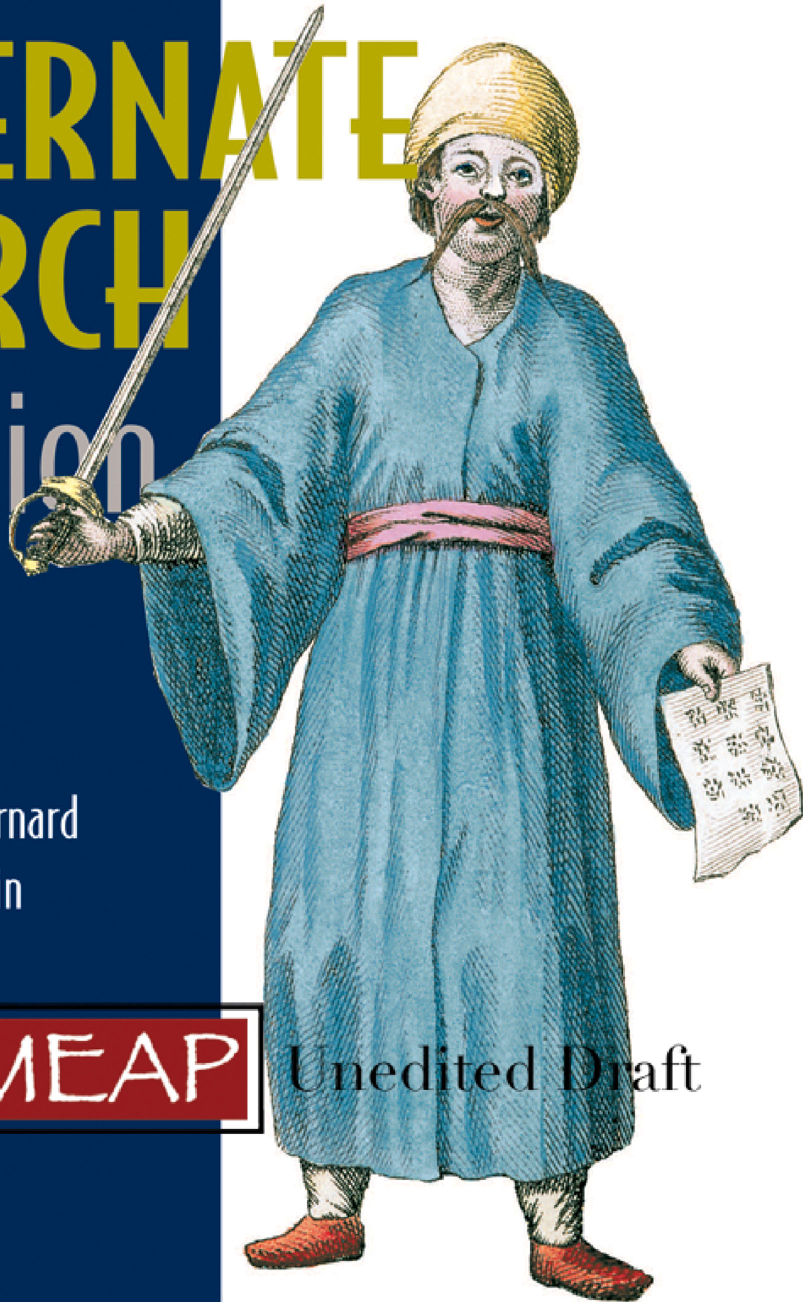
HIBERNATE SEARCH in Action



Emmanuel Bernard
John Griffin

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=409

MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

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

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=409

Hibernate Search in Action

Part I: Understanding Search technology

1. State of the art

2. Getting started with Hibernate Search

Part II: Paving the road to end the structural and synchronization mismatches

3. Mapping simple data structures

4. Mapping more advanced data structures

5. Indexing the world

Part III: Querying the world: taming the retrieval mismatch

6. Full text query in an object world

7. Writing a Lucene query

8. Filters: dynamic data filtering

Part IV: Advanced concepts

9. Performance considerations

10. Clustered environment

11. Seam integration

Part V: Back to the Lucene roots

12. Accessing Lucene natively

13. Document ranking

14. Save yourself time and work (get involved also)

Part I

Understanding Search technology

In part A, you will discover the place of search in modern applications and the different solutions at your disposal and their respective strengths. Chapter 1 will cover the reason behind the need for search, introduce the concepts behind the words full text search, then describe the different type of full text search solutions available to a user. Finally, and going closer to the Java developer's mind, explain some of the problems arising when integrating object oriented domain model and full text search. Equipped with this background, Chapter 02 will guide you through your first steps with Hibernate Search.

After reading this part of the book, you will be able to understand the concepts behind full text search and some benefits of this technology, integration issues between full text search and an object oriented world and will know how to set up and start using Hibernate Search in your Java application.

1
State of the art

This chapter covers:

- Need for search in modern applications
- Full text search concepts
- Full text search solutions
- Mismatches when integrating full text search and domain model

Search is a quite vague notion involving machine processes, human processes, human thoughts and even human feelings. As vague as it is, search is also a mandatory functionality in today's applications, especially since we are exposed and have access to much more information than we used to. Since the exposure rate does not seem to slow down these days, searching, or should we say finding, efficiently becomes a discriminatory element amongst applications, systems, and even humans. No wonder why your customers or your users are all about search.

Unfortunately, integrating efficient search solutions in our daily applications is not an easy task. In Java applications, where the model of your business domain is described by an object model, it can be particularly tricky to provide "natural" search capabilities without spending a lot of time on complex plumber code. Without breaking the suspense of this chapter, Hibernate Search is a for building advanced search functionalities in Java based applications (functionalities that will not shy against the big contenders in this field like Google or Yahoo!). But even more importantly, it relieves the application developer from the burdens of infrastructure and glue code and let him focuses on what matters in the end, optimizing the search queries to return the best possible information.

Before jumping into Hibernate Search, we want you to understand where it comes from and why this project was needed. This chapter will help you understand what search means today when speaking about interacting with an information system (whether it be a website, a backend application, or even a desktop). We will explore how various technologies address the search problem. From that background, you will be able to understand where Hibernate Search comes from and what solutions it provides. Take a comfortable position, relax and enjoy the show.

1.1. What is search?

Search: transitive verb. To look into or over carefully or thoroughly in an effort to find or discover something.

Whenever users interact with an information system, they need to access some information. Modern information systems tend to give users access to more and more data. Knowing precisely *where* to find *what* you are looking for is the edge case of search and there is practically no need for a search function in this situation. But most of the time, where and what are more blurry. Of course, before knowing where to look, you need to have a decent understanding of what you are looking for.

Surprisingly, some users barely know what they are looking for; they have vague (sometimes unorganized) ideas or partial information and seek for help and guidance based on those vague ideas: they seek for ways to

refine their search until they can browse a reasonably small subset of information. Too many and the gem is lost in the flow of data; too few and the gem might have been filtered out.

