources (CPU and memory), thus leading to a degraded search experience.

FIRST SEARCHER

There is also the concept of warming the first searcher during Solr initialization or after reloading a core. We'll leave it as an exercise for the reader to determine if there's value in configuring warming queries for the first searcher. Most Solr users just use the same queries for warming new and first searchers.

Including XML elements from other sources using XInclude

As a brief aside, Solr supports using XInclude to pull XML elements from other files into solrconfig.xml. For example, rather than duplicating your list of warming queries for new and first searchers, you can maintain the list in a separate file and XInclude it in both places, using: `<xi:include href="warming-queries.xml" xmlns:xi="http://www.w3.org/2001/XInclude">`

USECOLDSEARCHER

Before we turn our attention to Solr's cache management, we want mention two additional searcher-related elements in solrconfig.xml. The `<useColdSearcher>` element covers the case where a new search request is made and there is no currently registered searcher. If `<useColdSearcher>` is **false**, then Solr will block until the warming searcher has completed executing all warming queries; this is the default configuration for the example Solr server: `<useColdSearcher>false</useColdSearcher>`

On the other hand, if `<useColdSearcher>` is **true**, then Solr will immediately register the warming searcher regardless of how "warm" it is. Returning to our track-and-field analogy, **false** would mean the starting official waits to start the race until our sprinter is fully warmed-up, regardless of how long that takes. Conversely, a **true** value means that the race will start immediately regardless of how warmed-up our sprinter is.

MAXWARMINGSEARCHERS

It's conceivable that a new commit is issued before the new searcher warming process completes, which implies another new searcher needs to be warmed up. This is especially true if your searchers take considerable time to warm-up. The `<maxWarmingSearchers>` element allows you to control the maximum number of searchers that can be warming up in the background concurrently. Once this threshold is reached, new commit requests will fail, which is a good thing because allowing too many warming searchers to run in the background can quickly eat up memory and CPU resources on your server. Solr ships with a default of 2, which is a good value to start with:

`<maxWarmingSearchers>2</maxWarmingSearchers>`

If you find your server is reaching the maximum threshold too often, then revisit your warming logic to see if new searcher warming is taking too long.

So hopefully you now have a good sense for what a searcher is and how to configure Solr to manage searchers correctly for your application. Now let's look at more ways to optimize query performance using caching.

4.4 Cache management

Solr provides a number of built-in caches to improve query performance. Before we get into the details of specific Solr caches, it's important to understand cache management fundamentals in Solr.

4.4.1 Cache fundamentals

There are four main concerns when working with Solr caches:

1. Cache sizing and eviction policy
2. Hit ratio and evictions
3. Cached object invalidation
4. Auto-warming new caches

Broadly speaking, proper cache management in Solr is not a set-it and forget-it type process. You will need to keep an eye on your caches and fine-tune them based on actual usage of Solr. Remember that the Solr administration console is your friend when it comes to monitoring important components like caches and searchers.

CACHE SIZING

In terms of cache sizing, you don't want your caches to be so large that they consume all available memory in your JVM. Solr keeps all cached objects in memory and does not overflow to disk, as is possible with some caching frameworks. Consequently, Solr requires you to set an upper limit on the number of objects in each cache. Solr will evict objects when the cache reaches the upper limit using either a Least Recently Used (LRU) or Least Frequently Used (LFU) eviction policy.

Least Recently Used (LRU) evicts objects when a cache reaches its maximum threshold based on the time when an object was last requested from the cache. When a cache is full and a new object is added, the LRU policy will remove the oldest entry where age is determined by the last time each object in the cache was requested. The LRU policy is the default configuration for Solr.

Solr also provides a Least Frequently Used (LFU) policy that evicts objects based on how frequently they are requested from the cache. This is beneficial for applications that want to give priority to more popular items in the cache, rather than just those that have been used recently. For example, Solr's filter cache is a good candidate for using the LFU eviction policy because filters are typically expensive to create and store so you want to keep the filter cache small and give priority to the most popular filters in your application. We'll learn more about the filter cache in the next section.

A common misconception with cache sizing is to make your cache sizes really large if you have the memory available. The problem with this approach is that once a cache becomes invalidated after a commit, there can be many objects that need to be garbage collected by the JVM. Remember, closing a searcher invalidates all cached values. Without proper tuning of garbage collection, this can lead to long pauses in your server caused by full garbage collection. We'll learn more about tuning garbage collection parameters for Solr in chapter 12. For now, the important lesson is to avoid defining overly large caches and let some objects in the cache be evicted periodically.

HIT RATIO AND EVICTIONS

Hit ratio is the proportion of cache get requests that result in finding a cached value. Hit ratio indicates how much benefit your application is getting from its cache. Ideally, you want your hit ratio to be as close to 1 (100%) as possible. Conversely, a low hit ratio is an indication that Solr is not benefiting from caching.

Eviction count shows how many objects have been evicted from the cache based on the eviction policy described above. It follows that having a large number of evictions is an indication that the maximum size of your cache may be too small for your application. Also, eviction count and hit ratio are interrelated, as a high eviction count will lead to a sub-optimal hit ratio.

CACHED OBJECT INVALIDATION

In most cache management scenarios, you need to worry about how to invalidate a cached object so that your application does not return stale data. However, in Solr, this is not a concern because all objects in a cache are linked to a specific searcher instance and are immediately invalidated when a searcher is closed. Recall that a searcher is a read-only view of a snapshot of your index; consequently all cached objects remain valid until the searcher is closed.

AUTO-WARMING NEW CACHES

As we discussed in section 4.3, Solr creates a new searcher after a commit but it does not close the old searcher until the new searcher is fully warmed. It turns out that some of the keys in the soon-to-be-closed searcher's cache can be used to populate the new searcher's cache, a process known as "auto-warming" in Solr. Note that auto-warming a cache is different than using a warming query to populate a cache, as we discussed above in section 4.3.2.

Every Solr cache supports an **autowarmCount** attribute that indicates either the maximum number of objects or a percentage of the old cache size to auto-warm. How the objects are actually auto-warmed depends on the specific cache. We'll learn more about cache specific auto-warming strategies below. The key point for now is that you can configure Solr's caches to refresh a subset of cached objects when opening a new searcher, but as with any optimization technique, you need to be careful to not over do it.

At this point you should have a basic understanding of cache management concepts in Solr. Now, let's learn about the specific types of caches Solr uses to optimize query performance, starting with one of the most important caches: **filter cache**.

4.4.2 Filter cache

In Solr, a filter restricts search results to documents that meet the filter criteria but does not affect scoring. We saw an example of this in chapter 2 section 2.2.1, where our first example query filtered documents having manufacturer field (manu) set to "Belkin" using `fq=manu:Belkin` as seen in listing 4.12.

Listing 4.12 Example query using a filter query "fq" on the manu field

```
http://localhost:8983/solr/collection1/select?
q=iPod&
fq=manu%3ABelkin&                                    #A
sort=price+asc&
fl=name%2Cprice%2Cfeatures%2Cscore&
df=text&
wt=xml&
start=0&rows=10
```

#A Filter query "fq" on the manu field

When Solr executes this query, it computes and caches an efficient data structure that indicates which documents in your index match the filter. For the example server, there are 2 documents that match this filter.

Now consider what happens if another query is sent to Solr with the same filter query (`fq=manu:Belkin`) but a different query, such as `q=USB`. Wouldn't it be nice if the second query could use the results of the filter query from the first query? This is the purpose of Solr's filter cache! To see this cache in action, navigate to the **Query** page for the **collection1** core and submit the query as shown in figure 4.11.

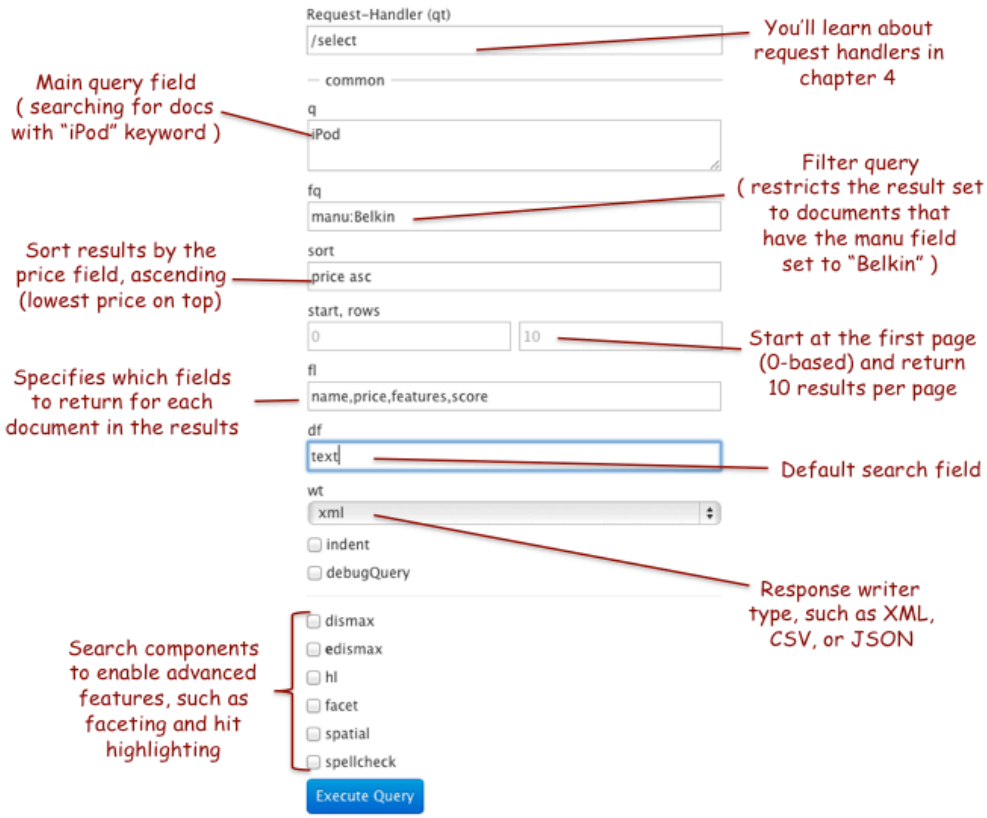


Figure 4.11 Execute a query with a filter query "fq" clause to see the filter cache in action

Next, navigate to the to the **Plugins / Stats** page for the **collection1** and click on the **CACHE** link. Figure 4.12 shows the properties and statistics for the **filterCache** MBean. Re-execute the same query several times and you will see the **filterCache** statistics change.

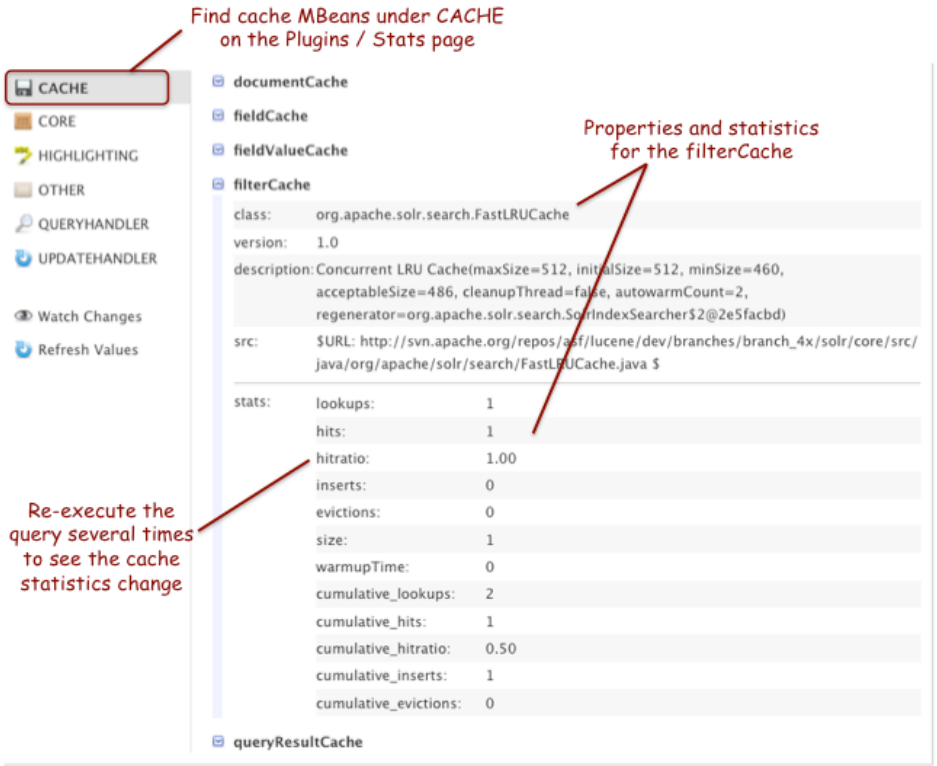


Figure 4.12 Inspecting the filterCache MBean from the Plugins / Stats page

Of course, it's difficult to fully appreciate the value of caching filters for a small index, but imagine an index with millions of documents and you can see how caching filters can really help optimize query performance. In fact, using filters to optimize queries is one of the most powerful features in Solr, mainly because filters are re-usable across queries. For now, we'll save a deeper discussion of filters for chapter 7 and turn our focus to how the filter cache is configured in **solrconfig.xml**. Listing 4.13 shows the default configuration for the filter cache.

Listing 4.13 Initial settings for the filter cache in the example server

```
<filterCache class="solr.FastLRUCache" #A
             size="512"                #B
             initialSize="512"
             autowarmCount="0"/>       #C
```

#A Uses the Least Recently Used (LRU) eviction policy
#B Maximum size is 512 objects
#C No objects are auto-warmed for this cache

AUTO-WARMING THE FILTER CACHE

A filter can be a powerful tool for optimizing queries, but you can also get into trouble if you don't manage cached filters correctly. Filters can be expensive to create and store in memory if you have a large number of documents in your index, or if the filter criteria is complex. If a filter is generic enough to apply to multiple queries in your application, then it makes sense to cache the resulting filter. In addition, you probably want to auto-warm some of the cached filters when opening a new searcher.

Let's look under the hood of the filter cache to understand what happens during auto-warming of objects in the filter cache. By now, you should know that objects cannot just be moved from the old cache to the new cache because the underlying index has changed, thus invalidating cached objects like filters. Of course each object in the cache has a key. For the filter cache, the key is the filter query, such as `manu:Belkin`. To warm the new cache, a subset of keys are pulled from the old cache and then executed against the new searcher, which re-computes the filter. In other words, auto-warming the filter cache requires Solr to re-execute the filter query with the new searcher. Consequently, auto-warming the filter cache can be a source of performance and resource utilization problems in Solr.

Imagine the scenario where you have hundreds of filters cached and your **autowarmCount** is set to 100. When warming the new searcher, Solr must execute 100 filter queries. Imagine it takes 65 seconds to execute the 100 filter queries and your application commits changes every minute. Under this scenario, you'll quickly run into problems where you are warming too many searchers in the background.

We recommend that you enable auto-warming for the filter cache but set the **autowarmCount** attribute to a small number, less than 10 to start out. In addition, we think the LFU eviction policy is more appropriate for the filter cache because it allows you to keep the filter cache small and give priority to the most popular filters in your application. Here are the recommended configuration settings for the filter cache:

```
<filterCache class="solr.LFUCache" #A
             size="100"            #B
             initialSize="20"
             autowarmCount="10"/>  #C
```

#A Change to use Least Frequently Used (LFU) eviction policy
#B Keep the size of this cache to a reasonable maximum
#C Only auto-warm the 10 most popular filters

Of course, you need to do some experimentation with these parameters depending on how many filters your application uses and how frequently you commit against your index.

In terms of memory usage per cached filter, Solr has different filter representations based on the size of the matching document set. As an upper limit, you can figure that any filter that matches many documents in your index will require MaxDoc bits of memory. For example, if your index has 10 million documents, then a filter can take up to 10 million bits of memory, or roughly 1.2MB.

4.4.3 Query result cache

The query result cache holds result sets for a query. Consider our sample query from listing 4.11. If you execute this query more than once, then subsequent results are served from the query result cache rather than re-executing the same query against the index. This can be a powerful solution for reducing the cost of computationally expensive queries. The query result cache is defined as:

```
<queryResultCache class="solr.LRUCache"
                  size="512"
                  initialSize="512"
                  autowarmCount="0"/>
```

Behind the scenes, the query result cache holds a query as the key and a list of internal Lucene document IDs as the value. Internal Lucene document IDs can change from one searcher to the next, so the cached values must be recomputed when warming the query result cache.

To auto-warm the query result cache, Solr needs to re-execute queries, which can be expensive. So the same advice we gave about keeping the **autowarmCount** attribute small for the filter cache applies the query result cache. That said, we do recommend setting the **autowarmCount** attribute for this cache to something other than the default zero so that you get some benefit from auto-warming recent queries.

Beyond sizing, Solr provides a few miscellaneous settings to help you fine-tune your usage of the query result cache.

QUERY RESULT WINDOW SIZE

In section 2.2.4, we stressed the importance of paging your search results in Solr to ensure optimal query performance. The `<queryResultWindowSize>` element allows you to prepare additional pages when you execute a query.

For example, imagine your application shows 10 documents per page and that in most cases your users only look at the first and second pages. You can set `<queryResultWindowSize>` to 20 to avoid having to re-execute the query to retrieve the second page of results. In general, you want to set this element to 2 or 3 times the page size used by your most important queries. However, if you set it too large, then every query is paying the price of loading more documents than you are showing to the user. If your users rarely go beyond page 1, then it is better to set this element to the page size. You can determine the percentage of queries for additional pages of results by searching the Solr log files looking for start parameter > 0.

QUERY RESULT MAX DOCS CACHED

As we discussed in section 4.4.1, you have to set a maximum size for Solr caches, however this doesn't affect the size of each individual entry in the cache. You can imagine a result set holding millions of documents in the cache would greatly impact available memory in Solr. The `<queryResultMaxDocsCached>` element allows you to limit the number of documents cached for each entry in the query result cache. As most users only look at the first couple of pages, you should set this to 2 or 3 times the page size.

ENABLE LAZY FIELD LOADING

A common design pattern in Solr is to have a query return a subset of fields for each document. For instance, in our example query from section 4.4.2 (listing 4.11), we requested the name, price, features, and score fields. However, the documents in the index have many more fields, such as category, popularity, manufacture date, etc. If your application adopts this common design pattern, then you want to set `<enableLazyFieldLoading>` to true to avoid loading unwanted fields. Of course, our example documents do not have many fields so it's easy to overlook the benefit of this setting. In practice, most documents have many fields so it is a good idea to load fields lazily.

4.4.4 Document cache

The query result cache holds a list of internal document IDs that match a query, so even if the query results are cached, Solr still needs to load the documents from disk to produce search results. The document cache is used to store documents loaded from disk in memory keyed by their internal document ID. It follows that the query result cache uses the document cache to find cached versions of documents in the cached result set.

This raises the question whether it makes sense to auto-warm the document cache? There's actually a good argument to be made against auto-warming this cache because there's no way to ensure the documents you are warming have any relation to queries and filters being auto-warmed from the query result and filter caches. In other words, you could be spending time re-creating documents that may not actually benefit your warmed filter and query result caches.

4.4.5 Field value cache

The last cache we'll mention is the field value cache, which is strictly used by Lucene and is not managed by Solr. The field value cache provides fast access to stored field values by internal document ID. The field value cache is used during sorting and when building documents for the response. As this is an advanced topic, we'll refer you to the Lucene JavaDoc[3] for more information.

At this point, you know how Solr processes queries using a request handling pipeline and you know how to optimize query performance using new searcher warming and caching. Let's finish our tour of **solrconfig.xml** by learning how to control what gets returned from a query.

4.5 Response formatting

Solr provides great flexibility in what is returned to a client in the response. In this section, we'll cover query response writers and document transformers.

---

[3] See JavaDoc for org.apache.lucene.search.FieldCache

4.5.1 Response writer

After query processing completes, Solr passes the results to a response writer based on the "wt" parameter in the request. A response writer transforms the Java objects into a serialized form to be returned to the client application. As we've seen in numerous examples, the default value for "wt" is XML. In addition to XML, the Solr example server includes response writers for JSON, XSLT, CSV, and Velocity.

Let's take a look at a few of the response writer definitions from **solrconfig.xml** to see how they work. Listing 4.14 provides the definition of the JSON response writer.

Listing 4.14 JSON response writer

```
<queryResponseWriter name="json"   #A
         class="solr.JSONResponseWriter"> #B
  <str name="content-type">text/plain; charset=UTF-8</str> #C
</queryResponseWriter>
```

#A The name of the writer should match the "wt" parameter in the request
#B Define the Java class that implements the writer
#C Set the HTTP content-type header for the HTTP response

The Solritas example uses the Velocity writer to demonstrate how Solr can be configured to produce a full Web-based UI using a custom response writer. The actual definition of the Velocity response writer is quite simple:

```
<queryResponseWriter name="velocity"
      class="solr.VelocityResponseWriter" startup="lazy"/>
```

Of course, all of the actual work is in the Velocity templates! If you're interested in how the Velocity templates work to produce the Solritas UI, take a look at `$SOLR_INSTALL/example/solr/collection1/conf/velocity/*.vm`.

Let's move on to a new feature in Solr 4 that allows you to enrich or otherwise transform documents in the response using a document transformer.

4.5.2 Document transformers

A document transformer allows you to add fields dynamically to documents in the response. For instance, you could use a document transformer to enrich a document with a value from an external database. Let's work with one of the built-in document transformers to see how they work.

Specifically, we'll use Solr's **ExplainAugmenterFactory** to add the Lucene explanation of the document score to each result. This could be useful for a search performance monitoring application to track the relationship between user clicks and document ranking. First, you need to activate the transformer by adding the following configuration to solrconfig.xml:

```
<transformer name="explain" #A
    class="org.apache.solr.response.transform.ExplainAugmenterFactory">
  <str name="args">nl</str> #B
</transformer>
```

#A The name of the field to produce in response documents
#B Return explanation as a named-list

Next, you need to reload the core to pick-up this configuration change, see section 4.1.2. After reloading the core, execute the query passing **[explain]** in the field list parameter "fl" as in listing 4.15:

Listing 4.15 Return an explanation of the score for each document using a transformer

```
http://localhost:8983/solr/collection1/select?
  q=iPod&
  fq=manu%3ABelkin&
  fl=name%2Cprice%2Cfeatures%2Cscore%2C[explain]&  #A
  wt=xml

<doc>
  <str name="name">iPod & iPod Mini USB 2.0 Cable</str>
  <arr name="features">
    <str>car power adapter for iPod, white</str>
  </arr>
  <float name="price">11.5</float>
  <float name="score">1.3722405</float>
  <str name="[explain]">   #B
 1.3722405 = (MATCH) weight(text:ipod in 4) [DefaultSimilarity],
  result of: 1.3722405 = fieldWeight in 4, product of:
    1.7320508 = tf(freq=3.0), with freq of:
      3.0 = termFreq=3.0
    3.1690538 = idf(docFreq=3, maxDocs=35)
    0.25 = fieldNorm(doc=4)
  </str>
</doc>
```

#A Request the [explain] field for each document in the field list parameter "fl"
#B [explain] field produced for each document using a document transformer

Table 4.2 provides a summary of Solr's built-in document transformers:

Transformer	Overview
[explain]	Explanation of the score calculation for each document in the results.
[value]	Add a constant value to each document; most frequently used to provide a default value in the response when the actual value is null.
[shard]	Provides the shard that served the document; we'll use this in chapter 13 when we learn about sharding with SolrCloud.
[docid]	Returnes the internal document ID from the Lucene index.

4.6 Core Admin API

Before we wrap up this chapter, we want to introduce the Core Admin API, which allows you to programmatically create, update, reload, rename, swap, and unload cores. You've already seen how to reload a core from the Solr administration panel in section 4.1.2. It turns out

that the Core Admin page just uses the Core Admin API behind the scenes. To begin, let's use the STATUS action to get some basic status information about the collection1 core. As you might have guessed the Core Admin API is based on HTTP so requesting status is as simple as sending the following HTTP GET request as in listing 4.16.

Listing 4.16 Get the status of a core using the Core Admin API

```
http://localhost:8983/solr/admin/cores? #A
action=STATUS&  #B
name=collection1  #C
```

#A Send a GET request to the /admin/cores handler
#B Action to execute in the Core Admin API
#C Name of the collection to get status for

This results in an XML response similar to:

```
<response>
  <lst name="responseHeader"> ... </lst>
  <str name="defaultCoreName">collection1</str>
  <lst name="initFailures"/>
  <lst name="status">
    <lst name="collection1">
      <str name="name">collection1</str>
      <bool name="isDefaultCore">true</bool>
      ...
      <date name="startTime">2013-02-12T21:31:03.957Z</date>
      <long name="uptime">7775928</long>
      <lst name="index">
        <int name="numDocs">35</int>
        <int name="maxDoc">35</int>
        <int name="deletedDocs">0</int>
        ...
      </lst>
    </lst>
  </lst>
</response>
```

The general format for a Core Admin API request is to specify an action and a core name. Let's use the API to create a new core. To begin, make sure the example server is running and enter the following URL into your browser:

Listing 4.17 Attempt to create a new core using the Core Admin API

```
http://localhost:8983/solr/admin/cores?    #A
     action=CREATE&                        #A
     name=SolrInAction&                    #A
     instanceDir=sia&                      #A
     config=solrconfig.xml&                #A
     schema=schema.xml&                    #A
     dataDir=data                          #A

<response>
   <lst name="responseHeader">
     <int name="status">400</int>
```

```
    <int name="QTime">8</int>
  </lst>
  <lst name="error">
    <str name="msg">
  Error CREATEing SolrCore 'SolrInAction':    #B
  Unable to create core: SolrInAction</str>
    <int name="code">400</int>
  </lst>
</response>
```

#A HTTP GET request to create a new core named "SolrInAction"
#B Expected error message because the sia directory does not exist

The operation should have failed because there is no directory named "sia" under Solr home. To resolve this problem, open a command-line and recursively copy the **collection1** directory to **sia**; on Unix/Linux based systems, you can do:

```
cd $SOLR_INSTALL/example/solr
cp -r collection1 sia
```

Now try the URL again and you should see a response message that looks like:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1861</int>
  </lst>
  <str name="core">SolrInAction</str>
  <str name="saved">.../example/solr/solr.xml</str>
</response>
```

The core admin API returns the response as XML but you can control that using the "wt" parameter, such as `wt=json` will return the response as JSON.

Verify that your new SolrInAction core is active by going to the Solr administration console. You should now see SolrInAction on the left navigation panel below collection1. Also, take a peek at the solr.xml and you should see something like:

```
<solr persistent="true">
  <cores adminPath="/admin/cores" defaultCoreName="collection1"
zkClientTimeout="${zkClientTimeout:15000}" hostPort="8983"
hostContext="solr">
    <core name="collection1" instanceDir="collection1" />
    <core schema="schema.xml" loadOnStartup="true" instanceDir="sia/"
transient="false" name="SolrInAction" config="solrconfig.xml"
dataDir="data"/>
  </cores>
</solr>
```

Next, execute the find all documents query (`*:*`) for the SolrInAction core. Notice that there are documents in your new core! This is because we copied the index data directory over from the collection1 core. Of course in practice you probably don't want two cores with the same data but there's no harm in having the same data for now.

Now, let's use the API to reload the SolrInAction core after making a configuration change. For the sake of this exercise, change the **autowarmCount** to 10 for each of the caches defined in solrconfig.xml for the SolrInAction core at:

`$SOLR_HOME/sia/conf/solrconfig.xml`. After making your changes, you need to reload the core to apply the configuration changes. This is done using the RELOAD action in the Core Admin API as shown in listing 4.18.

Listing 4.18 Programmatically reload a core to apply configuration changes

```
http://localhost:8983/solr/admin/cores?
   action=RELOAD&
   core=SolrInAction

   <response>

   <lst name="responseHeader">
     <int name="status">0</int>
     <int name="QTime">367</int>
   </lst>
   </response>
```

If you go to the plug-in stats page in the admin console for the SolrInAction core, you'll see the updated setting for **autowarmCount**.

4.7 Summary

Definitely give yourself a pat on the back after working through this long chapter! We know learning about configuration is not the most interesting of topics. At this point you should have a solid understanding of how to configure Solr, especially for optimizing query-processing performance. Specifically, you learned that Solr's request processing pipeline is composed of a unified request dispatcher and a highly configurable request handler. We saw how a search request handler has four phases and you can customize each phase. The **/browse** handler for the Solritas application served as a good example of using default parameters and custom components (spellcheck) to enable a feature-rich search experience while simplifying client application code.

We also learned how Solr processes query requests using a read-only view of the index with a component called a **searcher**. There is only one active searcher in Solr at any point in time and a new searcher needs to be created before updates to the index are visible. Closing the currently active searcher and opening a new one can be an expensive operation that affects query performance. To minimize impact on query performance, Solr allows you to configure static queries to warm-up a new searcher. Properly managing the new searcher warm-up process is one of the most important configuration tasks you'll need to do for your search application.

Solr also provides a number of important caches that need to be fine-tuned for your application. We looked at the filter, query result, document, and field value caches. For each cache, you need to set a maximum size and eviction policy based on actual usage of your application. The Solr administration console provides key statistics, such as the hit ratio, to help you determine if your caches are sized correctly.

Caches can be auto-warmed when creating a new searcher, which also helps optimize query performance. For example, you can use cache auto-warming to pre-populate Solr's filter cache with the most popular filters used by queries in your application. Cache auto-warming, while powerful, can also lead to large wait times waiting for a new searcher to warm-up. So we advised to start with small **autowarmCount** values and monitor searcher warm-up time closely.

Next, we covered how to control the format of responses using a response writer. The default response writer produces XML, but you can easily request other formats such as JSON using the "wt" query parameter.

We closed the chapter with a quick overview of the Core Admin API. Specifically, we showed you how to request the status for an existing core, create a new core, and reload a core using simple HTTP GET requests.

As we mentioned above, we chose to skip over the index-related settings in solrconfig.xml until we covered the basics of indexing. In the next chapter, you will learn about the Solr indexing process and index-related configuration settings.

5 Indexing

This chapter covers

- Basic concepts of indexing documents
- Designing your schema
- Defining fields and field types in schema.xml
- Field types for structured data like dates and language codes
- Advanced field and field type attributes
- How to index XML, JSON, CSV and other document types
- Update request handling, commits, and atomic updates
- Index management settings in solrconfig.xml

In Chapter 3, we learned how Solr finds documents using an inverted index, which in its simplest form, is a dictionary of terms and a list of documents where each term occurs. Solr uses this index to match terms in a user's query with documents where those terms occur. In this chapter, we learn how Solr processes documents to build the index. A key factor in indexing documents is text analysis. In this chapter we'll focus on the indexing process and non-text fields, saving a detailed discussion of text analysis until Chapter 6.

At the end of this chapter, you'll know how to get documents indexed in Solr and will understand key concepts like fields, field types, and schema design. As a prerequisite, this chapter will be easier to work through if you have the Solr example server running locally, which was covered in chapter 2. On the other hand, you will still be able to follow along with most examples without actually running Solr if you prefer to read this chapter and then come back to doing the hands-on activities another time.

5.1 Example micro-blog search application

Throughout this chapter and the next, we will design and implement an indexing and text analysis solution for searching micro-blog content from popular social media sites like Twitter. We use the term "micro-blog" as a generic term for the short, informal messages and other medium people share with each other on social networks. Examples of micro-blogs are tweets on Twitter, Facebook posts, and check-ins on Foursquare. In this chapter, we define the fields and field types to represent micro-blogs in Solr and learn how to add documents to Solr.

In chapter 6, we learn how to do text analysis on micro-blog content using built-in Solr tools. Let's get started by looking at the type of documents we will be working with in this example and how users might want to search them.

5.1.1 Determine how users will find documents

To begin, table 5.1 shows some fields from a fictitious tweet that we'll use throughout this chapter to learn about indexing documents in Solr. Even if you are not interested in analyzing social media content, the lessons we learn by working through this example have broad applicability for most search applications.

Table 5.1 Fields of a fictitious tweet.

Field	Value
id	1
screen_name	@thelabdude
type	post
timestamp	2012-05-22T09:30:22Z
lang	en
user_id	99991234567890
favourites_count	10
text	#Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach... Learning text analysis with #SolrInAction by @Manning on my i-Pad

Each document in a Solr index is made up of fields, where each field has a specific type that determines how it is stored, searched, and analyzed. In table 5.1, there are eight fields in our micro-blog document[1]. Take a moment to think about how a user might find micro-blogs using these fields. We think the **screen_name**, **type**, **timestamp**, **lang**, and **text**

[1] For a thorough discussion of all the available fields in a tweet, we recommend reading Map of a Tweet: www.slaw.ca/wp-content/uploads/2011/11/map-of-a-tweet-copy.pdf

fields are good candidates from a search perspective because they contain information that a typical user could use to build a query. For example, you can imagine a user wanting to see all English tweets (`lang:en`) from a specific user (`screen_name:thelabdude`) that occurred after a certain date (`timestamp:[2012-05-01T00:00:00Z TO *]`).

Of course you could just index all these fields but if you are developing a large-scale system to support millions of documents and high query volumes, then you only want to include the fields that will actually be searched by your users. For example, the **user_id** field is an internal identifier for Twitter so it's unlikely users will ever want to search on this field. In general, each field increases the size of your index so you should only include fields that add value for users.

The **favourites_count** field is the number of favorites the author of the tweet has, not the number of favorites for the tweet. This field is interesting because it has useful information from a user interface perspective but doesn't seem like a good candidate as a parameter to a search query. We'll address how to handle these display-oriented fields in section 5.2 when we discuss stored vs. indexed fields.

Now, let's think about how users might build a query using these fields, as that will help us decide how to represent these fields in our Solr index. Figure 5.1 depicts a fictitious search form based on the fields for our example micro-blog search application. Each field that we identified as being useful from a search perspective is represented on the form. This is a key point in designing your search application in that you need to think about how users will search a specific field in your index as that will help determine how the field is defined in Solr.

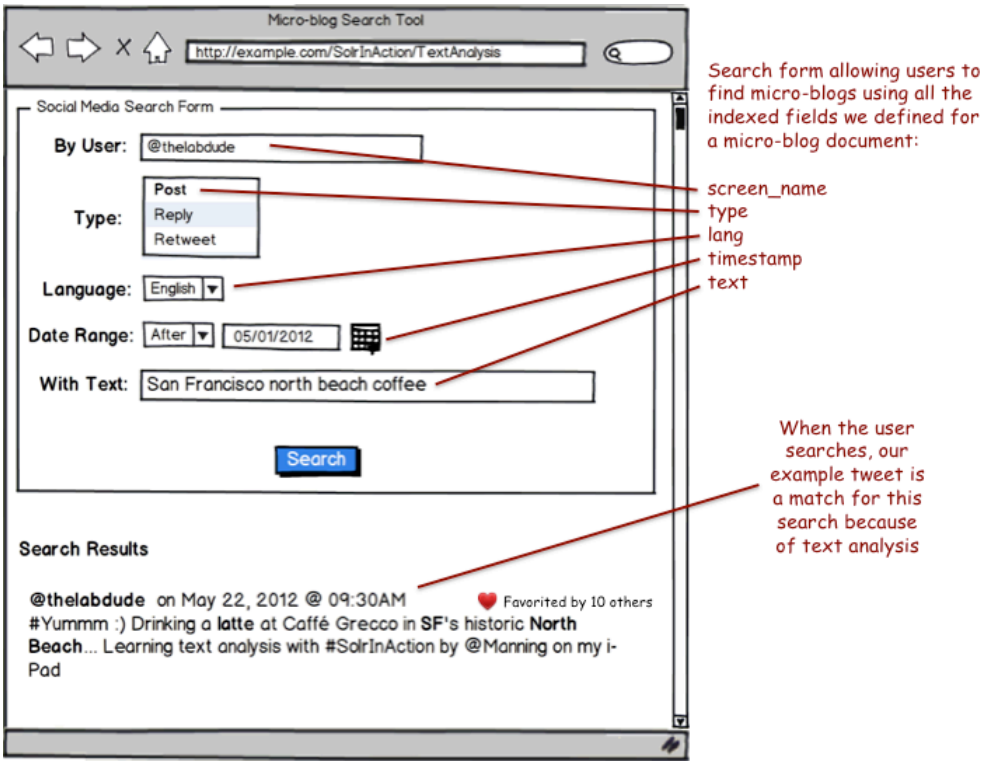


Figure 5.1 Fictitious Web form for finding micro-blogs using the screen_name, type, language, timestamp, and text fields. The favourites_count field is used for displaying results but is not used for searching.

So now we have a conceptual understanding of the fields in our example application and an idea of how users will search for documents using these fields. Next, let's get a high-level understanding of how to add documents to Solr.

5.1.2 Overview of the Solr indexing process

At a high-level, the Solr indexing process distills down to three key tasks:

1. Convert a document from its native format into in a format supported by Solr, such as XML or JSON
2. Add the document to Solr using one of several well-defined interfaces, typically HTTP POST
3. Configure Solr to apply transformations to the text in the document during indexing

Figure 5.2 provides a high-level overview of these three basic steps to getting your document indexed in Solr.

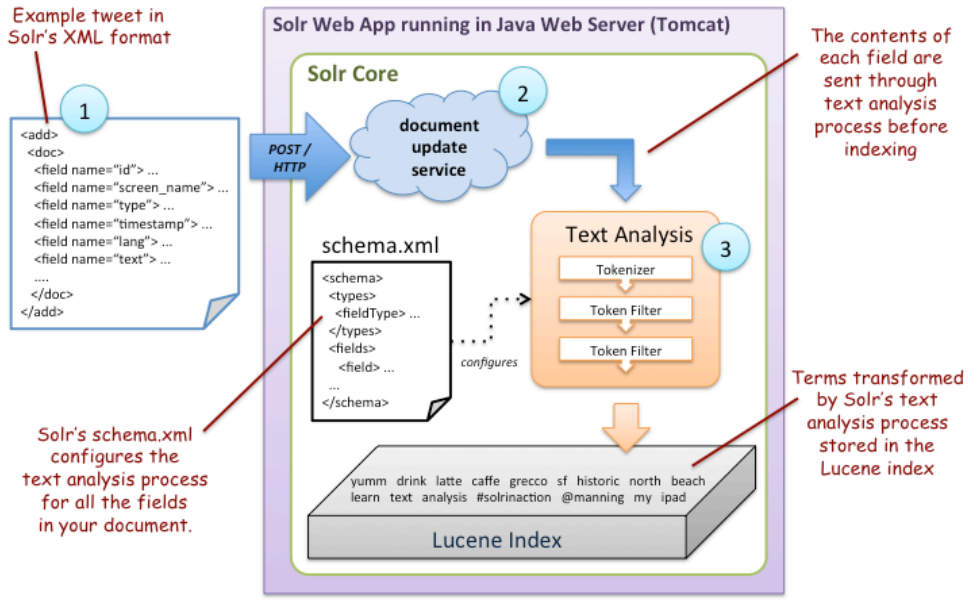


Figure 5.2 High-level overview of three main steps to indexing a document; in step 1, we represent the tweet in an XML format supported by Solr. Next, in step 2, we send the XML document to Solr's document update service using HTTP POST. In step 3, each field is analyzed based on the configuration defined in **schema.xml** before being added to the index.

Solr supports several formats for indexing your document including XML, JSON, and CSV. In Figure 5.2, we chose XML because its self-describing format makes it easy to understand. Here is how our example tweet would look using the Solr XML format:

Listing 5.1 XML document used to index the example tweet in Solr

