# Search Kit Programming Guide

User Experience > Carbon

2005-12-06

# Contents

# Figures, Tables, and Listings

# Introduction to Search Kit Programming Guide

**Note:** This document was previously titled *Adding Search to Your Application*.

Search Kit is Apple's content indexing and searching solution. It offers a powerful and streamlined procedural C framework that you can use to build information retrieval functionality into your Carbon, Cocoa, or Java application, or even into a command-line tool. In Mac OS X, Search Kit provides fast information retrieval in System Preferences, Address Book, Help Viewer, and Xcode. Apple's Spotlight technology is built on top of Search Kit to provide content searching in Finder, Mail, and the Spotlight menu.

Search Kit works in Mac OS X starting with version 10.3, Panther. Its features, starting with Mac OS X version 10.4, include:

- Fast indexing and asynchronous searching (improved in Mac OS X v10.4)
- Search mode determined by a Google-like query syntax, including phrase-based searching (new in Mac OS X v10.4)
- Text summarization (new in Mac OS X v10.4)
- Control over index characteristics including minimum term length, stopwords, and synonyms/substitutions
- Flexible management of document hierarchies and indexes
- Unicode support for language independence
- Relevance ranking and statistical analysis of documents to improve search quality

Search Kit is not for locating the position of search terms within a document or for finding documents based on their file-system attributes. Carbon and Cocoa offer other APIs for these other types of search, as described in "How Search Kit Works With Documents" (page 23).

## Who Should Read This Document

Whether you use Carbon, Cocoa, or a combination of the two, this guide provides the background you'll need to use Search Kit to add fast content searching to your application. If your application focuses on metadata rather than document content, you may want to consider using Spotlight instead.

- Use the Search Kit API when you want your application to have full control over indexing and searching. Also use Search Kit when your "documents" are not necessarily files on disk but web pages, database records, and so on.
- Use the simpler Spotlight API either when your focus is local file metadata or when your application does not need precise control over indexing or of the document hierarchy, or both.

## Organization of This Document

*Search Kit Programming Guide* contains the following chapters:

- "Search Basics" (page 9), an optional introductory chapter, gets you up to speed on some of the basics of information retrieval as a foundation for the rest of the book. If you are familiar with terms such as corpus, text extraction, inverted index, query, Boolean search, and relevance ranking, you can skip this chapter.
- "Search Kit Concepts" (page 19) describes the elements of Search Kit's approach to searching. Read this chapter to learn about Search Kit's notion of documents, terms, indexes, queries, searches, and search results. This chapter also provides an overview of the workflow behind a user's search, from text extraction to display of ranked results.
- "Search Kit Tasks" (page 35) provides detailed instructions on how to accomplish each of the steps involved in a practical usage scenario with Search Kit. It also provides sample code excerpts illustrating each step.

A glossary at the end lists terms that you need in order to understand information retrieval in general and Search Kit in particular.

## See Also

You may find this additional information from Apple helpful:

- *Search Kit Reference* describes the entire Search Kit API in detail.
- *Memory Management Programming Guide for Core Foundation* provides an introduction to memory management in Core Foundation. Search Kit uses Apple's Core Foundation style for memory management.
- *Debugging Programming Topics for Core Foundation* provides an introduction to debugging and error handling in Core Foundation. Search Kit uses Apple's Core Foundation style for error handling.
- Working With Spotlight provides an introduction to using Apple's Spotlight technology.

# Search Basics

Talking about an **information retrieval** (**IR**) technology such as Search Kit requires some terms and notions that may be unfamiliar to you. In this chapter you learn about these. Along the way, you learn that the workflow in an IR system isn't very different from the process of getting information from your local library.

If you understand such terms as corpus, text extraction, inverted index, query, Boolean search, and relevance ranking, you can skip this chapter and begin with "Search Kit Concepts" (page 19).

This chapter describes information retrieval as a three-step process:

1. Establishing a suitable source to search
2. Formulating a query
3. Invoking a search and providing results

This chapter steps you through a search workflow using a library metaphor, relating it to a generic computer-based IR system with occasional mention of Search Kit particulars.

## Establishing a Suitable Source to Search

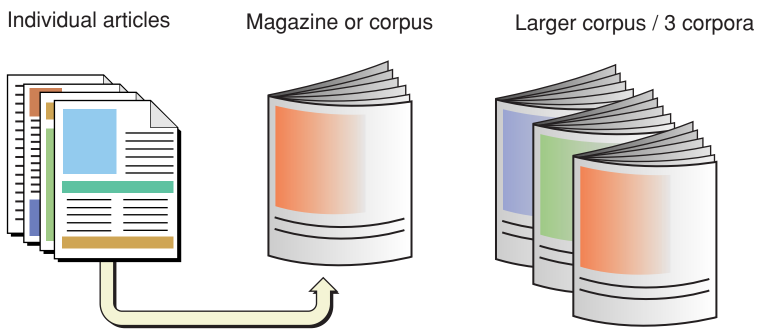
You have an information need. But before you can ask a question, you need someone or something to ask. That is, you need to establish who or what you will accept as an authority for an answer. So before you ask a question you need to define the target of your question.

Such a target can be as broad as the entire Internet for a simple search using Google, or as specific as a local mailbox in a user's Mail program. Using a library metaphor here, you'll ask a librarian, who will in turn consult some magazine indexes. The librarian plays the role of your application. The librarian's special reference skills play the role of Search Kit. The magazine indexes play the role of Search Kit indexes.

### Selecting a Document Collection

The target of a question corresponds to the information retrieval notion of a **document collection**, or more formally, a **corpus**, as depicted in Figure 1-1.

**Figure 1-1** A corpus is a collection of documents



If one considers a magazine article to be a **document**, then an issue of the magazine constitutes a corpus—it is a collection of one or more documents. The 12 issues published in a year are corpora ("Of these corpora, which has the most articles?"), but considered as a single, larger collection, they become a single, larger corpus ("In this corpus that includes 12 issues, which articles mention sports cars?").

Similarly, if one considers a mail message to be a document, then each of the mailboxes in a user's Mail application is a corpus. A set of mailboxes, when you search in one or another individually, are corpora. When you search a set of mailboxes as a single large collection, the set constitutes a single, larger corpus.

So generally speaking, information retrieval is a two step process that starts with specifying a corpus and proceeds to specifying a query.

If you call a librarian, you've chosen the library as your corpus. If you then ask, "Which make of car has the best trade-in value?" the librarian might narrow the effective corpus by looking through the articles in the past year's issues of a consumer magazine and an automotive magazine. In this scenario, the librarian, acting on your behalf, expects that one or more of these documents will contain a good answer—that is, the librarian expects that those issues constitute an appropriate corpus to search.

## Constructing Indexes

To get back to you in a reasonable amount of time, of course, the librarian wouldn't go to the magazine shelves and read every issue published over the past year, cover to cover. He'd use indexes. Computer-based information retrieval systems do the same.