Depending on the typical system usage, the search feature (or let's call it the reach feature) will have to deal with requests where the what is more or less clear in the user's mind. The clearer it is, the more important it is for the results to be returned by relevance.

WHAT IS RELEVANCE

Relevance is this barbarian word that simply means returning the information considered the most useful first in a result list. While the definition is simple, getting a program to compute relevance is not a trivial task, mainly because the notion of usefulness is hard to understand by a machine. Even worse, while most human will understand what usefulness means, most will disagree on the practical details: take two persons in the street and the notion of usefulness will differ sightly. Let's take an example: I am a customer of a wonderful online retail store and am looking for a "good reflex camera". As a customer, I am looking for a "good reflex camera" as cheap as possible, but the vendor might want to provide me a "good reflex camera" with the highest retail margin. Worse case scenario, the information system has no notion of relevance and the end user will have to do order the data manually.

Even when the users know precisely what they are looking for, they might not precisely know where to look and how to access the information. Based on the *what*, they expect the information system to provide access to the exact data as efficiently, as fast as possible with as few irrelevant pieces of information as possible. (This irrelevant information is sometimes called noise.)

Refining the what can be addressed in several ways, by:

- Categorizing information
- Using a detailed search screen
- Using a simple text box and hide the search complexity to the user

1.1.1. Categorizing information

One strategy is to categorize the information up front. You can see a good example of this approach in Figure 1.1.

The online retail website, Amazon, provides a list of departments and sub-departments that the visitor can go through to direct her search.

The categorization is generally done by business experts during data insertion. The role of the business expert is to anticipate searches and define a efficient category tree that will match the most common ones. However, there are several drawbacks when using this strategy:

- Predefined category might not match the search criteria or might not match the mindset of the user base. (I can navigate pretty efficiently through the mountain of papers on my desk and the floor because I made it, but I bet you will have a hard time seeing any kind of categorization.)
- Manual categorization takes time and comes close to impossible when there is simply too much data.

However, it is very beneficial if the user has no predefined idea as it helps him to refine what he is looking for.

Usually, the categorization is reflected as a navigation system in the application. If we want to make an analogy with this book, categories are the table of content. You can see a category search in action Figure 1.1).

Figure 1.1. Searching by category at Amazon.com: navigating across the departments and sub departments helps the user to structure his needs and refine his search

Unfortunately, this solution is not appropriate for all searches and all users. An alternative typical strategy is to provide a detailed search screen with various criteria representing fields of the data, for example, find by word and find by range.

1.1.2. Using a detailed search screen

A detailed search screen is very useful when the user knows what to look for. This is especially appreciated by expert users. They can fine tune their query to the information system. Such a solution, however, is not friendly to beginner or average users, especially users on the Internet. Users who know what they are looking for and know pretty well how data is organized will make the most out of this search mode (see for example the Amazon.com book search screen in figure 1.2).

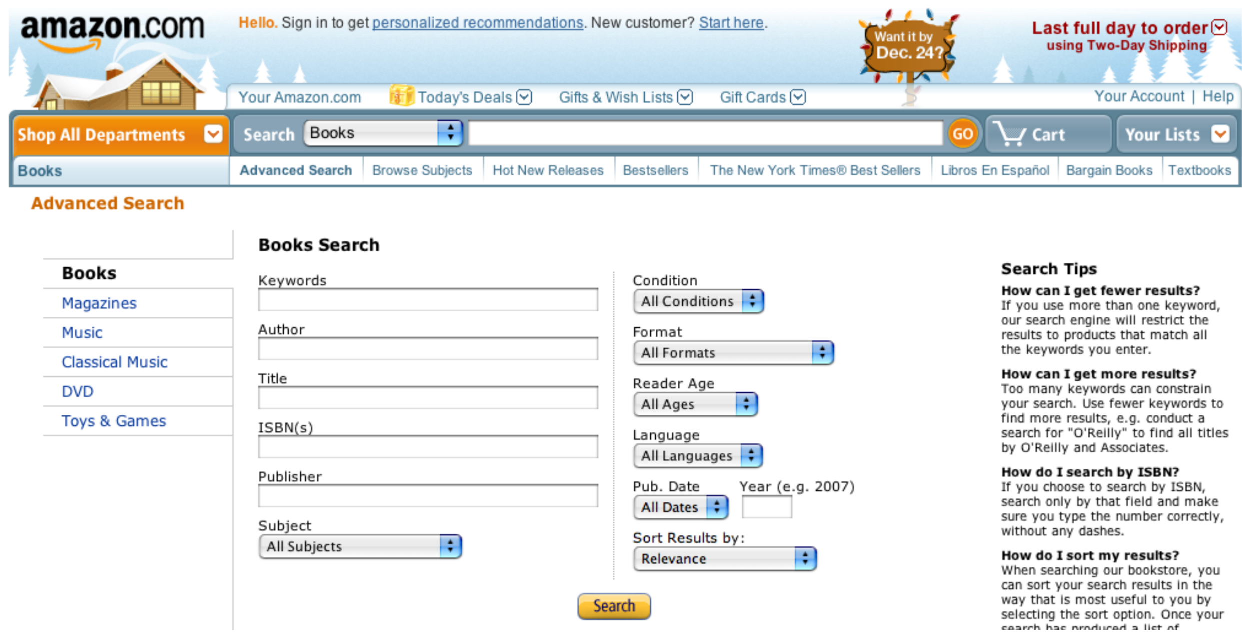


Figure 1.2. Using a detailed search screen: exposes advanced and fine-grained functionalities to the user interface. This strategy

does not fit beginners very well.

For beginners, a very simple search interface is key. Unfortunately it does add a lot of complexity under the hood as your simple user interface has to "guess" the user wishes. A third typical strategy is to provide a simple text box hiding the complexity of the data (and data model) and keeping the user free to express the search query in its own terms.

1.1.3. Using a user friendly simple text box

A simple text box, when properly implemented, provides better user experience for beginners and average users regardless of the qualification of their search (i.e. whether the what is vaguely or precisely defined). This solution however puts a lot more pressure on the information system: instead of having the user using the language of the system, the system has to understand the language of the user. Proceeding on our book analogy, such a solution is the 21st century version of a book index. See the search box at Amazon.com in Figure 1.3.

Figure 1.3. Using one search box: gives the freedom of expression to the user but introduces more complexity and work to the underlying search engine

While very fashionable these days, this simple approach has its limits and weaknesses. The proper approach is usually to use a mix of the previous strategies just like Amazon.com does.

1.1.4. Mixing search strategies

Those strategies are not mutually exclusive; au contraire, most information systems with a significant search feature implement these three strategies or a mix/variation of them.

While not always consciously thought by their designers, a search feature addresses the *where* problem. A user trying to access a piece of information through an information system will try to find the fastest / easiest possible way.

Application designers might have provided direct access to the data through a given path that does not fit the day to day needs of their users. Often, data is exposed the way it is stored in the system and the access path provided to the user is the easiest access path from an information system point of view. This might not fit the business efficiently.

Users will then work around the limitation by using the search engine to access information quickly.

