SAMPLE CHAPTER

Solr IN ACTION

Trey Grainger
Timothy Potter

FOREWORD BY Yonik Seeley

MANNING

Solr in Action

by Trey Grainger
Timothy Potter

Chapter 1

Copyright 2014 Manning Publications

brief contents

1 Introduction to Solr

This chapter covers

- Characteristics 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 such as social media, cloud computing, mobile applications, and big data, these are exciting, and challenging, times to be in computing. One of the main challenges facing software architects is handling the massive volume 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, nonrelational data storage and processing technologies, collectively known as NoSQL (Not only SQL). These systems share a common design pattern of matching storage and processing engines to specific types of data rather than forcing all data into the once-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 Apache Solr, a specific NoSQL technology. Solr, just as its nonrelational 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 that 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 tens of milliseconds.
- *Large volumes of documents*—Solr is designed to deal with indexes containing many millions of documents.
- *Text-centric*—Solr is optimized for searching natural-language text, like emails, web pages, resumes, PDF documents, and social messages such as 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. You'll begin by learning about the types of data and use 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?

Because you're looking at this book, we suspect that you already have an idea about why you need a search engine. 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 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, 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. Search engines like Solr are optimized to handle data exhibiting four main characteristics:

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, not strict rules. Let's dig into each to see why they're important for search. For now, we'll focus on the high-level concepts; we'll get into the "how" in later chapters.

TEXT-CENTRIC

You'll undoubtedly encounter the term *unstructured* used 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 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 *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 nontext data such as dates and numbers, but its primary strength is handling text data based on natural language.

The *centric* part is important because if users aren't interested in the information in the text, a search engine may not be the best solution for your problem. Consider an application in which employees create travel expense reports. Each report contains a number of structured data fields such as date, expense type, currency, and amount. In addition, each expense may include a notes field in which 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. Just because data contains text fields doesn't mean that data is a natural fit for a search engine.

Think about whether your data is text-centric. The main consideration 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. You'll see how to unlock the structure in text by using Solr's text analysis capabilities in chapters 5 and 6.

READ-DOMINANT

Another key aspect of data that search engines handle effectively is that data is read-dominant and therefore intended to be accessed efficiently, as opposed to updated frequently. Let's be clear that Solr does allow you to update existing documents in

your index. Think of *read-dominant* as meaning that documents are read far more often than 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* (NRT) 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 write data to Solr, it's intended to be read and reread myriad times over its lifetime. 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 must update existing data in a search engine often, 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

Until now, we've talked about data, but in reality, search engines work with documents. In a search engine, a *document* is a self-contained collection of fields, in which 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 subfields. 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. 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, 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, 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.

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 is *not* a general-purpose data-storage and processing technology.

The whole point of having such a 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 certain things and quite horrible at others. This means, in most cases, you're going to find that Solr complements relational and NoSQL databases 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 for. 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 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, not as strict rules. Before we get into specifics, we should remind you to 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's almost too 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 would 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, 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.
- Spelling correction is needed in case the user misspells some of the query terms.
- Autosuggestions save keystrokes, particularly for mobile applications.
- Synonyms of query terms must be recognized.

- Documents containing linguistic variations of query terms must be matched.
- Phrase handling is needed; that is, does the user want documents matching all words or any of the words in a phrase.
- Queries with common words like "a," "an," "of," and "the" must be handled properly.
- The user must have 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 basic 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: 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 an SQL query to a relational database, a row either matches a query or it doesn't, and results are sorted based on one or more of the columns. 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 the strength of the 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:

- 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 with no clear way to navigate them.
- Users are more comfortable with 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 in which users on the go will enter short queries with potential misspellings and expect it to simply work.

To influence ranking, you can assign more weight to, or boost, certain documents, fields, or specific terms. You can boost results by their age to help push newer documents toward the top of search results. You'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 results. For many users, though, this is only the first step in a more interactive session in which 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 documents from an initial query, as well as 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. You can, for example, categorize search results using document features to allow users to narrow down their results. This is known as *faceted search*, and it's one of the main strengths of Solr. You'll see an example of a faceted search for real estate in section 1.2. Facets are covered in depth in chapter 8.