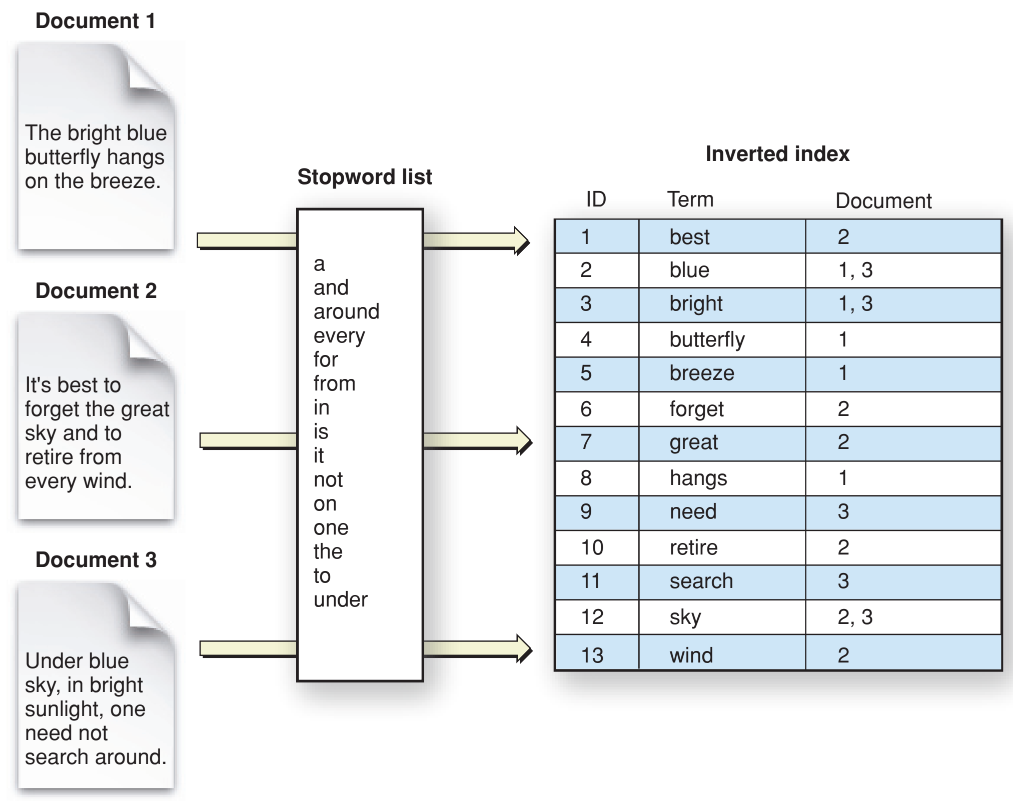
An **index** maps the salient information in a corpus into a format designed to let you quickly locate specific content. For example, an annual magazine index includes the key terms from every article in every issue published that year.

### Inverted Indexes

Each entry in a magazine index points you to one or more articles. You can think of an index as a list of terms, with each term followed by a list of the documents it appears in. This sort of index, the one that people usually think of, is formally known as an **inverted index**. The term "inverted" refers to the arrangement of information in the index, which is intended to locate documents by matching on terms. This is "inverted" compared to using documents directly: if you pick up a book, you "match" on the document and "locate" all the terms in it.

Figure 1-2 shows an inverted index schematically and depicts a simplified version of how such an index might represent terms from a set of documents.

**Figure 1-2** Inverted index



Document 1: The bright blue butterfly hangs on the breeze.

Document 2: It's best to forget the great sky and to retire from every wind.

Document 3: Under blue sky, in bright sunlight, one need not search around.

Stopword list: a, and, around, every, for, from, in, is, it, not, on, one, the, to, under

| ID | Term | Document |
|---|---|---|
| 1 | best | 2 |
| 2 | blue | 1, 3 |
| 3 | bright | 1, 3 |
| 4 | butterfly | 1 |
| 5 | breeze | 1 |
| 6 | forget | 2 |
| 7 | great | 2 |
| 8 | hangs | 1 |
| 9 | need | 3 |
| 10 | retire | 2 |
| 11 | search | 3 |
| 12 | sky | 2, 3 |
| 13 | wind | 2 |

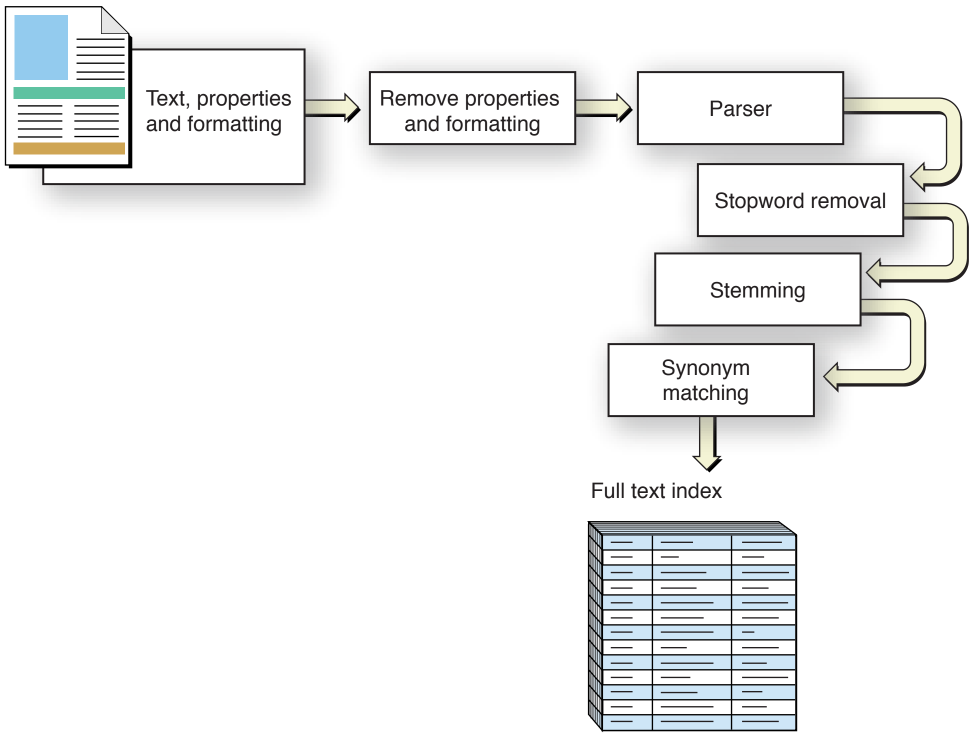
This figure includes something called a **stopword** list, used to filter out specified terms during text extraction. (S"Stopwords and Minimum Term Length" (page 12)topwords are described shortly.)

## Text Extraction

The earliest known use of inverted indexes is 1247 CE, when the first concordance of the Bible was created. Back then the process took the efforts of several hundred monks. (This historical information comes from a scholarly overview of text analysis from the University of Alberta, titled What Is Text Analysis?.)

An information retrieval system, instead of relying on multiple monks, employs a **text-extraction** algorithm to harvest relevant terms from a document. A **term** that becomes an entry in an index is typically a word. Figure 1-3 illustrates the basic steps in text extraction, starting with a formatted document and resulting in a full text index.

**Figure 1-3** Text extraction, from formatted document to full text index

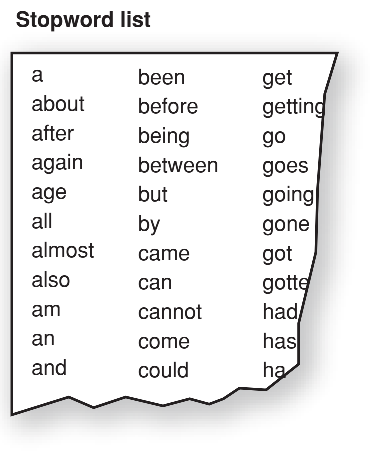

Text, properties and formatting
Remove properties and formatting
Parser
Stopword removal
Stemming
Synonym matching
Full text index

## Stopwords and Minimum Term Length

Text extraction entails a tradeoff between coverage and relevance. If you index every term in a corpus, you have perfect coverage and won't miss anything but you'll end up with an index whose percentage of useful entries is low. In many cases there's no benefit to extracting words like "the," "and," or "it" and placing them in an index.

Indexing systems, such as the one in Search Kit, let you specify a list of **stopwords**—terms to ignore during text extraction—and to specify a **minimum term length**. Terms in the corpus either shorter than the minimum term length or included in a stopword list are skipped during text extraction. Figure 1-4 shows an excerpt from a typical stopword list.

**Figure 1-4** A typical stopword list (excerpt)



Stopword list

| | | |
|---|---|---|
| a | been | get |
| about | before | getting |
| after | being | go |
| again | between | goes |
| age | but | going |
| all | by | gone |
| almost | came | got |
| also | can | gotte |
| am | cannot | had |
| an | come | has |
| and | could | ha |

Any word skipped in this way won't be in the index and so won't be searchable. This is sometimes just what you want. Such words rarely relate to the meaning or value of a document.

But if you want to support searching for exact phrases, excluding common words can be problematic precisely because they are common and appear often in phrases. One option some IR systems use to solve this dilemma is to index everything, then filter on stopwords and minimum term length at query time unless the query is phrase-based. For more on phrase-based searching, see "Phrase Searches" (page 14).