```
<add>     #A
  <doc>   #B
    <field name="id">1</field>                                  #C
    <field name="screen_name">@thelabdude</field>              #C
    <field name="type">post</field>                            #C
    <field name="timestamp">2012-05-22T09:30:22Z</field>       #C
    <field name="lang">en</field>                              #C
    <field name="user_id">99991234567890</field>               #C
    <field name="favourites_count">10</field>                  #C
    <field name="text">#Yummm :) Drinking a latte at Caffé
    Grecco in SF's historic North Beach... Learning text
    analysis with #SolrInAction by @Manning on my i-Pad</field>  #C
  </doc>
</add>
```

#A Tell Solr we are adding a new document to the index
#B You can add more than one document at a time, each wrapped in a <doc> tag
#C Provide the name and value for each field in the document

Notice that each field from Table 5.1 is represented in the XML and that the syntax is rather simple—you simply define the field name and value for each field! What you don't see is anything about text analysis or field type. This is because you define how fields are analyzed in the **schema.xml** document depicted in the diagram.

Recall from our discussion in Chapter 2 that Solr provides a simple HTTP-based interface to all of its core services, including a document update service for adding and updating documents. At the top left of the diagram in Figure 5.2, we depict sending the XML for our example tweet using an HTTP POST to a document update service in Solr. We'll go into more details about how to add specific document types, such as XML, JSON, and CSV later in the chapter. For now, think of the document update service as an abstract component that validates the contents of each field in a document and then invokes the text analysis process. After each field is analyzed, the resulting text is added to the index, thus making the document available for search.

We will spend more time on how the actual indexing process works in section 5.5 below. A high-level overview of the indexing process is sufficient for now, as we need to focus on more foundational concepts first. Specifically, we need to understand how Solr uses the schema.xml depicted in figure 5.2 to drive the indexing process.

The schema.xml defines the fields and field types for your documents. For simple applications, the fields to search and their types may be obvious. In general, though, it helps to do some up-front planning about your schema.

5.2 Designing your schema

With our example micro-blog search application, we dove right in and defined what a document is and which fields we want to index. In practice, this process is not always obvious for a real application, so it helps to do some up-front design and planning work. In this section, we learn about some key design considerations for search applications. Specifically, we'll learn to answer the following key questions about your search application:

1. What is a document in your index?
2. How is each document uniquely identified?
3. What are the fields in your document that can be searched by users?
4. Which fields should be displayed to users in the search results?

Let's begin by determining the appropriate granularity of a document in your search application as that impacts how you answer the other questions.

5.2.1 Document granularity

Determining what is a document in your Solr index drives the entire schema design process. In some cases it is obvious, such as with our tweet example, where the text content is typically short, so each tweet will be a document. However, if the content you want to index is very large, such as a technical computer book, you may want to treat sub-sections of a large document as the indexed unit. The key is to think about what your users will want to

see in the search results. Let's look at a different example to help you think about what is a document for your index.

Imagine searching for "text analysis" on Web site that sells technical computer books. If the site treated each book as a single document, then the user would see ***Solr in Action*** in the search results, but would then need to page through the table of contents or index to find specific places where "text analysis" occurs in the book. In Figure 5.3, the left-side image depicts how search results might look when an entire book is indexed as a single document.

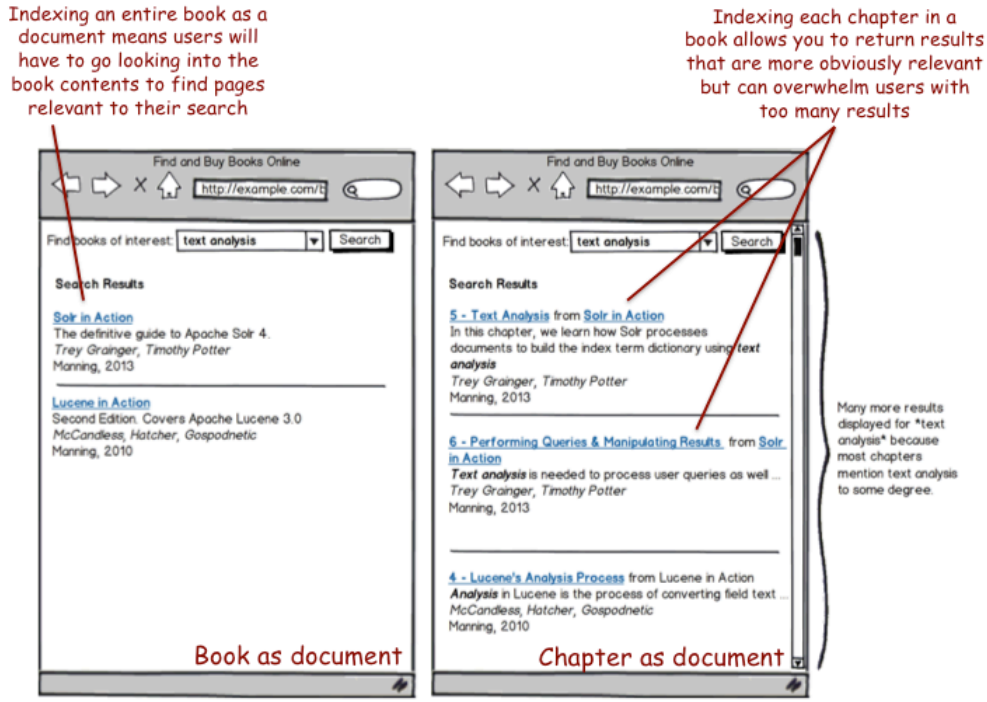


Figure 5.3 Comparison of search results when indexing entire book as document vs. indexing individual chapters as documents.

On the other hand, if the site treated individual chapters in each book as documents in the index, then the search results might show the user the **Text Analysis** chapter in ***Solr in Action*** as the top result, as seen on the right in Figure 5.3. However, since text analysis is a central concept in search, most of the other chapters in this book and other search books would be included as highly relevant results as well. So being too granular can overwhelm users with too many results to wade through.

You may also need to consider the type of content you are indexing, as splitting a technical computer book by chapter seems to make sense but splitting a fiction novel by chapter doesn't seem like a good approach. In the end, it's your choice on what makes a document in your index, but definitely consider how document granularity impacts user experience. In general, you want your documents to be as granular as possible without causing your users to miss the forest for the trees.

Solr Hit Highlighting

As a quick aside, we should note that Solr offers a feature called **hit highlighting** that allows you to highlight relevant sections of longer documents in search results. This is useful when you cannot break long documents up into smaller units but still want to help your users quickly navigate to highly relevant sections in large documents. For example, we could use hit highlighting to show the first paragraph of Chapter 6 in this book in the search results of a query for "text analysis". We discuss hit highlighting in depth in Chapter 9.

5.2.2 Unique key

Once you've identified what a document is, you *should* determine how to uniquely identify each document in the index. For a book, this might be the ISBN number. For a chapter, it might be the ISBN number plus the chapter number. Solr does not require a unique identifier for each document but if supplied, Solr uses it to avoid duplicating documents in your index. For those with a database background, the unique identifier is similar to a primary key for a row in a table. If a document with the same unique key is added to the index, then Solr overwrites the existing record with the latest document. We'll return to the discussion of unique keys when we discuss distributed search later in the book.

For our example micro-blog search application from section 5.1, the tweet already includes a unique identifier field: **id**. However, if we index content from a variety of social media sources, then we would probably add something like "twitter:" as a prefix to differentiate this document from a Facebook post with the same numeric id value**.**

5.2.3 Indexed fields

Once you have determined what a document is for your index and how to uniquely identify each document, the next step is to determine the ***indexed*** fields in the document. The best way to think about indexed fields is to ask whether a typical user could develop a meaningful query using that field? Another way to decide if a field should be indexed is if you removed this field from the search form would your users miss it?

For example, every book has a title and an author. When searching for books, people generally expect to find books of interest based on title and author, so these fields should be indexed. On the other hand, although every book has an editor, users typically do not search

by editor name when trying to find a book to read. Thus, editor name would need to not be an indexed field. Conversely, if you were building a search index for the book publishing industry, then it's very likely that your users would want to search by editor name so you would include that as an indexed field.

Determining which fields to include in the index is specific to every search application. Take a moment to think about the indexed fields for your documents. Keep these fresh in your mind, as you'll need to refer to them as you work through the rest of this chapter. As we already discussed, the **screen_name**, **type**, **timestamp**, **lang**, and **text** should be indexed for our micro-blog example. The **id** and **user_id** fields are used internally by Twitter and are unlikely to be missed if you don't allow users to search by these fields.

5.2.4 Stored fields

Although users probably won't search by editor name to find a book to read, we may still want to display the editor's name in the search results. In general, your documents may contain fields that are not very useful from a search perspective but are still useful for displaying search results. In Solr, these are call "stored" fields. The **favourites_count** field is a good example of a stored field that is not indexed but is useful for display purposes. You can imagine users would find it useful to see which authors have more favorites than others in search results but it's unlikely that users would want to search by this field. In addition to displaying stored fields, you can also sort by them, such as to see tweets from authors with more favorites at the top of search results. Of course a field may be indexed and stored, such as the **screen_name**, **timestamp**, and **text** fields in our micro-blog search application. Each of these fields can be searched and displayed in results.

As a search application architect, one of your goals should be to minimize the size of your index. If you're considering Solr, then most likely you have an application that needs to scale to handle large volumes of documents and users. Each stored field in your index consumes disk space and requires CPU and I/O resources to read the stored value for returning in search results. Thus, you should choose your stored fields wisely especially for large-scale applications.

At this point you should have a good idea about the types of questions you need to answer to design your search application. Once you've settled on a plan, it's time to rollup your sleeves and work with Solr's **schema.xml** to implement your design. As we saw in figure 5.2, schema.xml is the main configuration document Solr uses to understand how to index your documents. Let's take a quick preview of the main sections of schema.xml so we have an idea of what's in-store for us over the next couple of sections in this chapter.

5.2.5 Preview of schema.xml

In the next few sections, we build a valid **schema.xml** document for our example micro-blog search application. The schema.xml file is in the conf directory for your Solr core. For instance, the schema.xml for the example Solr server is in:

`$SOLR_INSTALL/example/solr/collection1/conf`. Listing 5.2 is a condensed version of the example schema.xml provided with Solr to give a feel for the XML syntax and important elements[2].

Listing 5.2 Major sections of Solr's schema.xml document

```
<schema name="example"       #A
 version="1.5">              #B
  <fields>                   #C
   <field name="id" type="string" indexed="true" stored="true" .../>
   <field name="name" type="text_general" indexed="true" stored="true"/>
   <field name="cat" type="string" indexed="true" stored="true" .../>
   ...
   <dynamicField name="*_s" type="string" indexed="true" stored="true"/>
   <dynamicField name="*_t" type="text_general" indexed="true"  .../>
   ...
 </fields>
 <uniqueKey>id</uniqueKey>              #D
 <copyField source="cat" dest="text"/>  #E
 ...
 <copyField source="manu" dest="text"/> #E
  <types>                               #F
    <fieldType name="string" class="solr.StrField" .../>
    <fieldType name="boolean" class="solr.BoolField" .../>
    ...
    <fieldType name="tint" class="solr.TrieIntField" .../>
    <fieldType name="tfloat" class="solr.TrieFloatField" .../>
    <fieldType name="text_general" class="solr.TextField">     #G
      <analyzer type="index">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" .../>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" .../>
        <filter class="solr.SynonymFilterFactory" .../>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
    </fieldType>
    ...
 </types>
</schema>
```

#A schema name is only used for display purposes
#B version is used internally by Solr to enable specific features
#C field elements represent the fields in your documents
#D define which field to use to uniquely identify each document
#E copy field directive
#F fieldType elements govern if and how a field is analyzed
#G fieldType useful for analyzing general text

[2] To see the full example schema, click on the **[SCHEMA]** link from the Solr administration page, which loads the schema.xml into your browser.

At a quick glance, it's easy to be overwhelmed by all the details in this document. By the end of this chapter, you'll have a clear understanding of all these details and will be well-equipped to craft your own schema.xml. For now, notice that there are three main sections of the schema.xml document:

1. The <fields> element contains <field> and <dynamicField> elements used to define the basic structure of your documents
2. Miscellaneous elements, such as <uniqueKey> and <copyField> are listed after the <fields> element
3. Field Types under the <types> element determine how dates, numbers, and text fields are handled in Solr

We'll work through each of these main sections below. Let's begin by looking at the <fields> section.

5.3 Defining fields in schema.xml

The <fields> section in schema.xml defines <field> elements for all fields in your document. Solr uses the field definitions from schema.xml to build the internal structure of the index for your documents. In this section, we learn how to define fields, dynamic fields, and copy fields in schema.xml. Since we already worked through the main concepts behind schema design in section 5.2, we're ready to define the <field> elements for our example micro-blog search application. Listing 5.3 defines the indexed and stored fields for our example application.

Listing 5.3 Field elements for our example micro-blog search application

```
<schema name="example" version="1.5">
  <fields>  #A
   <field name="id" type="string" indexed="true" stored="true"
          required="true"/>  #B
   <field name="screen_name" type="string" indexed="true" stored="true"/> #C
   <field name="type" type="string" indexed="true" stored="true"/>
   <field name="timestamp" type="tdate" indexed="true" stored="true"/>
   <field name="lang" type="string" indexed="true" stored="true"/>
   <field name="favourites_count" type="int"
          indexed="false" stored="true"/> #D
   <field name="text" type="text_microblog" #E
          indexed="true" stored="true"/>
    ...
  </fields>
  ...
</schema>
```

#A Define the fields of our example micro-blog application under the <fields> element
#B Field used to uniquely identify documents in the index
#C Fields must define a name and type and whether they are indexed and/or stored
#D Field is not indexed but is stored for display purposes
#E Text field will be analyzed using a field type called "text_microblog"

With these field definitions, Solr knows how to index micro-blog documents so they can be searched using a form similar to figure 5.1. When defining a field in schema.xml, there are a few required attributes you must provide to Solr.

5.3.1 Required field attributes

Each field has a unique name that is used when constructing queries. For instance, the query `screen_name:thelabdude` searches the "screen_name" field for value "thelabdude". In listing 5.3, we defined the **screen_name** field as:

```
<field name="screen_name"
       type="string"
       indexed="true"
       stored="true" />
```

In layman's terms, this definition means that the **screen_name** field is indexed and stored and contains "string" values. Each field must define a **type** attribute that identifies the `<fieldType>` to use for that field; we cover field types in detail in the next section. Each field must also define whether it is indexed and/or stored. As we discussed in section 5.2, indexed fields can be searched and stored fields can be returned in search results for display and sorting purposes. Of course, a field can be both, as is the case for most of the fields in our example.

Also, notice that there are no "nested" fields in Solr; all fields are siblings of each other in schema.xml implying a flat document structure. As we discussed in Chapter 3, documents in Solr need to be de-normalized into a flat structure and must contain all the fields needed to support your search requirements. In other words, there is no relational structure that allows you to join with other documents to pull in additional information to service queries or generate results.

Joins in Solr

You'll undoubtedly encounter content on the Web about doing document joins in Solr. We'll address this functionality in detail in Chapter 15. For now, it's important to understand that Solr joins are more like sub-queries in SQL than joins. A typical use case is to find the parent documents of child documents that match your search criteria using Solr joins. For example, in our example micro-blog application, we could use Solr joins to bring back the original post instead of re-tweets.

One thing that can be confusing is that when a field is stored, Solr stores the original value and not the analyzed value. For example, in listing 5.3, we declared the **text** field with `indexed="true"` and `stored="true"`. This means that the text field will be searchable and you can return the original text in search results. You cannot return the analyzed value in search results. Of course, if you don't return a field in search results, then it doesn't need to be stored.

There's also a case to be made for storing all fields for a document. If you plan to update documents after they are indexed, then you will need to store all fields. We'll learn more about updating documents in section 5.6.3 later in the chapter.

5.3.2 Multi-valued Fields

So far, our micro-blog search application only uses a small number of simple fields. Let's add a few more fields into the mix to exercise some of Solr's strengths in dealing with more complex document structures. Specifically, let's add support for a "links" field that contains zero or more links associated with each document. As you've probably seen on Twitter, users can embed links to related content, such as a photo or article on the Web. Here's an example of another fictitious tweet with two links embedded:

```
Just downloaded the MEAP for #SolrInAction from @Manning
http://bit.ly/15tzw to learn more about #Solr http://bit.ly/3ynriE
```

The links in this document are shortened URLs provided by bitly.com that resolve to the following Web sites shown in table 5.2:

Table 5.2 Actual URLs for shortened links in the example tweet

Shortened URL	Actual URL
http://bit.ly/15tzw	http://manning.com/
http://bit.ly/3ynriE	http://lucene.apache.org/solr/

From a search perspective, adding resolved links to your index allows users to find social media content that links to a specific Web site or page. For example, you can imagine users wanting to find all tweets that link to the *Solr In Action* MEAP page on manning.com. Since this example contains two links, we need a way to encode two values for one field. In Solr, fields that can have more than one value per document are called multi-valued fields. In schema.xml, you declare a multi-valued field by setting `multiValued="true"` on the field definition:

```
<field name="link"
       type="string"
       indexed="true"
       stored="true"
       multiValued="true"/>
```

When adding a document that has multiple links, you simply add multiple "link" fields in the XML document, as is depicted in listing 5.4.

Listing 5.4 Representing multi-valued fields in XML during indexing

```
<add>
  <doc>
    <field name="id">2</field>
    ...
    <field name="link">http://manning.com/</field>            #A
    <field name="link">http://lucene.apache.org/solr/</field> #A
    ...
```

```
  </doc>
</add>
```

#A Reuse the same field name to populate a multi-valued field during indexing

When searching, you can query for `link:"http://manning.com/"` and Solr will look for matches across all values in a multi-valued field.

So far, our micro-blog documents have only a small number of fields, which made it easy to declare each field separately in schema.xml. In practice, not all documents are so simple or sparse. Let's look at another type of field, called dynamic fields, that help deal with larger and more complex document structures.

5.3.3 Dynamic Fields

In Solr, dynamic fields allow you to apply the same definition to any fields in your documents whose name matches either a prefix or suffix pattern, such as s_* or *_s. In other words, dynamic fields uses a special naming scheme to apply the same field definition to any fields that match a glob-style pattern. Dynamic fields help address a few common problems that occur when building search applications, including:

1. Documents with many fields that share the same field definition
2. Documents from a variety of diverse sources
3. Adding new data sources with new unique fields to index

Let's look at each one of these use cases to get a good feel for what you can do with dynamic fields. However, you should note that you do not need to use dynamic fields with Solr. It's perfectly acceptable to not use dynamic fields if none of these use cases applies to your application.

Also, Solr ignores the dynamic field definitions in your schema.xml until you start indexing documents that make use of them. Thus, in practice, most Solr users keep the extensive list of dynamic fields provided with the Solr example schema so they are there when you need them, but are simply ignored otherwise.

MODELING DOCUMENTS WITH MANY FIELDS

Dynamic fields help you model documents having many fields by allowing you to match a prefix or suffix pattern to apply the same field definition in schema.xml to any matching fields. For example, in listing 5.3 we used the "string" field type for **type**, **screen_name**, and **language** fields. Moreover, each of these fields are both stored and indexed. That is, other than the name, each field definition is exactly the same.

Now imagine that in addition to these three fields, we have dozens of "string" fields that are also stored and indexed. Of course you can type in an explicit definition for each field or you can define a single `<dynamicField>` element to account for all of these string fields using a suffix pattern on the field name:

```
<dynamicField name="*_s" type="string" indexed="true" stored="true" />
```

With this glob pattern, any field with the name ending in "_s" will inherit this field definition, such as **subject_s**. Alternatively, you could use a prefix pattern "s_*". So

dynamic fields help save some typing and simplify your schema.xml when you have many fields.

You can also use dynamic fields for multi-valued fields, such as our links field in the previous section. The following dynamic field definition has multivalued="true":

```
<dynamicField name="*_ss" type="string" indexed="true" stored="true"
              multiValued="true"/>
```

For multi-valued links, your XML document would need to use **link_ss** as the field name for multiple links, as seen in listing 5.5.

Listing 5.5 Using dynamic fields to represent multi-valued fields during indexing

```
<add>
  <doc>
    <field name="id">9999012345679</field>
    ...
    <field name="link_ss">http://manning.com/</field>            #A
    <field name="link_ss">http://lucene.apache.org/solr/</field> #A
    ...
  </doc>
</add>
```

#A Including multiple links using dynamic field naming

SUPPORTING DOCUMENTS FROM DIVERSE SOURCES

Another benefit of dynamic fields is that they help you support a mixture of documents that share a common base schema, but also have some unique fields. Of course, if your documents don't have a common base schema, then they should probably not be in the same index! In our social media example, if we index documents from Twitter, Facebook, YouTube, and Google+, then documents from each of these sources will have some fields that are unique to each social network. We think it's more intuitive and maintainable to handle these source-specific fields as dynamic fields. For example, instead of defining many fields for each source as in the example below:

```
<field name="facebook_f1" type="string" indexed="true" stored="true" /> #A
<field name="facebook_f2" type="string" indexed="true" stored="true" /> #A
<field name="facebook_fn" type="string" indexed="true" stored="true" /> #A
...
<field name="twitter_f1" type="string" indexed="true" stored="true" /> #B
<field name="twitter_f2" type="string" indexed="true" stored="true" /> #B
<field name="twitter_fn" type="string" indexed="true" stored="true" /> #B
```

#A Many Facebook specific string fields that are stored and indexed
#B Many Twitter specific string fields that are stored and indexed

You can accomplish the sample using a single "string" dynamic field with the "*_s" suffix pattern as the name:

```
<dynamicField name="*_s" type="string" indexed="true" stored="true" />
```

When indexing, you need to send fields with the "_s" suffix, such as:

Listing 5.6 Include source-specific fields during indexing using dynamic fields

```
<add>
  <doc>
    <field name="id">9999012345678</field>
```

```
      <field name="screen_name">@thelabdude</field>
      <field name="type">post</field>
      ...
      <field name="facebook_f1_s">hello</field>  #A
      <field name="facebook_f2_s">world</field>  #A
      <field name="twitter_f1_s">foo</field>  #A
      <field name="twitter_f2_s">bar</field>  #A
    </doc>
  </add>
```

#A Matches the "*_s" <dynamicField> definition, which is a string field

ADDING NEW DOCUMENT SOURCES

If you add a new data source for your application that has fields you haven't encountered before, then you can just include them during indexing and they will be picked up automatically. New social networks seem to come online everyday so we wouldn't want to re-work our schema.xml just to handle documents from these new sources. With dynamic fields, you can include new fields introduced by your new document source without making any changes to the schema.xml.

For example, suppose you wanted to add support for a new social network that includes a field that captures the phase of the moon when the content was posted to the network (perhaps it's a dating site). With dynamic fields, you can just include this field as a string in your documents by doing: `<field name="moon_phase_s">waxing crescent</field>`.

Lastly, although dynamic fields can be a handy feature on the indexing side, there's no real magic on the query side. When querying for documents that were indexed with dynamic fields, you must use the full field name in the query. In other words, you cannot formulate queries to find a match in all string fields by querying with a prefix or suffix pattern: `*_s:coffee`. Rather, you need to explicitly identify which string fields to query, such as: `subject_s:coffee keyword_s:coffee`, etc. However, if you want to find matches in more than one field, dynamic or static, then Solr provides a clever way to do that with copy fields.

5.3.4 Copy Fields

In Solr, copy fields allow you to populate one field from one or more other fields. Specifically, copy fields support two use cases that are common in most search applications:

1) Populate a single "catch-all" field with the contents of multiple fields
2) Apply different text analysis to the same field to create a new searchable field

CREATE A CATCH-ALL FIELD FROM MANY FIELDS

In most search applications, users are presented with a single "search" box to enter a query. The intent of this approach is to help your users find documents quickly without having to fill-out a complicated form; think about how successful a simple search box has been for Google. In our tweet example, you might think it's an easy decision—just search the tweet text and you're done. However, with this approach, users would not find our example tweet if they searched for "@thelabdude" because that information is contained in the

screen_name field and not the **text**. In addition, if a tweet contains shortened bit.ly-style URLs, then searches in the text field for the actual "resolved" URL will not match as those are stored in the links field. What we really want here is a *catch-all* search field that contains the text from the **screen_name**, **text**, and resolved **link** fields. Thankfully, Solr makes it easy to create a single catch-all search field from many other fields in your document using the `<copyField>` directive.

First, you need to define a destination field that other fields will be copied into—let's name this field **catch_all**:

```
<field name="catch_all"
       type="text_en"
       indexed="true"
       stored="false"          #A
       multiValued="true"/>    #B
```

#A Catch all field should not be stored as it is populated from another field
#B Destination field must be multivalued if any of the source fields are multi-valued

This looks like any other field except there are two important aspects to the definition. First, notice that this field is not stored (`stored="false"`), which makes sense because we probably don't want to display a blob of many fields concatenated together to our users. In fact, even if you wanted to do this, you can't because there really is no original value for Solr to return for copy fields. Remember that Solr returns the original value for a stored field.

Second, if any of the source fields are multi-valued, then the destination field must be a multi-valued field (`multiValued="true"`). In our case, the link field is multi-valued, so we must define our copy field as multi-valued as well.

Now that we've defined the destination field, we need to tell Solr which fields to copy from using the `<copyField>` directive. Listing 5.7 shows how we would copy the values from the **screen_name**, **text**, and **link** fields into our **catch_all** field using Solr's copy field directive:

Listing 5.7 Using the copyField directive to populate the catch_all field

```
<schema>
  <fields>
    ...
  </fields>
  <copyField source="screen_name" dest="catch_all" /> #A
  <copyField source="text" dest="catch_all" />        #A
  <copyField source="link" dest="catch_all" />        #A
  <types>
    ...
  </types>
</schema>
```

#A raw contents of screen_name, text, and link fields copied into catch_all

Note that the `<copyField>` element is a sibling of the `<fields>` and `<types>` elements in schema.xml. Take a moment to think about why this is the case. The best way to make sense of this is that you must define the source and destination fields first and then you connect them with the copy field directive after all fields have been defined.

APPLY DIFFERENT ANALYZERS TO A FIELD

You also may want to analyze the contents of a single field differently. For instance, as we'll see in chapter 6, stemming is a technique that transforms terms into a common base form, known as a "stem", in order to improve recall. With stemming, the terms "fishing", "fished" and "fishes" all have a common stem of "fish". Thus, stemming can help your users find documents without having to think about all the possible linguistic forms of a word, which is a good approach for general text search fields.

On the other hand, consider how stemming would affect a type-ahead suggestion box (auto-suggest). In this case, stemming would work against your users in that you could only suggest the stemmed values and not the full terms. For example, with stemming enabled, your search application would not be able to suggest "humane" or "humanities" when the user started typing "human" in the auto-suggest box. Solr copy fields give you the flexibility to enable or disable certain text analysis features like stemming without having to duplicate storage in your index. Consider the following snippets from schema.xml:

Listing 5.8 Using copyField to apply different text analysis to the same text

```
<field name="text"
       type="stemmed_text"
       indexed="true"
       stored="true"/>

<field name="auto_suggest"
       type="unstemmed_text"
       indexed="true"
       stored="false"
       multiValued="true"/>
 ...
<copyField source="text" dest="auto_suggest" /> #A
```

#A raw text content from the text field copied into the auto_suggest field to be analyzed using the unstemmed_text field type

In this case, the **text** field has field type "stemmed_text" which presumably means the text is stemmed. The **auto_suggest** field is not stemmed. We use the `<copyField>` directive to populate the auto_suggest field with unstemmed text from the text field. Under the covers, Solr sends the raw, un-analyzed contents of the text field to the auto_suggest field, which allows a different text analysis strategy. The original text value is only stored once. To reiterate, you cannot return the original value of the auto_suggest field in search results so it has `stored="false"`.

5.3.5 Unique Key Field

In section 5.2.2, we discussed how it is a good idea to make your documents uniquely identifiable in the index using some unique ID value. To recap, if you provide a unique identifier field for each of your documents, then Solr will avoid creating duplicates during indexing. In addition, if you plan to distribute your Solr index across multiple servers, then you must provide a unique identifier for your documents. For these reasons, we recommend defining a unique identifier for your documents from the start. In our micro-blog example,

the "id" field is unique so we configure Solr to use that field as the unique key for documents using the <uniqueKey> element in schema.xml:

Listing 5.9 The <uniqueKey> element identifies the unique ID field for documents

```
<uniqueKey>id</uniqueKey>
```

One thing to note is that it's best to use a primitive field type, such as "string" or "long", for the field you indicate as being the <uniqueKey/> as that ensures Solr does not make any changes to the value during indexing. In fact, we've seen instances where Solr does not return results correctly if you don't use "string" as the type for text-based keys. So save yourself some trouble and just use string or one of the other primitive field types for your unique key field.

At this point, we've covered all the basic aspects of defining fields in schema.xml. You should also have a good understanding of when to use multi-valued, dynamic, and copy fields. It's now time to dig into the next major section of Solr's schema.xml to learn how to define field types.

5.4 Field Types for Structured "Non-Text" Fields

In this section, we learn to define field types for handling structured data like dates, language codes, and usernames. In chapter 6, we learn how to define field types for text fields like the body text of our example tweet. In general, Solr provides a number of built-in field types for structured data, such as numbers, dates, and geo-location fields. Figure 5.4 shows a class diagram of some of the more commonly used field types in Solr.

Commonly Used Field Type classes from the org.apache.solr.schema package:

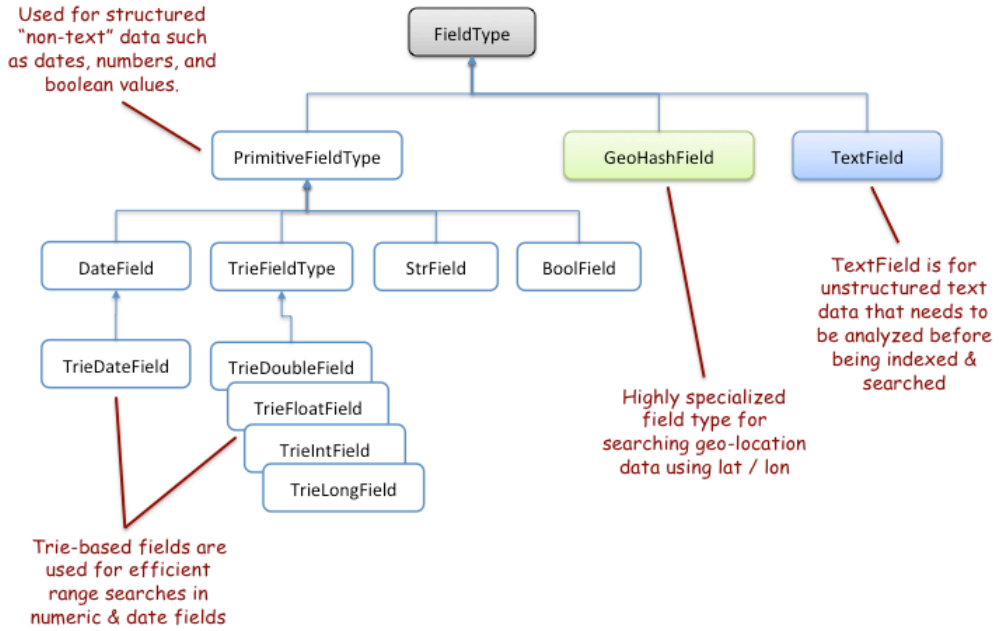


Figure 5.4 Class diagram of commonly used field types from the org.apache.solr.schema Java package.

Let's begin our discussion of field types for non-text data by looking at one of the most common field types: string.

5.4.1 String Fields

For our example tweet, in addition to the text field, we decided the **screen_name**, **type**, **timestamp**, and **lang** should also be indexed fields. Now we need to decide the appropriate type for each field. It turns out that each of these fields contains structured data that does not need to be analyzed. For example, the **lang** field contains a standard ISO-639-1 language code used to identify the language of the tweet, such as "en". Users can query the **lang** field to find English tweets as shown in Figure 5.5.

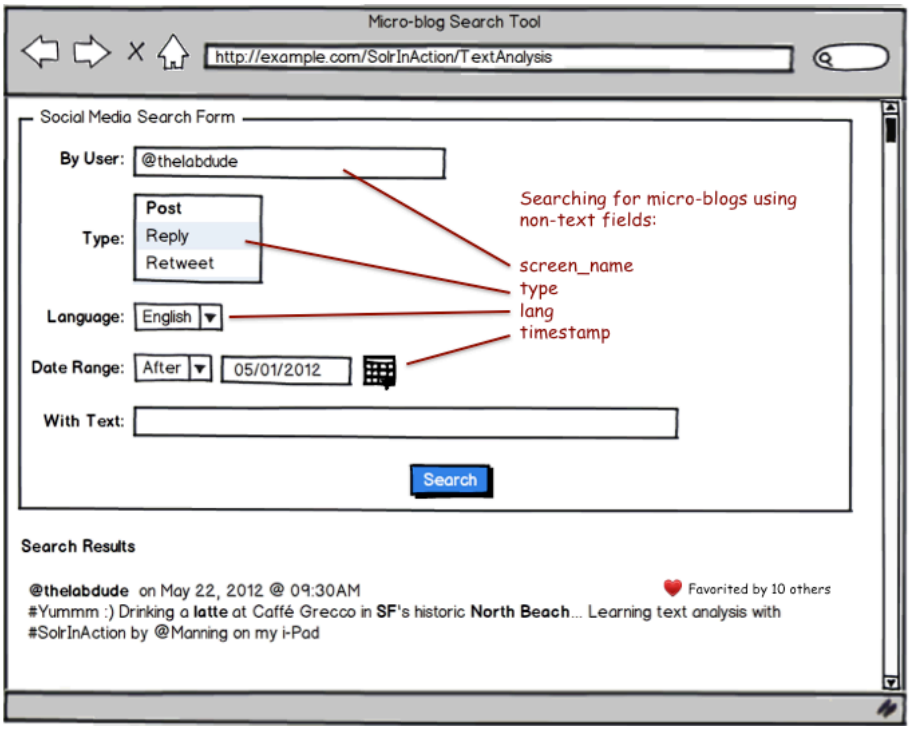


Figure 5.5 Mirco-blog search application Web form used to find documents using structured "non-text" fields.

Since the language code is already standardized, we don't want Solr to make any changes to it during indexing and query processing. Solr provides the **string** field type for fields that contain structured values that should not be altered in any way. Here is how the **string** field type is defined in schema.xml:

Listing 5.10 Field type definition for string fields in schema.xml

```
<fieldType name="string" class="solr.StrField"
      sortMissingLast="true" omitNorms="true"/>
```

Behind the scenes, all field types are implemented by a Java class, in this case "solr.StrField". At runtime, "solr.StrField" resolves to the built-in Solr class: `org.apache.solr.schema.StrField`. Anytime you see "solr." as a prefix of a class in schema.xml you know this translates to the fully qualified Java package: `org.apache.solr.schema.` This shorthand notation helps reduce clutter in your schema.xml. The sortMissingLast and omitNorms attributes are advanced options that we'll discuss in more detail in section 5.4.4.

If we use the string field type for our **lang** field, Solr will take the value "en" from our document and store it in the index unaltered as "en" during indexing. At query time, you also need to pass the exact value "en" to match English documents. In Figure 5.5, the user selected "English" which needs to be translated into "en" when processing the form. The **string** `<fieldType>` also seems like a good type for the **screen_name** and **type** fields, but what about **timestamp**?

5.4.2 Date Fields

A common approach to searching on date fields is to allow users to specify a date range. For example, in figure 5.5, the user searched for micro-blogs that occurred after a specified date (05-01-2012). On the query side, this would be a range query on the timestamp field:

```
timestamp:[2012-05-01T00:00:00Z TO *]
```

Since searching within a date range is such a common use case, Solr provides an optimized built-in `<fieldType>` called **tdate**:

Listing 5.11 Field type definition for date fields in schema.xml

```
<fieldType name="tdate" class="solr.TrieDateField" omitNorms="true"
           precisionStep="6" positionIncrementGap="0"/>
```

Admittedly, this looks a little scarier than the **string** type! The additional attributes like **precisionStep** and **positionIncrementGap** are advanced options that we'll address later in section 5.4.4. Again, the `solr.TrieDateField` is Solr's short-hand notation for specifying the Java class name that implements this field type, which in this case is: `org.apache.solr.schema.TrieDateField`. A "trie" is an advanced tree-based data structure that allows for efficient searching for numeric and date values by varying degrees of precision.

During indexing, Solr needs to know how to parse a date. Recall from section 5.1, the example XML document we sent to Solr for indexing included the timestamp field as:

```
<add>
  <doc>
    ...
    <field name="timestamp">2012-05-22T09:30:22Z</field>
    ...
  </doc>
</add>
```

In schema.xml, the timestamp field is configured to use the **tdate** type:

```
<field name="timestamp" type="tdate" indexed="true" stored="true" />
```

In general, Solr expects your dates to be in the ISO-8601 DateTime format (yyyy-MM-ddTHH:mm:ssZ); the date in our tweet (`2012-05-22T09:30:22Z`) breaks down to:

```
yyyy = 2012
  MM = 05
  dd = 22
  HH = 09 (24-hr clock)
  mm = 30
  ss = 22
   Z = UTC Timezone (Z is for Zulu)
```

If you send Solr a date in some other format, you will get a validation error during indexing and the document will be rejected.

DATE GRANULARITY

Next you need to decide the granularity of dates in your index. This goes back to understanding how your users need to query using dates. For example, if your users only expect to query for documents by day, then there's no point in indexing a date with second or millisecond precision. On the other hand, if you need to sort documents by date, then hour-level granularity may be too coarse, in which case you may want to do minute-level granularity.

During indexing, Solr supports Date Math operations to help you achieve the correct precision for a date field with little effort. For example, let's say you decide that you only need to index your dates at the hour-level of granularity. This saves space in your index, but also means that users cannot get more specific than hour ranges when searching. When indexing, you can send your date with `/HOUR` on the end; the / tells Solr to "round-down" to a specific granularity. Let's see how we can index our example tweet with hour granularity:

```
<field name="timestamp">2012-05-22T09:30:22Z/HOUR</field>
```

In the index, the value of the timestamp field for the example document will be equivalent to: `2012-05-22T09:00:00Z`. In addition to specifying the date/time exactly, Solr also supports the **NOW** keyword to represent the current system time on your Solr server. You can combine specific dates or the NOW keyword with Solr's Date Math operations to accomplish very powerful date calculations. For example, `NOW/DAY` evaluates to midnight of the current day and `NOW/DAY+1DAY` evaluates to midnight tomorrow. To query for all documents from today, you could do: `timestamp:[NOW/DAY TO NOW/DAY+1DAY}`. We'll dig more into the details of Solr's Date Math in chapter 7 when we discuss range queries.

Lastly, it should be noted that the **tdate** field is a good choice for fields that you need to do date range queries on but it comes at the cost of requiring more space in your index because more tokens are stored per date value. According to the Solr JavaDocs, a `precisionStep="6"` is a good value for long fields which is how Solr stores dates in the index. We dive into the details of how to choose the right precisionStep in section 5.4.4 below.

5.4.3 Numeric Fields

For the most part, numeric fields behave as you would expect in Solr. For example, in section 5.1, we previously discussed how the **favourites_count** field indicates the number of times the author of a tweet has been favorited by other users. This is not an intuitive field to search by but is useful from a display and sorting perspective. In other words, you can imagine users wanting to sort matching tweets by this field to see content from more popular authors. In schema.xml we declared the field as:

```
<field name="favourites_count" type="int" indexed="false" stored="true" />
```

The "int" field type is defined as:

```
<fieldType name="int" class="solr.TrieIntField"
           precisionStep="0" positionIncrementGap="0"/>
```

Because we don't need to support range queries on this field, we chose `precisionStep="0"` which works best for sorting without incurring the additional storage costs associated with a higher precision step used for faster range queries. Also, note that you shouldn't index a numeric field that you need to sort by as a string field because Solr will do a lexical sort instead of a numeric sort if the underlying type is string-based. In other words, if you index numeric fields using a string-based field type, then sorting will return results like 1, 10, 2, 3, … instead of 1, 2, 3, …, 10.

Up to this point, we've discussed the main concepts for indexing fields that contain structured information. We'll return to specific cases for these types of non-text fields in later chapters. For example, we'll discuss a `<fieldType>` used to represent latitude and longitude when we discuss Solr's geo-spatial search in Chapter 15. For now, let's wrap-up this section with a short discussion of some of the advanced configuration options for field types.

5.4.4 Advanced FieldType Attributes

Solr supports optional attributes for field types to enable advanced behavior. Table 5.4 covers advanced attributes for `<fieldType>` elements.

Table 5.4 Overview of advanced attributes for field type elements in schema.xml

Attribute	Behavior when Enabled (="true")
sortMissingFirst	When sorting results, Solr will list documents that do not have a value for the field at the top of the results.
sortMissingLast	When sorting results, Solr will list documents that do not have a value for the field at the bottom of the results.
precisionStep	Determines the number of terms created to represent a numeric value in the index for doing fast range queries on Trie-based fields like TrieDate and TrieLong; see JavaDoc for the NumericRangeQuery class for more details about the precisionStep attribute.
positionIncrementGap	Used to prevent phrase queries from matching the end of one value and the beginning of the next value in multi-valued fields.

Let's take a closer look at **precisionStep** as that is a common source of confusion for new Solr users. You can safely skip the following discussion and come back to this after you have your search application implemented and are looking for ways to improve performance of sorting and range queries.

CHOOSING THE BEST PRECISIONSTEP FOR NUMERIC FIELDS

Two common use-cases you'll undoubtedly encounter is finding documents that match a range of values for a numeric or date field, called a range query, and sorting results by numeric and date fields. As we discussed in section 5.4.3, Solr uses a **trie** data structure to support efficient range queries and sorting of numeric and date values. Let's learn how to choose the best value for **precisionStep** to support range queries and sorting in your Solr instance.

First, decide if you even need to worry about precisionStep by asking whether you have any numeric or date fields in your index that users would like to find documents across a range of values in those fields. For each of these fields, think about the range of possible values that will be indexed—are there potentially millions of unique values or just a handful? In Solr terminology, the number of unique values in a field as called the "cardinality" of the field.

For example, consider a Solr index for finding houses for sale across the United States. Homebuyers typically search for houses in a specific area and price range. For instance, a typical query in this application might look like: `city:Denver AND price:[250000 TO 300000]`. Home price seems like a good example of a field that needs to support efficient range searches. As this application will have listings from across the US, the price field will have a broad range of values from the low $10,000's to over $10,000,000, so its cardinality will be large.

Next, we need to decide on the best field type for listing price. You can imagine that most house prices will be rounded to the nearest dollar, as it's rare to see a home price with cents, so an int or long field should suffice. Also, the maximum value for an integer in Solr is 2,147,483,647 (2.1B) so it's unlikely to see a listing that exceeds this maximum. Keep in mind that you want to be as frugal with your field types as possible, i.e. avoid using an 8-byte long when a 4-byte int will suffice. This reduces the size of your index on disk and reduces memory usage during searching and sorting. We can define a field for home listing price as:

```
<field name="listing_price" type="tint" indexed="true" stored="true" />
```

To support range queries, the field must be indexed and since we want to display the home price in search results, the field must also be stored. In the Solr example schema.xml, the "tint" field type is defined as:

```
<fieldType name="tint" class="solr.TrieIntField"
           precisionStep="8"
           positionIncrementGap="0"/>
```

Let's take a look at what gets indexed for a home price of $327,500 using a TrieInt field with `precisionStep="8"`. The intuition behind a trie-based field is that Lucene generates multiple terms for each value in the field, where each term has less precision. In other words, two different home prices will have overlapping terms at lower precisions. Lucene does this to reduce the number of terms that have to be matched to satisfy a range query. Table 5.5 shows the terms that were indexed for a listing price of $327,500 using `precisionStep="8"`:

Table 5.5 Terms indexed for listing price $327,500 using TrieInt field with precisionStep 8

Precision Step (8)	Operation	Indexed Term
0 – no bits removed	`327500 & 0xFFFFFFFF`	327500
1 – 8 least significant bits removed	`327500 & 0xFFFFFF00`	327424
2 – 16 least significant bits removed	`327500 & 0xFFFF0000`	262144
3 – 24 least significant bits removed	`327500 & 0xFF000000`	0

Table 5.5 shows that at each precision step, Lucene removes (8 * step count) least significant bits from the original value, which reduces the "precision" of the indexed term. For example, at step 2, home prices $327,500 and $326,800 both would have term 262144 in the index, which means that a range query only needs to match 262144 instead of both prices. In fact, the single term 262144 would be the same for potentially thousands of homes priced between $262,144 and $327,679. In other words, using a precisionStep of 8 allows Solr to match potentially thousands of homes priced between $262,144 and $327,679 using a single term: 262144.

Let's compare this to using `precisionStep="4"` for listing price to see the impact of using a smaller step size. Table 5.6 shows the terms that would be indexed for price $327,500 if we use a `precisionStep="4"`:

Table 5.6 Terms indexed for listing price $327,500 using TrieInt field with precisionStep 4

Precision Step (4)	Java Bitwise Operation	Indexed Term
0 – no bits removed	`327500 & 0xFFFFFFFF`	327500
1 – 4 least significant bits removed	`327500 & 0xFFFFFFF0`	327488
2 – 8 least significant bits removed	`327500 & 0xFFFFFF00`	327424
3 – 12 least significant bits removed	`327500 & 0xFFFFF000`	323584
4 – 16 least significant bits removed	`327500 & 0xFFFF0000`	262144
5 – 20 least significant bits removed	`327500 & 0xFFF00000`	0

Table 5.6 shows that using a smaller precision step equates to more terms being indexed, 6 vs. only 4 when using precision step of 8. In general, a smaller precision step leads to more terms being indexed per value, which increases the size of your index. However, more terms also equates to faster range queries because Lucene can narrow the search space quicker with more terms. The intuition here is that Lucene can search the center of a range using the lowest possible precision in the trie. However, the upper and lower boundaries of

the range must be searched more precisely, so having more terms indexed allows the range boundaries to be matched more efficiently.

To verify this, we conducted an informal benchmark by indexing 500,000 randomly generated listing prices between $110,000 and $5,000,000. After indexing, we generated 10,000 random range queries to get a feel for average query performance; table 5.7 summarizes the results.

Table 5.7 Results from our informal benchmark to compare index size, term count, and query performance using precisionStep 4 and 8 for our trie-based home listing price field.

precisionStep	Number of Terms Indexed	Index Size (KB)	RangeQuery Performance
8	68,074	28,612	7.0ms
4	118,170	32,496	6.3ms

Notice that the index sizes differ by about 8-bytes per document, which agrees with table 5.6 where we saw two extra 4-byte integer terms created per document when using a smaller step size. To summarize, when selecting a precision step, you have to balance space considerations with range query performance. For our TrieInt listing price field, a precision step of 4 leads to more terms being indexed per unique price but slightly faster range searches, especially when the cardinality of a field is large (many unique values).

5.5 Sending documents to Solr for indexing

We now have enough background on the indexing process to begin adding documents to Solr. In this section we learn how to send documents to Solr for indexing and get a glimpse of what happens behind the scenes. At the end of this section, you'll be able to start indexing documents to Solr from your application. Let's begin by learning how to index the example tweets we've been working with in this chapter.

5.5.1 Indexing documents using XML or JSON

As we touched on in section 5.1, Solr allows you to add documents using a simple XML document structure. Listing 5.12 shows this structure for the two example tweets we used previously in this chapter. However, in this case, we changed the names of the fields to use dynamic fields. For instance, **screen_name_s** will be a string because of the **_s** suffix on the name. We do this for convenience as you can add these documents to the example Solr server without making any changes to schema.xml. If you were building a real application, then you may want to declare the fields from listing 5.3 explicitly in your schema.xml, but using dynamic fields works fine for this example.

Listing 5.12 XML document used to index example tweets in Solr using dynamic fields

```
<add>  #A
  <doc> #B
    <field name="id">1</field>
    <field name="screen_name_s">@thelabdude</field> #C
    <field name="type_s">post</field>
    <field name="lang_s">en</field>
    <field name="timestamp_tdt">2012-05-22T09:30:22Z/HOUR</field>
    <field name="favourites_count_ti">10</field>
    <field name="text_t">#Yummm :) Drinking a latte at Caffe Grecco in SF's
historic North Beach... Learning text analysis with #SolrInAction by
@Manning on my i-Pad</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="screen_name_s">@thelabdude</field>
    <field name="type_s">post</field>
    <field name="lang_s">en</field>
    <field name="timestamp_tdt">2012-05-23T09:30:22Z/HOUR</field>
    <field name="favourites_count_ti">10</field>
    <field name="text_t">Just downloaded the MEAP for #SolrInAction from
@Manning http://bit.ly/15tzw to learn more about #Solr
http://bit.ly/3ynriE</field>
    <field name="link_ss">http://manning.com/</field>
    <field name="link_ss">http://lucene.apache.org/solr/</field>
  </doc>
</add>
```

#A Tell Solr we are adding new documents to the index
#B You can add more than one document at a time, each wrapped in a <doc> tag
#C Use dynamic fields to determine the field type for each field based on the suffix in the name

Let's send this XML document to Solr to index these two tweets. The Solr example includes a simple command-line application that allows you to post XML documents into the example server. Open a command-line on your workstation and execute the following commands as seen in listing 5.13.

Listing 5.13 Commands to index the example tweets in Solr

```
cd SOLR_IN_ACTION_HOME/example-docs/     #A
java -jar post.jar ch5/tweets.xml        #B

SimplePostTool: version 1.4                #C
SimplePostTool: POSTing files to http://localhost:8983/solr/update.. #C
SimplePostTool: POSTing file ch5/tweets.xml  #C
SimplePostTool: COMMITting Solr index changes..  #D
```

#A replace SOLR_IN_ACTION_HOME with the actual path on your computer
#B post the ch5/tweets.xml to Solr for indexing
#C output from the post.jar application
#D Commit the documents to make them visible in search results

The two example tweets should now be indexed in your example Solr server. To verify, navigate with your favorite Web browser, to the Solr admin panel at http://localhost:8983/solr/#/, click on **Query** under **collection1** in the menu on the left, and execute query "`type_s:post`" query as seen in figure 5.6.

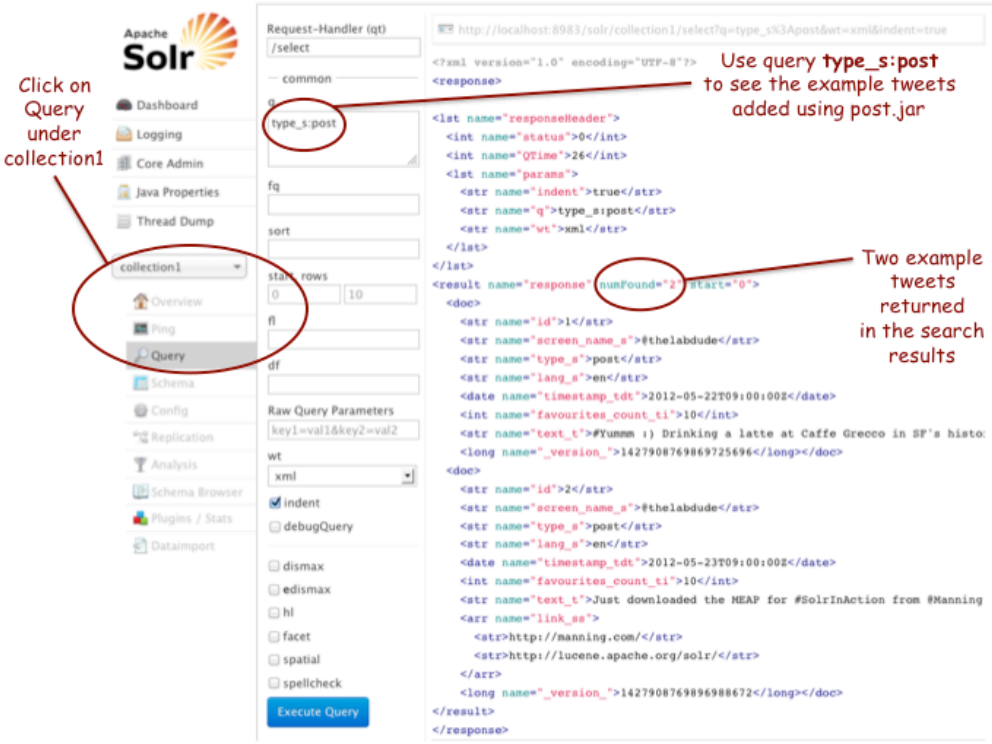


Figure 5.6 Screen shot of using query **type_s:post** to see the example tweets we added using the post.jar command-line utility provided with the Solr example.

Behind the scenes, the post.jar application sent the XML document over HTTP to Solr's update handler at URL: http://localhost:8983/solr/collection1/update. The update handler supports adding, updating, and deleting documents; we cover the update request handler in more detail in section 5.6 below.

Beyond XML, Solr's update request handler also supports the popular JSON (JavaScript Object Notation) and CSV (comma-separated values) data formats. For example, instead of indexing our example tweets using XML, we could have used JSON as shown in listing 5.14.

Listing 5.14 Using JSON to index documents instead of XML

```
[    #A
  {   #B
    "id" : "1",
    "screen_name_s" : "@thelabdude",
    "type_s" : "post",
    "lang_s" : "en",
    "timestamp_tdt" : "2012-05-22T09:30:22Z/HOUR",
    "favourites_count_ti" : "10",
```

```
    "text_t" : "#Yummm :) Drinking a latte at Caffe Grecco in SF's historic
North Beach... Learning text analysis with #SolrInAction by @Manning on my
i-Pad"
  },
  {
    "id" : "2",
    "screen_name_s" : "@thelabdude",
    "type_s" : "post",
    "lang_s" : "en",
    "timestamp_tdt" : "2012-05-23T09:30:22Z/HOUR",
    "favourites_count_ti" : "10",
    "text_t" : "Just downloaded the MEAP for #SolrInAction from @Manning
http://bit.ly/15tzw to learn more about #Solr http://bit.ly/3ynriE",
    "link_ss" : [ "http://manning.com/",
                  "http://lucene.apache.org/solr/"] #C
  }
]
```

#A Wrap all documents to index in a JSON array
#B Each document to index is a JSON object
#C Multi-valued fields encoded as a JSON array

We'll use the post.jar utility to send the JSON to Solr, but since XML is the default type, we have to explicitly tell the application that we are sending JSON by setting the "type" system property to "application/json":

```
java -Dtype=application/json -jar post.jar ch5/tweets.json
```

The placement of the `-Dtype=application/json` parameter to this command is important—it must be before the `-jar` argument. For a full list of options supported by post.jar, you can do: `java -jar post.jar --help`.

If you walked through both exercises of adding the XML and JSON documents, then you might think there are 4 documents in your index. However, because we are using the "id" field in our documents, there will only be 2 documents in the index. Verify this by yourself by re-issuing the `type_s:post` query as before. This demonstrates how Solr will update an existing document using the unique key field, which in the example schema.xml is "id".

If you look closely at the output from the post.jar application, you'll notice that it also sends a commit to Solr after POSTing the documents. Regardless of how you send documents to Solr, they are not searchable until they are committed. The commit process is quite involved and is covered in detail in section 5.6 below. Let's continue our discussion of how to index documents by learning about a popular Java-based client for Solr called **SolrJ**.

5.5.2 Using the SolrJ client library to add documents from Java

SolrJ is a Java-based client library provided with the core Solr project to communicate with your Solr server from a Java application. In this section, we'll implement a simple SolrJ client to send documents using Java. If you're not a Java developer or your application is not written in Java, then you'll be happy to know there are many other Solr client libraries

available for other languages, such as rsolr for Ruby and KoPHP for PHP. For a complete list of client libraries, see the Integrating Solr page in the Solr wiki[3].

Listing 5.15 provides a simple example using SolrJ to add our two example tweet documents to the index and then doing a hard commit. After committing, the example code sends the match all docs query (*:*) to Solr to get the documents we indexed back in the search results.

Listing 5.15 Example SolrJ client application

```
package sia.ch5;
...
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.SolrServer;
import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrServer;
import org.apache.solr.client.solrj.response.QueryResponse;
import org.apache.solr.common.SolrDocument;
import org.apache.solr.common.SolrDocumentList;
import org.apache.solr.common.SolrInputDocument;

public class ExampleSolrJClient {

    public static void main(String[] args) throws Exception {
        String serverUrl = (args != null && args.length > 0) ? args[0] :
                           "http://localhost:8983/solr/collection1";

        SolrServer solr = new HttpSolrServer(serverUrl);  #A

        SolrInputDocument doc1 = new SolrInputDocument(); #B
        doc1.setField("id", "1");
        doc1.setField("screen_name_s", "@thelabdude");
        doc1.setField("type_s", "post");
        doc1.setField("lang_s", "en");
        doc1.setField("timestamp_tdt", "2012-05-22T09:30:22Z/HOUR");
        doc1.setField("favourites_count_ti", "10");
        doc1.setField("text_t", "#Yummm :) Drinking a latte at Caffe Grecco"
in SF's historic North Beach... Learning text analysis with #SolrInAction by
@Manning on my i-Pad");

        solr.add(doc1);     #C

        SolrInputDocument doc2 = new SolrInputDocument();
        doc2.setField("id", "2");
        doc2.setField("screen_name_s", "@thelabdude");
        doc2.setField("type_s", "post");
        doc2.setField("lang_s", "en");
        doc2.setField("timestamp_tdt", "2012-05-22T09:30:22Z/HOUR");
        doc2.setField("favourites_count_ti", "10");
        doc2.setField("text_t", "Just downloaded the MEAP for #SolrInAction
from @Manning http://bit.ly/15tzw to learn more about #Solr
http://bit.ly/3ynriE");
```

[3] http://wiki.apache.org/solr/IntegratingSolr

```
        doc2.addField("link_ss", "http://manning.com/");            #D
        doc2.addField("link_ss", "http://lucene.apache.org/solr/"); #D

        solr.add(doc2);

        solr.commit(true, true);              #E

        for (SolrDocument next : simpleSolrQuery(solr, "*:*", 10)) {
            prettyPrint(System.out, next);
        }
    }

    static SolrDocumentList simpleSolrQuery(SolrServer solr,
            String query, int rows) throws SolrServerException {
        SolrQuery solrQuery = new SolrQuery(query);  #F
        solrQuery.setRows(rows);                     #F
        QueryResponse resp = solr.query(solrQuery);  #F
        SolrDocumentList hits = resp.getResults();
        return hits;
    }

    static void prettyPrint(PrintStream out, SolrDocument doc) { #G
        List<String> sortedFieldNames =
             new ArrayList<String>(doc.getFieldNames());
        Collections.sort(sortedFieldNames);
        out.println();
        for (String field : sortedFieldNames) {
            out.println(String.format("\t%s: %s",
                  field, doc.getFieldValue(field)));
        }
        out.println();
    }
}
```

#A Connect to the Solr server at the specified URL, such as http://localhost:8983/solr/collection1
#B Use a SolrInputDocument object to build a document to be indexed
#C Send the SolrInputDocument to the Solr update request handler over HTTP
#D Use the addField method to add multi-values for multi-valued fields
#E Do a normal or "hard" commit to make these new docs searchable
#F Use a SolrQuery object to construct the match all docs query
#G Pretty-print each SolrDocument in the results to stdout

As you can see from this basic example, the SolrJ API makes it very easy to connect to Solr, add documents, send queries, and process results. To begin, all you need is the URL of the Solr server, which in our example was http://localhost:8983/solr/collection1. Behind the scenes SolrJ uses the Apache HttpComponents Client library to communicate with the Solr server using HTTP. In section 5.5.1, we saw how Solr supports XML and JSON, so you may be wondering if SolrJ is using one of those formats to connect to Solr. It turns out that SolrJ actually uses an internal binary protocol called **javabin** by default. When doing Java-to-Java communication, the javabin protocol is more efficient than using XML or JSON.

Lastly, for better scalability for handling applications that need to index many documents very quickly, SolrJ provides a **ConcurrentUpdateSolrServer** that is more efficient for doing bulk adds and updates. In listing 5.15, each invocation of the add method resulted in a

separate HTTP request to the Solr server. In contrast, if we used the **ConcurrentUpdateSolrServer**, then add requests would be buffered on the client and sent to Solr in a bulk transfer. However, while the **ConcurrentUpdateSolrServer** is well suited for efficiently indexing many documents, it should not be used to make queries to Solr. Looking ahead, we'll see another type of SolrServer provided by SolrJ called **CloudSolrServer** when we learn about SolrCloud in chapter 13.

5.5.3 Other tools for importing documents into Solr

We've seen how we can send documents to Solr using basic HTTP POST using the post.jar application and using the popular SolrJ client from Java. These are not the only ways to get your documents into Solr. Being a mature, widely-adopted open source technology, Solr offers a number of powerful utilities for adding documents from other systems. In this section we introduce you to three popular tools available for populating your Solr index:

- Data Import Handler (DIH)
- Extracting Request Handler aka Solr Cell
- Nutch

Each of these tools is powerful and could easily justify taking an entire chapter to describe each tool. So for now, we just want to give brief mention of these tools so that you are aware of these options for populating your index.

DATA IMPORT HANDLER (DIH)

The Data Import Handler is an extension that pulls data from one or more relational databases into Solr using SQL. The DIH works with any database that provides a modern JDBC driver, such as Oracle, Postgres, MySQL, and MS SQL Server. At a high-level, you provide the database connection parameters and a SQL query to Solr and the DIH component queries the database and transforms the results into documents. We cover the DIH in detail in chapter 12, so for now let's look at another tool for indexing rich binary documents like PDF and MS Word documents.

EXTRACTING REQUEST HANDLER

The ExtractingRequestHandler, commonly called "Solr Cell" allows you to index text content extracted from binary files like PDF, MS Office, and OpenOffice documents. Behind the scenes, Solr Cell uses the Apache Tika project to do the extraction. Specifically, Tika provides components that know how to detect the type of document and then parse the binary documents to extract text and metadata. For example, you can send a PDF document to the ExtractingRequestHandler and it will automatically populate fields like title, subject, keywords, and body text in your Solr index. We will also work through an example using the ExtractingRequestHandler in chapter 12.

NUTCH

Apache Nutch is a Java-based open source Web crawler designed to crawl the Web, or at least very large portions of it. Nutch integrates with Solr out-of-the-box to make the Web

pages it crawls searchable using Solr. Thus, if your application needs to crawl hyper-linked pages on a massive scale, then Nutch is probably a good place for you to start.

Now that you've seen how to send documents to Solr for indexing, let's learn how those request are actually processed in Solr using a component called the update handler.

5.6 Update handler

In the previous section, we sent new documents to Solr using HTTP POST requests. The request to "add" these new documents was handled by Solr's update handler. In general, the update handler processes all updates to your index as well as commit and optimize requests. Table 5.8 provides an overview of common request types supported by the update handler.

Table 5.8 Overview of common requests processed by the update handler

Request Type	Description	XML Example

Add	Add one or more documents to the index, see listing 5.12 for a full example	`<add>`
		`  <doc>`
		`  <field name="id">1</field>`
		`  ...`
		`  </doc>`
		`</add>`

Delete	Delete a document by ID, such as deleting document with ID=1	`<delete>`
		`  <id>1</id>`
		`</delete>`

Delete by Query	Delete documents that match a Lucene query, such as deleting all micro-blog documents from user with screen_name=@thelabdude	`<delete>`
		`<query>screen_name:@thelabdude</query>`
		`</delete>`

Atomic Update	Update one or more fields of an existing document using optimistic locking; see section 5.6.4 below.	`<add>`
		`  <doc>`
		`  <field name="id">1</field>`
		`  <field update="set"`
		`name="favourites_count">12</field>`
		`  </doc>`
		`</add>`

Commit	Commits documents to the index with options to do a soft or hard commit and whether to block on the client until the new searcher is open and warmed.	`<commit waitSearcher="true"` `  softCommit="false" />`

Optimize	Optimize the index by merging segments and removing deletes	`<optimize waitSearcher="false"/>`

Although table 5.8 shows examples of update requests using XML, the update request handler supports other formats, such as JSON, CSV, and javabin. Behind the scenes, the update request handler looks at the Content-Type HTTP header to determine the format of the request, such as `Content-Type: text/xml`.

Listing 5.16 shows the configuration of the update handler in **solrconfig.xml**:

Listing 5.16 Configuration elements for the update handler in solrconfig.xml

```
<updateHandler class="solr.DirectUpdateHandler2">
  <updateLog>                                  #A
    <str name="dir">${solr.ulog.dir:}</str>    #A
  </updateLog>                                 #A
  <autoCommit>                                 #B
    <maxTime>15000</maxTime>                   #B
    <openSearcher>false</openSearcher>         #B
  </autoCommit>                                #B
  <autoSoftCommit>                             #C
    <maxTime>1000</maxTime>                    #C
  </autoSoftCommit>                            #C
  <listener event="postCommit" ...>            #D
   ...
  </listener>
</updateHandler>
```

#A Enable the transaction log, covered in section 5.6.2
#B Configure auto-commit policy, covered in section 5.6.1
#C Configure the soft auto-commit policy, see section 5.6.1
#D Register an update event listener

One of the most important tasks performed by the update handler is to process requests to commit documents to the index to make them visible in search results.

5.6.1 Committing documents to the index

In this section, we dig into the details of how Solr makes documents available for searching by committing them to the index. When a document is added to Solr, it will not be returned in search results until it is committed to the index. In other words, from a query perspective, a document is not visible until it is committed. In Solr 4, there are two types of commits: soft and normal or sometimes called "hard" commits. Let's look at how normal commits work as that will help you understand soft commits.

NORMAL COMMIT

A normal or "hard" commit is one where Solr flushes all un-committed documents to disk and refreshes an internal component called a "searcher" so that the newly committed documents can be searched. For our purposes, you can think of a searcher as a read-only view of all committed documents in the index. Refer to chapter 4.3 for a detailed discussion of how a searcher works. For now, let it suffice to say that a hard commit can be an

expensive operation that can impact query performance because it requires opening a new searcher.

After a normal commit succeeds, the newly committed documents are safely persisted to durable storage and will survive server restarts due to normal maintenance operations or a server crash. For high availability, you still need to have a solution to fail-over to another server if the disk fails. We discuss Solr's high-availability features in chapter 13.

SOFT COMMIT

A soft commit is a new feature in Solr 4 to support near-real-time (NRT) searching. We discuss near-real-time searching in more depth in chapter 13. For now, you can think of a soft commit as a mechanism to make documents searchable in near real-time by skipping the costly aspects of hard commits, such as flushing to durable storage and warming a new searcher. As soft commits are less expensive, you can issue a soft commit every second to make newly indexed documents searchable within about a second of adding them to Solr. However, keep in mind that you still need to do a hard commit at some point to ensure documents are eventually flushed to durable storage.

To summarize, a hard commit makes documents searchable but is expensive because it has to flush documents to durable storage and warm-up a new searcher. In contrast, a soft commit also makes documents searchable but they are not flushed to durable storage and a new searcher is not warmed up.

AUTO-COMMIT

For either normal or soft commits, you can configure Solr to automatically commit documents using one of three strategies.

1. Commit each document within a specified time
2. Commit all documents once a user-specified threshold of uncommitted documents is reached
3. Commit all documents on a regular time interval, like every 10 minutes

Solr's auto-commit behavior for hard and soft commits is configured in solrconfig.xml. The following XML snippet shows an example configuration where Solr will commit every 50,000 documents and every 10 minutes:

```
<autoCommit>
  <maxTime>600000</maxTime>       #A
  <maxDocs>50000</maxDocs>        #B
  <openSearcher>true</openSearcher> #C
</autoCommit>
```

#A Commit every 10 minutes (value in milliseconds)
#B Commit every 50,000 documents
#C Open a new searcher after committing

When performing an auto-commit, the normal behavior is to open a new searcher. However, Solr does allow you to disable this behavior by specifying `<openSearcher>false</openSearcher>`. In this case, the documents will be flushed to disk, but will not be visible in search results. Solr provides this option to help minimize the

size of its transaction log of uncommitted updates (see next section) and to avoid opening too many searchers during a large indexing process.

Imagine you have 5 million documents to index and have configured Solr to auto-commit every 50,000 documents. This means that Solr will perform 100 auto-commits during the process of indexing 5M documents. In this scenario, it may make sense to only pay the penalty of warming up a new searcher once, after all documents are indexed rather than warming up a new searcher 100 times. Of course, your client application can also send an intermittent hard commit request every 1M documents so that some documents are visible in search results sooner. The main point is that you want to think about whether you need to open a new searcher after every auto-commit. If the number of documents you are indexing is much larger than your auto-commit threshold, then you might consider setting `openSearcher="false"` and have your client issue a final "hard" commit once all documents are indexed.

Also, don't confuse the **openSearcher** attribute for the `<autoCommit>` element with the **waitSearcher** attribute for the `<commit>` request described in table 5.8. A new searcher is always opened and warmed when you send a `<commit>` request; the **waitSearcher** attribute indicates whether your client code should block until the new searcher is fully warmed up. As you learned in chapter 4, warming a new searcher can take a long time so use `waitSearcher="true"` with caution.

You can also configure Solr to do soft-commits automatically using the `<autoSoftCommit>` element in solrconfig.xml. However, you will want to use much smaller values for soft commits, such as every second (1000 ms) as shown in the XML snippet below.

```
<autoSoftCommit>
  <maxTime>1000</maxTime> #A
</autoSoftCommit>
```

#A Do a soft commit every second (1000 ms)

Now let's turn our attention to another powerful feature of the update handler that helps ensure you don't lose un-committed updates.

5.6.2 Transaction log

Solr uses a transaction log to ensure updates accepted by Solr are saved on durable storage until they are committed to the index. Imagine the scenario where your client application sends a commit every 10,000 documents. If Solr crashes after the client sends some documents to be indexed but before your client sends the commit, then without a transaction log, these un-committed documents will be lost. Specifically, the transaction log serves three key purposes:

1. Used to support real-time gets and atomic updates
2. De-couples write-durability from the commit process

3. Supports synchronizing replicas with shard leaders in Solr Cloud (covered in chapter 13)

The transaction log is configured for a core in **solrconfig.xml**:

```
<updateLog>
  <str name="dir">${solr.ulog.dir:}</str> #A
</updateLog>
```

#A Default location is tlog sub-directory of data

Every update request is logged to the transaction log. The transaction log continues to grow until you issue a commit. During a commit, the active transaction log is processed and then a new transaction log file is opened. Figure 5.7 illustrates the steps involved in processing an update request.

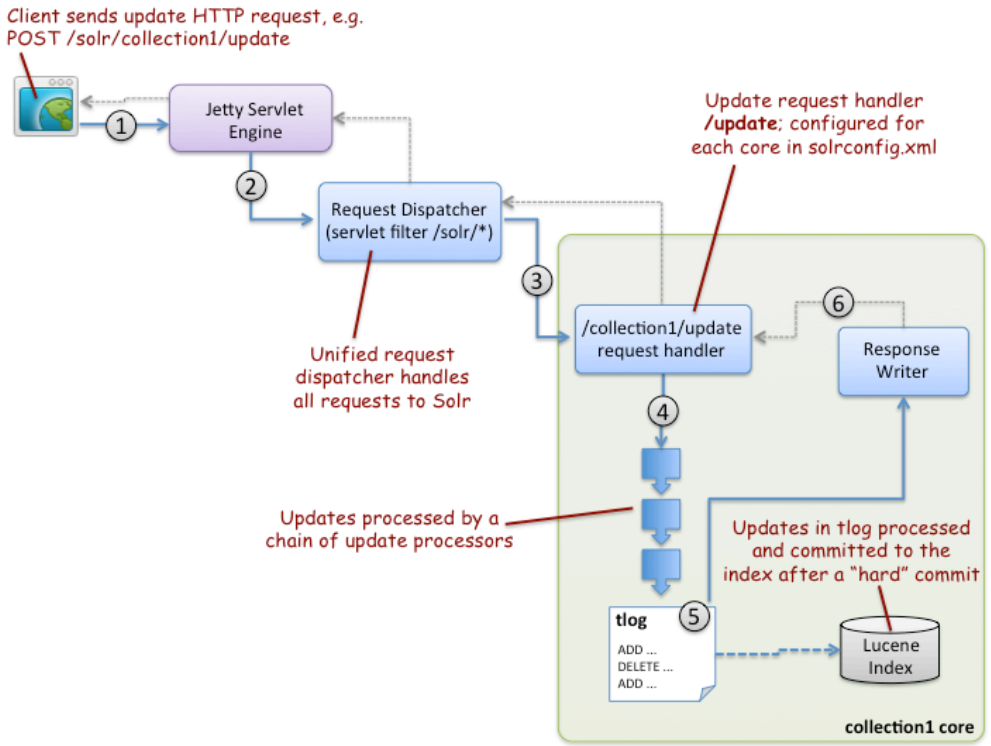


Figure 5.7 Sequence of events and main components used to process update requests, such as adding a new document.

A few of the components in figure 5.7 should be familiar to you, such as the request dispatcher and response writer. These are the same components we discussed in chapter 4

when learning about query request processing. Let's walk through the sequence of events in figure 5.7 to highlight some of the important concepts:

1. Client application sends an update request using HTTP POST. The client can send the request as JSON, XML, or Solr's internal binary **javabin** format. We saw an example client built using SolrJ in listing 5.15.
2. Jetty routes the request to the Solr Web application.
3. Solr's request dispatcher uses the "collection1" part of the request path to determine the core name. Next, the dispatcher locates the **/update** request handler registered in solrconfig.xml for the **collection1** core.
4. The update request handler processes the request. When adding or updating documents, the update handler uses **schema.xml** to process each field in each document in the request. In addition, the request handler invokes a configurable chain of update request processors to perform additional work on each document during indexing. We'll see an example of this in chapter 6 where we use an update request processor to do language detection during indexing.
5. The ADD request is written to the transaction log.
6. Once the update request is securely saved to durable storage, a response is sent to the client application using a response writer. At this point, the client application knows the update request is successful and can continue processing.

With the transaction log, your main concern is balancing the trade-off between the length of your transaction log, i.e. how many uncommitted updates, and how frequently you want to issue a hard commit. If your transaction log grows very large, then a restart may take a long time to process the updates, thus delaying your recovery process.

For instance, if the average size of your documents is 5KB and you commit every 100,000 documents, then your transaction log can grow to over 3.7GB! Of course, 5KB per document is a large document. In contrast, committing micro-blog documents like we used in this chapter every 100,000 documents would not be a problem. The key take-away is that you need to consider the size of your transaction log when configuring your auto-commit settings. As you learned in the previous section, you can issue a hard commit without affecting the query side by setting `openSearcher="false"` for your auto-commit settings. However, this implies that at some point, your client application must issue a full hard commit to make all updates visible in search results.

5.6.3 Atomic Updates

You can update existing documents in Solr by sending a new version of the document. However, unlike a database where you can update a specific column in a row, with Solr you must update the entire document. Behind the scenes, Solr deletes the existing document and creates a new one; this occurs whether you change one field or all fields.

From a client perspective, your application must send a new version of the document in its entirety. For some applications where documents can be created from other sources, this is not such a big deal. For others that use Solr as a primary data store, re-creating a document in its entirety just to update a single field can be problematic. In practice, this requires users to query for the entire document, apply the updates, and then send the fully specified document back to Solr.

This pattern of requesting all fields for an existing document, updating a subset of fields, and then sending the new version to Solr is very common in practice. Consequently, atomic updates are new feature in Solr that allow you to send updates to only the fields you want to change. This brings Solr more inline with how database updates work. Solr will still delete and create a new document, but this is transparent to your client application code.

FIELD-LEVEL UPDATES

Returning to our micro-blog search example, let's imagine that we want to index a new field on existing documents that holds the number of times the tweet has been re-tweeted. We'll use this new field as an indication of popularity of a tweet. To keep things simple, we'll update this field once a day. You can imagine a daily volume statistic would also be useful but we'll just stick with an aggregated value to keep things simple. Our focus here is to learn about atomic updates.

Let's name our new field **retweet_count_ti**, which indicates we are using a dynamic field so we don't have to update the **schema.xml** to add this new field. The "_ti" suffix applies the following dynamic field (from schema.xml):

```
<dynamicField name="*_ti" type="tint" indexed="true" stored="true"/>
```

Here's an example request to update the **retweet_count_ti** field using XML:

```
<add>  #A
  <doc>
    <field name="id">1</field>   #B
    <field update="set" name="retweet_count_ti">100</field>  #C
  </doc>
</add>
```

#A Slightly un-intuitive but updates must be wrapped in an <add> element.
#B Identify the existing document with id=1
#C Sets the retweet_count_ti field to 100

Behind the scenes, Solr locates the existing document with `id="1"`, retrieves all stored fields from the index, deletes the existing document, and creates a new document from all existing fields plus the new **retweet_count_ti** field. It follows that all fields must be stored for this to work because the client application is only sending the id field and the new field. All other fields must be pulled from the existing document.

We used the `update="set"` directive to set the **retweet_count_ti** field. Alternatively, since our update process runs daily, we can just count them for the previous day and increment the existing value using `update="inc"`. In addition to **set** and **inc**, you can also use "add" to append a new value to a multi-valued field.

OPTMISTIC CONCURRENCY CONTROL

Now consider a slightly more involved example where we want to use crowd sourcing as a way to classify the sentiment of micro-blogs. In a nutshell, we'll pay users to classify each document as either being positive, neutral, or negative. The sentiment field could be useful for allowing users to find negative information about a product or restaurant.

Once classified, each micro-blog document needs to be updated in Solr with the sentiment label. In our re-tweet count example, we updated the **retweet_count_ti** field once a day using an automated process. However, with sentiment classification, updates to the **sentiment_s** field can happen at anytime. Thus, it's conceivable that two users will attempt to update the sentiment label on same document at the same time. Of course we could implement some cumbersome process that requires users to explicitly lock a document before labeling, but that would slow them down unnecessarily. Also, we probably don't want to pay for a document to be classified twice. Hence, we need some way to guard against concurrent updates to the same document—enter optimistic concurrency control.

To avoid conflicts, Solr supports optimistic concurrency control using a special version tracking field, named **_version_**. The special version field should be defined in your schema.xml as:

```
<field name="_version_" type="long" indexed="true" stored="true"/> #A
```

#A With Solr 4, you should not change or remove this field from your schema.xml

When a new document is added, Solr assigns a unique version number automatically. When you need to guard against concurrent updates, you simply include the exact version the update is based on in the update request. Consider the following update request that includes a specific **_version_**:

```
<add>
  <doc>
    <field name="id">1</field> #A
    <field update="set" name="sentiment_s">positive</field>  #B
    <field update="set" name="classified_by_s">SomeUserID</field> #C
    <field name="_version_">1234567890</field>    #D
  </doc>
</add>
```

#A Identify the document to update using its unique ID
#B Set the sentiment_s field to "positive"
#C Track which user classified this tweet so they get paid
#D Return the _version_ of the document this update is based on

When Solr processes this update, it will compare the **_version_** value in the request with the latest version of the document, pulled from either the index or the transaction log. If they match, then Solr applies the update. If they do not match, then the update request fails and an error is returned to the user. A client application can handle the error response to let the user know the document was already classified by another user. This approach is called "optimistic" because it assumes that most updates will work on the initial attempt and that conflicts are rare.

Using the **_version_** field to enforce concurrency control raises the question of how does the client application get the current _version_ from Solr? The best technique is to use a

real-time get request. For instance, to get the _version_ field for our example document with ID 1, you would send HTTP GET request:

http://localhost:8983/solr/collection1/get?id=1&fl=id,_version_

Real-time get returns the latest version of a document regardless of whether it is committed to the index. Consequently, real-time get and atomic updates rely on the transaction log being enabled for your index.

Solr gives you a few other options for handling concurrent updates with the _version_ field. Table 5.9 gives an overview of how Solr behaves depending on the value of the **_version_** in an update request:

Table 5.9 Using the **_version_** field to enforce update semantics in Solr

If _version_ field is set to	Solr does
>1	Versions must match or the update fails
1	The document must exist
<0	The document must not exist
0	No concurrency control desired, existing value is overwritten

As illustrated by this simple example, atomic updates are a powerful new addition to Solr's arsenal of data management features. With Solr 4, you can now update existing documents simply by sending the fields that need to be updated along with the unique identifier of the document to update.

5.7 Index management

In chapter 4, we delayed a discussion of index management settings in **solrconfig.xml** until you had more background with Solr indexing. You are now ready to tackle Solr's index management settings. In this section, we focus on the index-related settings that you are most likely to need to change, beginning with how indexed documents are stored. It should be said that most of the index-related settings in Solr are for expert use only. What this really means is that you should take caution when making changes and that the default settings are appropriate for most Solr installations.

5.7.1 Index storage

When documents are committed to the index, they are written to durable storage using a component called a **Directory**. The Directory component provides the following key benefits to Solr:

- Hides details of reading from and writing to durable storage, such as using JDBC to store documents in a database

- Implements a storage-specific locking mechanism to prevent index corruption, such as OS-level locking for file system based storage
- Insulates Solr from JVM and operating system peculiarities
- Enables extending the behavior of a base Directory implementation to support specific use cases like near real-time search

Solr provides several different Directory implementations and there is no one best Directory implementation for all Solr installations! Thus, you will need to do some research to decide on the best implementation for your specific application of Solr. In practice, this depends on your operating system, JVM type, and use cases. However, as you learned in chapter 4, Solr tries to be well-configured out-of-the-box. Let's dig into how Solr's index storage is configured by default, which will help you decide if you need to change the default configuration.

DEFAULT STORAGE CONFIGURATION

By default, Solr uses a Directory implementation that stores data to the local file system in the **data** directory for a core. For instance, the example server stores its index in `$SOLR_INSTALL/example/solr/collection1/data`. The location of the data directory is controlled by the `<dataDir>` element in **solrconfig.xml**:

```
<dataDir>${solr.data.dir:}</dataDir> #A
```

#A Default setting from solrconfig.xml, resolves to collection1/data for the example server

The `solr.data.dir` property defaults to "data" but can be overridden in **solr.xml** for each core, such as:

```
<core loadOnStartup="true" instanceDir="collection1/"
      transient="false" name="collection1"
      dataDir="/usr/local/solr-data/collection1"/> #A
```

#A Directory to store data for the collection1 core

The first thing you need to consider is whether the data directory for your index has enough storage capacity for your index. Also, it's important that your data directory supports fast reads and writes, with a little more priority given to read performance. Strategies for optimizing disk I/O is beyond the scope of this book, but here are some simple pointers to keep in mind:

- Each core should not have to compete for the disk with other processes
- If you have multiple cores on the same server, it's a good idea to use separate physical disks for each index
- Use high quality, fast disks or even better, consider using Solid State Drives (SSD) if your budget allows
- Spend some quality time with your system administrators to discuss RAID options for your servers

CHOOSING A DIRECTORY IMPLEMENTATION

Once you've tackled storage concerns, you also need to consider the best Directory implementation for your storage solution. The default Directory implementation used by Solr

is `solr.NRTCachingDirectoryFactory`, which is configured with the `<directoryFactory>` element in **solrconfig.xml**:

```
<directoryFactory name="DirectoryFactory"
  class="${solr.directoryFactory:solr.NRTCachingDirectoryFactory}"/>
```

The `NRTCachingDirectoryFactory` is actually just a wrapper class around `solr.StandardDirectoryFactory` to add support for near real-time search. At runtime, the `StandardDirectoryFactory` selects a specific Directory implementation based on your operating system and JVM type, as depicted in figure 5.8 below.

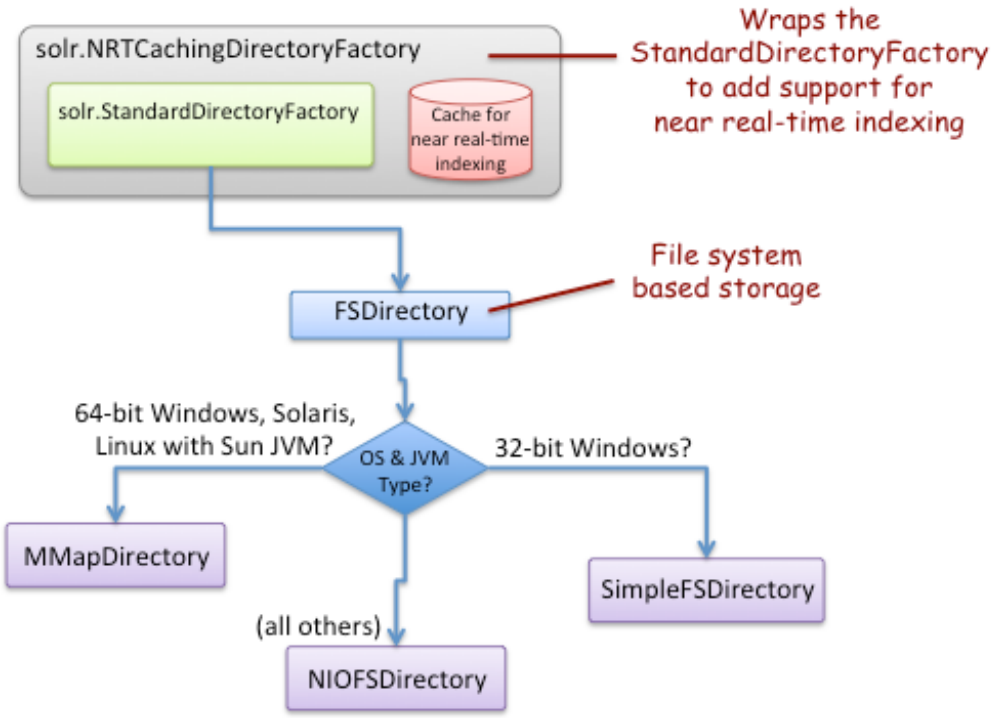


Figure 5.8 Solr Directory implementation selected at runtime depending on your specific operation system version and JVM type.

Based on figure 5.8, there are three possible file system based Directory options for Solr:

1. **MMapDirectory**: Uses memory-mapped I/O when reading the index; best option for installations on 64-bit Windows, Solaris, or Linux operating systems with the Sun JVM.
2. **SimpleFSDirectory**: Uses a Java RandomAccessFile; should be avoided unless you are running on 32-bit Windows

3. **NIOFSDirectory**: Uses java.nio optimizations to avoid synchronizing reads from the same file; should be avoided on Windows due to a long-standing JVM bug

You can determine which Directory implementation is enabled for your Solr server using the Core Admin page on the Solr administration console. Figure 5.9 shows where to find the Directory information for the example **collection1** core.

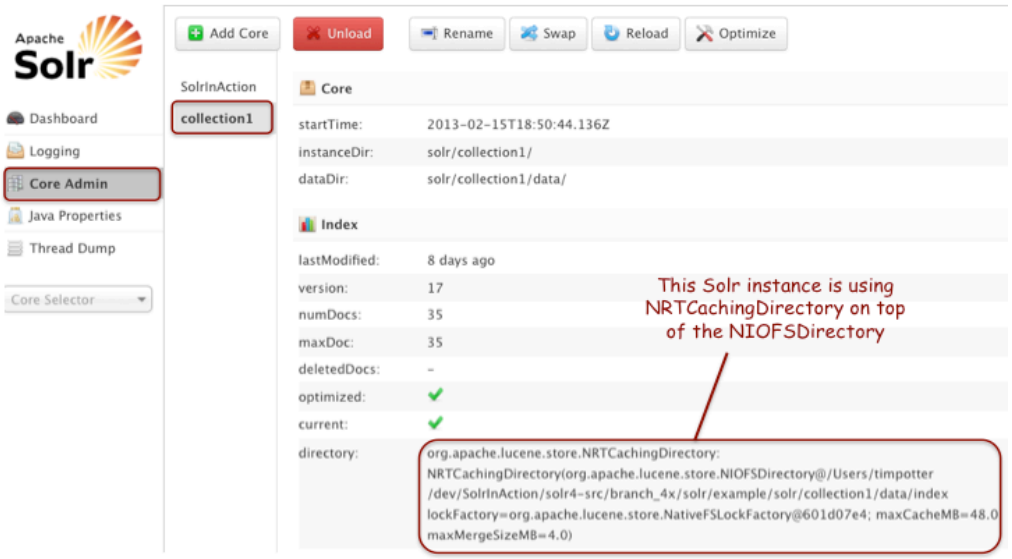


Figure 5.9 Screen shot of the Core Admin page showing the active Directory implementation for the collection1 core

You can override the default selection by explicitly setting the directory factory in solrconfig.xml. For example, in figure 5.9, Solr is using the NRTCachingDirectory implementation. If we want to change that to use MMapDirectory, then solrconfig.xml should be changed to:

```
<directoryFactory name="DirectoryFactory"
      class="${solr.directoryFactory:solr.MMapDirectoryFactory}"/> #A
```

#A Explicitly use the MMapDirectory implementation

The MMapDirectory implementation is your best option when running on 64-bit Windows, Solaris, and Linux because it optimizes read performance by using the virtual memory management features of these modern operating systems more intelligently[4].

[4] For a deeper understanding of MMapDirectory, please see Uwe Schindler's blog titled **Use Lucene's MMapDirectory on 64bit platforms, please!** at http://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectory-on-64bit.html

5.7.2 Segment merging

A segment is a self-contained, read-only sub-set of a full Lucene index; once a segment is flushed to durable storage, it is never altered. When new documents are added to your index, they are written to a new segment. Consequently, there can be many active segments in your index. Each query must read data from all segments to get a complete results set. At some point, having too many small segments can negatively impact query performance. Combining many smaller segments into fewer larger segments is commonly known as segment merging.

SHOULD I OPTIMIZE MY INDEX?

Optimize is an operation that forces Lucene to merge existing segments into a specified number of larger segments, with the default value being 1. For instance, an index with 32 segments will have only 1 large segment after optimizing. Let it suffice to say that optimize can be a very expensive operation in terms of memory, CPU, and disk I/O in Solr, especially for large indexes; it is not uncommon for a full optimization to take hours for a large index.

One of the most common questions on the Solr user mailing list is whether to "optimize" your index. This is understandable because who doesn't want an "optimized" index? However, current wisdom in the Solr community suggests that rather than optimizing your index, it is better to fine-tune Solr's segment merge policy. Moreover, having an optimized index doesn't mean that a slow query will suddenly become fast. Conversely, you may find that query performance is acceptable with an un-optimized index.

EXPERT-LEVEL MERGE SETTINGS

By default, all the segment-merging settings are commented out in solrconfig.xml. This is by design because the default settings should work for most installations, especially when you're just getting started. You should also notice that each element is labeled as an "expert" level setting. Table 5.10 provides an overview of segment merge related elements from solrconfig.xml.

Table 5.10 Overview of segment merge elements from solrconfig.xml

Element	Purpose
ramBufferSizeMB	Maximum amount of RAM used to buffer documents during indexing before they are flushed to the Directory; default is 100 megabytes (MB). This should not be confused with commits, which force all buffered documents to be written to durable storage. Increase this value to buffer more documents in memory and reduce disk I/O during indexing.
maxBufferedDocs	Maximum number of documents to buffer during indexing before flushing to the Directory (durable storage); default is 1000 documents. This should not be confused with commits, which force all buffered documents to be written to durable storage.

mergePolicy	Controls how Lucene performs segment merging, such as deciding how many segments to merge at once. Default is the **TieredMergePolicy**, see JavaDoc for `org.apache.lucene.index.TieredMergePolicy` for more information.
mergeFactor	Controls how many segments get merged at once; default is 10. Determining the best value for your index depends on average document size, available RAM, and desired indexing throughput.
mergeScheduler	Controls when segment merging runs; default setting is to run concurrently in the background using the **ConcurrentMergeScheduler**.

To be clear, just because these expert-level settings are commented out, segment merging is still enabled in your index, running in the background[5]. For now, we recommend that you avoid optimizing and just use the default configuration for segment merging until you have a good reason to change these settings. It's very likely that "doing nothing" when it comes to segment merging is the right approach for your server[6]. If indexing throughput becomes an issue for your application, then you can revisit these settings. Unfortunately, we're not able to be more specific when it comes to tuning the merge process because it depends on so many environment specific factors.

HANDLING DELETES

By now, you should know that segments are not changed after they are flushed to durable storage. This implies that deletes do not actually delete documents from existing segments. It turns out that deleted documents are not removed from your index until segments containing deletes are merged. In a nutshell, Lucene keeps track of deletes in a separate data structure and then applies the deletes when merging. For the most part, you don't have to worry about how this works.

5.8 Summary

At this point you should have a good understanding of the Solr indexing process. To recap, we began the chapter by learning about the schema design process. Specifically, we discussed considerations about document granularity, document uniqueness, and how to determine if a field should be indexed, stored, or both.

Next, we learned how to define fields in schema.xml, including multi-valued and dynamic fields. We saw how dynamic fields are useful for supporting documents with many fields and documents coming from diverse sources. You also learned how to use Solr's `<copyField>` directive in order to populate a catch-all text search field or to apply different text analysis to the same text during indexing.

---

[5] Check out Mike McCandless' blog entry **Visualizing Lucene's segment merges** to see a visualization of segment merging; http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html

[6] Refer to section 2.13.6 in **Lucene in Action**, second edition for more information about merging.

Next, we saw how to work with structured data using Solr's support for strings, dates, and numeric field types. We used Solr's round-down operator (/) to index date values at different precisions, such as hour-level precision using `/HOUR`. We also learned that Solr provides Trie-based fields to support efficient range queries and sorting on numeric and date fields. We saw how to use the **precisionStep** attribute for numeric and date fields to balance the trade-off between having a larger index size and range query performance.

Armed with an understanding of schema.xml, we learned how to send XML and JSON documents to Solr using HTTP and SolrJ. We briefly introduced some additional tools provided by Solr for importing documents from a relational database (DIH) and indexing rich binary documents like PDF and MS Word using the extracting request handler.

After documents are processed, they need to be committed before they can be searched using either normal commits or soft commits for near-real-time search. We showed how Solr uses a transaction log to avoid losing un-committed updates. Beyond adding new documents, you learned how to update existing documents using Solr's atomic update support. You can guard against concurrent updates using optimistic concurrency control using the special **_version_** field.

We closed out this chapter by returning to a discussion of index-related settings from solrconfig.xml. Specifically, we showed you where and how Solr stores the index using a Directory component. You also learned about segment merging and that it is a good idea to avoid optimizing your index or changing segment merge settings until you have a better understanding of your indexing throughput requirements.

In the next chapter, we continue learning about the indexing process by diving deep into text analysis. After finishing chapter 6, you will have a solid foundation for designing and implementing a powerful indexing solution for your application.

6 Text analysis

This chapter covers

- Goals of text analysis
- Basic text analysis tools in Solr
- Testing with Solr's analysis form
- Defining custom field types for advanced text analysis
- Language detection during indexing
- Extending text analysis with Solr's Plug-In framework

In Chapter 5, we learned how the Solr indexing process works and learned to define non-text fields in schema.xml. In this chapter, we get a little deeper into the indexing process by learning about text analysis.

Text analysis removes the linguistic variations between terms in the index and terms provided by users when searching, so that a user querying for *"buying a new house"* matches documents with terms *"home"* and *"purchase"*. In this chapter, you'll learn, for example, how to configure Solr to establish a match between queries containing *"house"* and documents containing *"home"*.

When done correctly, text analysis allows your users to query using natural language without having to think about all the possible forms of their search terms. You don't want your users to have to construct queries like: "*buying house or purchase home or buying a home or purchasing a house …*"

Allowing users to find information they seek using natural language is fundamental to providing a good user experience. Given the broad adoption and sophistication of Google and similar search engines, users are conditioned to expect search engines to be very intelligent and intelligence in search starts with great text analysis!

The state-of-the-art of text analysis goes well beyond removing superficial differences between terms to address more complex issues like language-specific parsing, part-of-speech tagging, and lemmatization. Don't worry if you're not familiar with some these terms as we'll cover them in more detail below. What's important is that Solr has an extensive framework for doing basic text analysis tasks, such as removing very common words, known as stop words, as well as doing more complex analysis tasks. To accommodate such power and flexibility, Solr's text analysis framework can seem overly complex and daunting to new users. As we like to say, Solr makes solving very difficult text analysis problems possible and simple tasks a little too cumbersome. However, after working through this chapter, we're confident that you'll be able to harness this powerful framework to analyze most any content you'll encounter.

The main goal of this chapter is demonstrate how Solr approaches text analysis and to help you think about how to construct analysis solutions for your documents. To this end, we'll tackle a somewhat complex text analysis problem to demonstrate the mechanics and strategies you need to be successful. Specifically, we'll cover these fundamental components of text analysis with Solr:

- Basic elements of text analysis in Solr: analyzer, tokenizer, and chain of token filters
- Defining a custom field type in schema.xml to analyze text during indexing and query processing
- Common text analysis strategies such as removing stop words, case-folding, synonym expansion, and stemming.

Once we have a solid understanding of the basic building blocks, we'll tackle a harder analysis problem to exercise some of the more advanced features Solr provides for text analysis. Specifically, we'll see how to analyze micro-blog content from sites like Twitter. Tweets present some unique challenges that require us to think hard about how users will use our search solution. Specifically, we show you how to:

- Collapse repeated characters down to a maximum of two in terms like "yummm"
- Preserve #hastags and @mentions
- Index non-English documents using language detection
- Use a custom token filter to resolve shortened bit.ly style URLs

6.1 Analyzing micro-blog text

Let's continue with the example micro-blog search application we introduced in chapter 5. To recap, we're designing and implementing a solution to search micro-blogs from popular social media sites like Twitter. Since the main focus of this chapter is on text analysis, let's take a closer look at the text field in our example micro-blog document. Here is the text we want to analyze:

```
#Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach...
Learning text analysis with #SolrInAction by @Manning on my i-Pad
```

As was discussed in the introduction above, a primary goal of text analysis is to allow your users to search using natural language without having to worry about all the possible forms of their search terms. In Figure 6.1, the user searched the text field for "*San Francisco north beach coffee*", which is a natural query to expect given all the great coffee houses in North Beach; maybe our user is trying to find a great place to drink coffee in North Beach by searching social media content.

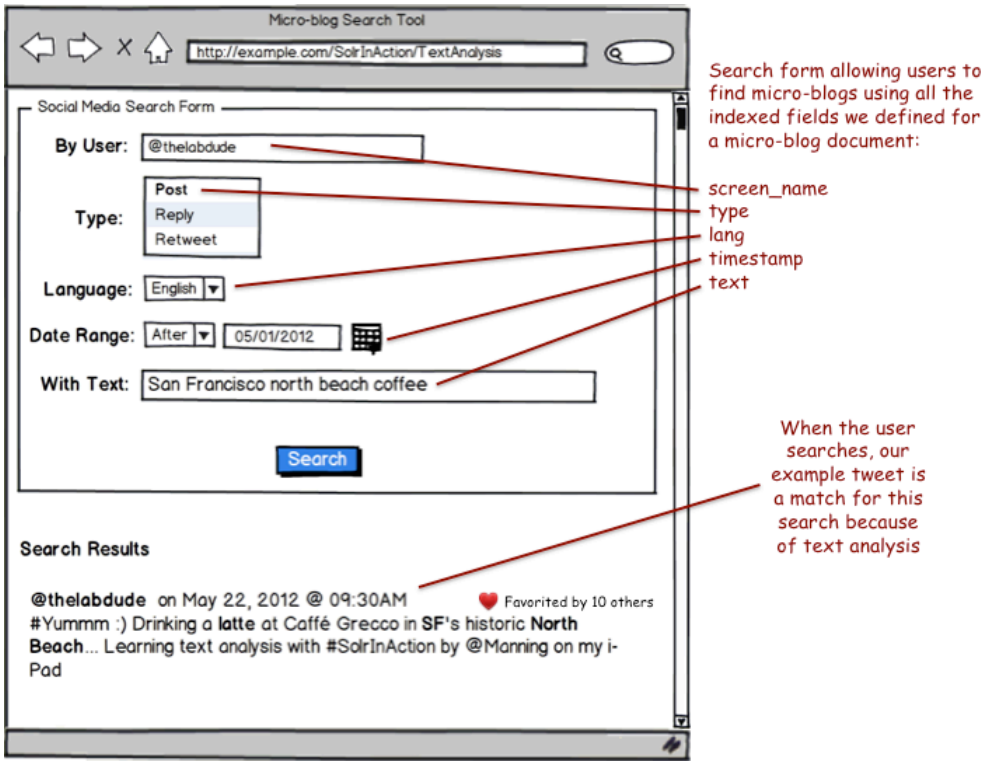


Figure 6.1 Example Web form to find micro-blogs in our example search application from chapter 5.

We assert that our example tweet should be a strong match for this query even though the exact terms "San Francisco", "north beach", and "coffee" do not occur in our sample tweet. Yes, "North Beach" is in our document, but case matters in search unless you take specific action to make your search index case-insensitive. We assert that our example document should be a strong match for this query because of the relationships between terms used in the user's search query and terms in the document shown in table 6.1:

Table 6.1 Query terms that match terms in our example tweet

User Search Term	Terms in Document
San Francisco	SF's
north beach	North Beach
Coffee	latte, Caffé

So the task ahead of us is to use Solr's text analysis framework to transform the tweet text into a form that makes it easier to find. Figure 6.2 shows the transformations we'll make to the text using Solr's text analysis framework, which for now you can think of as a black box. In the remaining sections of this chapter, we'll open up the black box to see how it works.

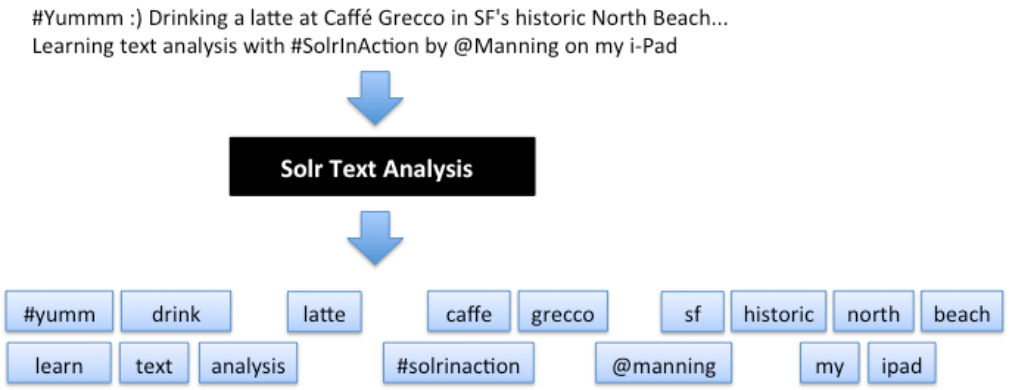


Figure 6.2 Tweet text transformed into a more optimal form for searching. Each box indicates a unique term in the Solr index after text analysis process is applied to the text. The extra space between some terms indicates stop words that were excluded from the index.

Can you spot all the transformations that were applied? Table 6.2 provides a summary of the key transformations that were applied to the text using various Solr text analysis tools that we'll go through in detail below. Notice that each transformation taken individually is usually quite simple but collectively they make a big difference in improved user experience with your search application.

Table 6.2 Overview of the transformations made to the micro-blog text using Solr's text analysis tools. Notice how all transformations are performed using built-in Solr text analysis tools, i.e. no custom code required!

Raw Text	Transformation	How we did it with Solr
all terms	lowercased (SF's -> sf's)	**LowerCaseFilterFactory**
a, at, in, with, by, on	*- removed from text -*	Very common terms called "stop words" removed from the text using the **StopFilterFactory**
Drinking, learning	drink, learn	Stemming with the **KStemFilterFactory**
SF's	sf, san francisco	Apostrophe s ('s) removed by the **WordDelimiterFilterFactory** and sf mapped as a synonym of san francisco using the **SynonymFilterFactory**
Caffé	caffe	Diacritic é transformed to e using the **ASCIIFoldingFilterFactory**
i-Pad	ipad, i pad	Hyphenated word correctly handled using the **WordDelimiterFilterFactory**
#Yummm	#yumm	Collapse repeated letters down to a maximum of two using the **PatternReplaceCharFilterFactory**
#SolrInAction, @Manning	#solrinaction, @manning	Hashtags and mentions correctly preserved using the **WhitespaceTokenizer** and **WordDelimiterFilterFactory**

Don't worry if some of the Solr class names look a bit daunting as we'll cover each one of the tools listed above as we progress through the chapter. For now, let's consider a few of the interesting transformations occurring on this text, all of which are provided by built-in Solr tools.

For example, we use Solr's **ASCIIFoldingFilterFactory** to transform "caffé" into "caffe" which means users won't have to enter the diacritical é when searching. Without this transformation, users searching for "caffe" would not find documents indexed with "caffé". The hyphenated term "i-Pad" is handled by adding two terms to the index using the **WordDelimiterFilterFactory**: "ipad" and "i pad". This means that queries containing "ipad", "i pad", or "i-pad" will all be a match. Of course "iPad" is the correct form, but with search, you want to be as accommodating of simple variations of terms as possible.

Solr also allows you to replace characters and terms using regular expressions. For instance, repeating letters in a word is very common in social media content like tweets in

order to express emotion, e.g. “yummm”. However, from a search perspective, “yummm” and “yumm” are basically equivalent so we can reduce the number of unique terms in our index by collapsing these repeated letters down to a maximum of two. Later in the chapter, we’ll see how to use regular expressions with Solr to make this transformation.

It’s worth highlighting that all of the transformations shown above were provided by built-in Solr tools, meaning that we only had to configure them and didn’t write any Java code. While Solr’s built-in arsenal is powerful, sometimes you need to extend its capabilities. We’ll see how to do this with Solr’s Plug-In framework to deal with bit.ly-style shortened URLs in social media content in section 6.4.3.

At this point, you should have a good feel for where we are headed with text analysis in Solr and might be wondering how to get started. In other words, now that we know Solr provides all these great tools to transform text, how do we actually apply these tools to our documents during indexing? In the next section, we’ll start to work with field types for doing text analysis.

6.2 Basic text analysis

As we learned in chapter 5, the `<types>` section in schema.xml defines `<fieldType>` elements for all possible fields in your documents where each `<fieldType>` defines the format and how the field is analyzed for indexing and queries. The example schema provided with Solr defines an extensive list of field types applicable for most search applications. Of course, if none of the pre-defined Solr field types meets your needs, then you can build your own field type using the Solr Plug-In framework. We’ll see an example of the Solr Plug-In framework in the last section of this chapter.

If all your fields contained structured data like language codes and timestamps, you actually wouldn’t need to use Solr since a relational database is very efficient at indexing and searching “structured” data. Dealing with unstructured text is where Solr really shines. Consequently, the example Solr schema pre-defines a number of powerful field types for analyzing text. Listing 6.1 provides the XML definition for one of the simpler field types named **text_general** as a starting point for analyzing our tweet text:

Listing 6.1 Example field type for analyzing general text

```
<fieldType name="text_general"  #A
           class="solr.TextField"
           positionIncrementGap="100">

  <analyzer type="index"> #B
    <tokenizer class="solr.StandardTokenizerFactory"/> #C

    <filter class="solr.StopFilterFactory" ignoreCase="true"
            words="stopwords.txt"
            enablePositionIncrements="true" />
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>

  <analyzer type="query"> #D
```

```
        <tokenizer class="solr.StandardTokenizerFactory"/>

        <filter class="solr.StopFilterFactory"
                ignoreCase="true" words="stopwords.txt"
                enablePositionIncrements="true" />
        <filter class="solr.SynonymFilterFactory"
                synonyms="synonyms.txt"
                ignoreCase="true" expand="true"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>

    </fieldType>
```

#A Use a short, descriptive name based on the type of data
#B Define the analyzer to use for indexing documents.
#C Tokenizer splits text in a field into tokens
#D Define the analyzer to use for analyzing queries.

Let's break this XML definition down into manageable parts. At the top, you define a `<fieldType>`. For fields that handle text data, you should specify the value of the class attribute to be `solr.TextField`. This tells Solr that you want the text to be analyzed. Also, you should use a name that gives a clue about the type of text that will be analyzed using this field type; for example, `text_general` is a good all-purpose type for when you don't know the language of the text you are analyzing.

ANALYZER

Inside of the `<fieldType>` element, you should define at least one `<analyzer>` that determines how the text will be analyzed. In practice, it's common to define two separate `<analyzer>` elements: one for indexing and another for analyzing the text entered by users when searching; the **text_general** field type uses this approach. Take a moment to think about why you might use different analyzers for indexing and querying. It turns out that you often need to do some additional analysis for processing queries than is needed for indexing a document. For example, adding synonyms is typically done during query text analysis only to avoid inflating the size of your index and to make it easier to manage synonyms. We'll see an example of this approach shortly.

Although you can define two separate analyzers, the analysis applied to query terms must be compatible with how the text was analyzed during indexing. For example, consider the case where an analyzer is configured to lowercase terms during indexing but does not lowercase query terms; users searching for "North Beach" would not find our example tweet because the index contains the lowercased forms "north" and "beach".

TOKENIZER

In Solr, each `<analyzer>` breaks the text analysis process into two phases: tokenization (parsing) and token filtering. Technically, there is also a pre-processing phase before tokenization where you can apply character filters. We'll discuss character filtering in more detail in section 6.3.1, so for now let's concentrate on tokenization and token filters.

In the tokenization phase, text is split into a stream of tokens using some form of parsing. The most basic tokenizer is a WhitespaceTokenizer that splits text on whitespace only. More common is the StandardTokenizer, which performs intelligent parsing to split

terms on whitespace, punctuation and correctly handles URLs, email addresses, and acronyms. To define a tokenizer, you need to specify the Java implementation class of the factory for your tokenizer. For example, to use the common StandardTokenizer, you specify `solr.StandardTokenizerFactory`.

```
<tokenizer class="solr.StandardTokenizerFactory"/>
```

In Solr, you must specify the factory class instead of the underlying Tokenizer implementation class because most tokenizers do not provide a default no-arg constructor. By using the factory approach, Solr gives you a standard way to define any tokenizer in XML. Behind the scenes, each factory class knows how to translate the XML configuration properties to construct an instance of the specific tokenizer implementation class. All tokenizers produce a stream to tokens that can be processed by zero or more filters that perform some sort of transformation of the token.

TOKEN FILTER

A token filter performs one of three possible actions on a token:

- Transformation – Changing the token to a different form such as lowercasing all letters or stemming
- Token injection – Adding a token to the stream, as is done with the synonym filter
- Token removal – Removing terms as is done by the stop word filter

Filters can be chained together to apply a series of transformations on each token. The order of the filters is important as you wouldn't want to have a filter that depended on the case of your tokens listed after a filter that lowercases all tokens.

Let's see this process in action to process our example tweet text, starting with the StandardTokenizer.

6.2.1 StandardTokenizer

At this point, you should have a good understanding of the schema design process and mechanics of defining fields and field types in schema.xml. Let's put this knowledge to work to do some basic text analysis of our example tweet text from chapter 5. The first step in basic text analysis is to determine how to parse the text into a stream of tokens using a tokenizer. Let's start by using the StandardTokenizer, which has been the classic go-to solution for many Solr and Lucene projects because it does a great job of splitting text on whitespace and punctuation but also handles acronyms and contractions with ease. To see this tokenizer in action, let's use it to parse our example tweet:

```
#Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach...
Learning text analysis with #SolrInAction by @Manning on my i-Pad
```

For those familiar with user-generated content on social networks like Twitter, you'll notice that this Tweet is actually quite well formed compared to most but still poses some interesting challenges from a parsing perspective. Figure 6.3 depicts the tokens produced by the StandardTokenizer:

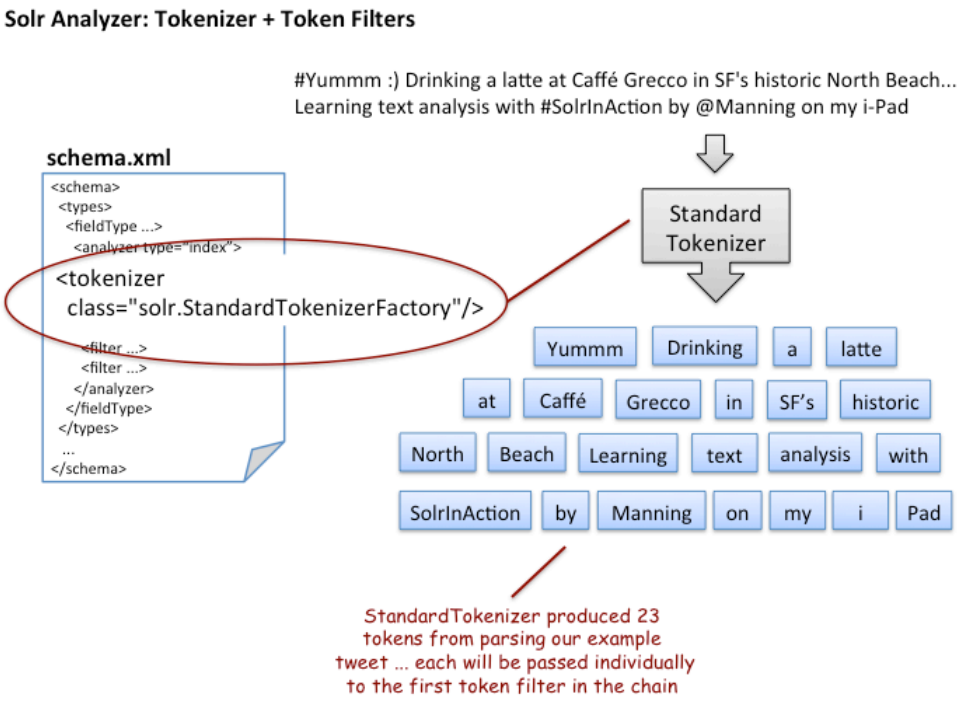


Figure 6.3 Stream of 23 tokens produced by the StandardTokenizer for our example tweet.

Not bad! The StandardTokenizer split the tweet successfully into a stream of 23 separate tokens. Specifically, the StandardTokenizer provides the following features:

- Splits on whitespace and standard punctuation characters such as period, comma, semi-colon, etc. (notice how the ellipsis "…" and emoticon ":)" is removed from the stream)
- Preserves Internet domain names and email addresses as a single token
- Splits hyphenated terms into two tokens, e.g. i-Pad becomes "i" and "Pad"
- Supports a configurable maximum token length attribute, default is 255
- Leading # and @ characters stripped off of hashtags and mentions

Next, let's look at several common token filters provided by Solr to do basic text analysis. Returning to our example tweet, there are a number of issues with the token stream that should be addressed before this text is added to the index. First, there are a few extremely common terms such as "a" or "in" that only serve a grammatical purpose and add little value in differentiating one document from another. In Solr, very common words that occur in

most of the documents in your index are known as **stop words** and can be removed from the token stream easily using the `StopFilterFactory`.

6.2.2 Removing stop words with the StopFilterFactory

Solr's StopFilterFactory removes stop words from the token stream during analysis as they add little value in helping your users find relevant documents. Removing stop words during indexing helps reduce the size of your index and can improve search performance as it reduces the number of documents Solr has to rank for queries that contain stop words. To analyze our example tweet, we defined the StopFilterFactory in listing 6.1 as:

```
<filter class="solr.StopFilterFactory" ignoreCase="true"
                words="lang/stopwords_en.txt"
```

Notice that we specify an English-specific stop word list (`words="lang/stopwords_en.txt"`). Out-of-the-box, Solr provides a basic stop word list that you can customize to meet your needs, see stopwords_en.txt in the **lang** sub-directory under conf (example/solr/conf/lang). Here are the English stop words included in the example Solr server:

```
a an and are as at be but by for if in into is it no not
of on or such that the their then there these they this to was
will with
```

In general, stop word removal is language-specific. If you are analyzing German text, then you will need a different stop word list, containing terms like "die" and "ein". Solr provides custom stop word lists for many languages in the lang sub-directory.

Advanced Approach to Removing Stop Words

Google owns a patent on its approach to handling stop words where they include all stop words during indexing and selectively remove stop words from queries based on comparing sets of documents retrieved with and without using the stop words, see: http://www.google.com/patents/US7945579. Google's patented approach to stop words is a great example of how search providers use advanced text analysis to achieve competitive advantage. Even with something as simple as removing stop words, there is not a one-size fits all solution.

6.2.3 LowerCaseFilterFactory – Lowercase Letters in Terms

This filter lowercases the letters in all tokens, e.g. in our example Manning became manning so we'll match user queries containing: MANNING, Manning, and manning. Defining this filter is trivial:

```
<filter class="solr.LowerCaseFilterFactory"/>
```

As with stop words, it's not always clear whether to apply the lowercase filter to all terms. For example, terms starting with a capital letter in the middle of a sentence typically indicate a proper noun that can greatly improve precision of your results if users seek the proper

noun form of common terms such as “North Beach”; both “north” and “beach” are common terms that appear in other contexts so the capital letter can help improve precision.

For example a user searching for “North Beach” is probably more interested in documents about the popular San Francisco neighborhood and would be less interested in a document containing “the waves are stronger on the **north beach** of the island”. However, using a more nuanced approach to lowercasing terms assumes that your users use the right case when searching. In most cases, you’ll want apply the lowercase filter but still need to determine where in the filter chain to apply it. If you have synonym list in all lowercase, then you’ll need to apply the lowercase filter before your synonym filter.

Figure 6.4 shows the resulting text after applying the stop word and lowercase filters to the example tweet:

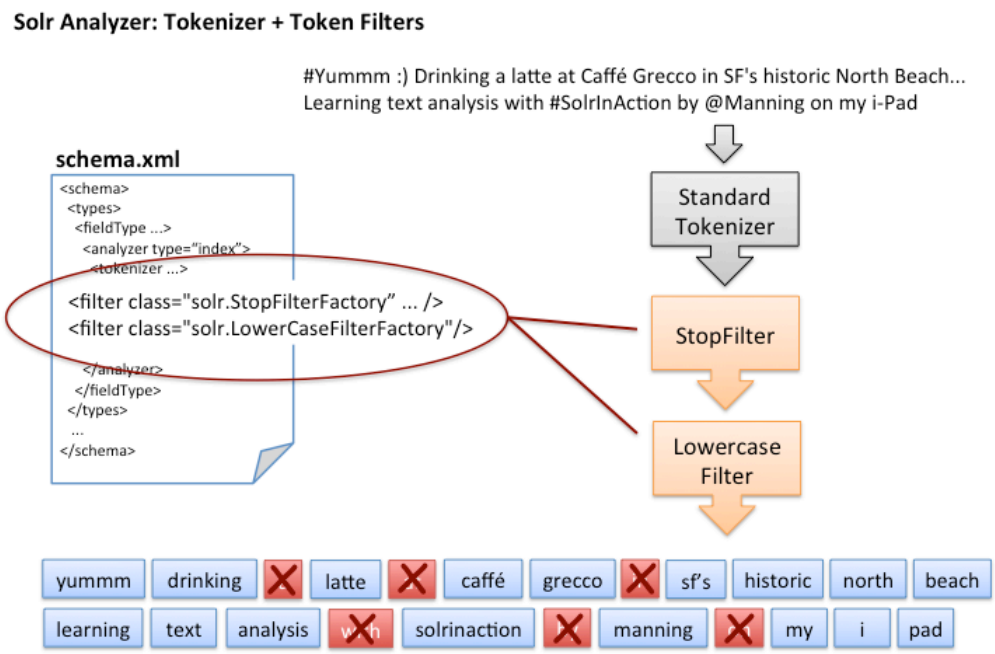


Figure 6.4 Resulting text to be indexed after splitting using the StandardTokenizer and applying the stop word and lowercase filter. The terms with red (x) are the stop words and would not be included in your index.

So now we have a basic text analysis solution for our sample tweet. Let’s apply what we learned so far to see some actual text analysis in action. Solr provides a simple form that

allows you to test your text analysis configuration on sample text without having to add a document to the index. The form also allows you to see if a query would match a sample document without having to actually index the document.

6.2.4 Testing your Analysis with Solr's Analysis Form

Let's try a query "San Francisco coffee" to see if it matches our example tweet as we would expect. The easiest way to get started is to use Solr's admin console, which provides a simple Web form to analyze your content. With the server running, navigate to the admin console at http://localhost:8983/solr/#/ and then click on collection1 as shown in the screen shot in figure 6.5:

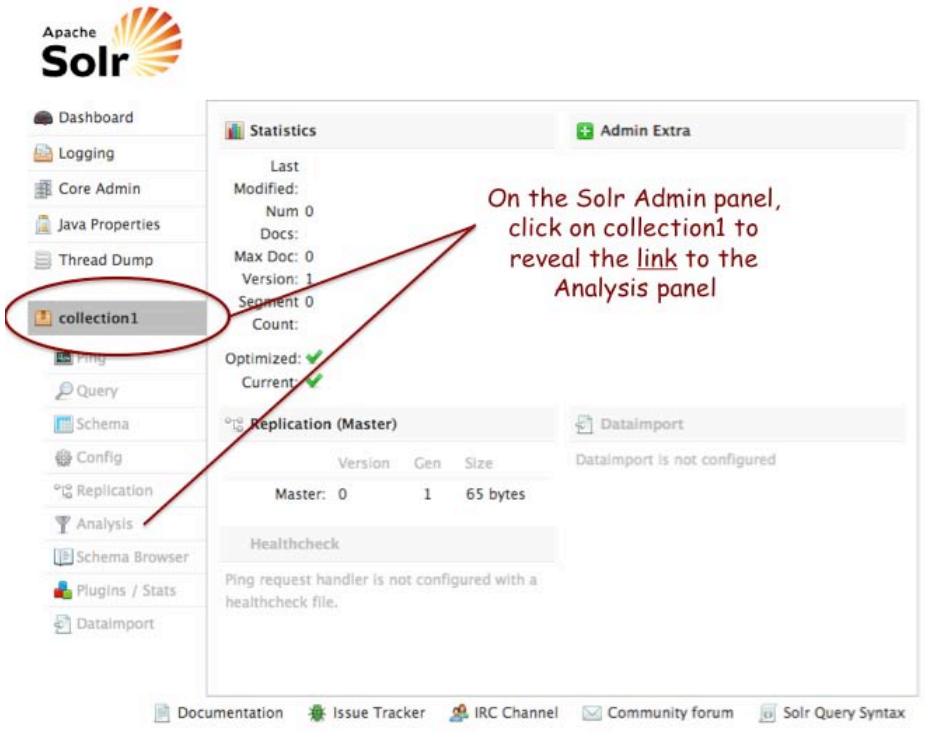


Figure 6.5 How to find the link to the Analysis form from the Solr admin panel

Once the form loads in your browser, enter the sample tweet text in the **Field Value (index)** text box and *"text_general"* in the **Field Type** box as seen in figure 6.6:

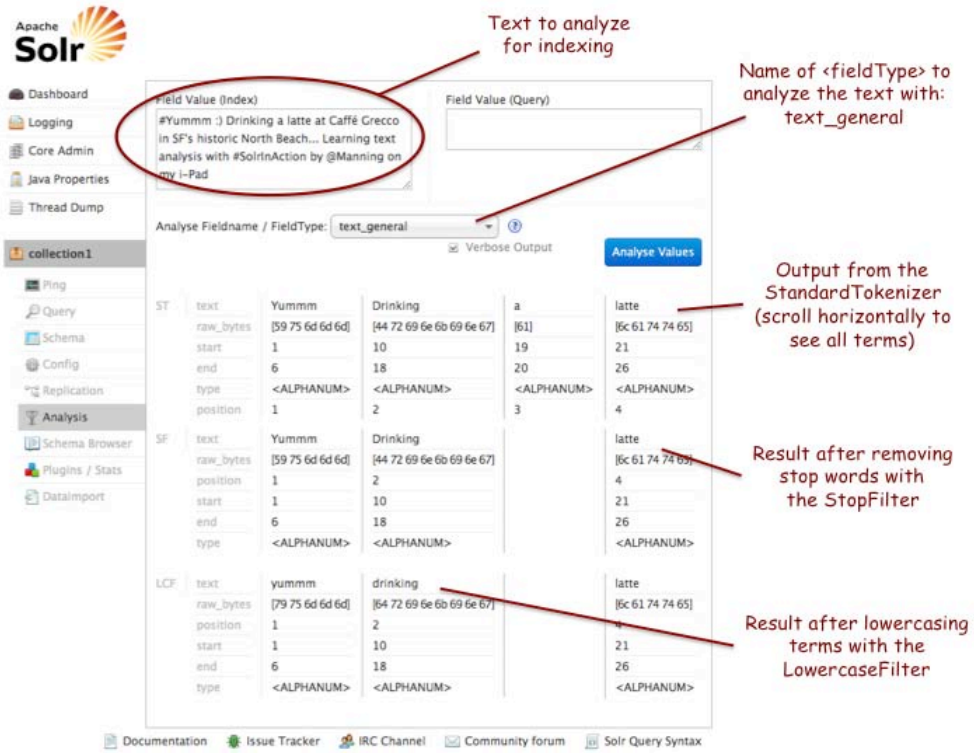


Figure 6.6 Analysis form showing the step-by-step process Solr applies to analyzing a document using the selected FieldType: text_general.

Once you enter the field type and text to analyze, click on the **Analyse Values** button to see the result. Below the form, Solr reports the steps it takes to analyze the text for indexing using the **text_general** field type. Notice how the text is first parsed with the StandardTokenizer, abbreviated as ST and then each token passes through the StopFilter (SF) and LowercaseFilter (LCF).

Next, let's try a query against our example tweet to see if it is a match; enter "drinking a latte" in the text box labeled **Field Value (query)** as shown in the screen shot in figure 6.7. When you click on the **Analyse Values** button, Solr will highlight any terms in the document that match the query, in this case: "drinking" and "latte".

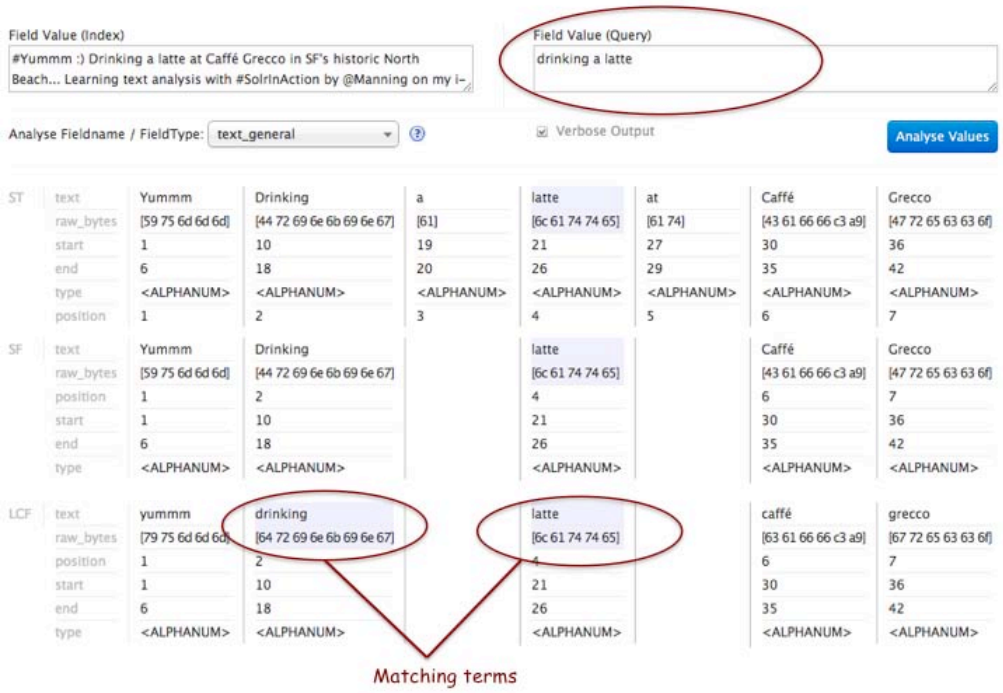


Figure 6.7 Search for "drinking a latte" on the analysis panel and Solr will highlight matching terms "drinking" and "latte" in the example document.

Next, enter "*San Francisco drink cafe ipad*" and click on the **Analyse Values** button again. Although this is a nonsensical query, we would expect our sample document to be a match. We're using this nonsensical query to demonstrate that seemingly small differences in terms can lead to highly relevant documents being missed by your users. Case in point, none of the query terms match the example document! The problem of course is that while it's easy for a human to see the similarity between the terms in the query and the example document, to Solr, the terms have no relation to each other! We will use better text analysis to overcome this mismatch.

Overall, as a first pass, we resolved a number of text parsing and basic analysis issues with very little effort. On the other hand, there are a number of outstanding issues that will make finding this tweet difficult for your users. How many potential issues do you see in the text? Compare your results to figure 6.8:

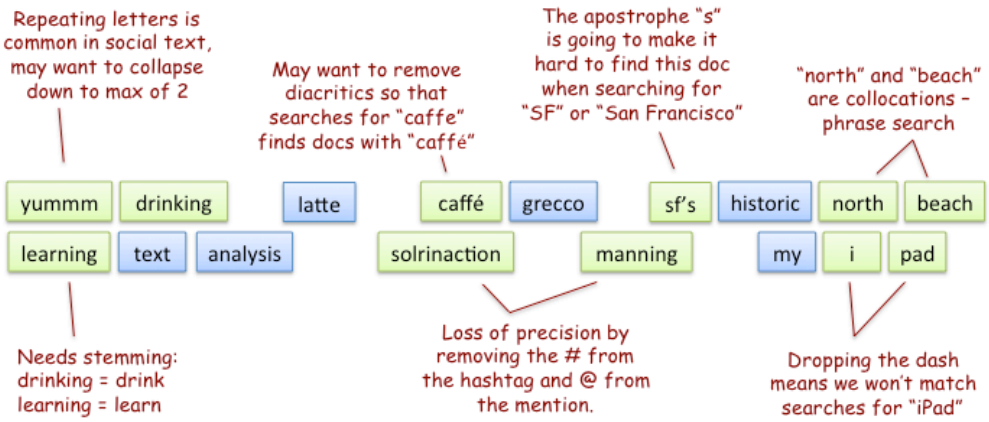


Figure 6.8 Remaining text analysis issues with our sample tweet after applying the lowercase and stop filters.

Unless you are indexing tweets or content from other social media sources, you may not encounter any of the analysis issues shown in figure 6.8. In general, the main point is that you need to study a representative sample of the documents in your index to determine the type of analysis needed as we've done here.

At this point, it should be clear that the basic text analysis provided by the **text_general** general field type is not sufficient to meet our needs. Consequently, we need to implement a new custom field type building on the tools we've already learned about as well as learning a few new ones.

6.3 Defining a custom field type for micro-blog text

We left off having made good progress with analyzing social media text, but there were a few nagging issues, depicted in figure 6.8, that we need to address. In this section, we address these remaining issues by introducing a few more of Solr's built-in text analysis tools. To begin, since none of the predefined field types meet all of our needs, we will define a new custom field type in schema.xml. Under the `<types>` element in schema.xml, add the following new type named **text_microblog**, as shown in listing 6.2.

Listing 6.2 Custom field type for analyzing micro-blog text

```
<types>  #A
  ...
  <fieldType name="text_microblog" class="solr.TextField"
           positionIncrementGap="100">
  <analyzer type="index">                    #B
    <charFilter class="solr.PatternReplaceCharFilterFactory"
                pattern="([a-zA-Z])\1+"
                replacement="$1$1"/>
```

```
      <tokenizer class="solr.WhitespaceTokenizerFactory"/>
      <filter class="solr.WordDelimiterFilterFactory"
              generateWordParts="1"
              splitOnCaseChange="0"
              splitOnNumerics="0"
              stemEnglishPossessive="1"
              preserveOriginal="0"
              catenateWords="1"
              generateNumberParts="1"
              catenateNumbers="0"
              catenateAll="0"
              types="wdfftypes.txt"/>
      <filter class="solr.StopFilterFactory"
              ignoreCase="true"
              words="lang/stopwords_en.txt"
              enablePositionIncrements="true"
              />
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.ASCIIFoldingFilterFactory"/>
      <filter class="solr.KStemFilterFactory"/>
    </analyzer>
    <analyzer type="query">        #C
      <charFilter class="solr.PatternReplaceCharFilterFactory"
                  pattern="([a-zA-Z])\1+"
                  replacement="$1$1"/>
      <tokenizer class="solr.WhitespaceTokenizerFactory"/>
      <filter class="solr.WordDelimiterFilterFactory"
              splitOnCaseChange="0"
              splitOnNumerics="0"
              stemEnglishPossessive="1"
              preserveOriginal="0"
              generateWordParts="1"
              catenateWords="1"
              generateNumberParts="0"
              catenateNumbers="0"
              catenateAll="0"
              types="wdfftypes.txt"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.ASCIIFoldingFilterFactory"/>
      <filter class="solr.StopFilterFactory"
              ignoreCase="true"
              words="lang/stopwords_en.txt"
              enablePositionIncrements="true"
              />
      <filter class="solr.KStemFilterFactory"/>
      <filter class="solr.SynonymFilterFactory" #D
              synonyms="synonyms.txt"
              ignoreCase="true"
              expand="true"/>
    </analyzer>
  </fieldType>
</types>
```

#A New field type is added below the existing field types inside of <types>
#B Analyzer for indexing documents
#C Analyzer for processing user queries
#D Synonym processing is performed on the query side, not during indexing

You should recognize the structure and some of the elements in this field type definition, but there are also few new ones that we haven't discussed yet. Table 6.3 provides an overview of the new tools we'll cover in this section.

Table 6.3 List of additional Solr text analysis tools needed to analyze micro-blog text

Solr Tool	Description
PatternReplaceCharFilterFactory	Use regular expression to replace characters before tokenizing
WhitespaceTokenizerFactory	Split text on whitespace only
WordDelimiterFilterFactory	Intelligently split tokens on punctuation, case change, and handle special characters like # and @ on hashtags and mentions
ASCIIFoldingFilterFactory	Transform diacritics into their ASCII equivalent if possible
KStemFilterFactory	Stemming on English text; less aggressive than the Porter stemmer
SynonymFilterFactory	Inject synonyms for common terms into queries

Don't worry if this list of complicated names looks overwhelming, as we'll work through each of these tools in the sections below. Let's start with a solution for removing repeated letters from words like "yummm" using a regular expression.

6.3.1 Collapsing Repeated Letters with PatternReplaceCharFilterFactory

In Solr, a CharFilter (or character filter) is a pre-processor on an incoming stream of characters before they are passed on to the tokenizer for parsing. Much like token filters, CharFilters can be chained together to add, change, or remove characters from text. In Solr 4, there are three built-in CharFilters:

- **solr.MappingCharFilterFactory**: Applies replacements of characters defined in an external configuration file.
- **solr.PatternReplaceCharFilterFactory**: Uses a regular expression to replace characters with an alternative value.
- **solr.HTMLStripCharFilterFactory**: Strips HTML markup from text.

As with most Solr features, you can implement your own using the Plug-In framework. Of the three filters, the PatternReplaceCharFilterFactory seems most appropriate for our current text analysis needs as tweets typically do not have HTML embedded in them and we do not need to map any characters. Consequently, we won't show how to use the MappingCharFilterFactory or HTMLStripCharFilterFactory filters in these chapters, so please see the Solr wiki for more details. Let's see how to apply the PatternReplaceCharFilterFactory to address a few of the issues with our example tweet.

The **solr.PatternReplaceCharFilterFactory** is used to filter characters using regular expressions. To configure this factory, you need to define two attributes: pattern and replacement. The pattern is a regular expression that identifies the characters we want to replace in our text. The replacement attribute specifies the value you want to replace the matched characters with. Don't worry if you're not a regular expression expert, in most cases you can find the expression you need online using Google or similar search engine.

Let's use this `<charFilter>` to solve those pesky terms with repeated letters like yummm. To collapse repeated letters down to a maximum of 2, we need a regular expression that identifies sequences of repeated letters, so `([a-zA-Z])\1+` will work nicely. In regex speak, the `[a-zA-Z]` is a character class that identifies a single letter in lower or uppercase. The parenthesis around the character class identifies the matching letter as a captured group. The `\1` part is called a numbered backreference that matches a repeat of the first group and the + part says the repeated letter can occur 1 or more times.

So that covers the pattern to match, what about the replacement? Our goal is to collapse repeated letters down to a maximum of two so what we need is a way to address the part of a term that matches `([a-zA-Z])`. With regular expressions, the ([a-zA-Z]) part of our expression is called a captured group and is addressable as $1. Thus, our replacement value is simply $1$1. For example, in yummm, ([a-zA-Z]) evaluates to the first "m" and the entire expression evaluates to "mmm", so "mmm" gets replaced to "mm". To make this work in our text_microblog fieldType, you simply add the XML shown in listing 6.3.

Listing 6.3 Define a charFilter to collapse repeated letters using regular expression

```
<charFilter class="solr.PatternReplaceCharFilterFactory"
            pattern="([a-zA-Z])\1+"
            replacement="$1$1"/>
```

6.3.2 Preserving Hashtags, Mentions, and Hyphenated Terms

It turns out that a few of the issues we are having with the example tweet are caused by the StandardTokenizer. Specifically, the StandardTokenizer is stripping the # and @ characters from hashtags and mentions respectively. In addition, it's splitting hyphenated terms into two tokens, such as "i-Pad" becoming "i" and "Pad". Unfortunately, the means that searches for "iPad" won't match our example document as we saw in the previous section. In terms of hashtags and mentions, you might wonder why we care about preserving those special characters on the front?

In the previous section, we saw that the @ was removed from @Manning by the StandardTokenizer, which means that we've lost some information about this term. Specifically, @Manning used in a social context has a very special meaning in that it identifies a specific social account, in this case the one used by Manning Publishers. This is very different than a tweet about Peyton Manning. Also, if we apply stemming, then "manning" will become "man". This is an extreme case in that our example tweet will be a match for queries with the term "man" in them!

The same goes for hashtags. For example, #fail is a hashtag commonly used to denote a person's dissatisfaction with another person, place or thing such as a brand. So if you wanted to find all tweets with #fail, then you probably don't want to match tweets with just "fail", as in "*I partied too late last night, I hope I don't fail today's mid-term exam*" Thus, during text analysis, you want to preserve that fact that #fail is different than just fail by preserving the leading # character. In general, we want to preserve hashtags and mentions so that we have flexibility to differentiate between documents with #fail and documents with just fail. The general lesson here is that sometimes you need to preserve context about special terms during text analysis.

So hopefully your convinced that we need to preserve the # and @ signs on our text. Now we just need to figure out how to do it. First, let's make sure we understand how they are being removed. The StandardTokenizer uses word delimiters to split text into tokens and is treating the # and @ signs as word delimiters. If you are a Java developer, then your first inclination might be to extend StandardTokenizer and override the behavior that strips off these two special characters. Unfortunately, the StandardTokenizer is not easy to extend and more importantly, we can do this without writing any custom code! And, we get a chance to learn about two more text analysis tools in Solr in the process, namely the WhitespaceTokenizerFactory and the WordDelimiterFilterFactory.

WHITESPACETOKENIZERFACTORY

The WhitespaceTokenizerFactory is a very simple tokenizer that splits text on whitespace only. If you recall, the StandardTokenizer split our example tweet into 23 separate tokens. In contrast the whitespace tokenizer produces a different set of tokens, as depicted in figure 6.9:

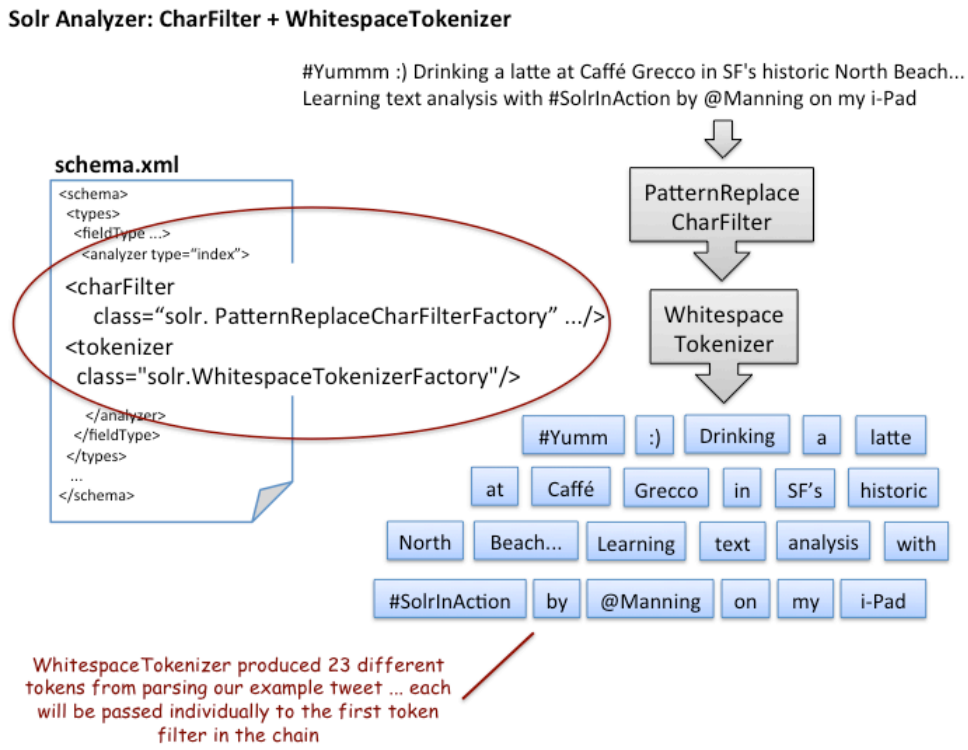


Figure 6.9 Tokens produced by the WhitespaceTokenizer; still 23 but some are different than what was produced by the StandardTokenizer

Progress? Maybe. So we solved our hashtag, mention, and hyphenated term issue but we introduced a few more issues in the process. Specifically, the emoticon ":)" is now a token and the ellipsis "…" is included as part of "Beach…". Luckily, with Solr, these issues are easily resolved using the WordDelimiterFilterFactory.

WORDDELIMITERFILTERFACTORY

To complement WhitespaceTokenizer's simplistic approach to splitting on whitespace, the WordDelimiterFilterFactory offers a powerful solution to resolving most issues caused by splitting on whitespace. At a high-level, this filter splits a token into sub-words using various parsing rules. We'll cover these rules in more detail shortly. First let's see how this filter helps us preserve the special characters on our hashtags and mentions. For our example, we configured the WordDelimiterFilterFactory using the following options shown in listing 6.4:

Listing 6.4 Define WordDelimiterFilterFactory with options set to preserve # and @

```
<filter class="solr.WordDelimiterFilterFactory"
        generateWordParts="1"
        splitOnCaseChange="0"
        splitOnNumerics="0"
        stemEnglishPossessive="1"
        preserveOriginal="0"
        catenateWords="1"
        generateNumberParts="0"
        catenateNumbers="0"
        catenateAll="0"
        types="wdfftypes.txt"/> #A
```

#A Define custom character types for splitting words in a file called wdfftypes.txt

It turns out that by default this filter will also strip off the # and @ characters from hashtags and mentions. However, unlike the StandardTokenizer, the WordDelimiterFilter provides any easy way to customize which characters it treats as word delimiters using a simple "types" mapping file, which in our example is "wdfftypes.txt". Our wdfftypes.txt file contains two mappings:

```
\# => ALPHA
@ => ALPHA
```

These settings map both the # and @ characters to the ALPHA class which means our instance of the WordDelimiterFilter won't treat them as word delimiters. The leading backslash on the hash sign is so that Solr won't interpret that line as a comment when reading the wdfftypes.txt file. With this simple mapping, hashtags and mentions are preserved in our text.

The WordDelimiterFilter also handles hyphenated terms in a robust manner. In our example tweet, the author incorrectly used "i-Pad" instead of "iPad"; what would be nice is for our tweet to match all possible forms used in queries, ie. "i Pad", "i-Pad", and "iPad". Think for a minute about what needs to happen during indexing and query analysis to ensure all three forms produce matches. Hopefully you figured out that the WordDelimiterFilter needs to split i-Pad on the hyphen into two tokens "i" and "Pad", but also injects a new token "iPad" into the stream. This is the exact behavior you get from the WordDelimiterFilter when you set `generateWordParts=1` and `catenateWords=1`.

In general, the WordDelimiterFilter provides a number of options used to fine-tune the transformations it makes on your tokens. Table 6.4 gives you an overview of how each of these options works.

Table 6.4 Configuration options for the WordDelimiterFilterFactory

Attribute	Behavior if enabled (="1")	Default
generateWordParts	Splits words using built-in parsing rules and other options to create sub-word parts	enabled (1)
splitOnCaseChange	Splits camel-cased terms when a change in letter case is encountered during parsing, e.g. SolrInAction is split into three tokens: Solr In Action	enabled (1)
splitOnNumerics	Splits terms having a mixture of letters and numbers when a number is encountered, e.g. R2D2 is split into four tokens: R 2 D 2	enabled (1)
stemEnglishPossessive	Removes the apostrophe s from a term, e.g. SF's becomes SF	enabled (1)
preserveOriginal	Includes the original token in the text in addition to any other tokens produced by this filter; e.g. SF's would be included as well as SF	disabled (0)
catenateWords	Concatenates sub-word parts into a single token, e.g. i-Pad would be split at the hyphen into two tokens "i" and "Pad" and then concatenated into "iPad" to produce a third token	disabled (0)
generateNumberParts	Splits tokens with numeric data separated by punctuation like dashes into multiple tokens, e.g. a phone number of 867-5309 would be split into two terms 867 and 5309	enabled (1)
catenateNumbers	If a number token was split, then concatenate the parts into a single term; keeping with our 80's reference, 867-5309 would be split into three tokens: 867, 5309, and 8675309	disabled (0)
catenateAll	When using generateWordParts="1" and generateNumberParts="1", concatenate all parts into a single token	disabled (0)

It may take a little experimentation to get a feel for how the WordDelimiterFactory works with your content. We recommend using Solr's analysis form to experiment with these settings as we did in section 6.2.4. For now, we've solved the problems with hashtags, mentions, and hyphenated terms, so let's turn our attention to how to handle terms with accent marks like caffé.

6.3.3 Removing Diacritcal Marks using the ASCIIFoldingFilterFactory

In search, it's often the case that doing simple things can make a big difference. This is the case when handling characters that have diacritical marks on them, such as "caffé" in our example or "jalapeño". In most cases, you can't be sure users will type characters with the diacritical mark when searching, so Solr provides the **ASCIIFoldingFilterFactory** to transform characters into their ASCII equivalent, if available. In schema.xml, you can include this filter in your analyzer by adding:

```
<filter class="solr.ASCIIFoldingFilterFactory"/>
```

It's best to list this filter after the lowercase filter so that you only have to work with lowercase characters.

6.3.4 Stemming with the KStemFilterFactory

Stemming transforms words into a base form using language-specific rules. Solr provides a number of stemming filters, each with their own strengths and weaknesses. For now, we'll use a filter based on the Krovetz Stemmer: solr.**KStemFilterFactory**. This stemmer is less aggressive in its transformations than other popular stemmers like the PorterStemmer. We'll discuss stemming and lemmatization in more depth in chapter 18. For now, we apply the KStemFilterFactory to remove "ing" from terms like "drinking" and "learning"[1].

```
<filter class="solr.KStemFilterFactory"/>
```

Table 6.5 shows some examples of stemming applied to terms using the KStemFilterFactory and the PorterStemFilterFactory.

Table 6.5 Comparing stems produced by the KStemmer and Porter algorithms

Original Term	Stem (KStemmer)	Stem (Porter)
drinking	drink	drink
requirements	requirement	requir
operating	operate	oper
operative	operative	oper
wedding	wedding	wed
@manning	@manning	@manning

It should be clear from these few examples that the Porter stemming algorithm is much more aggressive than the KStemmer. The problem with being too aggressive is that your search application may end up matching documents that have little to do with a user's query. For example, if a user searches for "wedding in July", the term "wedding" is stemmed to

[1] See: R. Krovetz, 1993: "Viewing morphology as an inference process," in R. Korfhage et al., Proc. 16th ACM SIGIR Conference, Pittsburgh, June 27-July 1, 1993; pp. 191-202.

"wed" using the Porter stemmer, which is also a common abbreviation for Wednesday. Also notice that "operating" and "operative" both stem to "oper" using Porter so documents containing "covert operative" and "operating system" will both be a match for a query for "operating capital". In general, a stemmer expands the set of documents that match a query but can negatively impact precision of the results.

6.3.5 Injecting Synonyms at Query Time with the SynonymFilterFactory

This filter injects synonyms for important terms into the token stream. For example, you may want to inject "home" into the token stream when you encounter "house". In most cases, synonyms are injected only during query time analysis. This helps reduce the size of your index and is easier to maintain changes to the synonym list. If you inject synonyms during indexing and you discover a new synonym for one of your terms, then you will have to re-index all of your documents to apply the change. On the other hand, if you only inject synonyms during query processing, new synonyms can be introduced without re-indexing. Here's how to define the SynonymFilter in your schema.xml:

```
<filter class="solr.SynonymFilterFactory"
                synonyms="synonyms.txt"
                ignoreCase="true" expand="true"/>
```

This filter is best applied to the query analyzer only. You also need to consider where to put it in the chain of filters. To determine this, you need to think about what transformations need to take place before you match synonyms for terms. We think it makes the most sense to list this filter last for your query analyzer so that your synonym list can assume all other transformations have already taken place on a token. Consider if we apply the ASCIIFoldingFilter after our synonym filter; this means that our synonym list will need to include the diacritics, such as caffé.

In our example, there are a few tokens that could benefit from synonyms including: SF, latte, and caffe. To map synonyms for these terms in Solr, add the following to the `synonyms.txt` file identified in your filter definition:

```
sf,san fran,sanfran,san francisco
latte,coffee
caffe,cafe
```

Looking at this, you might think that's great Solr provides a way to do this mapping but who would want to manually configure thousands of synonyms? A fair point and currently, this is a problem in Solr. There are some solutions available as community extensions to Solr, such as a utility to convert the WordNet database of English synonyms into Solr's format.

6.3.6 Putting it all together

After applying these additional filters to our example tweet, we are left with the following text, shown in figure 6.10:

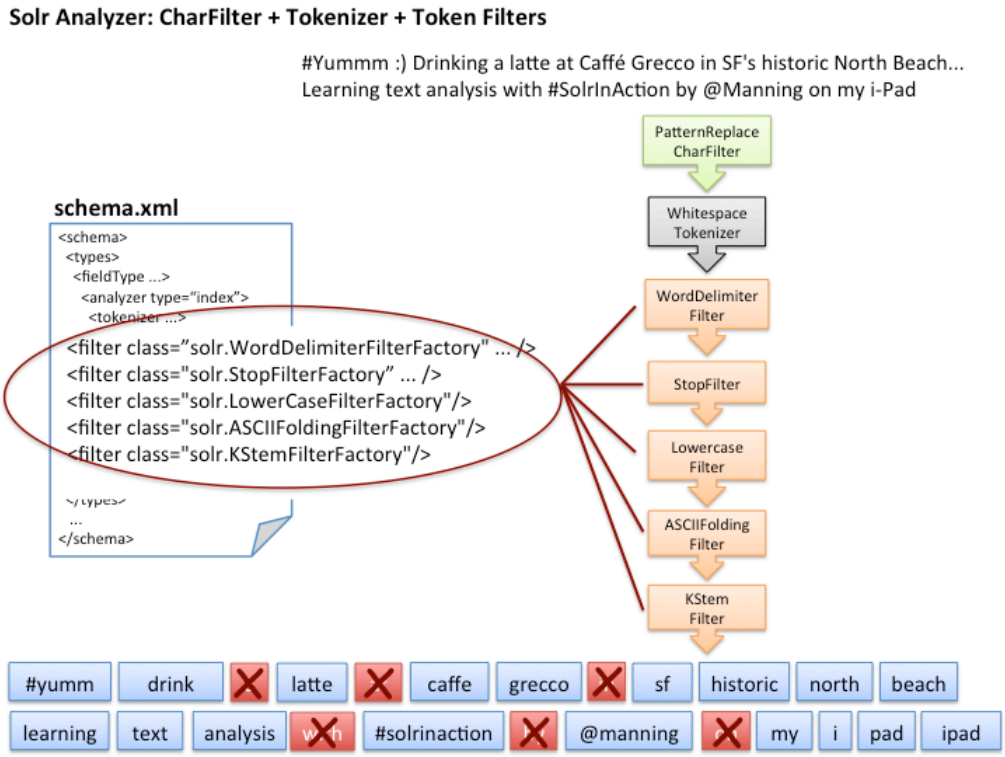


Figure 6.10 Results after applying the new custom **text_microblog** <fieldType> to the tweet text.

So now let's return to the Solr Analysis form to see if the previous query we tried is a match. Recall the query "*San Francisco drink cafe ipad*" was not a match when we used the simple **text_general** field type in section 6.2.4. However, using the **text_microblog** field type, our example document is a strong match for this query as shown in figure 6.11 below.

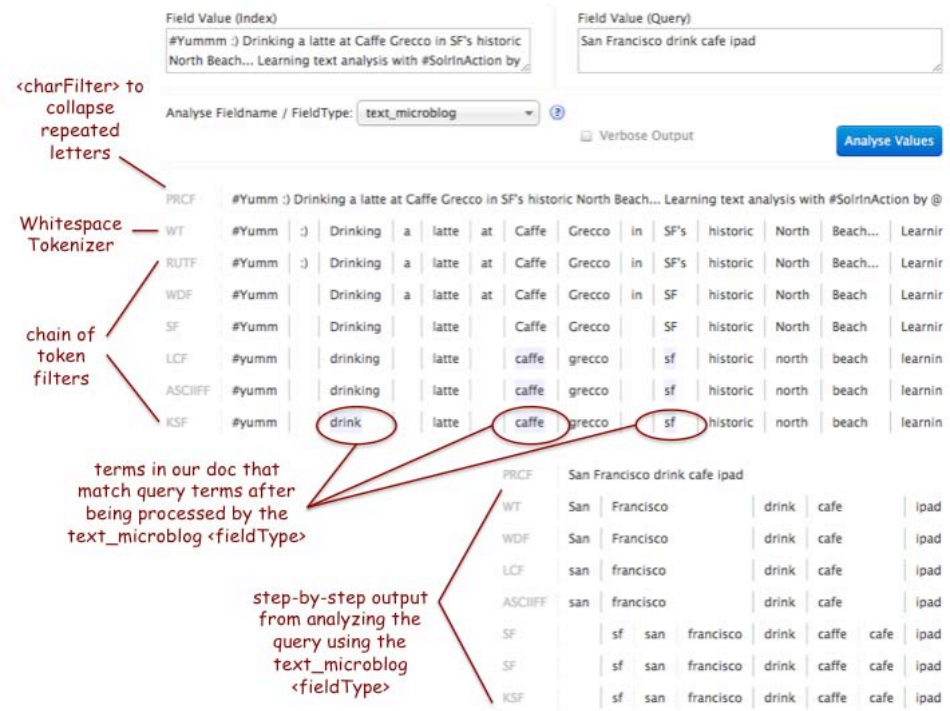


Figure 6.11 Solr Analysis form showing the results of analyzing our example tweet and a query with the text_microblog <fieldType>

The query terms shown in table 6.6 are now matches to terms in our example tweet based on text analysis.

Table 6.6 Query terms matching our example tweet based on text analysis

Query Term	Matching Term in Document	Explanation
San Francisco	SF's	WordDelimiterFilter removes the 's from SF's and the SynonymFilter injects sf as a synonym for "San Francisco" during query text analysis
drink	Drinking	LowercaseFilter changes "D" to "d" and the KStemFilter removes the "ing"
cafe	Caffé	LowercaseFilter changes "C" to "c", ASCIIFoldingFilter changes "é" to "e" and SynonymFilter injects "caffe" as synonym for "cafe" during query processing

iPad	i-Pad	WordDelimiterFilter splits i-Pad on the hyphen and then creates iPad as a new token by concatenating the split terms into one

Of course this query is a bit contrived for example purposes. The key take-away is that Solr provides a wealth of built-in text analysis tools that allow you to implement flexible solutions to handle complex text. The ultimate goal being a search application that makes it easier for your users to find relevant documents using natural language without having to think about linguistic differences in text.

If you are interested in analyzing micro-blog text or just want to see the text_microblog field type in action, then you can apply the `ch6/schema.xml` from the source code provided with the book to your local Solr server. After applying the schema.xml, restart your Solr server and then use the post.sh script to add the tweets.xml document to your server.