DON'T USE A SEARCH ENGINE TO ...

Let's consider a few use cases in which 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 from which it takes a long time to reconstruct many documents when generating results.

Another use case in which you shouldn't use a search engine is deep analytic tasks that require access to a large subset of the index (unless you have a lot of memory). 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 as is possible with SQL. In chapter 3, you'll learn techniques to adapt relational data to work with Solr's flat document structure.

Also, 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 for which a search engine is the right (or wrong) solution, it's time to dig into what Solr does and how it does it on a high level. In the next section, you'll learn what capabilities Solr provides and how it approaches important software-design principles such as 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*.

- Solr isn't a web search engine like Google or Bing.
- 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 for sale using a web browser. Figure 1.1 depicts a screenshot from this fictitious web application. Don't focus too much on the layout or design of the UI; it's only a mock-up to give visual context. What's important is the type of experience that Solr can support.

Let's tour the screenshot in figure 1.1 to illustrate some of Solr's key features. Starting at the top-left corner, working clockwise, 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 (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 geospatial 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 our fictitious neighborhood. With Solr's geospatial support, you can sort documents by geo distance, limit documents to those within a particular geo distance, or even return the geo distance per document from any location. It's also important that geospatial searches are fast and efficient, to support a UI that allows users to zoom in and out and move around on a map.

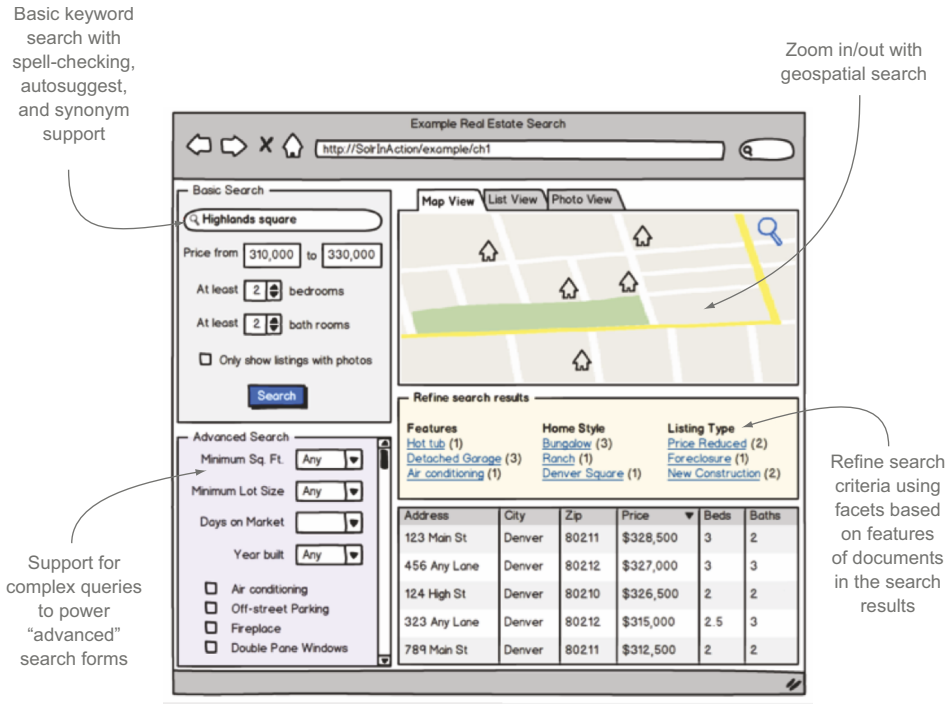

Figure 1.1 Mock-up screenshot of a fictitious search application to depict Solr features

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 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 can 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).[1]

In our example real estate application, the user's primary 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 United States, which definitely qualifies as a "large collection." In a nutshell, Solr uses Lucene to provide the core data structures for indexing documents and executing searches to find documents.

Lucene is a Java-based library for building and managing an *inverted index*, 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.

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 an 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 or more 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 queries.