Here is one example of such hidden usage. In the book industry, the common identifier is the ISBN (International Standard Book Number). Everybody uses this number when they want to share some data on a given book. Emmanuel saw a backend application specifically designed for book industry experts, where the common way to interact on a book was to share a proprietary identifier (namely, the database primary key value in the company's datastore).

The whole company interaction process was designed around this primary key. What the designers forgot was that book experts employed by this company have to interact outside the company boundaries very often. It turned out that instead of sharing the internal identifiers, the experts kept using the ISBN as the unique identifier.

To convert the ISBN into the internal identifier, the search engine was used extensively as a palliative. It would have been better to expose the ISBN in the process and hide the internal identifier for machine consumption: and this is what the humans have ended up doing.

1.1.5. Choosing a strategy: the first step of a long road

Choosing one or several strategies is only half the work though, and implementing them efficiently can become fairly challenging depending on the underlying technology used. In most Java applications, both simple text box search and detailed screen search are implemented using the request technology provided by the data store. The data store being usually a Relational Database Management System, a SQL query is built from the query elements provided by the user (after a more or less sophisticated filtering and adjustment algorithm). Unfortunately, the data source query technology often does not match properly the needs for a user centric search tool. This is particularly the case of relational databases.

1.2. Pitfalls of search engines in relational databases

SQL (Structured Query Language) is a fantastic tool for retrieving information. It especially shines when it comes to restricting columns to particular values or range of values and expresses data aggregation. But is it the right tool to find the information based on user inputs?

To answer this question, let's take a look at an example and see the kind of input a user can provide and how an SQL-based search engine would deal with it. A user is looking for a book at a favorite online store. The online store uses a relational database to store the books catalog. The search engine is entirely based on the SQL technology.

The search box on the upper right is ready to receive the user's request.

```
"a book about persisting objects with ybernate in Java"
```

A relational database groups information into tables, each table having one or several columns. A simple version of the website could be represented by the following model:

- A Book table containing a title and a description
- An Author table containing a first name and a last name
- A relation between books and their authors

Thanks to this example, we will be able to uncover typical problems that arise on the way to build an SQL based search engine. While this list is by no mean complete, we will face:

- Writing complex queries as the information is spread across a lot of tables
- Convert the search query into a search of individual words
- Keeping the search engine efficient by eliminating meaningless words (either too common or not relevant)
- Finding efficient ways to search for a given word as opposed to a column value
- Returning results matching words from the same family
- Returning results matching synonymous words
- Recovering from user typos and other approximations
- Returning the most useful information first

1.2.1. Query information spread across

Where to look for the search information our user has requested? Realistically, title, description, first name and last name potentially contain the information the user could base his search on. The first problem comes to light. The SQL based search engine needs to look for several columns and tables, potentially joining them and leading to somewhat complex queries. The more columns the search engine is targeting, the more complex the SQL query or queries will be.

```
select book.id
from Book book left join book.authors author
where book.title = ? OR book.description = ? OR author.firstname = ? OR author.lastname = ?
```

This is often one area where search engines limit the user in order to keep the query relatively simple (to generate) and efficient (to execute). Note that, this query does not make any difference if the information we are looking for is found in one, two, or more columns but it seems this information could be important (more on this later).

1.2.2. Searching words not columns

Our search engine now looks for the user provided sentence across different columns. It is very unlikely that either one of the columns contains the complete following phrase, "a book about persisting objects with ybernate in Java". Searching each individual word sounds like a better strategy. This leads to the second problem: a phrase needs to be split into several words. While this could sound like a trivial job, do you actually know how to split a Chinese sentence into words? After a small Java preprocessing, the SQL based search engine now has access to a list of words that can be searched for: "a, about, book, ybernate, in, Java, persisting, objects, with".

1.2.3. Filter the noise

Not all words seem equal though. "Book", "ybernate", "Java", "persisting", "objects" seem relevant to the search whereas "a", "about", "in", "with" are more noise and return results completely unrelated to the spirit of the search.

The notion of noisy word is fairly relative. First of all, it depends on the language, but it also depends on the subject(s) being searched. For an online book store, "book" might be considered a noisy word. As a rule of thumb, a word can be considered noisy if it is very common in the data and hence not discriminatory (a, the, or etc) or if it is not meaningful for the search ("book" in a bookstore). You have discovered yet another bump into the holy quest of SQL based search engines: a word filtering solution needs to be in place to make the question more selective.

1.2.4. Find by words... fast

Restricted to the list of meaningful query words, the SQL search engine can look for each word in each column. Searching a word inside the value of a column is not a primitive operation In SQL, the SQL `like` operator is used in conjunction with the wild card character '%' (e.g. `select ... from ... where title like '%persisting%'...`). And unfortunately for our search engine, this operation can be fairly expensive and you will understand why in a minute.

To verify if a table row matches `title like '%persisting%'`, a database has 2 main solutions:

- Walk through each row and do the comparison: this is called a table scan and can be a fairly expensive operation especially when the table is big.
- Use an index.

An index is a data structure that makes efficient the search by the value of a column usually by ordering the index data by column value (see Figure 1.4).

Book table

id	title	description
1	The Da Vinci Code	Detective novel basing the plot ...
2	Alice's adventures in Wonderland	Classic of the nonsense literature ...
...	...	...
234	Economics for Dummies	Introductory book demystifying ...
235	The real history behind the Da Vinci Code	Uncover reality behind ficiton ...

Index on title

title	id
Alice's adventures in Wonderland	2
Economics for Dummies	234
The Da Vinci Code	1
The real history behind the Da Vinci Code	235

Figure 1.4. Show the typical index structure in a database. Row ids an be quickly found by title column value thanks to the structure.

To return the results of the query `select * from Book book where book.title = 'Alice's adventures in Wonderland'`, the database can benefit from the index, to know which rows match. This operation is fairly efficient because the title column values are ordered alphabetically: the database will look into

the index in a roughly similar way you would look into a dictionary to find the words starting with A, then l, then i. This operation is called an index seek: the index structure is used to find the matching information very quickly.

Note that the query `select * from Book book where book.title like 'Alice%'` can use the same technique because the index structure is very efficient in finding values that start with a given string. Now let's look at the original search engine's query, `where title like '%persisting%'`. The database cannot reuse the dictionary trick here because the column value might not start with `persisting`. Sometimes, the database will use the index by reading every single entry and see which one has the word `persisting` somewhere in the key: this operation is called an index scan. While faster because the index is more compact, this operation is very similar to the table scan and thus often slow. Because the search needs to find a word inside a column value, our search engine query is reduced to using either the table scan or the index scan technique and suffering from their poor performances.

1.2.5. Search words from the same family, with the same meaning

After identifying all the previous problems we end up with a slow SQL based search engine, complex to implement and we need to apply some complex analysis to the human query before morphing it into a SQL query.