Stopword lists are, of course, language specific and perhaps even domain specific. A corpus of chemical descriptions would likely benefit from a different stopword list than a corpus of children's short stories would. If you plan to market your application to different cultural regions, or to widely divergent markets even within one region, keep this in mind.

## Synonyms and Stemming

Indexing systems also employ lists of **synonyms** to improve search quality. For example, if you ask a librarian a question about used-car trade-in value, he might look in some magazine indexes under "car"; but the text of a relevant article might mention only "passenger vehicle" or "automobile." If the indexing system associated these alternate terms as synonyms of "car," the relevant articles would also be attached to the index term "car."

Teaching an indexing system about synonyms amounts to giving it a list, the creation of which is a manual process. Search Kit supports synonyms as a so-called "substitution list" in the text analysis properties dictionary of an index. For more on this, see `kSKSubstitutions` and `SKIndexCreateWithURL` in *Search Kit Reference*.

There's an algorithmic technique as well for increasing the likelihood users will find what they're looking for. This technique is called **stemming** or, sometimes, **suffix stripping**. Search Kit does not perform stemming, but it's useful to know about it so you can better understand Search Kit's behavior in your application. You may want to provide stemming functionality yourself.

Most languages have closely related words, known as morphological or inflectional variants, based on a common portion known as a **stem**, or **root word**. In English, the stem tends to come at the beginning of words: "swim," "swimming," and "swimmer" share the common stem "swim."

An indexing system that knows how to stem will convert each alternate word form, during text extraction, to a common stem word. The associated query system will also convert words in a query string to the appropriate stem. For example, “swim,” “swimming,” and “swimmer” would all be transformed to “swim.” Some stemming systems can deal with irregular endings as well. For instance, a stemmer could equate “swam” with “swim” even though the stem “swim” does not appear in this variant.

Stemming not only increases the likelihood of a successful search, it also decreases index size.

If your application requires stemming behavior, you can add it yourself using a standard algorithm such as the one developed by Martin F. Porter in 1980. Using the Porter Stemming Algorithm, sometimes called the Porter Stemmer, your application would get the text from the documents in a corpus, stem it, and then hand it to Search Kit for indexing. Your application would also apply stemming to queries.

Keep in mind that stemming, like a useful list of synonyms, and like a stopword list, is language and domain dependent.

## Minimum Term Frequency

Another way to reduce index size and increase index quality is to employ a **minimum term frequency** during text extraction. An indexing system that supports minimum term frequency skips over terms that appear in a document fewer than a specified number of times.

The idea behind minimum term frequency is that if a term appears only once in a document, that document is not likely to be a useful source of information on that topic. Search Kit does not currently support a minimum term frequency, but you could add this behavior to your application using other Mac OS X frameworks.

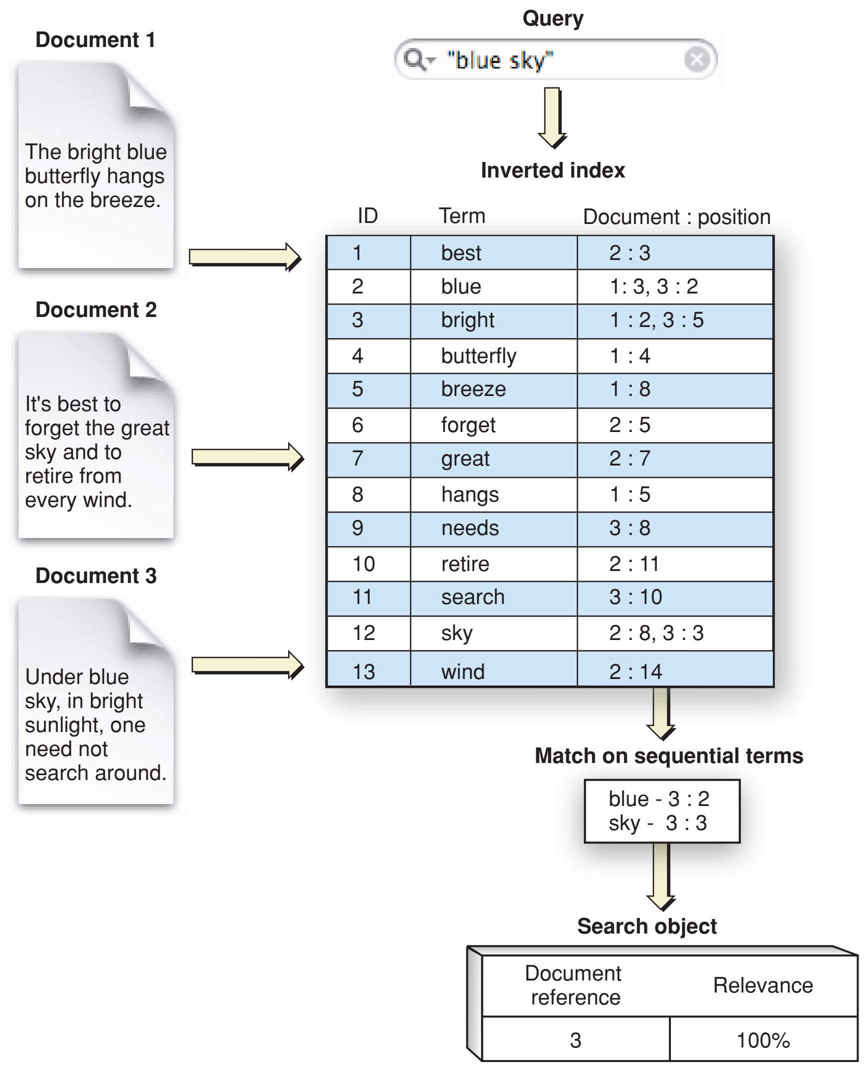
An information retrieval system that needs to support phrase searches should not exclude words from an index based on term frequency, just as it should not exclude words using a minimum term length or stopword list.

## Phrase Searches

Indexes can be constructed in a way that supports **phrase** or **proximity** searching. These allow users to search, for example, for “Search Kit” as a complete phrase, as opposed to searching for documents that contain the terms “search” and “kit” anywhere in their content.

In an index that supports phrase searching, a term’s linear position in a document is recorded along with a reference to the document the term appears in. See Figure 1-5 (page 15).

**Figure 1-5** An inverted index that supports phrase searches


Query
"blue sky"
Document 1
The bright blue butterfly hangs on the breeze.
Document 2
It's best to forget the great sky and to retire from every wind.
Document 3
Under blue sky, in bright sunlight, one need not search around.
Inverted index
Match on sequential terms
blue - 3 : 2
sky - 3 : 3
Search object

| ID | Term | Document : position |
|---|---|---|
| 1 | best | 2 : 3 |
| 2 | blue | 1: 3, 3 : 2 |
| 3 | bright | 1 : 2, 3 : 5 |
| 4 | butterfly | 1 : 4 |
| 5 | breeze | 1 : 8 |
| 6 | forget | 2 : 5 |
| 7 | great | 2 : 7 |
| 8 | hangs | 1 : 5 |
| 9 | needs | 3 : 8 |
| 10 | retire | 2 : 11 |
| 11 | search | 3 : 10 |
| 12 | sky | 2 : 8, 3 : 3 |
| 13 | wind | 2 : 14 |

| Document reference | Relevance |
|---|---|
| 3 | 100% |

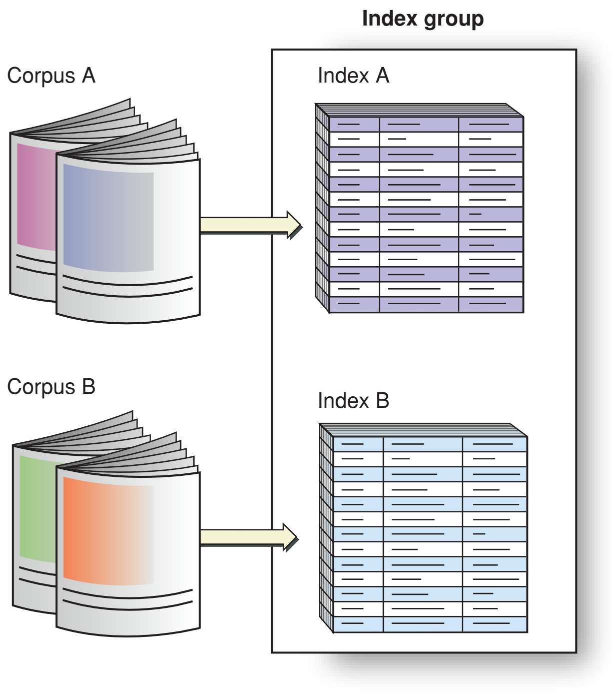
Searching for a phrase amounts to searching for a series of terms that appear in consecutive order. Similarly, searching for proximity amounts to searching for a pair of terms whose linear distance is small. Search Kit supports phrase searching in inverted and inverted-vector indexes but does not currently support proximity searching.