---

[1] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, *Introduction to Information Retrieval* (Cambridge University Press, 2008).

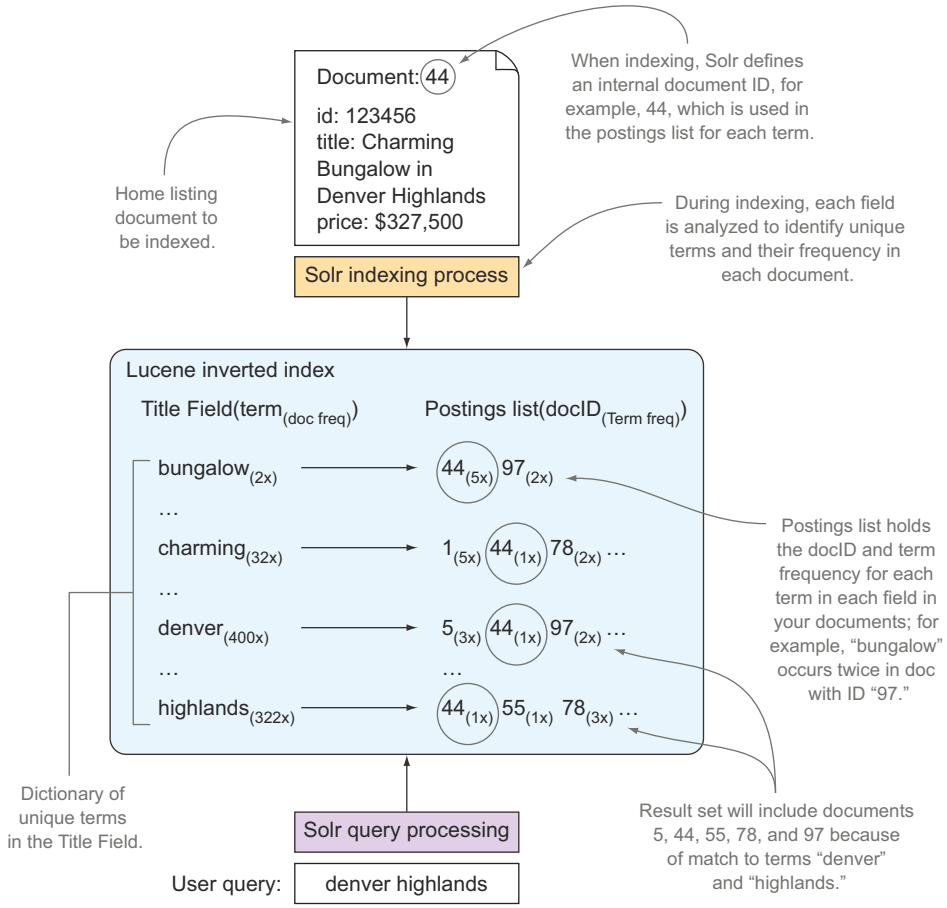

Figure 1.2 The key data structure supporting information retrieval is the inverted index.

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 its 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 a 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.

(continued)

Apache Hadoop provides an open source implementation of MapReduce, and it's used by the Apache Nutch open source project to build a Lucene inverted index for web-scale search using Solr. A thorough discussion of Hadoop and Nutch is beyond the scope of this book, but we encourage you to investigate these projects if you need to build a web-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 your index structure using Solr's flexible *schema.xml* configuration document.

1.2.2 Flexible schema management

Although Lucene provides the library for indexing documents and executing queries, what's missing 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: an XML-configuration document named *schema.xml*. Under the covers, Solr uses *schema.xml* to represent all of the possible fields and data types necessary to map documents into a Lucene index. This saves programming time and makes your index structure easier to understand and communicate to others. A Solr-built index is 100% compatible with a programmatically built Lucene index.

Solr also adds nice constructs on top of the core Lucene indexing functionality. Specifically, Solr provides copy 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 *schema.xml*. This shows how flexible Solr's schema support is; the example Solr server is designed to support product search, but it 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 to 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, such as Jetty or Tomcat, or a full J2EE application server like JBoss or Oracle AS. Figure 1.3 depicts the major software components of a Solr server.