```
cd SolrInAction/example-docs
./post.sh ch6/tweets.xml
```

You should see output similar to:

```
[~/dev/projects/SolrInAction/example-docs]$ ./post.sh ch6/tweets.xml
Posting file tweets.xml to http://localhost:8983/solr/collection1/update
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader"><int name="status">0</int><int
name="QTime">140</int></lst>
</response>

<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader"><int name="status">0</int><int
name="QTime">67</int></lst>
</response>
```

After indexing the documents, you can search for `catch_all:@thelabdude` and you should see the tweet examples we used in this chapter in the search results. Feel free to experiment with various queries to see Solr text analysis in action. Let's now turn our attention to more advanced topics in text analysis.

6.4 Advanced text analysis

Throughout this chapter we built a solution to parse and analyze very complex social media text without writing a single line of code. We covered the main tools Solr provides but in reality we've only touched on the most common tools available. Before we wrap up this chapter, we want to provide you with an overview of some of the advanced techniques you can use to fine-tune your analysis. Specifically, we cover the following topics:

- Advanced options for text fields in schema.xml
- Multi-lingual text analysis and language detection
- Extending text analysis using Solr's Plug-In Framework

Let's begin by looking at some of the more advanced attributes you can apply to text fields in schema.xml.

6.4.1 Advanced field attributes

Table 6.7 lists the advanced attributes you can set for `<field>` elements along with a short description of what to expect when the attribute is enabled. We think it is important to be aware of these options, as you will undoubtedly encounter them when looking at the Solr example schema.xml. Also, you should be aware that each of these advanced attributes only apply to text fields and do not impact non-text fields like dates and strings.

Table 6.7 Overview of advanced attributes for field elements in schema.xml

Attribute	Behavior when Enabled (="true")	Default Value
omitNorms	Disables length normalization and index-time boosting for the field, which helps save storage in your index. Norms help give shorter documents a little boost during relevance scoring. Thus, if most of your documents are of similar size, then you could consider omitting norms to help reduce the size of your index. However, you must not omit norms (`omitNorms="false"`) if you need index-time boosts on a field as Solr encodes the boost into the norm value. Norms are omitted by default for primitive types such as dates, strings, and numeric fields.	false
termVectors	Solr provides a MoreLikeThis feature to find documents that are similar to a specific document. The MoreLikeThis feature requires term vectors to be enabled so that Solr can compute a similarity measure between two documents. Storing term vectors can be expensive for large indexes, so only enable these if you really need them.	false
termPositions	Commonly used to improve the performance of the hit highlighting feature in Solr; covered in depth in Chapter 8. Enabling term positions increases the size of your index.	false
termOffset	Commonly used to improve the performance of the hit highlighting feature in Solr; covered in depth in Chapter 8. Enabling term positions increases the size of your index.	false

OMITTING FIELD NORMS

Let's take a little closer look at the optional `omitNorms` attribute as that is a common source of confusion for new Solr users. As we discussed in chapter 3, a norm is floating-point value (Java float) based on a document length norm, document boost, and field boost. Under the covers, Lucene encodes this floating point value into a single byte, which if you think about it is pretty cool. The document length norm is used to boost smaller documents. Without going into too much detail, Lucene gives smaller documents a slight boost over longer documents to improve relevance scoring. Conceptually, if a query term matches a short document and a long document, both containing the term once, Lucene considers the

short document to be slightly more relevant to the query because the matching term has more weight in the short document than it does in the long document. In this case, term weight is just the term frequency (1) divided by the total number of terms in the document (N). For a short document with 10 terms, the weight would be 0.10 and for a long document with 1000 terms, the weight would be 0.001. Thus, Lucene gives the shorter document a slight boost, which is encoded in the norm.

It stands to reason that if your documents are of similar length and you are not using field and document boosts at index time, then you can set `omitNorms=true` to save memory during searching. However, when just starting out, we recommend that you use the default value (`omitNorms=false`) so that results ranking benefits from the document length normalization. Let's take a quick look at another advanced attribute for fields, which is used to improve the performance of document similarity calculations.

TERM VECTORS

In chapter 3, we discussed how Solr uses term vectors to compute the similarity between documents and queries. Solr also provides a feature to compute a similarity between documents, commonly known as **More Like This**. The **More Like This** feature in Solr finds documents in the index that are very similar to a specific document. Under the covers, this feature uses document term vectors to compute the similarity. The term vector for any document can be computed at query time using information stored in the index. However, for better performance, term vectors can be pre-computed and stored during indexing.

The optional `termVectors` attribute allows you to enable term vectors to be stored for each document during indexing. Thus, if you plan to make heavy use of the **More Like This** feature in your search application, you should set `termVectors="true"` for text fields. If you decide to enable this feature after indexing documents, then you will need to need to re-index all documents.

We will revisit these attributes throughout the rest of the book, when applicable. For example, we'll look closely at the `termPositions` and `termOffsets` attributes in Chapter 9 when we discuss hit highlighting. For now, let's turn our attention to another advanced text analysis topic—namely multi-lingual analysis and language detection.

6.4.2 Multi-Lingual Text Analysis

Throughout this chapter, we focused on analyzing English text. The key take-away here is that the text analysis solution you choose is language specific. In other words, the solution we built for analyzing English micro-blog content won't work very well for German or French tweets. Each language will have its own parsing rules for tokenizing, stop words list, and stemming rules. In general, you will need to develop a specific `<fieldType>` for each language you want to analyze for your index. That said many of the techniques you learned in this chapter are still applicable for analyzing languages other than English.

This raises the question of how to select the right text analyzer during indexing? Assuming you want to index all your documents regardless of language in the same index, a simple solution would be to use a unique field for each language. For example, suppose we

want to index French tweets in our micro-blog search application. We could define the following field:

```
<field name="text_fr" type="text_microblog_fr"
      indexed="true" stored="true"/>
```

By now, you know this `<field>` definition means there is a corresponding `<fieldType>` named **text_microblog_fr** that is capable of analyzing French tweets; we'll leave it as an exercise for our French speaking readers to define the **text_microblog_fr** field type. Out-of-the-box, the Solr example does define the **text_fr** field type, shown in listing 6.5, that can be used for basic analysis of French text so we'll use that for this example.

Listing 6.5 Field type definition provided with the Solr example for analyzing French text

```
<fieldType name="text_fr" class="solr.TextField"
             positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <!-- removes l', etc -->
    <filter class="solr.ElisionFilterFactory" ignoreCase="true"
            articles="lang/contractions_fr.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
            words="lang/stopwords_fr.txt" format="snowball"
            enablePositionIncrements="true"/>
    <filter class="solr.FrenchLightStemFilterFactory"/>
  </analyzer>
</fieldType>
```

There are a few things to note about the **text_fr** field type:

- StandardTokenizer is used for tokenizing because it works well with most Latin-based languages, but if you wanted to preserve hashtags and mentions you would need to switch over to using the WhitespaceTokenizer
- Custom stopwords are loaded for French from the lang/stopwords_fr.txt file
- A French-specific stemmer, FrenchLightStemFilterFactory, is applied to the text as stemming rules differ for every language

So now that we have a separate field and field type for French text, we need to populate it during indexing. Of course, if you know a document is French ahead of time, then you can manually populate the **text_fr** field when constructing your document to be indexed. For example, assume we have the following tweet, which is a famous quote from Voltaire:

> Le vrai philosphe n'attend rien des hommes, et il leur fait tout le bien don't il est capable.
>
> Voltaire

The Solr XML document to index this French tweet is shown in listing 6.6:

Listing 6.6 Example of a French tweet

```
<add>
  <doc>
    <field name="id">3</field>
    <field name="screen_name">@thelabdude</field>
    <field name="type">post</field>
    <field name="lang">fr</field>     #A
    <field name="timestamp">2012-05-23T09:35:22Z</field>
    <field name="favourites_count">10</field>
    <field name="text_fr">Le vrai philosphe n'attend rien des hommes, et il
leur fait tout le bien don't il est capable. Voltaire</field> #B
  </doc>
</add>
```

#ASpecifying the language of the text explicitly during indexing
#B Populate the text_fr field so the text is analyzed with the correct field type for French

Notice how we specify the **lang** as "fr" and use the **text_fr** field explicitly. This approach is fine if you already know the text is French, but what do you do when you don't know the language ahead of time? Part of Solr's appeal is its large community of active users, contributors, and committers, so as you might have guessed, Solr has a built-in solution for language detection!

Built-In Language Detection

Solr language detection solution was designed to work with documents that are primarily one language or another. In other words, your mileage may vary if you send a document that contains a mixture of languages. Mixed language documents aside, let's apply Solr's language detection solution to our social media search application to see how it works. Solr's solution should work well for our social media search application in that blogs, tweets, and comments are typically written in a single language. What we hope to see is that we can send a document, such as a tweet, in any major language, have Solr detect the language, and then apply the correct text analysis based on the language. Continuing with our Voltaire quote, we want Solr to determine the document is French and then populate our **lang** field with the value "fr". Once the language is known, Solr will look for a field named "**text_fr**" to populate so that this text gets analyzed using the correct field type designed for French text.

To get started, you need to enable the language detection component on Solr's `updateRequestHandler` defined in **solrconfig.xml**, as shown in listing 6.7.

Listing 6.7 Solr's update request handler with language detection enabled

```
<requestHandler name="/update" class="solr.UpdateRequestHandler">
  <lst name="defaults">
    <str name="update.chain">langid</str> #A
  </lst>
</requestHandler>
```

#A activates the language detection component during indexing

This activates the "langid" `updateRequestProcessorChain` when documents are indexed. Next, you need to configure the langid updateRequestProcessorChain, which is located near the bottom of **solrconfig.xml** in the Solr example server; in your text editor, search for "langid". Solr provides two options for detecting language: 1) Tika-based language

detection, and 2) LangDetect-based language detection. For this example, we'll use the LangDetect-based solution as it supports 53 languages and has been shown to be more accurate and faster than the Tika-based solution[2]. Also, LangDetect is an open-source project released under the commercial friendly Apache 2.0 license; it's hosted at http://code.google.com/p/language-detection/. Listing 6.8 shows how to configure language detection in solrconfig.xml.

Listing 6.8 Configuration for the language detection component in solrconfig.xml

```
<updateRequestProcessorChain name="langid">
  #A
  <processor class="org.apache.solr.update.processor.
LangDetectLanguageIdentifierUpdateProcessorFactory">
    <str name="langid.fl">text</str> #B
    <str name="langid.langField">lang</str> #C
    <str name="langid.fallback">en</str>
    <bool name="langid.map">true</bool> #D
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

#A Specify the LangDetect solution
#B Name of field containing text to detect the language for
#C Name of field to populate the detected language
#D Enable language-specific field mapping so that text_fr gets populated with French text

During indexing, Solr passes each document through the "langid" updateRequestProcessorChain. The processor, an instance of `LangDetectLanguageIdentifierUpdateProcessor`, uses the value of the **langid.fl** parameter to determine which fields in the document contain text that should be used to detect the language. In our case, it is the **text** field. The processor pulls the text and determines the language using statistical analysis on the patterns in the text. Once the language is detected, the processor populates the language code in the field specified in the **langid.langField** parameter, which in our example is **lang**. Lastly, the processor is configured by the **langid.map** field to map the text to a specific field in the index by combining the name of the incoming field used for language detection "text" and the language code "fr". Thus, for our example, the processor will map French text to the **text_fr** field.

With these settings in place, you can now send documents containing text in any of the 53 languages supported by LangDetect and Solr will do the right thing. Of course, you still need to define the fields and field types to handle specific languages as we did for French.

6.4.3 Extending Text Analysis using a Solr Plug-In

We'll wrap-up this chapter with an answer to the question of what do you do when Solr doesn't provide a built-in solution for your text analysis problem? As you saw in this chapter,

[2] See: http://blog.mikemccandless.com/2011/10/accuracy-and-performance-of-googles.html

we accomplished some very powerful transformations on our micro-blog content without writing any code using only built-in Solr tools. Consequently, it will be rare to encounter text analysis requirements that cannot be addressed with one of the built-in tools. The Solr Plug-In framework can be used to develop extensions for a number of Solr components beyond text analysis. Thus, we'll only touch on the basics as they relate to text analysis. We'll use the Plug-In framework to extend other components of Solr in later chapters.

To begin, we need a requirement that Solr can't solve with built-in tools. Recall from our previous discussion of multi-valued fields where we indexed zero or more URLs into the "links" field. For tweets, any links in the text are going to be shortened links provided by an online service like bitly.com. For instance, the shortened bit.ly URL for the Solr home page http://lucene.apache.org/solr/ is http://bit.ly/3ynriE. From a search perspective, the shortened URL isn't very useful as it's hard to imagine someone entering "bit.ly/3ynriE" in a search box! What we want is for a document with a shortened link to be found when someone searches for the resolved URL, e.g. searching for "lucene.apache.org/solr" would find a tweet with link http://bit.ly/3ynriE. Thus, during indexing, we need to extract the shortened URL and replace it with the resolved URL. To get the resolved URL, we can use an HTTP HEAD request and follow redirects until we reach the resolved URL or, in the case of bit.ly, we can use their Web service API.

Now that we know the problem to solve and have a basic understanding of how we want to solve it, we need to determine where in the Solr text analysis process to plug-in our solution. In other words, we need to determine the type of plug-in we need to build. As we learned in section 6.2. Solr text analysis involves the following components: Analyzer, CharFilter, Tokenizer, and TokenFilter. Recall that an analyzer brings together a tokenizer and chain of token filters into a single component. That feels a little heavy-handed since we want to utilize our existing tokenizer and chain of filters. A tokenizer parses text into a stream of tokens. A token filter either transforms, replaces, or removes tokens.

Our solution involves replacing one token, a shortened URL, with a different token, a fully-resolved URL. Thus, for this requirement, it makes sense to build a custom TokenFilter, which in Solr is the most common and easiest way to customize text analysis. That said, you can also build your own analyzer, tokenizer, or char filter using a similar process to what we illustrate below.

To create a custom TokenFilter, you need to develop two concrete Java classes: one that extends Lucene's `org.apache.lucene.analysis.TokenFilter` class to perform the filtering and a factory for your custom filter that extends Lucene's `org.apache.lucene.analysis.util.TokenFilterFactory`. The factory class is needed so that Solr can instantiate configured instances of your TokenFilter using configuration supplied in the schema.xml file.

CUSTOM TOKENFILTER CLASS

Let's begin by building a custom TokenFilter. As our main focus here is to learn how to implement a custom solution for text analysis, we'll leave the actual implementation details

of resolving a URL to the motivated reader. Listing 6.9 shows a skeleton of the custom TokenFilter class for resolving shortened URLs:

Listing 6.9 A custom Lucene TokenFilter class to resolve shortened URLs

```
package sia.ch6;

import java.io.IOException;
import java.util.regex.Pattern;

import org.apache.lucene.analysis.TokenFilter;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute;

public class ResolveUrlTokenFilter extends TokenFilter { #A

    private final CharTermAttribute termAttribute =
                 addAttribute(CharTermAttribute.class);
    private final Pattern patternToMatchShortenedUrls;

    public ResolveUrlTokenFilter(TokenStream in,  #B
      Pattern patternToMatchShortenedUrls) {
        super(in);
        this.patternToMatchShortenedUrls = patternToMatchShortenedUrls;
    }

    @Override
    public boolean incrementToken() throws IOException { #C
        if (!input.incrementToken())
            return false;

        char[] term = termAttribute.buffer();
        int len = termAttribute.length();

        String token = new String(term, 0, len);
        if (patternToMatchShortenedUrls.matcher(token).matches()) {
            // token is a shortened URL, resolve it and replace
            termAttribute.setEmpty().
                 append(resolveShortenedUrl(token));
        }

        return true;
    }

    private String resolveShortenedUrl(String toResolve) { #D
        // TODO: implement a way to resolve shortened URLs #D
        return toResolve;                                  #D
    }
}
```

#A Custom filter should extend TokenFilter
#B Custom TokenFilterFactory knows how to construct your TokenFilter
#C Method is called to process each token in the stream
#D Implementation left for the motivated reader

If you choose to actually implement this filter, we encourage you to consider the impact your solution will have on indexing performance. If you have a large volume of documents

you need to index, as would typically be the case for social content, then your implementation will need to account for the high-latency required to resolve links. A rough outline of a solution would use a distributed caching solution, such as memcached, to avoid resolving links more than once and would take full advantage of Web service APIs provided by URL shortening services like bitly.com. Typically, the API-based approach will allow for batching up many shortened URLs into a single request, which will be more efficient than using HTTP HEAD requests to resolve the links by following redirects.

CUSTOM TOKENFILTERFACTORY CLASS

When developing a custom TokenFilter for Solr, you also need to implement a Factory class that knows how to instantiate an instance of your TokenFilter. The Factory is responsible for taking attributes specified in schema.xml and converting them into parameters needed to create the TokenFilter. Here's how we'll define our token filter in schema.xml:

```
<filter class="sia.ch6.ResolveUrlTokenFilterFactory"
        shortenedUrlPattern="http:\/\/bit.ly\/[\w\-]+" />
```

The factory uses the `shortenedUrlPattern` attribute to compile a Java Pattern object for matching shortened URLs you want to resolve during text analysis. The example shown here only supports bit.ly URLs so in a real application, you would want to extend this regular expression to support all possible shortened URL sources.

Next, take a moment to think about where in the chain of filters in the **text_microblog** field type you should put this filter. We think it should go immediately after the WhitespaceTokenizer (before the WordDelimiterFilter), as you don't want to perform any transformations on the shortened URL before trying to resolve it. The Java implementation for our factory is shown in listing 6.10.

Listing 6.10 Custom Lucene TokenFilterFactory to create our custom TokenFilter

```
package sia.ch6;

import java.util.Map;
import java.util.regex.Pattern;

import org.apache.lucene.analysis.TokenFilter;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.util.TokenFilterFactory;

public class ResolveUrlTokenFilterFactory extends TokenFilterFactory { #A

    protected Pattern patternToMatchShortenedUrls;

    @Override
    public void init(Map<String,String> args) { #B
        super.init(args);
        assureMatchVersion();
        patternToMatchShortenedUrls =
              Pattern.compile(args.get("shortenedUrlPattern"));
    }
```

```
    @Override
    public TokenFilter create(TokenStream input) {    #C
        return new ResolveUrlTokenFilter(input,
                       patternToMatchShortenedUrls);
    }
}
```

#A Custom class must extend TokenFilterFactory so that Solr knows how to instantiate your factory
#B Override the init method to access attributes supplied to your factory in schema.xml
#C Override the create method to return a fully configured instance of your TokenFilter

Only a few lines of code are needed to plug-in a custom TokenFilter! The key take-away here is that Solr uses your factory class as the intermediary between the filter definition in schema.xml and a configured instance of your custom TokenFilter used during text analysis.

The last thing you need to do is to place a JAR containing your Plug-In classes in a location where Solr can locate them during initialization. To keep things simple, we recommend adding a new directory called "plugins" and then adding that location to **solrconfig.xml** as we discussed in Chapter 4.

```
<lib dir="plugins/" regex=".*\.jar" />
```

When the server starts up, it makes all JAR files in the plugins directory available to the Solr ClassLoader. If Solr complains about not being able to find your custom classes during initialization, try putting the full path to your plugins directory.

6.5 Summary

Congratulations—you've just conquered some pretty complex concepts! At this point you should have a solid understanding of Solr text analysis and the indexing process in general. To recap, we began the chapter by learning how to define field types to do basic text analysis. This is where we learned that field types for unstructured text-based fields are composed of either one analyzer for both indexing and query processing or two separate, but compatible analyzers for indexing and query processing. Each analyzer is made up of a tokenizer and chain of token filters. To test our simple analysis solution, we used Solr's Analysis form to see an example document pass through the StandardTokenizer and chain of simple filters to remove stop words and lowercase terms.

Our testing demonstrated that the basic text analysis solution was not sufficient to deal with all the nuances of our micro-blog content. Consequently, we leveraged more built-in Solr tools to tackle these complex requirements. Specifically, we used the PatternReplaceCharFilterFactory with a simple regular expression to collapse repeated characters down to a maximum of two to deal with terms like "yummm" and "soooo". We used the WhitespaceTokenizer and WordDelimiterFilter to preserve the leading # and @ characters on hashtags and mentions in tweets. The WordDelimiter also proved useful for handling hyphenated terms in a more robust manner. A search for iPad will match documents with i-Pad. We also saw how to use stemming and synonym expansion to improve our search application's matching capabilities. Overall, we developed a powerful solution for analyzing micro-blog content using only built-in tools and not a single line of custom code.

Finally, we finished our tour of text analysis by looking at some advanced concepts. Specifically, we covered advanced attributes for fields, such as setting `omitNorms=true` to reduce memory and storage in your index if you don't need index-time boosts. Next, we saw how to use Solr's built-in solution for language detection to handle documents in other languages. Lastly, we developed a custom TokenFilter to resolve shortened URLs. The key take-away was that Solr makes it easy to implement a custom analyzer, tokenizer, token filter, or char filter to implement exotic requirements.

In the next chapter, we learn how to query Solr and process results.

8 Faceted Search

This chapter covers

- An introduction to faceting and an overview of common use cases
- Field Facets and the ability to see the top values in any field per query
- Using bucketized Range Facets to understand number and date ranges values in your search documents
- Getting matched document counts for any number of arbitrarily complex queries by utilizing Query Facets
- Implementing multi-select faceting and even faceting upon values not included in your search results
- How to best filter upon faceted values
- An introduction to advanced faceting topics covered in later chapters

Faceting is one of the most powerful features of Solr, as compared to traditional databases and other NoSQL data stores. Faceted search, also called faceted navigation or faceted browsing, is a capability which allows those running searches to see a high-level breakdown of their search results based upon one or more aspects (facets) of their documents, allowing them to select filters to drill into those search results.

When running a search on a news site, you would expect to see options to filter your results by timeframe (last hour, last 24 hours, last week) or by category (politics, technology, local, business). When searching on a job search site, you would likewise expect to see options to filter results by city, job category, industry, or even company name. These filtering options generally display not only the available values for each of these facets, but also a count of the total search results matching each of those values. In fact, since only a limited number of values can be displayed on the screen for each facet, search engines often sort the values for each facet based upon the most prevalent values (those matching the

most documents). This allows users to quickly see a birds-eye-view of their results set, without having to actually look through every single search result.

Faceting in Solr enables this kind of dynamic metadata to be brought back with each set of search results. While the most basic form of faceting simply shows a breakdown of unique values within a field shared across many documents (a list of categories, for example), Solr also provides many advanced faceting capabilities. These include faceting based upon the resulting values of a function, based upon ranges of values, or even based upon arbitrary queries. Solr also allows for hierarchical and multi-dimensional faceting and multi-select faceting, which allows returning facet counts for documents that may have actually been filtered out a search result. We will investigate each of these capabilities in this chapter, leaving you with the knowledge to implement your own world-class faceted search experience.

To get the most out of this chapter, you should be familiar with how to perform queries and how query parsers work (covered in chapter 7) and also how fields are defined in Solr's schema.xml to analyze text (covered in chapters 5 and 6). A basic understanding of how the Lucene index stores terms (covered in chapter 3) is also useful, but not required.

Let us begin with some examples demonstrating how faceting in Solr enables navigating through and visualizing search results at a glance.

8.1 Navigating your content at a glance

In this section, we you will see a high-level overview of faceting, including several example demonstrating some powerful ways to visualize facets for maximum usability. Faceted search is generally composed of two separate steps, calculating and displaying facets to users (which I will refer to going forward as "bringing back facets" or sometimes just "faceting"), and allowing users to select one or more facet values by which their results should be filtered (which I will refer to as "selecting" or "filtering" on a facet).

At this point, you may be wondering exactly what a facet looks like when it is brought back from the search engine. If you were searching on a set of restaurant documents, you might expect several facets to be represented as a navigational element in the user interface, such as the one depicted in figure 8.1 depicting the facets to be returned for the query "hamburger."

▶Restaurant Type	▶State	▶Price Range	▶City
Fast Food (10073)	New York (4020)	< $5 (4000)	New York, NY (2021)
Sit-down Chain (2530)	California (3459)	$5 - $10 (7000)	San Francisco, CA (1499)
Local Diner (998)	Illinois (2450)	$10 - $20 (1007)	Chicago, IL (850)
Up-scale (400)	Georgia (1620)	$20 - $50 (1300)	Atlanta, GA (620)
	Texas (1501)	$50+ (550)	Austin, TX (501)
	...	...	...

Figure 8.1 Navigational element in a search results user interface depicting a breakdown of various "facets" of the content returned for a search query of "hamburger."

This navigational element provides a clear visual demonstration of what faceting is, and it also provides an initial glimpse into the power faceting provides. Each of the categories brought back ("Restaurant Type", "State", "Price Range", and "City") are individually considered one facet. You can think of each facet as a slice of information that describes the results of a search. In this case, there are at least 14,000 search results matching the query for "hamburger," but you are able to easily see that most of them are for fast food restaurants, most are in large cities, and that most restaurants' menus fall within the $5 to $10 range.

It is important to note that, while the information in figure 8.1 is useful, there are many potentially better way to visualize these facets. One downside of including each of the values above is that you can only display a few at a time for each facet. Figure 8.2 demonstrates an alternate way to view the State facet.

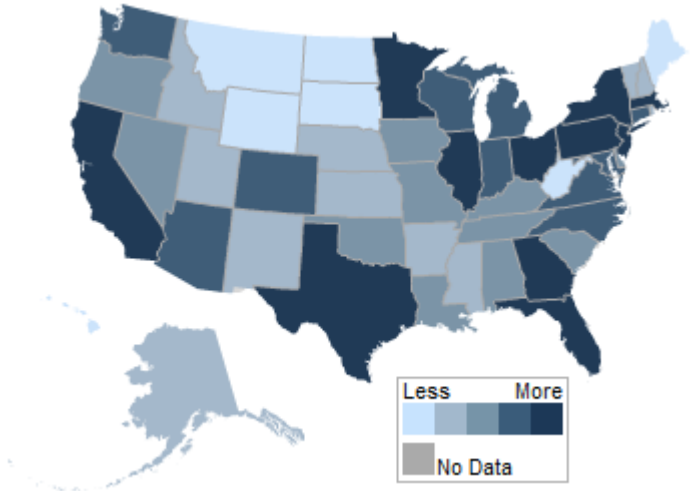


Figure 8.2 Representation of a facet on "State" in a visually appealing way. This visualization allows all 50 States within the United States to be displayed in the user interface at one time, preventing the user from being overwhelmed by information.

As you can see, by utilizing modern visualization techniques, it is possible to utilize facets as representing important meta-data about a user's search results to provide an enhanced searching experience. Figure 8.3 illustrates a similarly appropriate visualization for the Restaurant Type facet.

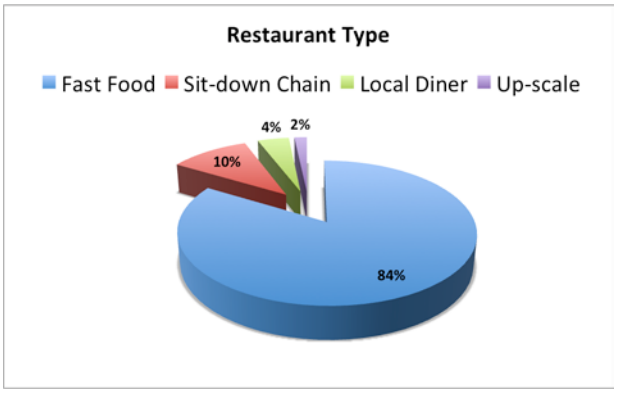


Figure 8.3 Visualization representing the Restaurant Type facet as a pie chart.

While geographic maps and pie charts may be useful for demonstrating discrete values, representing continuous values such as numbers and dates can often be best represented through line graphs, as demonstrated in figure 8.4.

Figure 8.4 The price range facet is visually represented as a continuous line graph, allowing users to interpret all of the values in the facet with one quick glance.

The line graph demonstrating the Price Range facet is particularly interesting because this visualization can be used to represent any range of values which can plotted in a continuous

series: numbers, dates and times, prices, distances, or even function values calculated inside of Solr. This is not a chapter on data visualization, so we will not belabor the point here, but it is important to take away from this section that faceting provides an incredibly powerful ability to generate real-time analytics on user searches which can greatly enhance your users' search experiences.

Before we dive into the mechanics of implementing facets in Solr, it is also important to note that facets are calculated in real-time for each set of search results. Each facet brings back a list of values (such as "Fast Food", "Sit-down Chain, "Local Diner", and "Up Scale" for the Restaurant Type facet), along with a count for each of those values indicating how many documents contain that value. The count is NOT how many times the value exists across all documents (as a value may exist multiple times within a single document), it is only a count of the number of documents matched. The fact that all facet values are calculated based upon a search result set means that every time a new search is fired, different facet values and counts will be returned for each facet. This allows users to run a search, see the facets that are returned from the search result, and then perform another search which filters on any values for which they want to limit the next set of search results.

Given our original example in figure 8.1, what would have happened if a user clicked on the value of "California" in the State facet and your application then executed another search filtering on that value? Figure 8.5 demonstrates the facet values that may have resulted from this second search.

▶Restaurant Type	▶State	▶Price Range	▶City
Fast Food (2500)	**California (3459)**	< $5 (800)	San Francisco, CA (1499)
Sit-down Chain (459)		$5 - $10 (1600)	Los Angeles, CA (701)
Local Diner (301)		$10 - $20 (580)	San Diego, CA (535)
Up-scale (199)		$20 - $50 (298)	San Jose (356)
		$50+ (181)	Sacramento (178)
			...

Figure 8.5 Demonstration of how facet values change when the state of California is filtered upon after the original search request in Figure 8.1. Notice that all cities in the City facet are now located in California since all documents within our search results must now be in California.

While figure 8.5 only filters on a single value (the state of California), you can actually apply as many filters as you want on any given search, and each facet will calculate its values based upon all of the filters being applied. If you explore the Price Range facet for the first search (nation-wide) and compare them to the results of the second search (state of California), you would be able to spot a noticeable price difference between California and the rest of the United States, as figure 8.6 will visually demonstrate.

Figure 8.6. Price Range Facet (nation-wide) vs. Price Range Facet (state of California). A noticeable trend is evident when the facet values are compared side-by-side: Restaurants in California, on the whole, are more expensive than restaurants in the rest of the country.

You will see in section 8.5 how to apply these filters to facet values, but the key take-away from this section is that facets can provide rich insights into the results of any given search, allowing your users to easily measure the quality of their search and drill-in to the aspects of the results they find most interesting.

8.2 Setting up some test data

Before we dive into the mechanics of faceting, we need to load up some sample data into Solr that will be used for the examples in rest of this chapter. Our sample data will be a small subset of documents similar to our restaurants example from section 8.1. This section will get you up and running with a handful of restaurant documents upon which you will be able to facet throughout the rest of this chapter. In order feed these example documents to Solr, we first need to include the fields definitions from listing 8.1 into Solr's schema.xml file (previously covered in chapter 5):

Listing 8.1 Schema.xml defining fields for restaurant documents examples

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="example" version="1.5">
 <fields>#A
     <field name="id" type="string" indexed="true" stored="true" />
     <field name="name" type="string" indexed="true" stored="true" />
     <field name="city" type="string" indexed="true" stored="true" />
     <field name="ischain" type="boolean" indexed="true" stored="true" />
     <field name="state" type="string" indexed="true" stored="true"/>
     <field name="tags" type="string" indexed="true" stored="true"
multiValued="true" />
     <field name="price" type="double" indexed="true" stored="true" />
 </fields>
 <uniqueKey>id</uniqueKey>#B
  <types>#C
     <fieldType name="string" class="solr.StrField" sortMissingLast="true"
/>
```

```
    <fieldType name="boolean" class="solr.BoolField"
sortMissingLast="true"/>
    <fieldType name="double" class="solr.TrieDoubleField" precisionStep="0"
positionIncrementGap="0"/>
   </types>
</schema>
```

#A We will have six fields
#B Should have a unique ID
#C We will have three field types

The schema in listing 8.1 defines each of the fields upon which we will attempt to facet in this chapter. Each of the documents upon which we will facet is contained in listing 8.2.

Listing 8.2 Documents upon which faceting will be demonstrated

```
[
  {
     "id":"1", "name":"Red Lobster", "city":"San Francisco, CA",
     "ischain":true, "state":"California", "tags":["sea food", "sit
     down"], "price":33.00#A
  },
  {
     "id":"2", "name":"Red Lobster", "city":"Atlanta, GA",
     "ischain":true, "state":"Georgia", "tags":["sea food", "sit-down"],
     "price":22.00#B
  },
  {
     "id":"3", "name":"Red Lobster", "city":"New York, NY",
     "ischain":true, "state":"New York", "tags":["sea food", "sit-
     down"], "price":29.00},#C
  {
     "id":"4", "name":"McDonalds", "city":"San Francisco, CA",
     "ischain":true, "state":"California", "tags":["fast food",
     "hamburgers", "coffee", "wi-fi", "breakfast"], "price":9.00#D
  },
  {
     "id":"5", "name":"McDonalds", "city":"Atlanta, GA", "ischain":true,
     "state":"Georgia", "tags":["fast food", "hamburgers", "coffee",
     "wi-fi", "breakfast"], "price":4.00#E
  },
  {
     "id":"6", "name":"McDonalds", "city":"New York, NY",
     "ischain":true, "state":"New York", "tags":["fast food",
     "hamburgers", "coffee", "wi-fi", "breakfast"], "price":4.00#F
  },
  {
     "id":"7", "name":"McDonalds", "city":"Chicago, IL", "ischain":true,
     "state":"Illinois", "tags":["fast food", "hamburgers", "coffee",
     "wi-fi", "breakfast"], "price":4.00#G
  },
  {
     "id":"8", "name":"McDonalds", "city":"Austin, TX", "ischain":true,
     "state":"Texas", "tags":["fast food", "hamburgers", "coffee", "wi-
```

```
   fi", "breakfast"], "price":4.00#H
},
{
   "id":"9", "name":"Pizza Hut", "city":"Atlanta, GA", "ischain":true,
   "state":"Georgia", "tags":["pizza", "sit-down", "delivery"],
   "price":15.00
},
{
   "id":"10", "name":"Pizza Hut", "city":"New York, NY",
   "ischain":true, "state":"New York", "tags":["pizza", "sit-down",
   "delivery"], "price":24.00
},
{
"id":"11", "name":"Pizza Hut", "city":"Austin, TX", "ischain":true,
   "state":"Texas", "tags":["pizza", "sit-down", "delivery"],
   "price":18.00
},
{
   "id":"12", "name":"Freddy's Pizza Shop", "city":"Los Angeles, CA",
   "ischain":false, "state":"California", "tags":["pizza", "pasta",
   "sit-down"], "price":25.00},
{
   "id":"13", "name":"The Iberian Pig", "city":"Atlanta, GA",
   "ischain":false, "state":"Georgia", "tags":["spanish", "tapas",
   "sit-down", "up-scale"], "price":45.00
},
{
   "id":"14", "name":"Sprig", "city":"Atlanta, GA", "ischain":false,
   "state":"Georgia", "tags":["sit-down", "gluten-free", "southern
   cuisine"], "price":15.00
},
{
   "id":"15", "name":"Starbucks", "city":"San Francisco, CA",
   "ischain":true, "state":"California", "tags":["coffee",
   "breakfast"], "price":7.50
},
{
   "id":"16", "name":"Starbucks", "city":"Atlanta, GA",
   "ischain":true, "state":"Georgia", "tags":["coffee", "breakfast"],
   "price":4.00
},
{
   "id":"17", "name":"Starbucks", "city":"New York, NY",
   "ischain":true, "state":"New York", "tags":["coffee", "breakfast"],
   "price":6.50
},
{
   "id":"18", "name":"Starbucks", "city":"Chicago, IL",
   "ischain":true, "state":"Illinois", "tags":["coffee", "breakfast"],
   "price":6.00
}
{
   "id":"19", "name":"Starbucks", "city":"Austin, TX", "ischain":true,
```

```
      "state":"Texas", "tags":["coffee", "breakfast"], "price":5.00
    },
    {
      "id":"20", "name":"Starbucks", "city":"Greenville, SC",
      "ischain":true, "state":"South Carolina", "tags":["coffee",
      "breakfast"], "price":3.00
    }
]
```

#A Each document has all seven fields
#B The "id" field is unique per restaurant
#C The "name" field has one non-unique value
#D The "city" has one non-unique value
#E The "ischain" field is true or false
#F The "state" has one non-unique value
#G The "tags" field contains multiple non-unique values
#H The "price" field is positive number

You can also find the example schema.xml (called restaurants_schema.xml) and the example documents (called restaurants.json) from listings 8.1 and 8.2 in the downloadable source code on *Solr in Action* section of Manning's website. After downloading or recreating files from the above listings in your current home directory, it is now time to start Solr and index the example restaurant documents.