Unfortunately, we are still far from the end of our journey: the perfect search engine is not there yet. One of the fundamental problem still present is that words provided by the user may not match letter to letter the words in our data. Our search user certainly expects the search engine to return not only book containing `persisting`, but also `persist`, `persistence`, `persisted` and any word whose root is `persist`. The process used to identify a root from a word (called stem) is named the stemming process. Expectations might even go further; why not consider "persist" and all of its synonyms? "Save", "store" are all valid synonyms of "persist". It would be nice if the search engine returned books containing the word "save" when the query is asking for "persist".

These are a new category of problems that would force us to modify our data structure to cope with them: a possible implementation could involve an additional data structure to store the stem and synonyms for each word but would involve a significant additional amount of work.

1.2.6. Recovering typos

One last case about words: "ybernate". You are probably thinking that the publication process was pretty bad at Manning to let such an obvious typo go through. Don't blame them, I asked for it. Because your search user will make typos. He will have heard of a new technology overhearing a conversation at Starbucks but have no clue on how to write it. Or he might simply have made a typo. The search engine needs a way to recover from ibernate, ybernate, hypernate. There are several techniques using approximation to recover from such mistakes. A very interesting one is to use a phonetic approach to match words by their phonetic (approximate) equivalent. Like the last two problems, there is no simple approach to solve this issue with SQL.

1.2.7. Relevance

Let's describe one last problem that has to be addressed and this is probably the most important one. Assuming the search engine manages to retrieve the appropriate matching data, the amount of data might be very large. Users will not scroll through 200 or 2000 results, and if they do, they will probably be very unhappy.

How to ensure data is ordered in a way that returns the most interesting results in the first 20 or 40 results? Ordering by a given property will most likely not have the appropriate effect. The search engine needs a way to sort the results by relevance.

While this is a very complex topic, let's have a look at some simple techniques to get a feel about the notion. For a given type of query, some parts of the data, some fields are more important than others. In our example, finding a matching word in the title column has more value than finding a word in the description column: so the search engine can give priorities to the former. Another strategy would be to consider that the more matching words are found in a given data entry, the more relevant they is for the search. An exact word certainly should be valued more than an approximated word. When several words from the query are found close to each other (maybe in the same sentence), it certainly seems to be a more valuable result. If you are interested in the gory details of relevance, this book dedicates a whole chapter on the subject: chapter 13.

Defining such a magical ordering equation is not easy, but SQL based search engines don't even have access to the raw information needed to fill this equation: word proximity, number of matching words per result and so on. This is it for our eighth problem.

1.2.8. Many problems. Any solution?

The list of problems could go on for a while but hopefully, you have been convinced that an alternative approach for search engines must be used in order to overcome the shortcomings of SQL queries. Don't feel depressed by this mountain of problem descriptions. Finding solutions to address each and every one of them is possible and technology exists today: Full Text Search also named as Free Text Search.

1.3. Full Text search: a promising solution

Full text search is a technology focused on finding documents matching a set of words. Because of its focus, it addresses all the problems we have had during out attempt to build a decent search engine using SQL. While sounding like a mouthful, full text search is more common than you might think. You probably have been using full text search today. Most of the web search engines like Google, Yahoo!, Altavista use full text search engines at the heart of their service. The difference between each of them are recipe secrets (and sometimes not so secret) like the Google PageRank™ algorithm. PageRank™ will modify the importance of a given web page (result) depending on how many web pages are pointing to it and depending on how important each of this page is.

Be careful, though; these so called web search engines are way more than the core of full text search: they have a web user interface, they crawl the web to find new pages and update existing ones, and so on. They provide business specific wrapping around the core of a full text search engine.

Given a set of words (the query), the main goal of Full text search is to provide access to all the documents matching those words. Because sequentially scanning all the documents to find the matching words is very inefficient, a full text search engine (its core) is split into two main operations: indexing the information into an efficient format and searching the relevant information from this pre-computed index. From the definition, you can clearly see that the notion of word is at the heart of full text search: this is the atomic piece of information that will be manipulated by the engine. Let's dive into those two different operations.

1.3.1. Indexing

Indexing is a multiple step operations whose objective is to build a structure that will make data search more efficient. It solves the fourth problem we had with our SQL based search engine: efficiency. Depending on the full text search tools, some of those operations are not considered to be part of the core indexing process and are sometimes not included (see Figure 1.5).

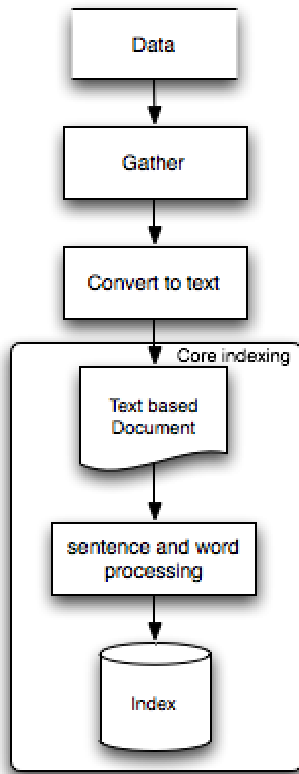


Figure 1.5. Indexing process. Gather data, convert it to a text. From the text-only representation of the data, apply word processing and store the index structure.

Let's have a look at each operation:

- The first operation needed is to gather information, for example by extracting information from a database, crawling the net for new pages, react to an event raised by a system. Once retrieved, each row, each HTML page, or each event will be processed.
- The second operation converts the original data into a searchable text representation: the *document*. A document is the container holding the text representation of the data, the searchable representation of the row, the HTML page, the event data and so on. Not all of the original data will end up in the document; only the pieces useful for search queries. While indexing the title and content of a book make sense, it is probably unnecessary to index the value of the URL pointing to the cover image. Optionally, the process might also want to categorize the data: the title of an HTML page may have more importance later on than the core of the page. They will probably be stored in different *fields*. Think of a document as set of fields. The notion of fields is step one of our journey to solve the first problem we hit in the SQL based search engine.
- The third operation will process the text of each field and extract the atomic piece of information a full text search engine understands: words (which was our second problem back in the SQL days). This operation is critical for the performance of full text search technology but also for the richness of the feature set. In addition to chunking a sentence into words, this operation also prepares the data to handle some of the additional problems we have been facing in the SQL based search engine: search by object root or stem (fifth problem) and search by synonyms (sixth problem). Depending on the full text search tool used, such additional features are available out of the box or not and can be customized, but the core sentence chunking is always there.
- The last operation in the indexing process is to store your document (optionally) and create an optimized structure that will make search queries fast. So what is behind this magic optimized structure? Nothing

much really than the index in the database we have seen in Section 1.2, but the key used in this index is the individual word rather than the value of the field (see Figure 1.6). The index also stores some additional information per word.

This information will help later on to fix the order by relevance problem we faced in our SQL based search engine. Amongst additional pieces of information, word frequency, word position and offset are worth noticing.

They allow the search engine to know how "popular" a word is in a given document, and what his position is compared to another word.