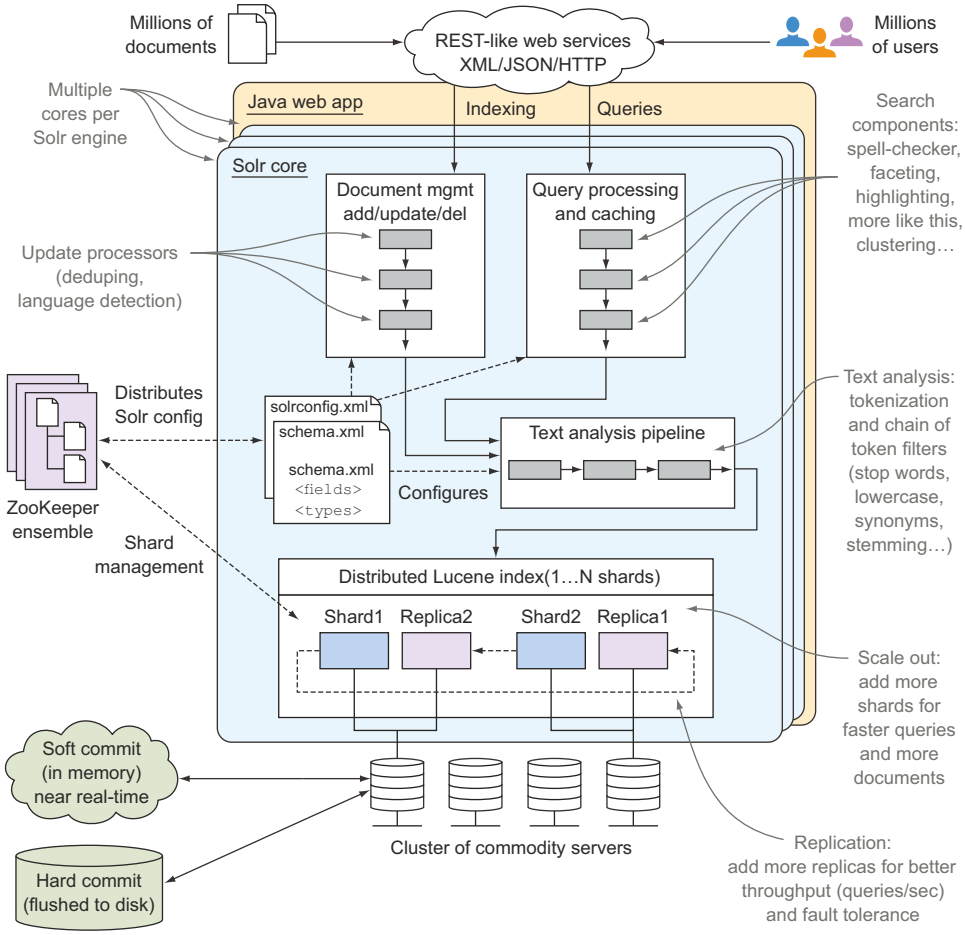

Figure 1.3 Diagram of the main components of Solr 4

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

As we mentioned in the introduction to this chapter, the Solr designers recognized that Solr is best as a complementary technology that works within existing architectures. In fact, you'll be hard put to find an environment in which Solr doesn't drop right in. As you'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 the 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 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 of the popular languages, including Python, PHP, Java, .NET, and Ruby, have a Solr client library.

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 in 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 such as monitoring and maintenance. Solr provides an API for creating and managing multiple cores, which will be covered in chapter 12.

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 (plugins)

Figure 1.3 depicted three main subsystems in Solr: document management, query processing, and text analysis. Of course, these are high-level abstractions for complex subsystems in Solr; we'll learn about each one later in the book. Each system 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 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 step to achieving scalability, Solr provides flexible cache-management features that help your server reuse computationally expensive data structures. 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 gets you only so far, and 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—query throughput and the number of documents indexed. Query throughput 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 that if your index is replicated across three servers, 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 a 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, then you'll likely reach a point at which 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, 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. Add more commodity servers to handle more traffic. Fueling this trend toward 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, you should consider deploying Solr on higher-end hardware with high-performance disks, ideally solid-state drives (SSDs). Hardware considerations for deploying Solr are discussed in chapter 12.