## Searching Multiple Indexes

Returning now to our library metaphor: If a librarian picks two annual indexes (one each from two magazines) to find an answer to a question about automobile trade-in values, he creates an **index group** consisting of two indexes, each of which contains the terms from multiple documents. Information retrieval systems can offer the same functionality. In the case of Search Kit, it is up to your application to define and manage index groups.

Just as a corpus is a collection of one or more related documents, an index group brings together one or more related indexes, as shown in Figure 1-6.

**Figure 1-6** An index group



This figure illustrates two indexes that each represent a different corpus. You can also create two indexes on the same corpus to include in a group. For example, one index might contain only article titles and another might contain article body text. If you create an index group with these two indexes, searches on it would cover both sorts of content. An index "group" containing only the title index would let a user find articles based strictly on titles.

Just as a librarian doesn't read magazines when searching for an article (but goes straight to an index instead), an IR system doesn't scan the documents in a corpus during a search. So there is a lag between the time of text extraction during index construction and the time of index scanning during a search. Depending on how volatile the information is in a corpus, and how quickly a user expects a search response, an application that uses an IR system may want to refresh its indexes before invoking a search.

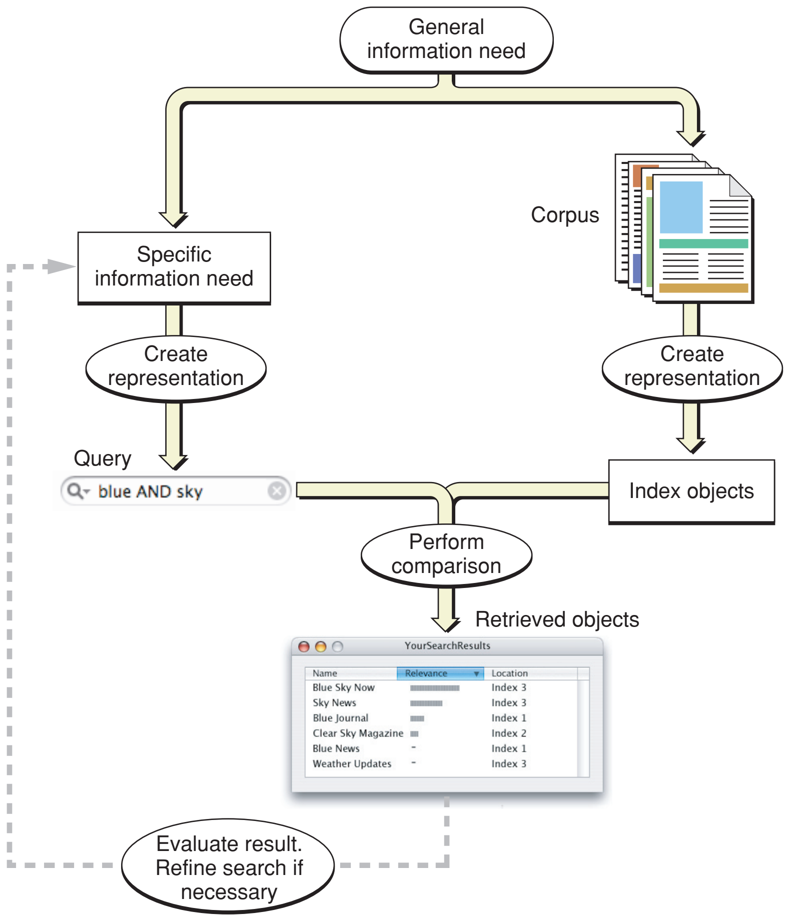
Often, the best time to refresh an index is after a change is made to any of its referenced documents. This is appropriate when an application exerts exclusive control over creating, deleting, and editing its documents. Spotlight, for example, uses this strategy whenever a document is changed, added to, or deleted from the Mac OS X file system.

## Formulating a Query

In our ongoing example, you started with a general information need and used it to identify a suitable, searchable corpus—namely, a library with a friendly librarian. You expected the library to contain the information you need, and you expected it to be efficiently accessible. Thanks to magazine publishers who create annual indexes, it is. The figure Figure 1-7 summarizes these steps from the top down on the right side.

The next step in using an IR system is to define a specific information need and represent it as a **query**, a process that the figure illustrates on the left side.

**Figure 1-7** An information retrieval system



| Name | Relevance | Location |
|---|---|---|
| Blue Sky Now | ▬▬▬▬ | Index 3 |
| Sky News | ▬▬▬ | Index 3 |
| Blue Journal | ▬▬ | Index 1 |
| Clear Sky Magazine | ▬ | Index 2 |
| Blue News | - | Index 1 |
| Weather Updates | - | Index 3 |

In this chapter, we've supposed that you want to know which make of car has the best trade-in value, and we've supposed that you are using a library to find your answer. Alternatively, you might ask a friend or use Google. In each case you expect your chosen corpus to contain the answer. And, in each case you need to formulate a specific query that expresses what you want to know in terms understood by whom or what you are asking.

You use natural language to ask a friend or a librarian, and a Boolean-like syntax for Google. To get the answer using a magazine index, you (or a librarian) would scan its alphabetical listings while holding one or more keywords in mind. In other words, the IR system determines the appropriate form for a query.

Some common sorts of queries for software-based IR systems are **Boolean**, **prefix**, and **phrase**. To get what you want, you need to pick the appropriate type of search and then use corresponding query syntax.

A librarian or an IR system can provide guidance to promote a successful search. Apple's search field widget, for example, provides a standard and flexible interface for invoking a search.

To review and then complete our library metaphor, you call your local library and ask for the information desk. The very cooperative librarian listens to your car trade-in value question and then puts you on hold for a few minutes. The librarian performs the search by comparing your query to the available indexes, represented in the figure Figure 1-7 as the "Perform comparison" step. He comes back on the phone, with your retrieved answer on a a scrap of paper, and says, “The make of automobile with the best trade in value this year is the XYZ.”

You have your answer, but that isn’t necessarily the end of the story. The answer might prompt you to realize you want to consider other criteria, such as purchase price. Or you might want to know which makes rank second and third.

The response you get becomes part of what is known and helps you focus more clearly on what you want to know. Information retrieval is often cyclical in this way. The IR systems that you develop should anticipate that answers lead to new questions. Google “Similar pages” links are an example of this.

## Providing Results

The two basic ways to provide results in an IR system are by inclusion/exclusion and by relevance.

An **inclusion/exclusion result** includes only documents that satisfy a query. The documents won’t be in any particular order—they will not be ranked by relevance. If a user wants to know which CDs in a catalog are by (fictional) rock band The Pink Widgets, then you simply want to return all the Pink Widgets CD titles as a result. You may order the results based on release date or popularity, but those are considerations outside the request encapsulated by the query.

In the case of the library search for used-car trade-in values, the librarian in our example didn’t bother saying which article was best. He just found the information and gave it to you.

A **relevance-based result**, on the other hand, includes documents with a range of relevance to a query. For example, an IR system might sort CD titles by The Pink Widgets toward the top of a results table but also list CDs containing song titles with the words ‘pink’ and ‘widget’ in them, or CDs containing cover versions of Pink Widgets songs, sorted lower in the table.

The best sort of result presentation will put the most satisfying matches at the top. More than this, it will present matches in a way that helps a user refine a search if need be—perhaps by providing context for each result, perhaps by providing associated information such as modification date or user-entered comments.