Book table

id	title
1	The Da Vinci Code
2	Alice's adventures in Wonderland
...	...
234	Economics for Dummies
235	The real history behind the Da Vinci Code

Full text index

Word	id	freq.	position
adventures	2	1/3	2
alice	2	1/3	1
code	1	1/3	3
	235	1/6	5
da	1	1/3	1
	235	1/6	4
dummies	234	1/2	2
economics	234	1/2	1
vinci	1	1/3	2
	235	1/6	5
wonderland	2	1/3	3

Figure 1.6. Full text index structure: each word in the titles is used as a key in the index structure. For a given word (key), the list of matching ids are store as well as the word frequency and position.

While indexing is quite essential for the performance of a search engine, searching is really the visible part of a search engine (and in a sense the only visible features your user will ever care about). While every engineer knows that the mechanics is really what makes a good car eventually, no user will fall in love with it unless is has nice, curvy lines and is easy to drive. Indexing is the mechanics of our search engine and searching is the user oriented polish that will hook our customers.

1.3.2. Searching

If we were using SQL as our search engine, we would have to write a lot of the searching logic by hand. Not only it would be reinventing the wheel but very likely our wheel would look more like a square that a circle. Searching takes a query from a user and returns the list of matching results efficiently and ordered by relevance. Like indexing, searching is a multi-step process that we picture in Figure 1.7. We will walk through them and see how they solve the problems we have seen during the development of our SQL based search engine.

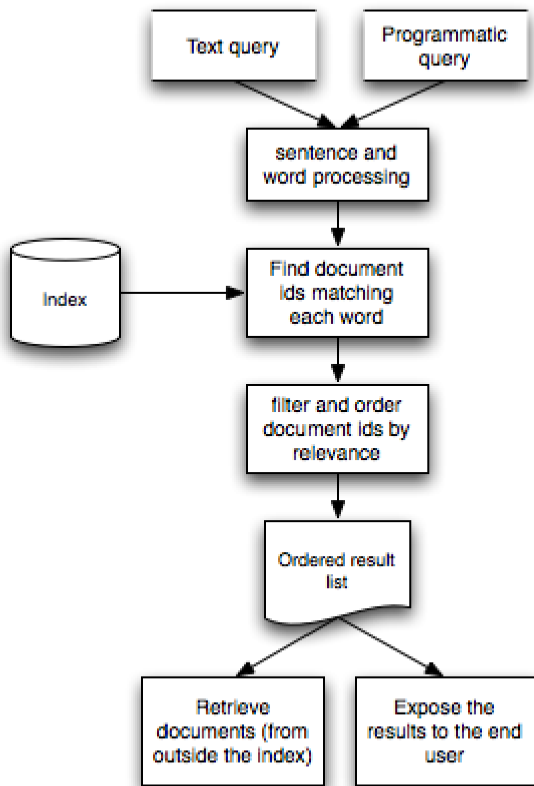


Figure 1.7. Searching process. From a user or program request, determine the list of words, find the appropriate documents matching those, eliminate the documents not matching an d order the results by relevance.

The first operation is about building the query, get the query from the client. Depending on the full text search tool, the way to express query is:

- String based: a text-based query language. Depending on the focus, such a language can be as simple as handling words and as complex as having boolean operators, approximation operators, field restriction and many more!
- Programmatic API based: for advanced and tightly controlled query, a programmatic API is very neat. It gives the developer a very flexible way to express complex queries and decide how to expose the query flexibility to their users (it might be a service exposed through a REST interface)

Some tools will focus on the string query, some on the programmatic API, some will provide both. Because the query language or API are focused on full text search, they end up being much more simple to write (in complexity) than their SQL equivalent and helps to reduce one of the pains we had with our SQL-based search engine: complexity.

The second operation, let's call it analyzing, is responsible for taking the sentences / list of words and applying the similar operation performed at indexing time on sentences (chunk into words, stems, phonetic description). This is critical because the result of this operation is the common language that indexing and searching will use to talk to each other and happens to be the one stored in the index. If the same set of operations is not applied, the search will not find the indexed words: not so useful! This common language is the cornerstone of full text search performances (the forth problem that hit us in our previous search engine).

Based on the common language between index and search, the third operation (finding documents) will read the index and retrieve the index information associated to each matching word (see Figure 1.6). Remember, for each word, the index stores the list of matching documents, the frequency, the word positions in a document and

so on. The implicit deal here is that the document itself is not loaded and that's one of the reason why full text search is efficient, the document does not have to be loaded to know whether it matches or not.

The next operation (filtering and ordering) will process the information retrieved from the index and build the list of documents (or more precisely handlers of documents). From the information available (matching documents per word, word frequency, word position), the search engine is able to exclude some documents from the matching list.

More importantly, it is able to compute a score for each document. The higher its score is, the higher a document will be in the result list. A lengthy discussion about scoring is available in chapter 13, but in the mean time let's have a look at some factors influencing its value:

- In a query involving multiple words, the closer they are in a document, the higher the rank is
- In a query involving multiple words, the more they are found in a single document, the higher the rank is
- The higher the frequency of a matching word is in a document, the higher the rank is
- The less approximate a word is, the higher the rank is

Depending on how the query is expressed and how the product computes score, these rules may not apply. This list is here to give you a "feeling" of what may affect the score and hence the relevance of a document. And that's it, this last part has solved the last problem faced by our SQL based search engine: ordering results by relevance.

Once the ordered list of documents is ready, the full text search engine exposes the results to the user. It can be through a programmatic API, or through a web page. Figure 1.8 shows a result page from the Google search engine.

Web Personalized Results **1** - **10** of about **516,000** for **hibernate search**. (**0.06** seconds)

hibernate.org - **Hibernate Search**
Hibernate Search brings the power of full text **search** engines to the persistence domain model ... **Hibernate Search** is using Apache Lucene(tm) internally, ...
search.**hibernate**.org/ - 15k - Cached - Similar pages

Hibernate Search
Hibernate Search indexes your domain model with the help of a few annotations, takes care of database/index synchronization and brings back regular managed ...
www.**hibernate**.org/hib_docs/ **search**/reference/en/html_single/ - 139k -
Cached - Similar pages

hibernate.org - **Hibernate**
The **Hibernate Search** team is pleased to announce version 3.0 final ... **Hibernate Search** 3.0.0.CR1 is mainly the last bits of new features and polishing ...
www.**hibernate**.org/ - Similar pages
More results from www.hibernate.org »

Hibernate Search 3.0 available: provides full-text **search**
Hibernate Search 3.0, which brings full text **search** capabilities to **Hibernate**-based applications, has been released. With **Hibernate Search**, developers can ...
www.theserverside.com/news/thread.tss?thread_id=46995 - 160k - Cached - Similar pages

Full Text **Search** with **Hibernate Search** « Cagatay Civici's Weblog
Doing Full Text **Search** in your domain model might be tricky in the past but now with **Hibernate Search** based on Lucene it's not a problem. ...
cagataycivici.wordpress.com/ 2007/03/06/full_text_**search**_with_**hibernate**/ - 44k -
Cached - Similar pages

Figure 1.8. Search results returned as a web page

Sounds like we have found the perfect solution to address our problem, let's have a look at the kind of full text search solutions on the market.