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 the most high-end hardware will experience failures.

Let's assume you have four shards for your index, and the server hosting shard2 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 shard2 fails, Solr reroutes indexing and query traffic to the replica, and your Solr cluster remains online. The result of this failure is that indexing and queries can still be processed, but they may not be as fast because you have one less server to handle requests. We'll discuss failover scenarios in chapters 12 and 13.

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 provide 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 including 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. If you have a nightly build off trunk in which all the automated tests pass, then you can be fairly confident that 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 improved 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 configurations across a cluster of Solr servers and to keep track of cluster state. Here are highlights of the new SolrCloud features:

- Centralized configuration.
- Distributed indexing with no single point of failure (SPoF).

- Automated failover 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 failover and load-balancing support built in.

This isn't to say that Solr scaling doesn't have room for improvement. SolrCloud still requires manual interaction when modifying the size of your search indexes (merging or splitting indexes), and not all Solr features work in a distributed mode. We'll get into all of the specifics of scaling Solr in chapter 12, and the new SolrCloud features in particular in chapter 13, but we want to make sure architects are aware that Solr scaling has come a long way in the past few years and now enables robust scaling with no SPoF.

1.3.2 Solr for the system administrator

As a system administrator, high among your questions about 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 such as JBoss or Apache Tomcat.

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.

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

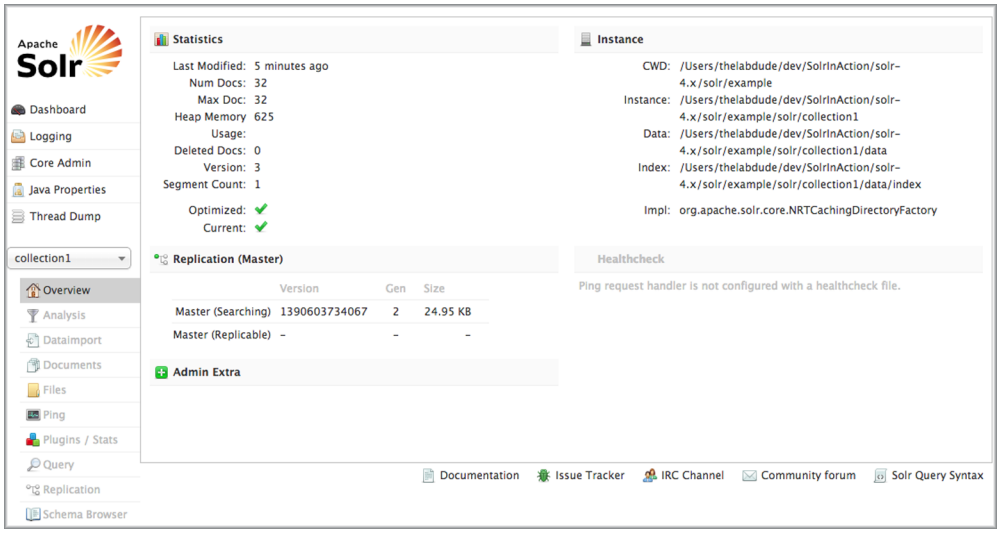

Figure 1.4 A screenshot of the Solr 4 administration console, in which you can send test queries, ping the server, view configuration settings, and see how your shards and replicas are distributed in a cluster.

1.3.3 Solr for the CEO

Although it's unlikely that many CEOs will be reading this book, here are some key talking points about Solr in case your CEO stops you in the hall.

- Executives like to know that an investment in a technology today is going to pay off in the long term. You can emphasize that many companies are still running on Solr 1.4, which was released in 2009; this means that Solr has a successful track record and is constantly being improved.
- CEOs 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.
- Solr has a large support community. 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 that you have help when you need it. You also have access to the source code, which means that if something is broken and needs fixing, you can do it yourself. Many commercial service providers can also help you plan, implement, and maintain your Solr installation, and many offer training courses for Solr.
- Solr doesn't require much initial investment to get started. (This one may be an argument of more interest to the CFO.) 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 subsecond response times.