Once you’ve added information retrieval to your application, you’ll probably want to fine-tune the presentation of results to provide maximum value. One way to approach such fine-tuning is by experimenting with a representative corpus and with a suite of expected user queries. With such experimentation you may discover, for example, that you can present sets of similar documents, or documents sharing a common attribute, as groups in an outline-style view. For example, a search in Mac OS X using the Spotlight search field shows results sorted by category such as Documents, Mail Messages, and PDF Documents.

# Search Kit Concepts

In this chapter you learn about Search Kit's capabilities, architecture, workflow, and internal workings. Before reading it you should understand the terms and ideas covered in "Search Basics" (page 9).

Apple designed Search Kit as a highly flexible information retrieval framework. When using Search Kit, a "document" is any text container as understood by your application. You are not limited to searching files on disk but can search any body of textual information—pages distributed across a network, live database content, or custom, application-defined data. Similarly, your application can define what counts as a "term" for Search Kit. For Japanese text, Search Kit parses terms using Apple's Japanese language analysis technology.

Search Kit uses **document URL objects**, similar to CFURL objects, to refer to documents. Using document URL objects, your application can define any sort of **document object hierarchy** and location scheme.

The first two sections of this chapter—"Search Kit Architecture" (page 19) and "Search Kit Application Workflow" (page 21)—provide a high-level understanding. The meat of this chapter is in the final section, "How Search Kit Works" (page 23).

## Search Kit Architecture

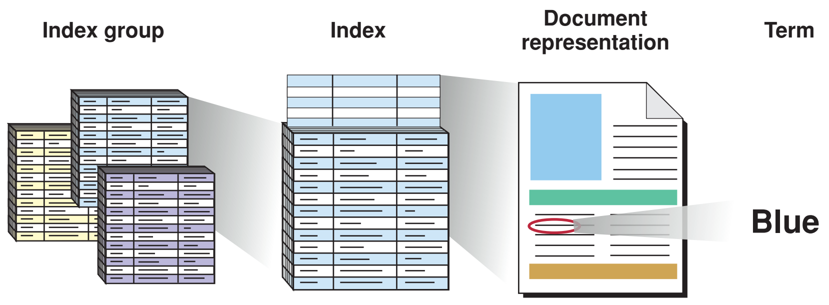
The Search Kit API is a C language framework within the Core Services umbrella framework. As such, it employs memory-management and error-handling conventions from Apple's Core Foundation technology and makes use of Core Foundation data types.

This section is a brief tour through Search Kit's architecture, providing just enough context to understand how Search Kit's pieces fit together. To dig deeper into any of the topics presented in this section, refer to "How Search Kit Works" (page 23) and "Search Kit Tasks" (page 35).

### Indexes, Documents, and Terms

Search Kit uses a simple information containment hierarchy to allow your application to manage the content, or corpus, it is responsible for. Figure 2-1 illustrates this hierarchy by zooming in successively from left to right.

**Figure 2-1** The Search Kit containment hierarchy



At the outermost level, as shown in the figure on the left, an application typically works with a group of indexes. An individual index, depicted in the figure under the heading "Index" and represented in Search Kit as an `SKIndexRef` opaque type, contains representations of one or more documents.

A document representation in an index is a key/value pair. The key is a lightweight, unique identifier of type `SKDocumentID`. The corresponding value includes a document URL object of type `SKDocumentRef`. Zooming in on one such document representation, the figure depicts an individual term associated with one document.

A Search Kit index associates each document with the terms extracted from it. A term is also represented as a key/value pair in an index. A term's key, like a document's key, is a lightweight, unique identifier. A term ID is of type `CFIndex`. The value for a term includes the term string itself. Given a term ID, your application can get the term by calling the `SKIndexCopyTermStringForTermID` function.

Index groups, as shown on the left in Figure 2-1, are not explicitly supported in Search Kit but have many uses. You implement them in your application for such things as:

- Simultaneously searching multiple fields in documents such as emails, where you have one index for the body content, another for the “To” header, and so on
- Simultaneously searching multiple corpora, such as multiple email mailboxes

## Index Types

You’ll typically create disk-based (persistent) indexes using the `SKIndexCreateWithURL` function, which creates an index in a file. One index file can hold any number of Search Kit indexes. The choice to put one or more than one index in a file has implications regarding how an application manages searches, as described later in this chapter.

Search Kit supports memory-based indexes as well, with its `SKIndexCreateWithMutableData` function.

Whether it is file or memory based, you set an index’s capabilities when you create it. There are two aspects to consider.

- *Index type* determines whether the index will be optimized for query searching or similarity searching.
- An index’s *text analysis properties dictionary* determines whether the index will support phrase-based searches. It also sets various index attributes that bear on index size and search efficiency.

### Indexes and Search Objects

A new Search Kit index is empty, ready to accept documents. Search Kit provides functions for creating document URL objects, for adding them and their associated text to indexes, and for reindexing documents that change or move.

To find documents, your application creates and then queries a **search object**. The search object, upon its creation, initiates search and then acts as a dynamic repository for results. Your application can query the search object immediately after its creation and then repeatedly to get additional results.

Each search object is associated with exactly one index. By creating multiple search objects, you can perform searches on multiple indexes sequentially or in parallel, according to the requirements of your application.

## Search Kit Application Workflow

Performing a search is a straightforward, two-step process:

1. Create a searchable corpus.
2. Using the user's query, perform a search and display the results.

### Create a Searchable Corpus

You make textual content searchable by indexing it. To do this, you:

1. Create an empty index using the `SKIndexCreateWithURL` function for a file-based index, or with the `SKIndexCreateWithMutableData` function for a memory-based index.
2. Create document URL objects to add to the index by calling the `SKDocumentCreateWithURL` or the `SKDocumentCreate` function, depending on how your application wants to manage your corpus.
3. Add document URL objects and text to the index with the `SKIndexAddDocumentWithText` or `SKIndexAddDocument` functions. In the general case, your application takes responsibility for getting text out of a document, using other Mac OS X frameworks, and then hands the text to an index in the form of a CFString object using the `SKIndexAddDocumentWithText` function. Alternatively, the `SKIndexAddDocument` function makes use of the Spotlight text importers to get the text from a local, file-based document.

### Perform a Search and Display the Results

A user’s request for information includes a query along with a specification of where to search. The query comprises words and perhaps **operators**, such as “`&`,” representing a logical `AND`, or “*”, the **wildcard character**. The specification of where to search—that is, which indexes to use—may be supplied by your application or supplied by the user. As an example of a user-supplied search location, the set of mailboxes a user selects in Mail becomes the active set of locations for search.

Different types of indexes support different types of search, so you may want your application to guide users appropriately in regard to query type. For example, if an index doesn't support phrase searching, your application could provide appropriate feedback if a user enters a query with a double-quote-delimited text string.

As preparation for a search, your application should use the `SKIndexFlush` function to update and flush to disk the indexes specified in the query. Updating and flushing ensure that you invoke the search on fresh data.

However, if the information in the corpus changes rarely, or if the corpus is so large that the time consumed by updating the indexes would frustrate user expectations, you may want to update more strategically. For instance, if your application controls when documents change, you can update an index incrementally each time a document changes. Mac OS X does this for file system searches that use Spotlight by way of file system notifications.

You create a search object with the asynchronous `SKSearchCreate` function. You then query it with the `SKSearchFindMatches` function, which provides results as an array of document IDs and a parallel array, if requested, of relevance scores. Your application can get document locations for these IDs, in the form of document URL objects, by calling the `SKIndexCopyDocumentURLsForDocumentIDs` function. Finally, your application can display these document locations as search results using other Mac OS X frameworks.

## Additional Workflows

In addition to query-based searches, Search Kit supports similarity searching and summarization. This section briefly describes these workflows as well as index maintenance in terms of removing and reindexing documents.

### Similarity Searching

In a **similarity search**, a user looks for documents similar to an example document. The workflow is nearly identical to the workflow for a query search, with these differences:

- When creating a search object with the `SKSearchCreate` function, use the `kSKSearchOptionFindSimilar` flag in the *`inSearchOptions`* parameter.
- Provide a string representing a document, or a portion of a document, to the `SKSearchCreate` function's *`inQuery`* parameter.