1.3.3. Full text search solutions

A variety of full text search solutions are available. Depending on their focus, they might fit better a different need. Some go beyond the core part of full text search and go all the way up to exposing the results in a web page for you. There are three main families of solutions:

- An integrated full text engine in the relational database engine
- A black box server providing the full text service
- A library providing a full text engine implementation

Let's explore these three classical approaches.

Full Text in Relational Databases

Integrating full text search with the relational engine sounds like a very appealing solution when full text searches aim at targeting… data stored in the database. When the objective is to enhance SQL queries of our application with some full text search capabilities, this solution is a serious contender. Let's go through some of the benefits:

- Less management duplication: since both your data and your index are handled by the same product, administrating the system should be quite simple (note that some full text search relational integration are not that… integrated and require a different backup / restore process)
- Data and index are always synchronized: since a database knows when you update your data, keeping the index up to date is very easy
- Mixing SQL queries and full text queries: the authors think this is really the most appealing benefit ; SQL provides a lot of flexibility when querying data, enhancing it with full text search keywords makes the querying experience very integrated.

Recent versions of the well known databases tend to include a full text search module. Oracle DB, Microsoft SQL Server, and MySQL to name a few embed such a module.

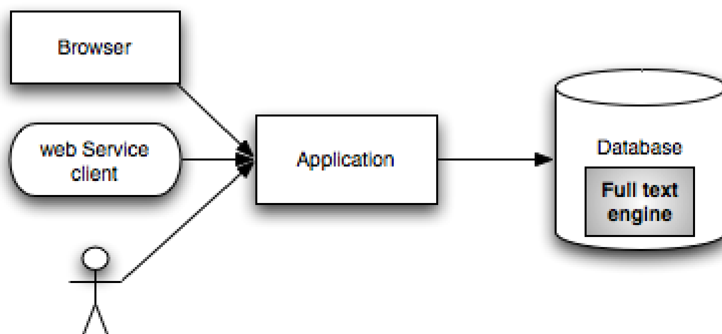


Figure 1.9. Full text embedded in a relational database

This solution unfortunately suffers from some problems:

- The first problem is scalability. In today's application architecture, the database tend to be the most critical path where scalability cannot be as easily achieved as in other tiers. Full text indexing and searching can be quite intensive in CPU, memory and Input/Output. Do we really want to spend database resources on such a feature set as depicted in Figure 1.9? Will it be a problem in the future and how fast will I reach the scalability limit?
- The second problem is portability. Unfortunately, there is no standard today to express a full text query. Relational vendors have extended SQL to support the ability to express those kind of queries, every single one in its own way. The end result for the user is the inability to build an application compatible with

multiple relational back ends. Even if the user is committed to a migration effort, the features themselves are not standard and their behavior might change from one product to another (if they are even present in both).

- The third problem is flexibility. Depending on the relational engine, indexing can be more or less flexible. Generally speaking though, flexibility is not the strong point of such engines. Flexibility is key to adapt your search engine to your business needs and to fulfill your user's requests. Flexibility is needed both at indexing time (how do you want your data to be prepared), and at searching time (what kind of full text operations are available).

Full text search engines embedded in a relational database is best for people specifically targeting search on the data embedded in their database, don't expect the requirements to go too far, are not ready to invest a lot in development time and of course are not concerned about database portability. Scalability is another concern for some implementations.

BLACK BOX

On the other side of the full text search spectrum are fully dedicated products whose focus is mainly on searching heterogeneous content on a website, intranet or the information system in general. As seen in Figure 1.10, they serve as the central indexing and searching service. Thanks to their focus, they tend to have very good performances both at indexing time and to process queries. The best known example today is the Google Search Appliance but the giant is not the only one on this market.

Such a tool is deployed on a dedicated server (included or not) and crawl your website, your intranet and the content of your documents (in a content management system or elsewhere) in the background pretty much like Yahoo! and Google.com do for the web. Those tools are very interesting for the out of the box experience they provide.

Beyond the core indexing and searching capabilities that belongs to full text search, these products usually provide some or all of those functionalities:

- Crawling for websites, CMS, wikis, databases
- Indexing a variety of document formats like presentations, spreadsheets and text documents
- Providing a unified web page to search this content and render the results

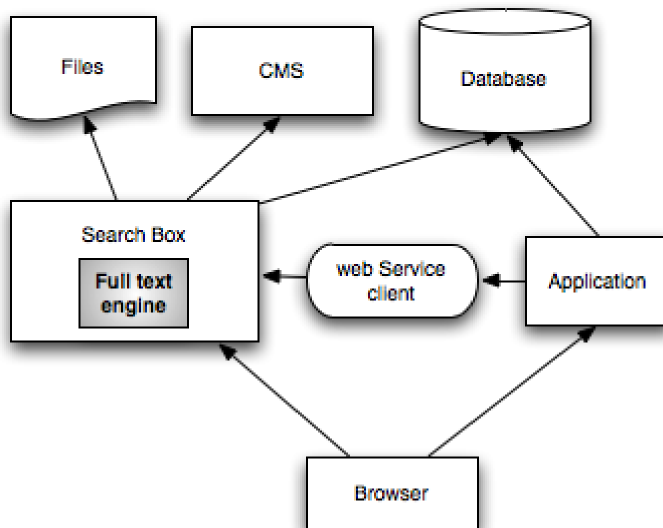


Figure 1.10 Server dedicated to full text search

Black box solutions really shine when you aim at finding data across the enterprise and when they are stored in a variety of areas. Maintenance of these solutions are usually quite low due to their out of the box focus, most

products come with an administration interface. They are not primarily focused on providing the business oriented search capability of a specific application and might lack flexibility and refinement to do so. For commercial products, pricing may vary but a price per document indexed or per index size is quite common.

NOTE

Apache Solr™, a product based on Apache Lucene™, is an hybrid product in between the Black box and the library category. Solr is a Black box product which takes care of a lot of the administration part but expose the indexing and searching through an XML based API.

LIBRARIES

A full text search library is an embeddable software that you can use in your application. This solution is by far the most flexible amongst the three when the search feature aims at being integrated into an application. Your application will call the library APIs when indexing a document or when searching a list of matching results. The query expressiveness and flexibility is the strong point of such a solution: the application developer can decide which full text feature will be exposed, which data specifically will be searchable (and potentially manipulate this data before indexing) and is free to decide how a user will express his query (which user interface, which natural language and so on). Flexibility when indexing is also quite strong since the application developer decides when and what to index and has control over the structure.