1.4 Features overview

Finally, let's do a quick rundown 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. You should note, however, 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

- Autosuggest
- Spell-checking
- Hit highlighting
- Geospatial search

PAGINATION AND SORTING

Rather than returning all matching documents, Solr is optimized to serve paginated requests, in which 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, 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; and (2) it helps you track how many queries result in requests for more pages, which may be an indication of a relevance-scoring problem. 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 subgroups 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. Faceting is one of the more popular and powerful features in Solr; we cover it in depth in chapter 8.

AUTOSUGGEST

Most users will expect your search application to "do the right thing," even if they provide incomplete information. Autosuggest allows users to see a list of suggested terms and phrases based on documents in your index. Solr's autosuggest features allow a user to start typing a few characters and receive a list of suggested queries with each keystroke. This reduces the number of incorrect queries, particularly because many users may be searching from a mobile device with a small keyboard.

Autosuggest gives users examples of terms and phrases available in the index. Referring to our real estate example, as a user types `hig`... Solr's autosuggestion feature can return suggestions like "highlands neighborhood" or "highlands ranch." We cover autosuggest in chapter 10.

SPELL-CHECKER

In the age of mobile devices and people on the go, spelling correction support is essential. Again, users expect the search engine to handle misspellings gracefully. Solr's spell-checker supports two basic modes:

- *Autocorrect*—Solr can make the spell correction automatically, based on whether the misspelled term exists in the index.
- *Did you mean*—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 your user typed in `hilands`.

Spelling 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, hit highlighting helps users find relevant documents by highlighting sections of search results that match the user's query. Sections are generated dynamically based on their similarity to the query. We cover hit highlighting in chapter 9.

Geospatial search

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 in which users, using geospatial search, can zoom in/out and move the map's center point to find nearby listings.

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 geospatial search features in chapter 15.

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:

- Result grouping/field collapsing
- Flexible query support
- Joins
- Document clustering
- Importing rich document formats such as PDF and Word
- Importing data from relational databases
- Multilingual support

Result grouping/field collapsing

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 the group. Result grouping, also referred to 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, in which emails matching a specific query can be grouped under the original email message

that started the conversation. You'll learn about result grouping/field collapsing in chapter 11.

FLEXIBLE QUERY SUPPORT

Solr provides a number of powerful query features, including

- Conditional logic using `AND`, `OR`, and `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

We'll cover these terms 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. In Solr, joins are more like SQL subqueries, in that you don't build documents by joining data from other documents. With Solr joins, you can return child documents of parents that match your search criteria. One example in which Solr joins are useful is returning all retweets of a Twitter message into a single response. We discuss joins in chapter 15.

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, 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 16.

IMPORTING RICH DOCUMENT FORMATS SUCH AS PDF AND WORD

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

IMPORTING DATA FROM RELATIONAL DATABASES

If the data you want to search with Solr is in a relational database, you can configure Solr to create documents using an 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 and multilingual text analysis in action in chapter 14.

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, Solr 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. If the price of a home in our example real estate application from section 1.2 changes, 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. 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 5.

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 such as 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 can impact query performance.

WRITE DURABILITY USING A TRANSACTION LOG

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 not 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 USING ZOOKEEPER

If you're new to Solr, you may not be aware that scaling previous versions of Solr was a manual and often cumbersome process. With SolrCloud, scaling is simple and automated because Solr uses Apache ZooKeeper to distribute configurations and manage shard leaders and replicas. The Apache website (http://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 keeping 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 13.

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 instead 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 SPoF when used in a distributed SolrCloud configuration.

We also touched on reasons 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 to where to go to learn more about each feature in this book.

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

SEARCH

Solr IN ACTION

Grainger • Potter

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

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

What's Inside

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

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

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

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

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

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

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

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

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

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]