### Using Summarization

Starting with Mac OS X v10.4, Search Kit supplants the summarization functionality previously available in Apple's Find By Content technology. You can use summarization independently of search or as an adjunct to your application's display of search results. Find By Content remains available in Mac OS X for backward compatibility only.

To perform summarization you create a **summarization object** of type `SKSummaryRef` by passing a text string to the `SKSummaryCreateWithString` function. You then use the `SKSummaryCopySentenceSummaryString` or `SKSummaryCopyParagraphSummaryString` functions to generate a sentence- or paragraph-based summary of the size you want. Each of these functions has a parameter that lets you specify summary length as an integer number of sentences or paragraphs.

You can see summarization in action by selecting a block of text in a Mac OS X application and then choosing the Summarize service from the Services submenu in the application menu.

For additional control over summarization, Search Kit supplies other functions that let you work with individual sentences and paragraphs from the summarization object. For more information, refer to *Search Kit Reference*.

### Index Maintenance

To reindex a document that has changed but whose location is the same, you simply replace it in the index. If a document has moved, or moved and changed, you remove the old version and then add the new version. For details refer to "Search Kit Tasks" (page 35).

## How Search Kit Works

Here you learn how Search Kit works with documents as abstract objects, how it indexes content, and how it manages queries and results.

### How Search Kit Works With Documents

Search Kit works with any collection of textual information as a document object hierarchy. This approach gives your application a great deal of flexibility in how it defines and manages documents.

There are two ways Search Kit can manage a document object hierarchy. One is specifically for documents that are disk-based files—those whose URLs use the `file` **scheme**. The other is general; it works with any sort of document, whether it is file based, memory based, or application defined.

- If your application works with documents that are disk-based files, you can use the file-system hierarchy directly. With this option you let Search Kit find documents and get their contents. Document URL objects in this approach are equivalent to file-system paths.
- If your documents are not disk-based files or you want more control over the document object hierarchy, use Search Kit's general approach. In this approach, your application specifies, using a flexible URL format, how to refer to documents. Your application takes responsibility for locating documents, and during indexing, your application hands the textual content of a document to Search Kit in the form of a CFString object.

With either approach, when you create a document URL object, you give Search Kit the information needed to find the document. This may be a file-system URL, an Internet URL, a SQL statement, an ID number, and so on—the format is up to your application. During a search, when Search Kit identifies a document in response to a user's query, your application can ask for the location information and use it to get the associated document.

## Working With Documents and Document URL Objects

Using Search Kit, the definition of "document" is largely up to your application. A document, to Search Kit, is simply a locatable chunk of text. The chunking depends on your application which serves as an intermediary between Search Kit and the information you want to search.

An email client, for example, might use a single plaintext file (such as in the Unix `mbox` format) to hold multiple documents—namely, the set of mail messages in a mailbox. Another application might employ a one-to-one correspondence between files and documents. Or an application might consider all the files within a folder to be a single, multipart document—in this case, a search hit in any of the referenced files might direct the user to the containing folder or bundle. It's up to your application.

To Search Kit, a document is atomic in that it defines the granularity of a search. Using Search Kit, your application can find documents—as your application understands them—but cannot locate the position of a term within a document. If you want to locate matches for a user's query within a found document, use the MLTE `TXNFind` function in Carbon or the `NSStringrangeOfCharacterFromSet:options:` method in Cocoa.

To Search Kit, a document's structure is irrelevant in that Search Kit indexes don't know anything about paragraphs, subtitles, tagging, or fields of information within a document. Search Kit sees a document's content simply as a bag of terms. If you want to let users search by various attributes of documents, you create an index for each attribute.

Search Kit is concerned only with textual content, so it does not keep track of file-system attributes such as modification timestamps. In Carbon, use the `PBCatSearch` function to search by file-system attributes. In Cocoa, use the `NSFileManager` class. You can also use the Spotlight API to search for documents according to file-system attributes or other metadata.

## Using Document Locations

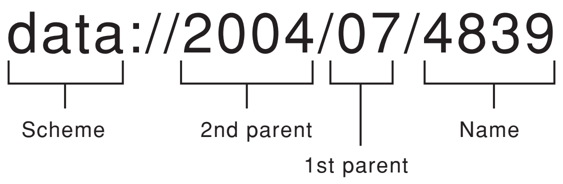
As described above, Search Kit's notion of document location is encapsulated in something called a document URL object. Document URL objects correspond to the simple data type `SKDocumentRef`. A document URL object is similar to a CFURL object but lets you use any format that you like to represent a document's location. It's up to your application to interpret the location to retrieve the document.

There are multiple ways to create document URL objects. You can create them by converting CFURL objects using the `SKDocumentCreateWithURL` function. Alternatively, you can give this function a URL directly. In this case you can use any URL scheme you like, including standard schemes such as `file`, `http` and `ftp`; or nonstandard schemes of your own design, such as `data`.

Yet another way to build document URL objects is to construct a document object hierarchy node by node. Search Kit supports the building of document object hierarchies with its `SKDocumentCreate` function. This function, rather than taking a complete URL as `SKDocumentCreateWithURL` does, builds a URL for you from a triple of information that you hand it: the document **name**, the **parent document URL object**, and an optional **scheme**.

Figure 2-2 illustrates these components as they appear in a URL. If you want to take advantage of Search Kit's ability to locate and read local, file-based documents, the Name portion should match the document filename. If your application manages the documents in its corpora, the Name portion may match the document filename or not, according to the application's needs.

**Figure 2-2** A Search Kit document URL



The full URL is the location information in the document URL object. The portion to the left of the Name, up to but not including the final slash, is the location information in the document URL object for the parent. To tell the `SKDocumentCreate` function the scheme for the document URL object, use the text starting from the left and up to, but not including, the colon.

A document object hierarchy can be flat, tree-based, or more complex. Your application defines and uses it.

You can determine document URLs by assembling them piece by piece if you want, using the functions `SKDocumentGetName`, `SKDocumentGetParent`, and `SKDocumentGetSchemeName` to query document URL objects. Start at the leaf document and traverse upward, parent to parent. This works with all document URL objects—those created by converting CFURL objects, those created from URLs directly, and those created by using name, parent, and scheme.

The parent-child-based control of document object hierarchies you get by using the `SKDocumentCreate` function can be useful when you want to attach information to nondocument nodes in a hierarchy. For example, you may want to record when the documents in a folder were last indexed; you can associate that information with the document URL object of the enclosing folder. This can also be useful when the documents in your corpus do not correspond to disk-based files—for example, when they are database records or tagged chunks of text within an enclosing file. In cases like these, a nondocument node is a good place to store metainformation about subordinate documents.

A side effect of Search Kit's powerful generality in handling document object hierarchies is that when you create a document URL object from a multipart URL, Search Kit creates a series of document URL objects, one for each element in the path. Use these if you want, as just described, or ignore them if they're not useful.

In the special case of `file` URLs, Search Kit knows how to find documents for you. It also knows how to harvest the content of local files, as described in "Terms, From Documents to Indexes" (page 26). In every other case, your application manages a document object hierarchy and interprets document locations from document URL objects.

You can get the URL itself from a document URL object by passing the object to the `SKDocumentCopyURL` function.

## Document Properties

You can associate information with each document in an index by way of a properties dictionary, using the `SKIndexSetDocumentProperties` function. The format of this optional dictionary can be as simple or as complex as you want. For example, an email program could include a property dictionary for each mailbox, describing metainformation such as number of read and unread messages, the file-system location of the mailbox, and so on.

You can use this property information as context for the user or for your application but you cannot directly search it. Retrieve property information using the `SKIndexCopyDocumentProperties` function.

## How Search Kit Extracts Terms From Documents

**Terms** are the currency of information retrieval. In a search, you provide a query consisting of terms, perhaps including operators, and Search Kit matches the query with indexed terms that have been extracted from documents. Here you learn how terms get from documents into indexes. You also learn about the Search Kit index types and about how they support various types of searching.

You may want to first review "Constructing Indexes" (page 10) in the Search Basics chapter, which provides an implementation-independent introduction to indexes.