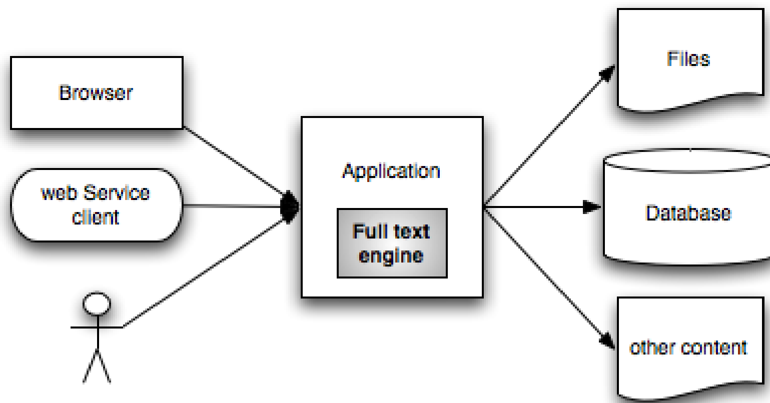


Figure 1.11. Library providing full text search

Depending on the richness and the popularity of the library you choose, some will be able to read complex formats like PDF, Word documents (either as part of the core product or as third party plugins).

Unfortunately, flexibility comes at a price. First of all, you need to be able to integrate the full text library to your application as pictured in Figure . While it's very easy for a new application, it might be more difficult for an application that has reached its end of life (or close to) or to an application you simply don't have the code source of...

While developers agreed to spend some time to work on the search part of the integration, they are often reluctant to spend too much time on the indexing part: said differently, developers would rather avoid the burdens of infrastructure code. But as we have seen previously, both operations (indexing and searching) are integral part of a full text search engine.

Amongst the libraries available to you, Apache Lucene™ is probably the most popular. Lucene is an open source library from the Apache foundation initially written and still maintained in Java. Due to its popularity, Lucene has subsequently been ported in different languages (C, C++, C#, Objective-C, Perl, PHP, Python and a few more). Lucene is also noticeable because this library is at the core of Hibernate Search, this is not the last time you will read about Lucene in this book.

Now that we know the three main full text solutions, the hard part is in front of us. Which one should be chosen?

WHICH ONE TO CHOOSE?

The authors will probably disappoint you at this stage but their answer is: it depends. For each solution, we have tried to give you the most common use case and the strong and weak points of each solution. Each product on the market fits more or less in one of those categories but some will sit in between. Instead of giving you a definitive answer, let's have a look at some of the questions you should answer before deciding to go for one product:

- Do I need to search data from one database or from multiple sources?
- Do I want to slightly improve an existing search feature or do I want to fully benefit from a full text search engine?
- How much (in time and money) this solution will cost me?
- Can I modify / influence the scoring (prioritize) algorithm to fit my needs? Do I care?
- Can I express search by phrase, approximation, range, exclusion, weak inclusion, mandatory inclusion and so on?
- Can I index non textual data? What about my own data format?
- What is the maintenance work?
- How fast is indexing? How fast is searching? How about after one, ten or one hundred millions records?
- How much integration and customization do I need for my business rules?
- How well integrated search needs to be with my user interface?

This list of questions is not exhaustive, the authors have both used Lucene extensively in their applications and like the flexibility and performance it provides. They also think that the way to implement the best search engine focused on your business needs is to use the library approach. They also were ready to pay the price of flexibility and dig more into the code. This is the approach described in this book. Hibernate Search and this book are indeed focused on reducing as much as possible the overhead paid for the flexibility gained when using Lucene and Hibernate.

The next section will focus on describing some of the problems and difficulties when integrating Lucene into a domain model centric applications. This will help you understand the reasons behind Hibernate Search.

1.4. Mismatches between the round object world and the flat text world

Full text search seem to be the magic bullet for our search solution when search is driven by a human input. It solves many of the problems we had with a SQL-based solution: performance, complexity to express the query, search by approximation, phonetic search, and ordering by relevance. And if we focus on the library solution, which is the one that seem to provide the most flexibility to achieve our goals, we will be able to extend our application with a custom search engine that will increase our user productivity. But how hard will it be?

To answer this question, we will take a typical Java application and try to integrate Lucene, a typical choice for a full text search engine. Our typical Java application is a web application (could very well be a rich client application) which uses Hibernate to persist a domain model into a relational database.

NOTE

A domain model is the object representation of your data model: hence represents the business domain of an application. This model, fairly close to the relational model, is mapped to the database thanks to an ORM like Hibernate in our application. This object model is at the heart of your application and is used by different modules to interact with the data.

This journey will show us three fundamental problems (called mismatches in this chapter):

- The structural mismatch
- The synchronization mismatch
- The retrieval mismatch

People used to Object Relational Mappers might find the idea of mismatch quite familiar. Not surprising at all since we try to exchange information from an object model to an index model, like ORM do from the object model to the relational model.

1.4.1. The structural mismatch

Lucene represents a record (an entry) as a Document. A Document is an API that can receive as many fields as it pleases you. Each field has a name and a value. This value is a String. A full text search engine like Lucene does not know a lot of types. Actually the only type understood by the engine is String. We need to find a way to convert the rich and strongly type domain model in Java to a string only representation that can be digested and indexed by the full text search engine (see Figure 1.12). While fairly easy for some types, it can be pretty questionable for others.

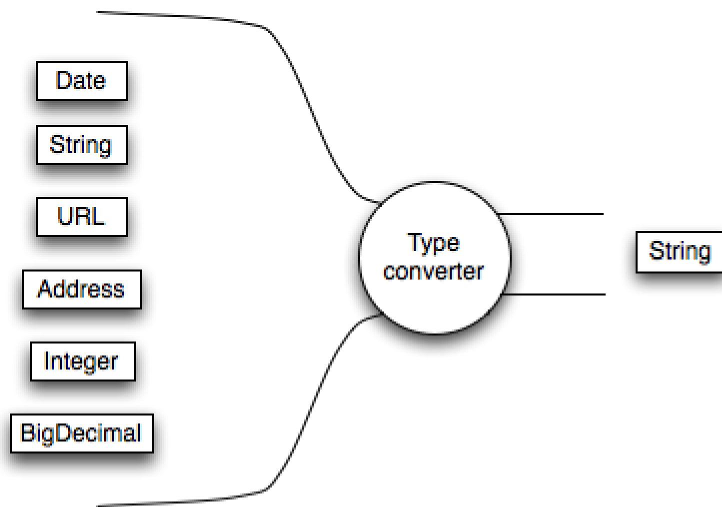


Figure 1.12. All rich types have to be converted into a string-only world

Date is one example amongst them. When doing a search query on a date or date range, we don't always need the date with its full precision up to millisecond. Maybe providing a way to store only the date up to the day or the hour could make sense. Date is not the only type with problems, number is another one. When comparing numbers, it's fairly admitted that 23 is lower than 123. Not for Lucene! Remember, everything is a string and in the string world, 23 is indeed higher than 123 because the lexical order of 2 is higher than 1. Beyond built-in Java types, custom classes like `Address` or `MoneyAmount` also have to be somewhat translated into a string representation into the index.

Let's explore a bit more our domain model. Some entities have relationships with each other. In a relational model, these relations are represented by foreign keys while in an object model, these relation are "pointers" from one object to another. Unfortunately, Lucene does not support the notion of relation between documents. The consequence is quite strong and means that Lucene does not support the notion of JOIN (like in SQL). A workaround has to be found when a query relies on constraint to related entities / documents and they can be quite common: return the list of matching orders where one item has Alice in the title and the customer lives in Atlanta.