```
cd {SOLR_ROOT}/example/solr/collection1/conf/
cp schema.xml schema.xml.original
cp ~/example-docs/ch8/restaurants_schema.xml schema.xml
cd ../../../
cp ~/example-docs/ch8/restaurants.json ./exampledocs/
java –jar start.jar
cd exampledocs
java -Dtype=application/json -jar post.jar restaurants.json
```

The last line above actually indexes the restaurants from a text file using the post.jar utility that comes with Solr. If everything is successful, you should see an output like the following:

```
SimplePostTool version 1.5
Posting files to base url http://localhost:8983/solr/update using content-type
application/json..
POSTing file ch8/restaurants.json
1 files indexed.
COMMITting Solr index changes to http://localhost:8983/solr/update..
```

Once you have successfully indexed the restaurant documents, you should be able to hit the standard Solr search handler and verify that your documents in the engine, along with all of the fields from the example documents, using the following url:

```
http://localhost:8983/solr/select?q=*:*&echoParams=none
```

Changing Solr's response format

Please note that all listings and other Solr responses in this chapter are in JSON format and not XML. Even though XML is Solr's default response type, JSON is a more human-readable format which is more compact and is thus easier to use for demonstration purposes. A parameter of "`wt=json`" was added onto the Solr request to change the response type from XML to JSON format. If you have not changed your default response type to JSON (Solr's default is a XML), then it will be necessary to add this "wt=json" parameter to all of the Solr requests in this chapter to return them in the same format.

You will also notice that all Solr responses in this chapter appear nicely indented. This can be accomplished by adding an "`indent=on`" parameter to your Solr url. Requesting this indented response format is not recommended for a production application (it works, but adds unnecessary extra processing time to your application), but it does make Solr responses more human-readable, which is why it has been used for all the examples in this chapter.

To eliminate redundancy, this chapter assumes that you have set "&wt=json&indent=on" as default parameters (see chapter 7 for how to do this) on all of your queries.

The response you receive from Solr should include all twenty of the example restaurant documents, with each containing the seven fields defined in the schema and the example restaurant dataset.:

```
{
   "responseHeader":{
     "status":0,
     "QTime":0},
   "response":{"numFound":20,"start":0,"docs":[
       {
            "id":"1",
            "name":"Red Lobster",
            "city":"San Francisco, CA",
            "ischain":true,
            "state":"California",
            "tags":["sea food", "sit down"],
            "price":33.0},
     },
     ...
   ]}
}
```

With the twenty example documents now searchable, we are ready to begin running faceted searches. We will begin with the most common form of faceting: faceting upon each of the unique values within a field.

8.3 Field faceting

Field faceting is the most common form of faceting: requesting the unique values found in a particular field back, along with the number of documents in which they are found, when a search is performed. In section 8.1, several Field Facets were demonstrated from the restaurants example: a facet on "Restaurant Type" field, a facet on the "State" field, and a facet on the "City" field. In this section, you will learn how to construct a Solr query requesting a Field Facet, and you will learn the various faceting parameters that allow you to tweak how the facet values are calculated and returned from Solr.

For demonstration purposes for the rest of the chapter, we are going to be faceting upon the documents we indexed in section 8.2. Let us start by running our very first facet:

```
http://localhost:8983/solr/select?q=*:*&echoParams=none&rows=0&
facet=true&facet.field=name
```

The results of this query appear in listing 8.3.

Listing 8.3 Facet results for a Field Facet on a single-valued field

```
{
   "responseHeader":{
     "status":0,
     "QTime":43},
   "response":{"numFound":20,"start":0,"docs":[]
   },
   "facet_counts":{#A
     "facet_queries":{},
     "facet_fields":{#B
       "name":[#C
         "Starbucks",6,#D
         "McDonalds",5,
         "Pizza Hut",3,
         "Red Lobster",3,
         "Freddy's Pizza Shop",1,
         "Sprig",1,#E
         "The Iberian Pig",1]},
     "facet_dates":{},
     "facet_ranges":{}}
}
```

#A The root node for all facets
#B Lists all Field Facets
#C We requested a facet on the "name" field
#D "Starbucks" was found in six documents
#E "Sprig" was found in one document

This example demonstrates the most basic form of faceting in Solr: Field Faceting on a single-valued field. In this kind of faceting, each unique value is examined, along with a count of documents in which that value is found. Since there is only one value per document, the sum of all of the counted values (company names in this case) will also equal the total number of documents.

Not all fields in Solr contain a single value, however. The tags field is an example of a multi-valued field. Let us see what happens when we try to facet upon the tags field in listing 8.4.

Listing 8.4 Faceting upon a multi-valued field

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.field=tags
```

Results:

```
...
"facet_fields":{
       "tags":[#A
         "breakfast",11,#B
         "coffee",11,#B
         "sit-down",8,
         "fast food",5,
         "hamburgers",5,
         "wi-fi",5,
         "pizza",4,
         "delivery",3,
         "sea food",3,
         "gluten-free",1,
         "pasta",1,
         "sit down",1,
         "southern cuisine",1,
         "spanish",1,
         "tapas",1,
         "up-scale",1]
}
...
```

#A The name of the field we faceted upon
#B Adds up to > 20 due documents matching multiple tags

It is interesting to note that the tags “breakfast” and “coffee” were the two most prevalent across the entire corpus of restaurant documents. This is because two of the restaurants, Starbucks and McDonalds, fell into both of these categories. If you were to search for breakfast or for coffee, this would become readily apparent from the search results returned from Solr.

It is also important to note that sum of the counts for each of the tags facet values is much greater than 20, which is the total number of documents in the search engine. This is because each document contains more than one term, which means that many of the individual terms map to the same document (and thus most of the documents are counted more than once).

In section 8.1 we discussed several methods for visualizing facets. This tag faceting example lends itself to what is probably a distinctly obvious kind of visualization: a tag cloud.

Figure 8.7 demonstrates mapping the facet values from this multi-valued Field Facet result on the tags field.

Figure 8.7 A tag cloud representation of a Field Facet on a multi-valued tags field. The size of the text is relative to the number of documents in which the phrase was found. This demonstrates again that there are many ways to visualize facets.

Tag clouds are common ways for users to see a high-level overview of the results of their search from a categorical standpoint. In addition to using tags like in this example, it is also common to facet upon a raw content field containing everyday language to glean these kinds of insights (such as the full-text of a restaurant's description in this case). The problem with using a raw content field, of course, is that much more noise exists in such a field, since everyday language contains junk words (such as stopwords like "and", "of", and "like"), making it less reliable in describing the document than explicit tags would be.

An additional important item to keep in mind when you are faceting is that the values returned for a field facet are based upon the indexed values for a field. In other words, if you pass in the term "San Francisco, CA", but you actually tokenize the field as a text field which splits on spaces and commas and also lowercases the text, then the actual values in the Solr index for that field would be "ca", "francisco", and "san". Thus, unless you want to actually bring back facet counts for each of those terms individually and lowercased, you have to consider how you may want to facet on a field when you create the field definition in Solr's schema.xml. It is fairly standard for Solr developers to create a separate field into which they will put a duplicate copy of certain content for the sole purpose of faceting (so that the original text can be preserved in a facetable form.

At this point you should have a solid grasp of what a facet is, and you should also have a good feel for requesting a facet on either a single-valued field or a multi-valued field. Thusfar, however, you have only been exposed to the default settings for bringing back a facet. Faceting seems easy when you are only dealing with twenty documents, but what happens when there are thousands or millions of unique terms that would come back from Solr on a faceting request? Fortunately, Solr has many faceting options which allow fine-tuning how facets are returned on a per-query basis. This list of Field Facet options is given in table 8.1.

Table 8.1 A listing of the Field Faceting parameters which can be specified on the Solr url to modify faceting behavior.

Solr Parameter	Possible Values	Description
`facet`	true, false	Enables or disables all faceting for the current search
`facet.field`	the name of any indexed field	Determines which field a facet should be calculated upon. This parameter may be specified multiple times to return multiple facets.
`facet.sort`	index, count	Sorts the facet values by highest number of occurrences (`count`), or by order in the index, which is generally alphanumerically (`index`). This parameter can be specified on a per-field basis.
`facet.limit`	An integer >= -1	Determines how many unique facet values will be returned for each facet. This parameter can be specified on a per-field basis.
`facet.mincount`	An integer >= 0	Sets a minimum number of documents in which a facet value must appear before it will be returned. By default, terms with zero matches in the current search will be included in facet results, so it is common to set facet.mincount to at least 1. This parameter can be specified on a per-field basis.
`facet.method`	enum, fc	The `enum` method loops over all terms in the index, calculating a set intersection with those terms and the query. The `fc` (field cache) method loops over the documents that match the query and finds the terms within those documents. The `fc` method is faster for fields which contain many unique values, whereas the `enum` method is faster for fields which contain few values. The `fc` method is the default for all fields except Boolean fields. This parameter can be specified on a per-field basis.
`facet.enum.cache.minDf`	An integer >= 0	Advanced: Specifies the minimum number of documents required to match a term before the filterCache (see section 8.8) should be used for that term. The default is 0, meaning that the filterCache should always be used. Setting this value > 0 can reduce memory consumption at the expense of slower queries. This parameter can be specified on a per-field basis.

One important take-away from table 8.1 is that multiple facets can be requested by specifying the facet.field parameter multiple times. Additionally, several of the facet

parameters are listed as being specifiable on a per-field basis. This can be accomplished using the following syntax:

```
f.<fieldName>.<FacetParameter>=<value>
```

Thus, if you wanted to bring back a facet for all 50 United States (in alphabetical order) with at least one restaurant, all restaurant names even if they don't match the query, and the top 5 matching tags, you could perform the query in listing 8.5.

Listing 8.5 Mixing Field Faceting parameters on a field-by-field basis

Query:

```
http://localhost:8983/solr/select?q=*:*&
facet=true&
facet.mincount=1&#A
facet.mincount=1&#A
facet.field=state&
f.state.facet.limit=50&#B
f.state.facet.sort=index&#B
facet.field=name&
f.name.facet.mincount=0&#C
facet.field=tags&#C
f.tags.facet.limit=5#D
```

Results:

```
...
"facet_fields":{
  "state":[
    "California",4,
    "Georgia",6,
    "Illinois",2,
    "New York",4,
    "South Carolina",1,
    "Texas",3],
  "name":[
    "Starbucks",6,
    "McDonalds",5,
    "Pizza Hut",3,
    "Red Lobster",3,
    "Freddy's Pizza Shop",1,
    "Sprig",1,
    "The Iberian Pig",1],
  "tags":[
    "breakfast",11,
    "coffee",11,
    "sit-down",8,
    "fast food",5,
    "hamburgers",5]
}
...
```

#A Default for all facets unless overriden
#B Only modifies the "state" facet
#C Only modifies the "name" facet
#D Only modifies the "tags facet

As you can see, this listing combines multiple facet options on multiple fields in a custom way to return exactly what was desired in the search results. While the query required many parameters, the flexibility these faceting options provide can be well worth the additional complexity.

At this point you have learned how to request Field Facets back in Solr so that you can see how many documents match each unique value in any of your indexed fields. While this is a powerful feature, faceting in Solr is actually much more powerful. The next realm of faceting we will explore is Query Faceting, the ability to bring back facet counts for literally any query – no matter how complex – which also matches your search results.

8.4 Query faceting

While it is great to be able to return the top values within any indexed field as a facet, as discussed in the last section, it can also be extremely useful to bring back counts for arbitrary sub-queries so that you know how many results might match a future search. Solr provides this through its implementation of Query Facets.

The best way to demonstrate this capability is through an example. Referring back to our restaurant searching dataset from section 8.2, let us say you wanted to run a query for restaurants falling within the price range of $5 to $25, but you also wanted to know how many of those documents were on the East Coast, on the West Coast, or in the Mid Western United States. You could certainly accomplish this by running three different queries, as indicated in listing 8.6.

Listing 8.6 Running multiple queries to obtain document counts for sub-queries

```
http://localhost:8983/solr/select?q=*:*&fq=price:[5 TO 25]
 ... "response":{"numFound":11 ...

http://localhost:8983/solr/select?q=*:*&fq=price:[5 TO 25]&
fq=state:("New York" OR "Georgia" OR "South Carolina")

... "response":{"numFound":5 ...


http://localhost:8983/solr/select?q=*:*&fq=price:[5 TO 25]&
fq=state:("Illinois" OR "Texas")

... "response":{"numFound":3 ...


http://localhost:8983/solr/select?q=*:*&fq=price:[5 TO 25]&
fq=state:("California")

... "response":{"numFound":3 ...
```

The example in listing 8.6 demonstrates the most brute-force method for finding search result counts for sub-queries in Solr – running each sub-query as a separate search and

seeing how many results are found. While the pain may not seem enormous in this small, contrived example, it is nevertheless unnecessary. Listing 8.7 demonstrates how such a query can be easily combined into a single query using Query Facets.

Listing 8.7 Running a single Query Facet to obtain document counts for sub-queries

Query:

```
http://localhost:8983/solr/select?q=*:*&fq=price:[5 TO 25]&facet=true&
facet.query=state:("New York" OR "Georgia" OR "South Carolina")&
facet.query=state:("Illinois" OR "Texas")&
facet.query=state:("California")
```

Results:

```
... "response":{"numFound":11,"start":0,"docs":[]},
   "facet_counts":{
     "facet_queries":{
       "state:(\"New York\" OR \"Georgia\" OR \"South Carolina\")":5,
       "state:(\"Illinois\" OR \"Texas\")":3,
       "state:(\"California\")":3},
     ...
}
```

As you can see from listing 8.7, multiple sub-queries can be combined into a single request to Solr through the use of Query Facets. The above example is very specific to our data set (since it requires knowing all possible values at query time), but you have already seen an example in section 8.1 which is a great use-case for Query Facets: faceting upon price, where the price ranges are not evenly spaced out. We can re-create this example using our test data, as indicated in listing 8.8.

Listing 8.8 Query Facets based upon multiple price ranges

Query:

```
http://localhost:8983/solr/select?q=*:*&rows=0&facet=true&
facet.query=price:[* TO 5}&
facet.query=price:[5 TO 10}&
facet.query=price:[10 TO 20}&
facet.query=price:[20 TO 50}&
facet.query=price:[50 TO *]
```

Results:

```
..."response":{"numFound":20,"start":0,"docs":[]   },
      "facet_counts":{
        "facet_queries":{
          "price:[* TO 5}":6,
          "price:[5 TO 10}":5,
          "price:[10 TO 20}":3,
          "price:[20 TO 50}":6,
          "price:[50 TO *]":0}, ...
```

This example demonstrates how Query Facets can effectively be used to create new buckets of information at query time in any Solr query. In reality, you could have just as easily created a new field in Solr called "pricerange" which contained each of these bucketized values. Had you done so, you could have just performed a field facet upon the new pricerange field to pull back each of the five bucketized values. Of course, this would also required you to map these bucketizing rules at index time when you are first feeding your content to Solr, a process which can be painful, especially as your amount of content in Solr begins to grow. Query Facets provide a nice alternative which allows you complete flexibility at query time to specify and re-define which buckets should be calculated and returned.

While the examples in this section have been simple, they demonstrate the complete flexibility that faceting upon any arbitrary query provides. Because Solr provides many powerful query capabilities including nested query handlers and function queries, the possibilities for advanced query-based faceting are only limited by one's imagination. Imagine taking a radius query in Solr and creating concentric circles (<5 kilometers, 5 – 10 kilometers, 10 – 20 kilometers, 20+ kilometers) around a particular location to create a facet upon "distance away." Alternatively, imagine generating a Query Facet on the calculated values of a custom relevancy function you have generated as a function query. The ability to extend faceting in this way is tremendously powerful - anything you can search upon, you can facet upon.

While Query Facets are incredibly flexible, they can become burdensome at times to request from Solr, as every single value upon which you want to generate facet counts must be explicitly specified. As we will see in the next section, Solr also provides a convenient Range Faceting capability that makes faceting upon numeric and date values much easier in this regard.

8.5 Range Faceting

Range faceting, as its name implies, provides the ability to bucketize numeric and date field values into ranges such that the ranges (and their counts) get returned from Solr as a facet. This can be particularly useful as a replacement for creating many different Query Facets to represent multiple ranges.

In the last section, a Query Facet was demonstrated based upon the `price` field in our example data from section 8.2. Had the values needed from the search engine been evenly spread out, similar facet counts could have been returned from Solr using a Range Facet, as indicated in listing 8.9.

Listing 8.9 Example Range Facet on the price field

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&
facet.range=price&
facet.range.start=0&
```

```
facet.range.end=50&
facet.range.gap=5
```

Results:

```
...{"response":{"numFound":20,"start":0,"docs":[]   },
      "facet_counts":{
        ...
        "facet_ranges":{
          "price":{
            "counts":[
              "0.0",6,
              "5.0",5,
              "10.0",0,
              "15.0",3,
              "20.0",2,
              "25.0",2,
              "30.0",1,
              "35.0",0,
              "40.0",0,
              "45.0",1],
            "gap":5.0,
            "start":0.0,
            "end":50.0}
        }}}
```

The output of this example is similar to the result of listing 8.8, with some notable exceptions. First, Range Faceting returns counts for every value falling between the start `facet.range.start` and `facet.range.end` parameters on the query, even those ranges containing no documents. Second, unlike the Query Facet example in listing 8.7, the buckets in range faceting are equally spaced out based upon the `facet.range.gap` parameter. You can adjust the gap to create larger or more granular buckets based upon the needs of your application. This is a great time saver if you want to bring back all of the range buckets within your range, as it prevents you from writing countless `facet.query` parameters manually to accomplish a similar effect.

In addition to the Range Faceting options already described, several additional range faceting parameters are available, including `facet.range.hardened`, `facet.range.other`, and `facet.range.include`. Table 8.2 is included as a reference for each available Range Faceting parameter and its available options.

Table 8.2 A listing of the Range Faceting parameters which can be specified on the Solr url to modify faceting behavior.

Solr Parameter	Possible Values	Description
`facet`	true, false	Enables or disables all faceting for the current search
`facet.range`	the name of any indexed numerical or date field	Determines which field a Range Facet should be calculated upon. This parameter may be specified multiple times to return multiple facets.

`facet.range.start`	The numerical or date value at which the first range should begin	This parameter specifies the lower bound of your ranges. No value lower than this will be included in the counts for this facet. This parameter can be specified on a per-field basis.
`facet.range.end`	The numerical or date value at which the first range should end	This parameter specifies the upper bound of your ranges. No value higher than this will be included in the counts for this facet. This parameter can be specified on a per-field basis.
`facet.range.gap`	For dates, a DateMath expression (+1DAY, +2Weeks, +1Hour, etc.). For other numeric fields, a number is expected.	The size of each range. To create the ranges, this gap will be added to the lower bound (`facet.range.start`) successively until the upper bound (`facet.range.end`) is reached. This parameter can be specified on a per-field basis.
`facet.range.hardened`	true, false	If the gap (`facet.range.gap`) does not divide evenly between the lower bound and the upper bound, the size of the last bucket different than previous buckets. If hardened is set to "true" then the final range will stop at the upper bound, leaving a potentially smaller final bucket. If hardened is false, then final bucket size will be increased above the upper bound such that its size it is the same size as the other buckets (the size of the gap). This parameter can be specified on a per-field basis.
`facet.range.other`	before, after, between, all, none	Indicates additional ranges that should be included in the ranges. The "`before`" option creates a bucket for all values prior to the lower bound. The "`after`" option creates a bucket for all values after the upper bound. The between option creates a bucket for all values between the lower and upper bounds. This parameter may be specified multiple times to include multiple values. The "`all`" option is a shortcut for saying "`before`, `after`, and `between`". If the "`none`" option is present, it will override any other parameters which are specified. This parameter can be specified on a per-field basis.
`facet.range.include`	lower, upper, edge, outer, all	"`lower`" means all ranges include their lower bound. "`upper`" means all ranges include their upper bound. "`edge`" means the first range includes its lower bound and the last range includes its upper bound. "`outer`"

		means that the "`before`" and "`after`" buckets (from `facet.range.other`) include their lower and upper bounds, respectively. This parameter may be specified multiple times to include multiple values. "`all`" is a shortcut for specifying each of the parameters separately. This parameter can be specified on a per-field basis.

The Solr parameters in table 8.2 demonstrate the rich options available when performing Range Faceting in Solr. As with Field Faceting, several of the Range Faceting parameters can be specified on a per-field basis utilizing the `f.<fieldName>.<FacetParameter>=<value>` syntax.

Range Faceting often provides a more convenient and succinct query syntax than Query Faceting when faceting upon ranges of number or date values. Query Faceting can alternatively be used when the range queries become overly complicated, allowing for some very powerful queries, as we saw in section 8.5. Of the three types of faceting we discussed, Field Faceting is the most widely used and simplest to use. For each of these three faceting types, we have explored how you can requests facets back from Solr. What we have yet to discuss is how you would go about refining your subsequent search once a facet is selected, which will be the topic of the following section.

8.6 Filtering upon faceted values

Returning facets from Solr is the first step toward allowing your users to refine their search results. Once you've shown the breakdown of faceted values to your users, however, the next step is to allow them to actually click on one or more facet values to apply that value as a filter. In this section, we will discuss the best approaches for applying these filters.

8.6.1 Applying filters to your facets

At the most basic level, applying filters upon a facet is no more difficult than just adding an extra filter (the `fq` parameter) to your query. In other words, assuming we wanted to return three facets with our searches, one a Field Facet on the "state" field, one a Field Facet on the "city" field, and one a Query Facet on the "price" field. Our initial query and results would look similar to listing 8.10.

Listing 8.10 Faceting upon the tags field in the example restaurant data

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10}&
facet.query=price:[10 TO 25}&
facet.query=price:[25 TO 50}&
facet.query=price:[50 TO *]
```

Results:

```
...
 "facet_counts":{
    "facet_queries":{
      "price:[* TO 10}":11,
      "price:[10 TO 25}":5,
      "price:[25 TO 50}":4,
      "price:[50 TO *]":0},
    "facet_fields":{
      "state":[
        "Georgia",6,
        "California",4,
        "New York",4,
        "Texas",3,
        "Illinois",2,
        "South Carolina",1],
      "city":[
        "Atlanta, GA",6,
        "New York, NY",4,
        "San Francisco, CA",3,
        "Austin, TX",3,
        "Chicago, IL",2,
        "Greenville, SC",1,
        "Los Angeles, CA",1]},
    "facet_dates":{},
    "facet_ranges":{}}
    ...
```

Search results will also be returned for this query (not shown in listing 8.10), but your user interface is likely to display these facet values for your users to select. In this example, how would you go about filtering upon a facet value? You actually already know how to do this – you simply add a filter to your query for each selected facet value. Listing 8.11 demonstrates a second search, where the user has clicked the state of "California".

Listing 8.11 Filtering upon a Field Facet

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field=state&
facet.field=city&
facet.query=price:[*% TO 10}&
facet.query=price:[10 TO 25}&
facet.query=price:[25 TO 50}&
facet.query=price:[50 TO *]&
fq=state:California
```

Results:

```
...
  "facet_counts":{
     "facet_queries":{
       "price:[* TO 10}":2,
       "price:[10 TO 25}":0,
       "price:[25 TO 50}":2,
```

```
      "price:[50 TO *]":0},
    "facet_fields":{
      "state":[
        "California",4,],
      "city":[
        "San Francisco, CA",3,
        "Los Angeles, CA",1]},
    "facet_dates":{},
    "facet_ranges":{}}
    ...
```

The point to take away from listing 8.10 and 8.11 is that users typically utilize facets in succession. First, they run a base search bringing back facets, and then they select a facet to run a subsequent search and narrow down their search results with a filter. This could continue, with the user running a third search, such as in listing 8.12.

Listing 8.12 Filtering upon both a Field Facet and a Query Facet

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10}&
facet.query=price:[10 TO 25}&
facet.query=price:[25 TO 50}&
facet.query=price:[50 TO *]&
fq=state:California&
fq=price:[* TO 10}
```

Results:

```
...
  "facet_counts":{
    "facet_queries":{
      "price:[* TO 10}":2,
      "price:[10 TO 25}":0,
      "price:[25 TO 50}":0,
      "price:[50 TO *]":0},
    "facet_fields":{
      "state":[
        "California",2,],
      "city":[
        "San Francisco, CA",2,]},
    "facet_dates":{},
    "facet_ranges":{}}
    ...
```

As the user drills down even further, applying both a filter of `state:California` and also a filter of `price:[* TO 10}`, you can see in from listing 8.12 that the results narrow down even further, leaving only two restaurants in California which match this narrowed query, both of which happen to be in San Francisco, CA.

It is worth noting that each of the examples you have seen thus far operate on a field containing only one value. As such, as soon as you filter upon that value, no documents

matching any other values for that facet will be returned. This makes sense in our single valued fields – if a document can only have one "price" or only appear in one "state", there is no way it could appear in a facet where another price or state was selected. Not all fields contain only a single value, however. Any field which is marked as multivalued in Solr's schema.xml file or that is of a field type which is analyzed into multiple tokens may actually contribute multiple terms to a facet. We can see this in action by utilizing the multivalued "tags" field in the example Solr index. Listing 8.13 shows several searches that successively apply filters on the "tags" field.

Listing 8.13 Applying several filters on a faceted field containing multiple values

NO FILTERS APPLIED

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field=name&
facet.field=tags
```

Results:

```
  ...
  "facet_fields":{
    "name":[
      "Starbucks",6,
      "McDonalds",5,
      "Pizza Hut",3,
      "Red Lobster",3,
      "Freddy's Pizza Shop",1,
      "Sprig",1,
      "The Iberian Pig",1],
    "tags":[
      "breakfast",11,
      "coffee",11,
      "sit-down",8,
      "fast food",5,
      "hamburgers",5,
      "wi-fi",5,
      "pizza",4,
      "delivery",3,
      "sea food",3,
      "gluten-free",1,
      "pasta",1,
      "sit down",1,
      "southern cuisine",1,
      "spanish",1,
      "tapas",1,
      "up-scale",1]},
  ...
```

ONE FILTER APPLIED

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field=name&facet.field=tags&fq=tags:coffee
```

Results:

```
    ...
   "facet_fields":{
      "name":[
        "Starbucks",6,
        "McDonalds",5],
      "tags":[
        "breakfast",11,
        "coffee",11,
        "fast food",5,
        "hamburgers",5,
        "wi-fi",5]},
    ...
```

TWO FILTERS APPLIED

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field=name&facet.field=tags&fq=tags:coffee&fq=tags:hamburgers
```

Results:

```
    ...
   "facet_fields":{
      "name":[
        "McDonalds",5],
      "tags":[
        "breakfast",5,
        "coffee",5,
        "fast food",5,
        "hamburgers",5,
        "wi-fi",5]},
     ...
```

From this example, you can see that fields containing multiple values will allow facets to continue to be returned for other values that all of the filters on the Solr url match. It is also worth noting that, even though these examples specify each selected filter value as its own `fq` parameter on the Solr url, there is no requirement that this be done. In fact, the filters applied in the last search of listing 8.13 could easily be converted from `fq=tags:coffee&fq=tags:hamburgers` to `fq=tags:(coffee AND hamburgers)` (or any logically equivalent Boolean expression). This will require less lookups in the Solr Filter Cache (discussed in chapter 13), and will also provide more control over how your filter values interact.

For example, there is nothing requiring you to "AND" together each of the facets selected. It could be a perfectly valid use case for you to "OR" values together, such as allowing users to select multiple cities in their facet while filtering to only the cities selected. The discussion of multi-select faceting in the following section will highlight how you might accomplish the display of such a facet with multiple filters selected at once, even on a single-valued field.

8.6.2 Safely filtering on faceted values

All of our examples of applying facet filters thus far have applied filters based upon a single valued term, such as "coffee" or "hamburgers". In reality, the terms brought back in facets may be more challenging to deal with, such as multi-word terms. What happens, for example, if you want to facet upon the multi-word term Los Angeles? As you already know, the filter `fq=city:Los Angeles` is invalid, as it syntactically says to find documents containing a city of "Los" and the term "Angeles" in the default field(s). In order to allow for phrases separated by a space, most Solr developers decide to quote all terms they facet upon. Thus, the query syntax would look like `fq=city:"Los Angeles"`.

Unfortunately, there is even a problem with just blindly quoting the terms you are filtering upon: if the term has quotes within it, the syntax will break unless you escape it. Thus, if you were searching on a search index containing songs and faceting upon the song 'the "in" crowd', you would need to escape the quotes to pass this term safely to Solr: `fq=name:"the \"in\" crowd"`. If quoting and escaping quotes were not already enough trouble, you also have to be mindful of all text processing taking place on the field upon which you are faceting. As you saw in chapter 5, text analysis upon a field can be defined differently for content indexing vs. querying. Thus, if any kind of mis-match occurs (which is generally a bad sign, of course), it is possible that a value you are trying to filter upon does not actually match the same number of documents as reported by the facet.

Thankfully, there is a fairly simple solution to ensuring the values returned by a facet and matched by a subsequent filter find the exact same documents: utilizing Solr's Term Query Parser (TermQParserPlugin), which was discussed in chapter 7. One of the benefits of this query parser is that it bypasses the defined text analysis chain for your field and instead matches the term passed in directly against the Solr index. This saves text-processing time, and it also prevents the other quoting and escaping logic discussed in the previous paragraphs. The syntax using the Term query parser for the Los Angeles example would be `fq={!term}Los Angeles`, and the syntax for the 'the "in" crowd' example would simply be `fq={!term}the "in" crowd`.

The one downside of using the Term query parser for all of your facet filters is that this query parser does not support Boolean syntax, so if you want to combine multiple facet values together in a filter, you must utilize the Nested query parser and its syntax. Listing 8.14 demonstrates utilizing both approaches: using a separate filter for each facet term and also combining multiple facet terms into a single filter utilizing the Term query parser.

Listing 8.14 Utilizing the Term query parser to filter on facet values

APPROACH 1: SEPARATE FILTERS PER TERM

Query:

```
http://localhost:8983/solr/select?q=*:*facet=true&facet.mincount=1&
facet.field=name&facet.field=tags&
fq={!term f=tags}coffee&fq={!term f=tags}hamburgers
```

APPROACH 2: ONE FILTER FOR ALL TERMS

Query:

```
http://localhost:8983/solr/select?q=*:*&
facet=true&facet.mincount=1&facet.field=name&facet.field=tags&
fq=_query_:"{!term f=tags}coffee" AND _query_:"{!term f=tags}hamburgers"
```

Regardless of whether you utilize approach 1 or approach 2 from listing 8.14 for implementing your facet filters, by utilizing the Term query parser you should make your queries faster. If you end up utilizing the Nested query parser syntax in approach 2, you will still need to escape quotes within your terms (since the whole nested query is in quotes), but this is a minor inconvenience if you want the capability to still use Boolean logic in your facet filters.

In this section, you have seen that applying filters to your facets is no different than applying any other filter in Solr, while seeing that it is possible to apply a separate filter per facet value or one filter for multiple facet terms. You also saw that it is possible to use the Term query parser to bypass text processing and avoid difficult-to-handle character escaping when applying a faceting filter, since you already know the exact term you need to match from the Solr index since facets are pulled directly from the index. At this point, you should be able to request and filter upon all of the basic facet types, but there is still more to explore. In the next section, you will uncover some useful ways to re-name facets for display purposes and to even bring back facet counts for documents which have already been filtered out.

8.7 Multi-select faceting, keys, and tags

When requesting a facet back from Solr, the name of the facet is not always the most useful for purposes of displaying results back to the user or even handling them within your application stack. Solr actually provides a very convenient ability to rename facets when they are returned, making facets much more user friendly for many use cases. Solr also provides the ability to bring back facet counts for documents that have been filtered out. This can be incredibly useful for implementing multi-select faceting – an ability to filter search results but still see the number of documents that would have matched had the filter not been applied. In this section, you will be introduced to the concepts of the keys, tags, and excludes local params (local params were introduced in chapter 7), which enable these useful facet re-naming and multi-select capabilities.

8.7.1 Keys

All facets have a name which allows developers to distinguish them from each other. By default the name of a facet is the field name (for Field Facets and Range Facets) or the query (for Query Facets) upon which the facet values and counts are calculated. Using the `key` local param, however, it is easy to rename any facet, as demonstrated in listing 8.15.

Listing 8.15 Renaming the key of a facet

DEFAULT SOLR FACETING NAMES

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field=city&#A
facet.query=price:[* TO 10]&#B
facet.query=price:[10 TO 25]&#B
facet.query=price:[25 TO 50]&#B
facet.query=price:[50 TO *]#B
```

Results:

```
  ...
  "facet_counts":{
     "facet_queries":{
       "price:[* TO 10]":11,#B
       "price:[10 TO 25]":5,#B
       "price:[25 TO 50]":4,#B
       "price:[50 TO *]":0},#B
     "facet_fields":{
       "city":[#A
         "Atlanta, GA",6,
         "New York, NY",4,
         "Austin, TX",3,
         "San Francisco, CA",3,
         "Chicago, IL",2,
         "Greenville, SC",1,
         "Los Angeles, CA",1]},
     "facet_dates":{},
     "facet_ranges":{}}}
  ...
```

RENAMING FACETS BY SPECIFYING AN EXPLICIT KEY

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field={!key="Location"}city&#C
facet.query={!key="<$10"}price:[* TO 10}&#D
facet.query={!key="$10 - $25"}price:[10 TO 25}&#D
facet.query={!key="$25 - $50"}price:[25 TO 50}&#D
facet.query={!key=">$50"}price:[50 TO *]#D
```

Results:

```
  ...
  "facet_counts":{
     "facet_queries":{
       "<$10":11,#D
       "$10 - $25":5,#D
       "$25 - $50":4,#D
       ">$50":0},#D
     "facet_fields":{
```

```
        "Location":[#C
          "Atlanta, GA",6,
          "New York, NY",4,
          "Austin, TX",3,
          "San Francisco, CA",3,
          "Chicago, IL",2,
          "Greenville, SC",1,
          "Los Angeles, CA",1]},
      "facet_dates":{},
      "facet_ranges":{}}}
  ...
```

#A A Field Facet is named after its Field by default
#B A Query Facet is named after its Query by default
#C A Field Facet being renamed from "city" to "Location"
#D Query Facets being given more readable names

The ability to rename a facet demonstrated in listing 8.15 can be very useful in many scenarios. It allows your search application to request Query Facets, for example, without requiring the application to interpret the queries from the result set during a post-processing stage. It also allows for user-friendly names to be assigned to facets regardless of the underlying field or query associated with the facet, which can make displaying the results in a user interface more straightforward. Additionally, by enabling each facet to be assigned a unique name, this capability to specify keys allows for more than one facet to be defined on the same field (which can be useful for Field Facets or Range Facets), each coming back under a unique name. One last advantage of this approach is that it allows multiple fields to be mapped into the same name depending upon query-time rules.

For example, say you have a search index with a field called "SecretInformationOnlyAvailableToSomeUsers" and another field called "InformationAvailableToAllUsers". With such a setup, you could create a facet at query time which either specified `facet.field={!key="Information"}InformationAvailableToAllUsers`, or else `facet.field={!key="Information"}SecretInformationOnlyAvailableToSomeUsers`. This layer of indirection can by handy in many scenarios, including any time you may want to re-define a facet to point to a different field with minimal changes to your application stack. In addition to renaming facets, Solr also provides the ability to tag certain filters so that you can control their interaction with other Solr features.

8.7.2 Tags, excludes, and multi-select faceting

When filters are applied to a Solr query request, the results must include every single filter. By default, the same holds true for facets. One of the problems this presents, however, is that often times it is useful to see facet counts for values which have been already been excluded from the query. Figure 8.8 demonstrates this problem by requesting some facets on our restaurant test data from section 8.2 and applying a filter for the state of California.

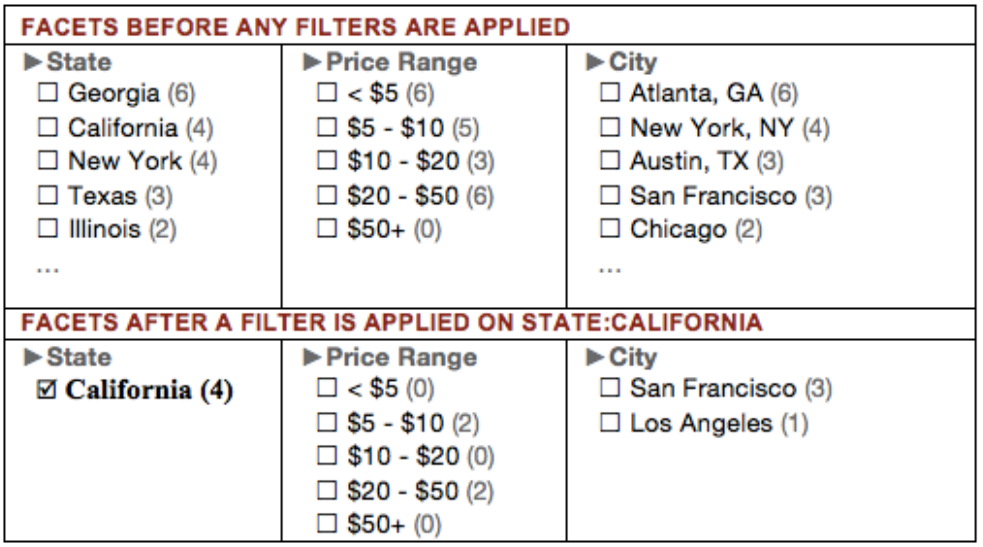


Figure 8.8. The problem with basic filtering on a facet. By default, after filtering upon a facet, the facet values which are returned for that facet no longer include the documents which were filtered out. This is problematic if you want to allow your user to select multiple values to include in the search, as they will never be able to "OR" any additional facet filters for the values since they are no longer visible as options.

Even though you would expect your search results to only display documents in California given the above user interface, you would also expect the facets to continue displaying the other states so that you could expand your query. The same principle applies for price ranges and cities – it is silly to only allow your users to search for one value per facet at a time.

Fortunately, Solr has a solution to this problem through a feature called Facet Exclusions. Facet Exclusions allow you to "add back" documents removed by any groups of filters that were applied on your search request. By adding back the removed documents to the facet counts, you can make facet counts on each facet effectively ignore any filters applied based upon that facet. The parameters necessary to implement this are the "tag" local param and the "ex" local param. Listing 8.16 demonstrates using these parameters to implement multi-select faceting on our example from figure 8.8.

Listing 8.16 Using tags and excludes to implement multi-select faceting

Query:

```
http://localhost:8983/solr/select?q=*:*&facet=true&facet.mincount=1&
facet.field={!ex=tagForState}state&#A
facet.field={!ex=tagForCity}city&
facet.query={!ex=tagForPrice}price:[* TO 5}&
facet.query={!ex=tagForPrice}price:[5 TO 10}&
```

```
facet.query={!ex=tagForPrice}price:[10 TO 20}&
facet.query={!ex=tagForPrice}price:[20 TO 50}&
facet.query=price:[50 TO *]&
fq={!tag="tagForState"}state:California#B
```

Results:

```
  ...
  "facet_counts":{
     "facet_queries":{
       "{!ex=tagForPrice}price:[* TO 5}":0,
       "{!ex=tagForPrice}price:[5 TO 10}":2,
       "{!ex=tagForPrice}price:[10 TO 20}":0,
       "{!ex=tagForPrice}price:[20 TO 50}":2,
       "price:[50 TO *]":0},
     "facet_fields":{
       "state":[
         "Georgia",6,#C
         "California",4,#C
         "New York",4,#C
         "Texas",3,#C
         "Illinois",2#C,
         "South Carolina",1],#C
       "city":[
         "San Francisco, CA",3,
         "Los Angeles, CA",1]},
     "facet_dates":{},
     "facet_ranges":{}}}
  ...
```

#A The "state" facet should ignore filters tagged "tagForState"
#B The query results will be limited to the state of California
#C The state facet ignores the filter on "state:California"
#D Other facets respect the filter on "state:California"

As you can see, a filter was applied on the query in listing 8.16 which limited the search to the state of California, but the total count of all documents matching the rest of the query was still returned for each state – not just for California. In terms of the mechanics of the query, a tag of "tagForState" was applied on the filter of for the state of California. This was accomplished using the syntax `fq={!tag="tagForState"}state:California`. Then, when the state facet was requested, it was told to exclude all filters tagged with "tagForState" using the syntax `facet.field={!ex=tagForState}state`.

Even though you cannot see the actual search results in listing 8.16, it is important to keep in mind that the documents returned from Solr are still constrained to the state of California (because the filter was applied to the query), even though state facet is not limited by that filter. It is also important to understand that all of the other facets not tagged with the "tagForState" tag are constrained by still constrained to the state of California for the same reason.

You may also note that each of the other requested facets (on `city` and `price`) also contained exclusion tags, but that those exclusion tags did not correspond with any tagged filters. While these exclusion tags were unnecessary, they did not cause any problems and

were simply ignored. If someone were to re-run the query and add a filter on one of the additional facet values, the currently unused exclusion tag would kick into effect. Whether you actually choose to apply exclusion tags on facets prior to the existence of any filters containing the excluded tags is up to you, but the point here is that doing so, while possibly wasteful syntax-wise, will not negatively impact the returned results.

It is possible to build some very interesting user interfaces and data analytics capabilities by mixing and matching tags and facets, but those use cases go well beyond what this chapter can cover. You should feel free to experiment with these capabilities in Solr if you want to learn more.

At this point you have seen all of the major use cases for facets, including Field Faceting, Query Faceting, and Range Faceting. There are several additional aspects of faceting that have not yet been discussed. In the next section, we will touch on some of the more advanced topics related to faceting which will be discussed in later chapters.

8.8 Beyond the basics

This chapter provides a solid overview of the most used faceting capabilities in Solr, but this is not the last time you will see faceting discussed. Faceting makes heavy use of Solr's caches, so you will need to optimize your use of Solr's built in caches in order to maximize the performance of your faceting requests. Working with caches will be discussed in-depth in chapter 13.

In addition to performance tuning, you will also see some more advanced forms of faceting in chapter 14. One of these advanced faceting capabilities is called Pivot Faceting, and it provides the ability for you to facet in many dimensions. For example, say it was not sufficient for your application to just know the top values from the "tags" field, and that you really needed to know the top tags per city. How would you go about accomplishing this? You could run a first search and get a facet for all the cities, and then run subsequent searches for each city to get its tags facet. Unfortunately, this approach does not scale very well and can easily result in you having to run dozens or hundreds of searches as your document set gets larger. Pivot facets, however, allow you to facet across multiple dimensions to pull back these kinds of calculations in a single search. If you would like to learn more, please checkout the Advanced Faceting section of chapter 14.

8.9 Summary

Congratulations on wrapping up an in-depth chapter on one of Solr's most powerful features. As you have seen, faceting provides a fast way to allow users to see a high-level overview of what kinds of documents their queries match. You would be hard pressed to find a major search-powered website today which does not provide some form of faceting to allow users to drill down and explore their search results. Using Solr, you have the ability to bring back the top values within each field using Field Facets, to bring back bucketed ranges of numbers or date values utilizing Range Facets, or to bring back the counts of any number of arbitrarily complex queries by utilizing Query Faceting.

You also saw that it is possible to utilize keys to rename facets as they are being returned, learned how to use tags and Facet Excludes to implement multi-select faceting which returns counts even for documents which are filtered out by a query, and explored multiple ways of applying filters to a query once facets are clicked upon by your users.

Finally, you encountered a brief introduction to the importance of effectively utilizing Solr's caches (covered further in chapter 12) for faceting performance optimization, and you heard about multi-dimensional Pivot Faceting, which will be discussed in detail in the "Complex data operations" chapter. At this point, you should be able to implement some fairly sophisticated search application utilizing all but the most complex forms of faceting available in Solr.

In the next chapter, you will learn how to use another very common Solr feature, Hit Highlighting, which allows the snippets of text matched in each document during a search to be returned for display in your list of search results, providing a potentially important insight to your users as to whether a document in your search results is worth exploring.

11

Result Grouping / Field Collapsing

This chapter covers

- How to exclude duplicate documents from search results
- Returning multiple groups of query results in a single search request
- Ensuring variety in search results by ensuring multiple categories are represented
- Grouping query results by field values, queries, or functions
- Data partitioning strategies necessary for scaling Solr's grouping functionality

Result Grouping is a useful capability in Solr for ensuring an optimal mix of search results is returned for a user's query. Result Grouping, also commonly referred to as Field Collapsing, is the ability to ensure only one document (or some limited number) is returned for each unique value (actual or dynamically computed) within a field. This can be useful if you have multiple similar documents – say products from the same company, locations for the same restaurant chain, or companies with multiple offices – but do not want an entire page of search results to only represent a single product, restaurant, or company.

You have probably seen implementations of this capability when using your favorite web search engine at some point. If you have ever seen search results telling you that many results matched your query, but only one (or the top few) were being displayed, you have encountered Field Collapsing. Often times, such a message will provide a link users can click to see the fully-expanded list of search results, along with a count of how many additional documents would have been returned had the search results not been collapsed.

In addition to collapsing search results to remove duplicate documents, the Result Grouping functionality in Solr provides several other useful features. In many ways, you can view Result Grouping in Solr as a more verbose form of Faceting. Instead of only returning separate Facet sections along with the counts for each value, however, Result Grouping actually returns the unique values and their counts (like Faceting) plus some number of

documents that contain each of the specified values. One way in which grouping is very different than Faceting, however, is that it returns the requested groups within the search results section, which means the groups and values within the groups are sorted based upon the sorts specified for the documents in the query. Grouping can be performed based upon field values, functions, or queries, making its applications numerous. While this may sound complex initially, we will cover several use cases to demonstrate this incredibly useful and (mostly) straightforward feature. Before jumping in, however, it will be useful to clarify the sometimes confusing distinction between the names "Result Grouping" and "Field Collapsing."

11.1 Result Grouping vs. Field Collapsing

One question commonly asked is why the Result Grouping functionality in Solr is referred to by two different names: "Result Grouping" and "Field Collapsing." The reason is partly historical but now mostly semantic. Field Collapsing is the more commonly used term for this functionality in other search engines, referring to the act of returning a normal set of results with duplicate values removed. An early version of this functionality was being developed for Solr and was unofficially called the "Field Collapsing" patch. The decision was later made to make this capability more generic, however, and the "Result Grouping" name was chosen to signify the more generic capabilities now available.

While returning a single results set collapsed on the duplicate values within a field ("Field Collapsing") is certainly a supported option using Solr's Result Grouping functionality, it is also possible to return multiple result sets – or "groups" from a single query. The "Result Grouping" name thus signifies a more generic capability than the traditional "Field Collapsing" use-case. This chapter will demonstrate multiple uses for Solr's Result Grouping functionality, starting with the most common one – "Field Collapsing" to removing duplicate documents from a set of search results.

11.2 Skipping Duplicate Documents

Let's say you are running an online e-commerce website that sells user-posted items. While on the one hand it may be nice to run a search for an item and see thousands of copies of identical products, you may actually provide a much better user experience if instead you only show one document per unique item (possibly along with a count of how many of each item exists). This will allow some diversity in the search results, just in case the item that shows up at the top is not exactly what the user was trying to find. As an example, say a user searched for the term "Spider-man." If the top match was "The Amazing Spider-man - 2012" and you had 100 copies of this same item, the top 100 results would be for this one product, assuming they all had the same relevancy score based upon their content. If, instead, you were to group on the name of the product and ask for only one item per group, "The Amazing Spider-man - 2012" would only show up once amidst other possibilities like "Spider-man – 2002", "Spider-man 2 – 2004", and "Amazing Spider-man Comic #2". In

most scenarios, this will provide a better user experience than the default option of listing "The Amazing Spider-man - 2012" over and over.

In order to demonstrate this capability in action, some example product documents have been added in the source code that accompanies this book. After following the steps in Appendix A to obtain a clean version of Solr, you can start up Solr and add the example documents using the commands in Listing 11.1.

Listing 11.1 Indexing example documents for e-commerce site

```
cd {SOLR_ROOT}/example/solr/collection1/conf/
cp schema.xml schema.xml.original
cp ~/example-docs/ch11/ecommerce_schema.xml schema.xml
cd ../../../
cp ~/example-docs/ch11/ecommerce.xml ./exampledocs/
java –jar start.jar
cd exampledocs
java -jar post.jar ecommerce.xml
```

With our e-commerce search engine up and running, we can now demonstrate Solr's Result Grouping capabilities in action. Listing 11.2 demonstrates a standard search request for the query "Spider-man" without grouping turned on.

Lsting 11.2 Standard search containing duplicate documents

Request:

```
http://localhost:8983/solr/collection1/select?
   wt=json&
   indent=true&
   echoParams=none&
   df=content&fl=id,product,format&
   sort=popularity asc&
   q=spider-man
```

Response:

```
   {
      "responseHeader":{
        "status":0,
        "QTime":2},
      "response":{"numFound":18,"start":0,"docs":[
          {
            "id":"4",
            "format":"dvd",
            "product":"The Amazing Spider Man - 2012"},
          {
            "id":"5",
            "format":"blue-ray",
            "product":"The Amazing Spider Man - 2012"},
          {
            "id":"6",
            "format":"dvd",
            "product":"Spider Man - 2002"},
```

```
        {
          "id":"7",
          "format":"blue-ray",
          "product":"Spider Man - 2002"},
        {
          "id":"8",
          "format":"dvd",
          "product":"Spider Man 2 - 2004"},
        {
          "id":"9",
          "format":"blue-ray",
          "product":"Spider Man 2 - 2004"},
        {
          "id":"11",
          "format":"xbox 360",
          "product":"The Amazing Spider-Man"},
        {
          "id":"12",
          "format":"ps3",
          "product":"The Amazing Spider-Man"},
        {
          "id":"13",
          "format":"xbox 360",
          "product":"Spider-Man: Edge of Time"},
        {
          "id":"14",
          "format":"ps3",
          "product":"Spider-Man: Edge of Time"}]
    }}
```

The results from listing 11.2 almost certainly represent a bad user experience, as they seem to present every format of the same product as separate products, listing it multiple times. By turning Grouping on, the results appear much more diverse, providing a cleaner user experience, as demonstrated in Listing 11.3.

Listing 11.3 Grouped search results collapsing on the product field

Request:

```
http://localhost:8983/solr/collection1/select?
     wt=json&
     indent=true&
     echoParams=none&
     df=content&
     fl=id,product,format&
     sort=popularity asc
     q=spider-man&
     group=true&
     group.field=product
```

Response:

```
{
   "responseHeader":{
```

```
  "status":0,
  "QTime":3},
"grouped":{
  "product":{
    "matches":18,
    "groups":[{
        "groupValue":"The Amazing Spider Man - 2012",
        "doclist":{"numFound":2,"start":0,"docs":[
            {
              "id":"4",
              "format":"dvd",
              "product":"The Amazing Spider Man - 2012"}]
        }},
      {
        "groupValue":"Spider Man - 2002",
        "doclist":{"numFound":3,"start":0,"docs":[
            {
              "id":"6",
              "format":"dvd",
              "product":"Spider Man - 2002"}]
        }},
      {
        "groupValue":"Spider Man 2 - 2004",
        "doclist":{"numFound":3,"start":0,"docs":[
            {
              "id":"8",
              "format":"dvd",
              "product":"Spider Man 2 - 2004"}]
        }},
      {
        "groupValue":"The Amazing Spider-Man",
        "doclist":{"numFound":2,"start":0,"docs":[
            {
              "id":"11",
              "format":"xbox 360",
              "product":"The Amazing Spider-Man"}]
        }},
      {
        "groupValue":"Spider-Man: Edge of Time",
        "doclist":{"numFound":2,"start":0,"docs":[
            {
              "id":"13",
              "format":"xbox 360",
              "product":"Spider-Man: Edge of Time"}]
        }},
      {
        "groupValue":"Spider-Man Halloween Costume",
        "doclist":{"numFound":1,"start":0,"docs":[
            {
              "id":"15",
              "format":"costume",
              "product":"Spider-Man Halloween Costume"}]
        }},
      {
        "groupValue":"Boys Spider-Man T-shirt",
        "doclist":{"numFound":1,"start":0,"docs":[
            {
```

```
        "id":"21",
        "format":"shirt",
        "product":"Boys Spider-Man T-shirt"}]
  }},
{
  "groupValue":"Amazing Spider-Man Comic #1",
  "doclist":{"numFound":1,"start":0,"docs":[
      {
        "id":"22",
        "format":"paperback",
        "product":"Amazing Spider-Man Comic #1"}]
  }},
{
  "groupValue":"Amazing Spider-Man Comic #2",
  "doclist":{"numFound":1,"start":0,"docs":[
      {
        "id":"23",
        "format":"paperback",
        "product":"Amazing Spider-Man Comic #2"}]
  }},
{
  "groupValue":"Amazing Spider-Man Comic #3",
  "doclist":{"numFound":1,"start":0,"docs":[
      {
        "id":"24",
        "format":"paperback",
        "product":"Amazing Spider-Man Comic #3"}]
  }}]}}}
```

Listing 11.3 demonstrates several noteworthy properties of grouping. First, notice that grouping must be turned on by specifying the `group=true` parameter. Second, the `group.limit` parameter was set to `1`, indicating that only one document should be returned for each unique value within a Group. It can be useful to set the `group.limit` to an integer higher than `1`, as you will see in section 11.4, but for removing all duplicate documents you would generally only want one document returned per unique value upon which you are collapsing.

You will also notice that the results format in listing 11.3 is substantially different than the default Solr results format. This more verbose format is necessary to communicate all of the information associated with grouped search results – the name of the field that is grouped upon, the unique terms within that field (`groupValue`) which defines each group, and the total number of results (`matches`) before collapsing occurred in each group.

Unfortunately, it can often be inconvenient to support parsing out two separate Solr results formats to handle grouping. Thankfully, if you are looking to remove duplicates and do not need all of this additional grouping information, Solr provides a `group.main` parameter which, if set to `true`, will merge the results from each group back into a flat list and return it in the main results format. Listing 11.4 demonstrates the same query as listing 11.3, but with `group.main` set to `true`.

Listing 11.4 Flattening grouped results into the main search results format

Request:

```
http://localhost:8983/solr/collection1/select?
     wt=json&
     indent=true&
     echoParams=none&
     df=content&
     fl=id,product,format&
     sort=popularity asc&
     q=spider-man&
     group=true&
     group.field=product&
     group.main=true
```

Response:

```
{
   "responseHeader":{
     "status":0,
     "QTime":8},
   "response":{"numFound":18,"start":0,"docs":[
       {
         "id":"4",
         "format":"dvd",
         "product":"The Amazing Spider Man - 2012"},
       {
         "id":"6",
         "format":"dvd",
         "product":"Spider Man - 2002"},
       {
         "id":"8",
         "format":"dvd",
         "product":"Spider Man 2 - 2004"},
       {
         "id":"11",
         "format":"xbox 360",
         "product":"The Amazing Spider-Man"},
       {
         "id":"13",
         "format":"xbox 360",
         "product":"Spider-Man: Edge of Time"},
       {
         "id":"15",
         "format":"costume",
         "product":"Spider-Man Halloween Costume"},
       {
         "id":"21",
         "format":"shirt",
         "product":"Boys Spider-Man T-shirt"},
       {
         "id":"22",
         "format":"paperback",
         "product":"Amazing Spider-Man Comic #1"},
       {
         "id":"23",
         "format":"paperback",
```

```
          "product":"Amazing Spider-Man Comic #2"},
        {
          "id":"24",
          "format":"paperback",
          "product":"Amazing Spider-Man Comic #3"}]
    }}
```

The main disadvantages of using the `group.main` option, of course, is that you lose access to the total number of un-collapsed results within each group, but if this is not important in your search application then this may be a fair trade-off in return for not having to handle two different search results formats. You also lose the name of the group in this simple format, but this can often be derived from the results by returning the field in each document that you grouped upon (assuming it is a single-valued field). The other disadvantage of using the `group.main` format is that it only supports a single group being requested. Thusfar, we have only specified a single `group.field=product` parameter, but Solr actually supports returning multiple groups. For example, you could ask for both a `group.field=type` and a `group.field=format`, and Solr would return both groups in the grouped results format. If you specify `group.main=true`, however, Solr will only return the last group that you specify in the Solr url.

There is also another option in-between the advanced grouping format (the default) and the `group.main` format: the simple grouping format. By specifying `group.format=simple`, you can return multiple groups (like the default advanced grouping format) while still returning the results for each group request in a flat list like the `group.main=true` option. In fact, from an implementation standpoint, setting `group.main=true` actually just uses the `group.method=simple` capability and returns the last specified group in the main results list. Listing 11.5 demonstrates the same query as in listings 11.3 and 11.4, but utilizing the simple grouping format.

Listing 11.5 The simple grouping format

Request:

```
http://localhost:8983/solr/collection1/select?
     wt=json&
     indent=true&
     echoParams=none&
     df=content&
     fl=id,product,format&
     sort=popularity asc&
     q=spider-man&
     group=true&
     group.field=product&
     group.format=simple
```

Response:

```
{
   "responseHeader":{
     "status":0,
     "QTime":2},
```

```
"grouped":{
  "product":{
    "matches":18,
    "doclist":{"numFound":18,"start":0,"docs":[
        {
          "id":"4",
          "format":"dvd",
          "product":"The Amazing Spider Man - 2012"},
        {
          "id":"6",
          "format":"dvd",
          "product":"Spider Man - 2002"},
        {
          "id":"8",
          "format":"dvd",
          "product":"Spider Man 2 - 2004"},
        {
          "id":"11",
          "format":"xbox 360",
          "product":"The Amazing Spider-Man"},
        {
          "id":"13",
          "format":"xbox 360",
          "product":"Spider-Man: Edge of Time"},
        {
          "id":"15",
          "format":"costume",
          "product":"Spider-Man Halloween Costume"},
        {
          "id":"21",
          "format":"shirt",
          "product":"Boys Spider-Man T-shirt"},
        {
          "id":"22",
          "format":"paperback",
          "product":"Amazing Spider-Man Comic #1"},
        {
          "id":"23",
          "format":"paperback",
          "product":"Amazing Spider-Man Comic #2"},
        {
          "id":"24",
          "format":"paperback",
          "product":"Amazing Spider-Man Comic #3"}]
    }}}}
```


You have seen throughout this section how to collapse the results of a query into groups so as to remove duplicate documents. You have also seen three formats in which grouped search results can be returned: the default advanced grouping format, the simple grouping format, and the returning of a single collapsed group in the main search results. Each of these formats strikes a balance between backwards compatibility with the standard search results format and the richness of information available to describe the identified groups.

The collapsing of results down to a single document per unique field value represents the essence of what is meant by the use of the traditional term “Field Collapsing.”

Throughout the rest of this chapter, we will see some more advanced use cases for the more generic grouping capabilities available in Solr. The next section will begin by demonstrating how to request multiple documents per group in a single query.

11.3 Returning multiple documents per group

Collapsing to a single document per unique field value is not the only practical use-case for Solr’s Grouping functionality, of course. Returning to our e-commerce search engine example from the previous section, imagine that if instead of only collapsing duplicate documents, we could actually guarantee that we would return a fixed number of results from each product category. For example, going back to our earlier search (in section 11.2) for “spider-man”, imagine that instead of returning a flat list of collapsed (de-duplicated) documents, that instead, we returned up to three documents per unique category. If you still have your Solr instance up and running from listing 11.1 (if not, you can easily go back and restart it), we can run this query. The example documents contain a `type` field that we can group upon, requesting a limit of three documents per group. Additionally, let us request a maximum of 5 total groups be returned. Listing 11.6 demonstrates this query.

Listing 11.6 Returning multiple values per group

Request:

```
http://localhost:8983/solr/collection1/select?
   q=spider-man&
   wt=json&
   indent=true&
   echoParams=none&
   df=content&
   fl=id,product,format&
   sort=popularity asc&
   group=true&
   group.field=type&
   group.limit=3&
   rows=5&
   start=0&
   group.offset=0
```

Response:

```
   {
      "responseHeader":{
        "status":0,
        "QTime":2},
      "grouped":{
        "type":{
          "matches":18,
          "groups":[{
              "groupValue":"Movies",
              "doclist":{"numFound":8,"start":0,"docs":[
                  {
```

```
          "id":"4",
          "format":"dvd",
          "product":"The Amazing Spider Man - 2012"},
        {
          "id":"5",
          "format":"blue-ray",
          "product":"The Amazing Spider Man - 2012"},
        {
          "id":"6",
          "format":"dvd",
          "product":"Spider Man - 2002"}]
    }},
  {
    "groupValue":"Video Games",
    "doclist":{"numFound":4,"start":0,"docs":[
        {
          "id":"11",
          "format":"xbox 360",
          "product":"The Amazing Spider-Man"},
        {
          "id":"12",
          "format":"ps3",
          "product":"The Amazing Spider-Man"},
        {
          "id":"13",
          "format":"xbox 360",
          "product":"Spider-Man: Edge of Time"}]
    }},
  {
    "groupValue":"Clothing",
    "doclist":{"numFound":2,"start":0,"docs":[
        {
          "id":"15",
          "format":"costume",
          "product":"Spider-Man Halloween Costume"},
        {
          "id":"21",
          "format":"shirt",
          "product":"Boys Spider-Man T-shirt"}]
    }},
  {
    "groupValue":"Comic Books",
    "doclist":{"numFound":3,"start":0,"docs":[
        {
          "id":"22",
          "format":"paperback",
          "product":"Amazing Spider-Man Comic #1"},
        {
          "id":"23",
          "format":"paperback",
          "product":"Amazing Spider-Man Comic #2"},
        {
          "id":"24",
          "format":"paperback",
          "product":"Amazing Spider-Man Comic #3"}]
    }},
  {
```

```
            "groupValue":"Action Figures",
            "doclist":{"numFound":1,"start":0,"docs":[
                {
                  "id":"25",
                  "format":"n/a",
                  "product":"Marvel Legends Icon: Spider Man 12\" Action
Figure"}]
            }}]}}}
```

Several important points can be gleaned from listing 11.6. First, notice that no group contains more than three results, but that not all groups contain three results. Even though the `group.limit` is set to 3, this only sets the upper limit, as any group without three documents cannot return a full three documents. Second, notice that the `rows=5` parameter is controlling not how many documents are returned, but instead it is controlling how many groups are returned. In the default advanced grouping format, both the rows parameter and the start parameter apply to the groups instead of to the documents within each group. That is, `rows` controls how many groups are returned, while `group.limit` controls how many documents are returned within each group. Likewise, the start parameter controls the group offset for paging through groups, while the `group.offset` parameter controls the document offset for paging through the documents within each group.

One final very important aspect of grouping can be gleaned from listing 11.6: the way sorting interacts with grouping. Conceptually, all groups are sorted based upon the order in which their top document is sorted. What this means is that, assuming groupings were not enabled, all of the documents would appear in their sorted order (by relevancy score by default). If the highest sorted document were to have a `type` field value of “Movies”, the second highest a value of “Comic Books”, the third highest again a value of “Movies” and the fourth highest a value of “Clothing”, then the sorted order of the groups would be “Movies”, “Comic Books”, and then “Clothing”.

Inside of each group, the documents are also sorted, by default, in the order of their first appearance. This means that since “Movies” is the top category group and since three movies are requested to appear within the Movies category group, that the second and third movie are likely to be less relevant (sorted lower) in an absolute sense even though they show up higher in the results because they get promoted up to the top group of “Movies.” For this reason, it is often uncommon for someone to request a `group.limit` of greater than `1` when using the `group.main` or the `group.format=simple` formats, as the sorted order of results may appear quite strange without the structure provided by the advanced grouping format to distinguish when one group ends and another begins.

While grouping on the values within a field provides so very useful search capabilities, as shown in the above examples, much more is possible with Solr’s Result Grouping capabilities. The next section will begin by demonstrating the ability to group on more than just field values - you will see how to group by arbitrary queries and functions, as well.

11.4 Grouping by functions & queries

In addition to supporting grouping by unique field values, Solr also supports two additional grouping use cases. The first of these is similar to grouping on a field, but it instead allows grouping by dynamically computed values by utilizing function queries. The second additional use case is Solr's query grouping capability, which essentially allows multiple queries to be run concurrently and returned as separate result sets.

11.4.1 Grouping by function

Grouping based upon functions is accomplished using the `group.func` parameter. We will not cover all of the possible functions here (see chapter 15 for in-depth coverage of functions in Solr), but listing 11.7 demonstrates a search result grouped by a function. In this case, the function is trying to group the search results into three groups by popularity (most popular = 1, somewhat popular = 2, least popular = 3).

Listing 11.7 Search result grouped by a function

Request:

```
http://localhost:8983/solr/collection1/select?
     wt=json&
     indent=true&
     echoParams=none&
     df=content&
     fl=id,product,format&
     sort=popularity asc&
     q=spider-man&
     group=true&
     group.limit=3&
     rows=5&
     group.func=map(map(map(popularity,1,5,1),6,10,2),11,100,3)
```

Response:

```
{
   "responseHeader":{
     "status":0,
     "QTime":3},
   "grouped":{
     "map(map(map(popularity,1,5,1),6,10,2),11,100,3)":{
       "matches":18,
       "groups":[{
           "groupValue":1.0,
           "doclist":{"numFound":2,"start":0,"docs":[
               {
                 "id":"4",
                 "format":"dvd",
                 "product":"The Amazing Spider Man - 2012"},
               {
                 "id":"5",
                 "format":"blue-ray",
                 "product":"The Amazing Spider Man - 2012"}]
           }},
         {
```

```
      "groupValue":2.0,
      "doclist":{"numFound":4,"start":0,"docs":[
          {
            "id":"6",
            "format":"dvd",
            "product":"Spider Man - 2002"},
          {
            "id":"7",
            "format":"blue-ray",
            "product":"Spider Man - 2002"},
          {
            "id":"8",
            "format":"dvd",
            "product":"Spider Man 2 - 2004"}]
      }},
    {
      "groupValue":3.0,
      "doclist":{"numFound":12,"start":0,"docs":[
          {
            "id":"11",
            "format":"xbox 360",
            "product":"The Amazing Spider-Man"},
          {
            "id":"12",
            "format":"ps3",
            "product":"The Amazing Spider-Man"},
          {
            "id":"13",
            "format":"xbox 360",
            "product":"Spider-Man: Edge of Time"}]
      }}]}}}
```

As you can see in listing 11.7, grouping by a function is conceptually the same as grouping by a field value, except that the value you group upon is computed dynamically. In this case, all popularity values were mapped into three groups (popularity 1-5 was mapped into group 1, popularity 6-10 was mapped into group 2, and popularity 11-100 was mapped to group 3). Functions can be nested, as you see in this case which nests three map functions, which means you have full control over the values that are computed if you want to manipulate them by combining multiple functions together. If function grouping is too limiting for your use case, it is also possible to group by a query so that you can specify your own arbitrary values upon which to group.

11.4.2 Grouping by query

In section 11.3, you saw that it is possible to collapse search results so that no more than a few documents are returned matching a particular value within a field (using the `group.field` parameter). In addition to grouping on pre-defined field values, it can also be useful to dynamically group on arbitrary queries. For example, a customer-centric user experience could be highly customized to return three sets of search results: those within 50 kilometers of the user, those within the customer's price range, and those within the customer's favorite category. Because Solr allows multiple `group.query` parameters to be

sent in the same request, it is possible to bring back each of these sets of search results as a separate group in the search results.

To demonstrate this capability utilizing our dataset for this chapter, let us see what it would look like to request three somewhat arbitrary query groups: one query that matches all movies, one query that matches anything in a DVD format, and one query which is for the specific product named "The Hunger Games." Listing 11.8 demonstrates how to accomplish this.

Listing 11.8 Search results grouped by multiple queries

Request:

```
http://localhost:8983/solr/collection1/select?
   sort=popularity asc&
   wt=json&
   indent=on&fl=id,type,format,product&
   echoParams=none
   group.limit=2&
   df=content&
   q=*:*&
   group=true&
   group.query=type:Movies&
   group.query=Games&
   group.query="The Hunger Games"
```

Results:

```
{
   "responseHeader":{
     "status":0,
     "QTime":1},
   "grouped":{
     "type:Movies":{
       "matches":26,
       "doclist":{"numFound":11,"start":0,"docs":[
           {
             "id":"1",
             "type":"Movies",
             "format":"dvd",
             "product":"The Hunger Games"},
           {
             "id":"4",
             "type":"Movies",
             "format":"dvd",
             "product":"The Amazing Spider Man - 2012"}]
       }},
     "Games":{
       "matches":26,
       "doclist":{"numFound":7,"start":0,"docs":[
           {
             "id":"1",
             "type":"Movies",
             "format":"dvd",
             "product":"The Hunger Games"},
           {
```

```
          "id":"2",
          "type":"Video Games",
          "format":"xbox 360",
          "product":"Dance Dance Revolution"}]
    }},
  "hunger":{
    "matches":26,
    "doclist":{"numFound":2,"start":0,"docs":[
        {
          "id":"1",
          "type":"Movies",
          "format":"dvd",
          "product":"The Hunger Games"},
        {
          "id":"3",
          "type":"Books",
          "format":"paperback",
          "product":"The Hunger Games"}]
    }}}}
```

Listing 11.8 demonstrates three important takeaways. First, it is possible to request multiple groups back from Solr. This is actually true for any kind of grouping query (`group.field`, `group.func`, or `group.query`) – any number of groups can be returned from Solr within a single request. Second, a query group is essentially a way to perform multiple sub-searches of the original search. In this case, the initial query was a wide-open search (`q=*:*`), which allows you to essentially run as many queries as you want within a single request, each returning separate sets of results. Third, while it is true that a document will only appear once with a grouped result set, it is important to note that when multiple group parameters are used in a Solr request, each set of grouped results can contain that document again. In other words, it is as if you are literally running multiple searches within the same request, because each of the separately requested query groups can contain the same documents if the query in their corresponding `group.query` parameter matches those documents.

With this ability to run multiple sub-searches in a single Solr request, some interesting interactions take place between Solr's grouping functionality and the results paging and document sorting which also occur during the request. The next section will dive into these interactions.

11.5 Paging & sorting grouped results

Grouping, because of its richer search results structure, introduces some additional complexity when paging through and sorting search results. You will recall from chapter 7 that Solr utilizes the `rows` parameter to determine how many documents to return from Solr for a standard search query. When grouping results, however, there is an extra layer of complexity – what is it you are actually trying to put a limit upon? Do you wish to return a certain number of documents per group, a certain number across all groups, or a certain number of groups? Similar questions exist when using Solr's `start` parameter to page through grouped search results and the `sort` parameter to sort the grouped search results.

What does it mean to page through and sort search results in the context of multiple groups of search results?

In order to handle this additional complexity, Solr's grouping functionality applies these global parameters – rows, start, and sort – to the groups themselves. The `rows` parameter determines how many groups to return, the `start` parameter controls paging through available groups, and the `sort` parameter controls how groups are sorted (based upon their top document) as opposed to how documents are sorted across groups.

Should you additionally need to increase the number of documents per group, page through the results within a group, or sort the results within the groups differently, Solr's Result Grouping functionality has separate parameters to control this. As indicated in section 11.3, the `group.limit` parameter specifies the maximum number of results to return per group, performing the behavior that the `rows` parameter performs on a non-grouped search. The `group.offset` parameter allows you to page through the results within a group, performing the behavior that the `start` parameter provides on a non-grouped search. Finally, the `group.sort` parameter allows you to re-sort the documents within your groups, even though they have already been sorted initially by the sort parameter to determine the order in which the groups will appear. Some very interesting uses for this two-pass sorting could be implemented, with one phase finding the relevant groups of documents and the other phase sorting the documents within that group based upon some other business need.

At the end of the day, perhaps the easiest way to think of paging, sorting, and limiting the results in a grouped request is to think of the groups as the actual documents Solr is returning. You can page, sort, and limit those "document groups" just like you would page, sort, and limit single documents in a standard search request. Solr then provides the additional group parameters (`group.limit`, `group.offset`, and `group.sort`) to help you refine the documents within the groups for display.

11.6 Grouping gotchas

While Solr's Grouping capabilities prove useful for many use-cases, there are a few aspects of this functionality that can be a bit tricky to navigate. Understanding these details is important for determining how to partition your data, what Facet counts represent, and how your query performance will be impacted.

11.6.1 Faceting Upon Result Groups

By default, Facet counts are based upon the original query results, not the grouped results. This means that whether you turn grouping on for a query or not, the facet counts will be the same. It is possible, however, to only return facet counts for "collapsed" results by setting the parameter `group.facet=true`.

If you were to turn faceting on and facet on the `type` field using the ecommerce data for this chapter, you would get the results in listing 11.9.

Listing 11.9 Standard Faceting on grouped search results

Request:

```
http://localhost:8983/solr/collection1/select?
   wt=json&
   indent=true&
   echoParams=none&
   df=content&
   fl=product&
   group=true&
   sort=popularity asc&
   q=*:*&
   facet=true&
   facet.mincount=1&
   group.format=simple&
   fq=type:Movies&
   facet.field=type&
   group.field=product
```

Response:

```
   {
      ...
      "grouped":{
        "product":{
          "matches":11,
          "doclist":{"numFound":11,"start":0,"docs":[
              {
                "product":"The Hunger Games"},
              {
                "product":"The Amazing Spider Man - 2012"},
              {
                "product":"Spider Man - 2002"},
              {
                "product":"Spider Man 2 - 2004"},
              {
                "product":"Top Gun"},
              {
                "product":"A Beautiful Mind"}]
          }}},
      "facet_counts":{
        ...
        "facet_fields":{
          "type":[
            "Movies",11]},
        ...
   }}
```

The first characteristic of listing 11.9 that you should notice is that while eleven documents matched the filter for `type:Movies`, only six documents were returned in the grouped search results when the results were collapsed by the `product` field. This is because several of the movies contained duplicate documents corresponding with different movie formats (DVD, Blue-ray, and even VHS), so the duplicate documents with the same product name were were collapsed out. The second characteristic you may notice is that the facet counts

are still based upon the total number of documents and not the collapsed number of documents.

From a customer's perspective, however, there are not eleven unique movies upon which they should be able to facet. The customer most likely does not care about all the format variations and simply wants to see that there are six unique movies. Thankfully, Solr provides a parameter, `group.facet=true`, which you can specify to achieve this desired result, as shown in listing 11.10.

Listing 11.10 Limiting Facet counts to grouped search results

Request:

```
http://localhost:8983/solr/collection1/select?
   wt=json&
   indent=true&
   echoParams=none&
   df=content&
   fl=product&
   group=true&
   sort=popularity asc&
   q=*:*&
   facet=true&
   facet.mincount=1&
   group.format=simple&
   fq=type:Movies&
   facet.field=type&
   group.field=product&
   group.facet=true
```

Response:

```
   {
      ...
      "grouped":{
        "product":{
          "matches":11,
          "doclist":{"numFound":11,"start":0,"docs":[
              {
                "product":"The Hunger Games"},
              {
                "product":"The Amazing Spider Man - 2012"},
              {
                "product":"Spider Man - 2002"},
              {
                "product":"Spider Man 2 - 2004"},
              {
                "product":"Top Gun"},
              {
                "product":"A Beautiful Mind"}]
          }}},
      "facet_counts":{
        ...
        "facet_fields":{
          "type":[
            "Movies",6]},
        ...
   }}
```

This ability to group on a field and then facet on the collapsed group can come in very handy for situations like the above. Unfortunately the `group.facet=true` parameter is global (it is turned on or off for all requested facets), and it cannot be applied to multiple requested results groups in the current version of Solr. As such, if you do turn on grouped faceting, you should be aware that it only applies to the first requested result grouping.

11.6.2 Distributed Result Grouping

One very important consideration when utilizing Solr's Result Grouping functionality is how it interacts with distributed search. Unlike standard searches, Result Grouping cannot be said to fully work in distributed mode… instead, it is more accurate to say that it works in a pseudo-distributed mode. Grouping does return aggregated results in distributed mode, but the results are only the aggregates of the groups calculated locally on each Solr core.

Why does this matter? It matters because if the values you are grouping on are randomly distributed across Solr cores then the counts of grouped documents are going to be inaccurate. If you were grouping a query for products by a field containing the product manufacturer's name (to see all the unique products for that manufacturer), your total count of groups would be roughly the sum of the group counts for each Solr core you search against in a distributed search. If and only if your documents are partitioned into separate shards by manufacturer name would you get the correct group count, since each group is guaranteed to only exist on one shard. This is an important consideration to keep in mind if you plan on using Solr's grouping functionality and require that the count of groups (returned by `group.ncount=true`) is accurate. If your data is not sharded by the field you are grouping on and you are performing a distributed search, the count of groups that is returned will merely be an upper limit. In addition to these data partitioning limitations, a few of the grouping parameters simply do not currently work in a distributed mode: `group.truncate` and `group.func`, so be careful using these if you think your data will grow beyond what one Solr core can handle.

11.6.3 Returning a flat list

You saw in section 11.2 how to return grouped results in the simple (flat) grouped format instead of the default advanced grouping format by setting the parameter `group.format=simple`. You also saw that you could return the first group in the main default results format (like a non-grouped query) by setting the `group.main=true` parameter. While both of these options are useful, you should consider wisely whether you can live without the extra information provided by the advanced grouping format. Without the advanced format, you cannot request the number for groups, for example. If you are using grouping to collapse documents, this means that you will not know how many unique values were found without using the advanced format. Because it can often be challenging to change the response format in your application down the line, you should carefully consider which format to use during your initial application development.

11.6.4 Grouping performance

While grouping is a very powerful feature, it is considerably slower than a standard Solr query. Based upon unscientific measurements, some have seen an average query take about three times longer to execute when grouping was requested for the purpose of collapsing down to one unique value (sometimes more, sometimes less depending upon the complexity of the query).

In order to help speed up grouping, a cache can be turned on for your grouping queries utilizing the `group.cache.percent` query parameter. This parameter defaults to 0, so setting it to any value between 1 and 100 will turn it on. Because Result Grouping is internally implemented as two actual searches, turning this cache on can increase the speed of queries by caching one of those searches. It is an advanced feature, but it has been demonstrated to improve the performance of Boolean queries, wildcard queries, and fuzzy queries. Be careful, though, as it has also been shown to decrease performance on very simple queries such as term queries and match all (`*:*`) queries.

You will need to measure the performance impact for your own search application, including whether to turn group caching on or not, but a query using Result Grouping is likely to demonstrate a performance slowdown relative to a non-grouped query. You should certainly take the performance impact into consideration when determining whether Solr's grouping capabilities make sense for your use case.

11.7 Summary

This chapter has demonstrated Solr's Result Grouping / Field collapsing capabilities. This functionality enables many query options, including removing duplicate or near duplicate documents dynamically at query time, ensuring results come from diverse categories, or even executing multiple queries with a single request. Result Grouping can also be used to modify the results used for faceting, essentially preventing duplicate documents from being considered each time for facet counts. You also saw how to return multiple documents per group and how to group on fields, functions, and queries. Finally, you got to see some of the trickier gotchas associated with using Solr's Result Grouping functionality, and you saw both the performance impacts of Result Grouping as well as the formats in which grouped results can be returned.

The last five chapters have covered many of Solr's key search features, and by now you should feel prepared to build a world-class search application based upon Solr. Taking that search application to production will require some additional work, however, and that effort is the subject of our next chapter.