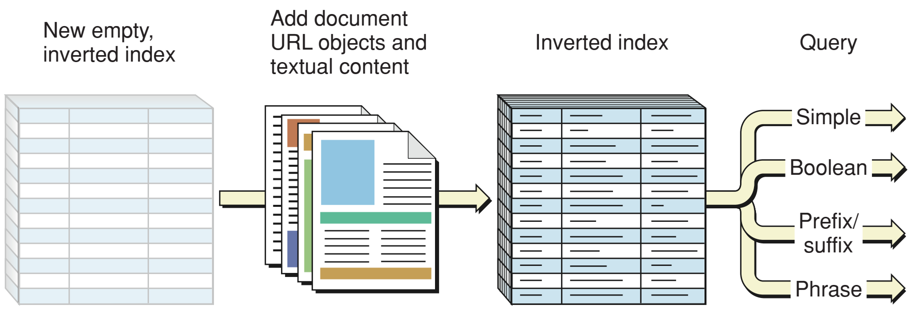
### Terms, From Documents to Indexes

Terms get from documents into Search Kit indexes through a three-step process:

1. You ask Search Kit to create a new, empty index if you need one. Otherwise, get the reference for an existing index.
2. You create a document URL object for each of the documents you want to index.
3. You add the document URL objects and textual content to the index.

Figure 2-3 depicts this process.

**Figure 2-3** Adding documents to a new index


New empty, inverted index
Add document URL objects and textual content
Inverted index
Query
Simple
Boolean
Prefix/ suffix
Phrase

In the general case and for documents that do not correspond to on-disk files—webpages, database records, tag-delimited subsets of files, data in memory, or custom, application-specific content—use the `SKIndexAddDocumentWithText` function. Your application explicitly sends document text in the form of a CFString object to the function and provides the document location as a document URL object. You can also use this function for on-disk, file-based documents when you want more control over the indexing process. You might do this, for example, for an XML document when your application understands the tagging semantics.

To ask Search Kit to get textual content for you from an on-disk file using Spotlight importers, use the `SKIndexAddDocument` function. This function converts a file-system path to a document URL object, and (with help from the Spotlight importers) gets the text to be placed in the index.

If your application relies on Spotlight importers, when your program launches, tell Search Kit to load them by calling the `SKLoadDefaultExtractorPlugIns` function.

In either case, the "add-document" functions—`SKIndexAddDocument` and `SKIndexAddDocumentWithText`—parse textual content into terms before placing it into an index. If your application is reading Japanese text, Search Kit uses Apple's Japanese language analysis technology.

Each term in a Search Kit index has a unique ID and is associated with a list of document URL objects, as illustrated in the Search Basics chapter in Figure 1-2 (page 11). Various Search Kit functions let you convert between terms and IDs, determine which documents contain a term, get the number of times a term appears in a document, and so on.

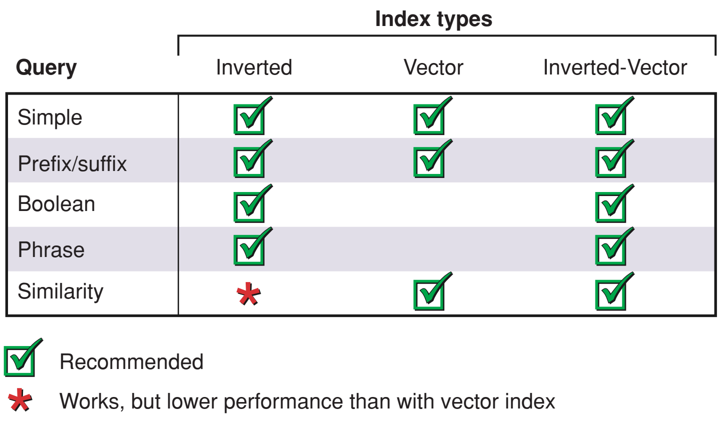
## Index Types

Three types of Search Kit indexes handle various functionality and efficiency requirements:

- *Inverted indexes* map terms to documents. Use an inverted index to allow users to discover which documents match their queries. This is the preferred index type for most applications.
- *Vector indexes* map documents to terms. Use a vector index to let users find documents based on a similar, specified document—that is, to perform similarity searching.
- *Inverted-vector indexes* combine the characteristics and capabilities of inverted and vector indexes. They consume more memory and file space than either of their constituent types.

When you create an index, you specify one of these three index types using the *`inIndexType`* parameter in either the `SKIndexCreateWithURL` or `SKIndexCreateWithMutableData` function. The type determines how each added document will be indexed and indirectly determines which sorts of searches your users can perform. For example, a vector index does not support Boolean queries or phrase-based searching. Figure 2-4 lists the various index types and the search capabilities for each.

**Figure 2-4** Index types and their available search types



| | Index types | | |
|---|---|---|---|
| **Query** | Inverted | Vector | Inverted-Vector |
| Simple | ✓ | ✓ | ✓ |
| Prefix/suffix | ✓ | ✓ | ✓ |
| Boolean | ✓ | | ✓ |
| Phrase | ✓ | | ✓ |
| Similarity | * | ✓ | ✓ |

✓ Recommended
* Works, but lower performance than with vector index

**Inverted indexes** are optimized for fast query-based searches and for minimal index size. They map terms to document. That is, terms are the keys in the key-value pairs in inverted indexes.

Use an inverted index unless your application's primary use is similarity searching. Although inverted indexes work for similarity searching, performance is slower than when searching by similarity using a vector index.

A Search Kit inverted index lists each constituent term exactly once—no matter how many of the documents contained in the index include the term and no matter how frequently the term appears in any of the documents. The value in a term's (key, value) pair in an inverted index includes a number indicating how many of the index's documents contain the term, IDs of the documents that use the term, and how often the term appears in each document.

If you specify proximity-searching support when you create an index, the index also tracks the position of each instance of the term in each document.

**Vector indexes** map documents to terms. That is, documents are the keys in the (key, value) pairs in vector indexes. Their primary use is fast similarity searching.

The value in a document's (key, value) pair in a vector index includes a number indicating how many terms the document contains, IDs of the terms in the document, and how often each term appears in the document.

Vector indexes do not support Boolean or phrase-based searching. These limitations, along with their relatively large size, make them a bad choice unless your primary need is fast similarity searching.

**Inverted-vector indexes** support every type of Search Kit query but are larger still than vector indexes. Their only practical application is to support Boolean, phrase-based, and fast similarity searching on the same index, and when index size will not be an issue.

### Text Analysis Properties

In addition to having a type, each Search Kit index has a **text analysis properties dictionary** that defines a variety of index characteristics and capabilities. Among these are phrase-based searching, synonyms, words to exclude from an index ("stopwords"), and so on.

You specify the text analysis properties dictionary for an index using the *`inAnalysisProperties`* parameter in either the `SKIndexCreateWithURL` or `SKIndexCreateWithMutableData` function. The available keys for the dictionary are defined in the Text Analysis Keys constants, described in *Search Kit Reference*.

You must ensure that the set of attributes you confer on an index makes sense. For example, because Vector indexes do not support phrase-based searching, do not use a `kCFBooleanTrue` value for the `kSKProximityIndexing` key in a Vector index's text analysis properties dictionary.

## Designing Index Architecture

To design the index architecture for your application, begin by answering these questions:

- Which documents should be in a given index?
- Should the indexes be persistent or memory based?
- Which sorts of queries should the indexes support?
- Do you want to filter the content as it is added to the index (using stopwords or a minimum term length)?
- Do you need a list of term substitutions?

Your answers to these questions will guide your choice of index type and text analysis properties.

### Flushing and Compacting Indexes

When your application adds or removes a document URL object from an index, the on-disk or memory-based representation of the index becomes stale. A search on an index in such a state won't have access to the nonflushed updates. The solution is to call the `SKIndexFlush` function before searching. `SKIndexFlush` flushes index-update information and commits memory-based index caches to disk, in the case of an on-disk index, or to a memory object, in the case of a memory-based index. In both cases, calling this function makes the state of an index consistent.

Indexes can develop **fragmentation** (that is, they can become bloated with unused text) as your application adds and removes document URL objects. Search Kit **compacts** an index, if needed, when your application calls the `SKIndexCompact` function. Because this function typically takes significant time to do its work, call it only when you find that an index is significantly fragmented.