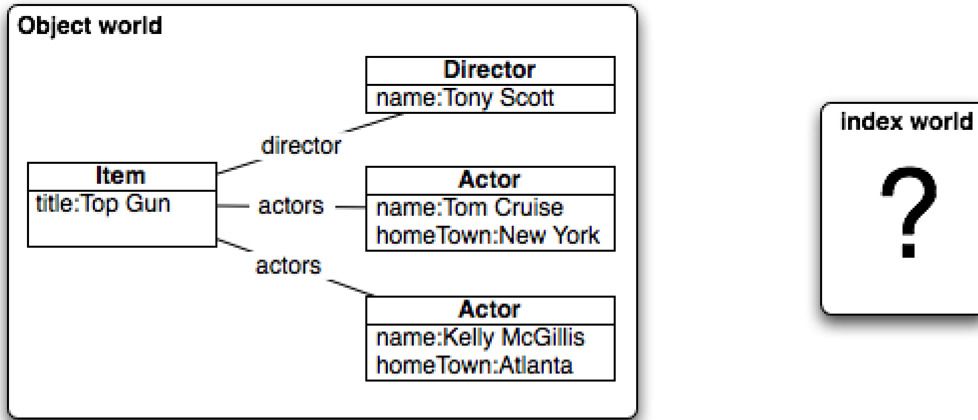


Figure 1.13 The notion of relationship between entities is not available in a full-text index like Lucene. How can query on associated objects be expressed?

To integrate the conversion from the object model to the index model, some boilerplate code needs to be written.

1.4.2. The synchronization mismatch

Converting the data is one part of the problem, the system also has to keep the data synchronized between the database and the index. Remember, Lucene is not part of the relational engine and does not know when the data is updated.

The application knows when a use case updates some data, this database update could be coupled with the Lucene index update as shown in Figure 1.14.

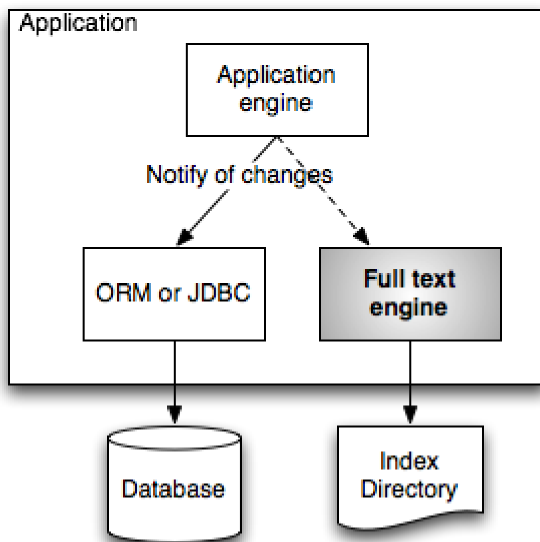


Figure 1.14. Every change made to the database by the application has to be done to the full-text index as well

This solution is a bit fragile though if multiple paths update the data in the system, especially since some path could benefit from the update by cascade feature provided by ORMs: it can be fairly hard to know which object is updated or created in a whole object graph.

Our integration process now has to take care of the synchronization between the two data structures, one way or another.

1.4.3. The retrieval mismatch

Once the data is property converted to the index and kept synchronized with the database, let's think about doing what we planned from day one: finding information. The full text search engine will return a list of matching documents through a Lucene specific API. A document is essentially a Map of field name, field value. The developer has two choices. Accept to use a untyped version of its domain model (the map) and adjust the code to deal with two different models depending on the search method (full text vs SQL). Or do a conversion back to the domain model. In addition to requiring writing the code to convert the index data back into an object model, some problem will arise: the index needs to store the same amount of data as the database potentially wasting Lucene performance, be sure that the text version of the data can be converted back into its original type and deal with the notion of lazy loading.

LAZY LOADING

Lazy loading is a technique heavily used in ORMs to avoid to load too much information in memory. Without lazy loading, loading an object means loading the object and all the associated objects recursively. Provided that the entities have rich relationships, this would mean loading the whole database in memory. To avoid that, an ORM will only load the object or object graph up to a certain level. This level is defined statically and/or dynamically by the developer. If the program reaches one of the non loaded associations, the ORM will load the needed object graph section transparently.

Don't try this at home; you will soon discover that lazy loading is a quite complex problem to solve! There is an additional inconvenience, if you manage to solve the lazy loading problem. The loaded objects are not objects taking care of by Hibernate: if the application changes one of those objects, how to make sure the changes will be propagated both to the database and the index? One last problem with this architecture, you could end up having two instances of an object representing the same database row, one from Hibernate and one from Lucene leading to synchronization hell.

1.5. Summary

We have seen on this chapter the various strategy to provide search functionalities to an application and help the user to access information. While categorization and a detailed search screen address some of the needs, the simple one box search feature popularized by search websites like Google or Yahoo! has become increasingly important.

Traditional relation database query solutions do not address efficiently (and sometime not at all) the search needs when it comes to interaction with a human demand. Full text search technology address those concerns by providing solutions to search by relevance, search by approximation, efficiently search by words in a document and so on.

Full text search opens a lot of doors that where simply not accessible by other technologies, it pushes the limits of the user experience by providing a semi natural way to express queries. Unfortunately, properly integrating a full text search engine like Lucene into a classic Java architecture whether it be SE (Standard Edition) or EE (enterprise Edition) is not an easy task and most people have to accept some inconsistency between their programmatic model (one from the Java Persistence / Hibernate part and one from the Lucene part). The authors believe that these problems are the major reasons for the lack of adoption in a large scale of full text search engines like Lucene by Java applications despite a constant pressure by the customers: the price to pay is high enough for project leaders to think twice before jumping.

Why not getting rid of the database?

A lot of the problems encountered are due to the fact that data has to be maintained both in the full text index and the database. Why not storing all the data in the Lucene index and removing the database from the equation?

> A database is a very complex system that guaranties some very important properties while playing with your data.
>
> Data are accessible and updatable by many concurrent users as if they were the only ones playing around. Modern relational databases have become very good at ACID concurrency and offer scalability that are unattained by other storage systems. Beyond concurrency and referential integrity, relational databases have years of known record of stability in keeping data safe.
>
> Every major development platform had access to the main relational databases, data and databases hosting them will most likely outlive your application: compatibility is key, don't jeopardize it.

Fortunately, Hibernate Search addresses the three mismatches and makes the use of Hibernate and Lucene a pleasant experience where the developers can focus on the business value of search instead of spending most of their time in boring conversions and infrastructure code. Sounds like the perfect time to be introduced to Hibernate Search! The next chapter will get you started with Hibernate Search in a pragmatic way, from the set up and configuration process to the mapping and query writing process. While walking through this getting started guide, we will start to see how Hibernate Search addressed the three mismatches we have discovered in this chapter.