To check for bloat you can take advantage of the way Search Kit allocates document IDs. It does so starting at 1 and without reusing previously allocated IDs for an index. Simply compare the highest document ID, found with the `SKIndexGetMaximumDocumentID` function, with the current document count, found with the `SKIndexGetDocumentCount` function.

## How Search Kit Performs Searches

Once you have a searchable corpus in the form of populated indexes, you're ready to search. Searching is a multistep process:

1. Get the user's query, including, if applicable, their specification of where to search.
2. Create (or reuse) an appropriate index group.
3. Update the indexes to query, as necessary.
4. Invoke the search.
5. Display results based on information from the returned search object, and continue to update the results as appropriate by continuing to query the search object.

This section describes working with indexes and focuses on the general case of searching multiple indexes. It also describes the various types of query available in Search Kit and briefly describes working with search results.

### Searching Multiple Indexes

In many cases, applications need to invoke a search over multiple indexes. For example, say a car buyer wants to learn about cars from America, Germany, and Japan. Your application might manage its automobile data by using a separate index for each country of manufacture. In this case, you'd include indexes from the specified countries in the user's search.

As a slightly more complex example, say you have a large set of static webpages (not generated on demand from a database) composing an online catalog. Each page lists several products. Each product, in turn, might be a member of one or more product categories—sports equipment, home and garden, sale items, and so on. Say you'd like your users to be able to search by product name, price, and category.

One strategy for providing search on such a website would be to define and appropriately tag each individual product description as a document. Recall that, when using Search Kit, a "document" is anything your application defines it to be. You would include tagging within each product description indicating product name, price, and a list of categories the product belongs to.

You could then create a separate index across the website for each such field of information, along with an index of all the visible text content. A user's search could specify which fields to search on, and your application would add the corresponding indexes to the index group used for the search.

Search Kit's asynchronous architecture lets you search multiple indexes in sequence or in parallel, depending on your application's needs and on the structure of your information.

- If your application is searching multiple indexes in a single file or in separate files but all on the same physical disk, Apple recommends that you search the indexes in sequence for best performance.
- When an index group is distributed across multiple disks or across a network, or when the indexes are all in memory, search the indexes in parallel.

For a parallel search, your application can use a separate thread for each index to be searched. Alternatively, you can create a search object for each index in a group, then repeatedly query the search objects in turn by making use of the timeout option in the `SKSearchFindMatches` function.

## Queries

Search Kit responds to a query by interpreting the query's terms, the explicit and implicit operators, and the order of the query's terms and operators. Using an enhanced, Google-like syntax, Search Kit supports a variety of types of query as well as arbitrary combinations of these types. For example, the following query includes Boolean, prefix, and suffix searching:

```
appl* OR *ing
```

Using the asterisk (`*`) wildcard operator and the Boolean `OR` operator, this query returns documents containing words that begin with “appl” as well as documents that contain words that end with “ing”.

**Note:** In versions of Mac OS X prior to version 10.4, Tiger, Search Kit used the now-deprecated `SKSearchResultsCreateWithQuery` function. That function required explicit setting of search type by way of a search type parameter. Refer to *Search Kit Reference* for more information.

The following table lists the operators for nonsimilarity searches. (Similarity searches do not respond to operators.) Synonyms, separated by commas here, all have the same order of evaluation.

**Table 2-1** Query operators in Search Kit, from highest to lowest precedence

| Operator† | Meaning |
|---|---|
| " | Opening and closing delimiter for phrase-based searching. |
| `(`, `)` | Opening and closing delimiters for logical grouping. |
| `!`, `NOT` | Boolean `NOT`. |

| Operator† | Meaning |
|---|---|
| `&`, `AND`, `<space>` | Boolean `AND`. The `<space>` character represents a Boolean operator when there are terms to both sides of the `<space>` character. In this case, `<space>` represents a Boolean `AND` by default, or a Boolean `OR` if specified by `kSKSearchOptionSpaceMeansOR`. |
| \| | Boolean inclusive `OR`. |
| `*` | Wildcard for prefix or suffix; surround term with wildcard characters for substring search. Ignored in phrase-based searches. |

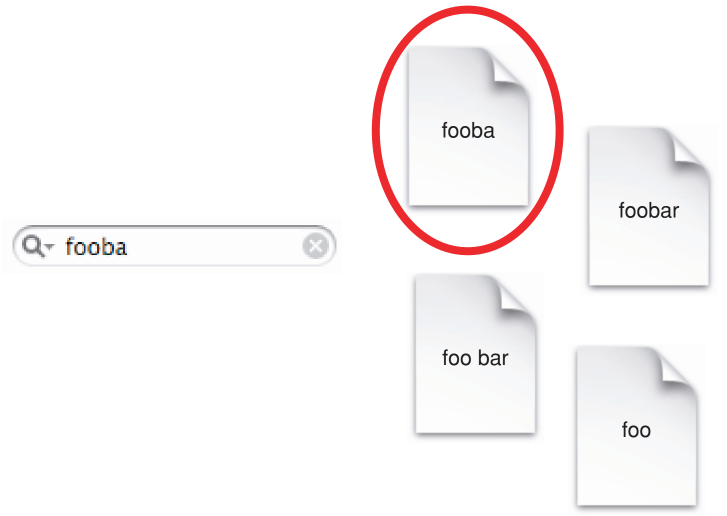
A Search Kit query can be as complex as you want, combining all the various operator types. For the purposes of explanation, this section discusses each query type separately.

The simplest sort of query consists of:

- One or more terms
- No operators other than the `<space>` character between the terms
- The default, `AND`-based behavior for <space>

Such a **simple search** looks for documents in the targeted set of indexes that contain all of the terms entered in the query string. Indexed terms match the query only if they match exactly. For example, if you search for "fooba" and a document contains "foobar" but not "fooba," you don't get a hit. Figure 2-5 depicts the behavior of a simple search.

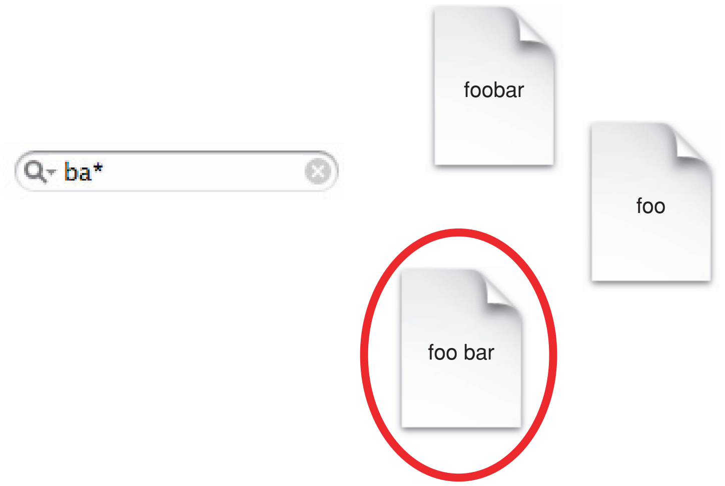
**Figure 2-5** A simple search



If you specify the "space means `OR`" option using the `kSKSearchOptionSpaceMeansOR` constant in the `SKSearchCreate` function, Search Kit finds not only documents that contain all of the query terms. It also finds documents that contain some, but not all, of the query terms; it ranks such documents lower than documents that contain all the query terms.

A **prefix search** looks for documents represented in the targeted set of indexes that contain terms beginning with the characters in a query. This is especially useful for the sort of live searching you see in Mail and Xcode. As the user types each successive character, the found set narrows. See Figure 2-6.

**Figure 2-6** A prefix search



Prefix searching looks at each term in the query separately, `AND`ing multiple terms by default, and matches on the beginnings of terms in the currently targeted set of indexes. In Figure 2-6, the word "bar" in the circled document is the only term that begins with the same characters as in the query, "ba."

To invoke a prefix search directly, a user would append the asterisk (*) character to the end of each term to be used as a prefix. An application can implicitly add the trailing asterisk to query terms before sending the query on to Search Kit.

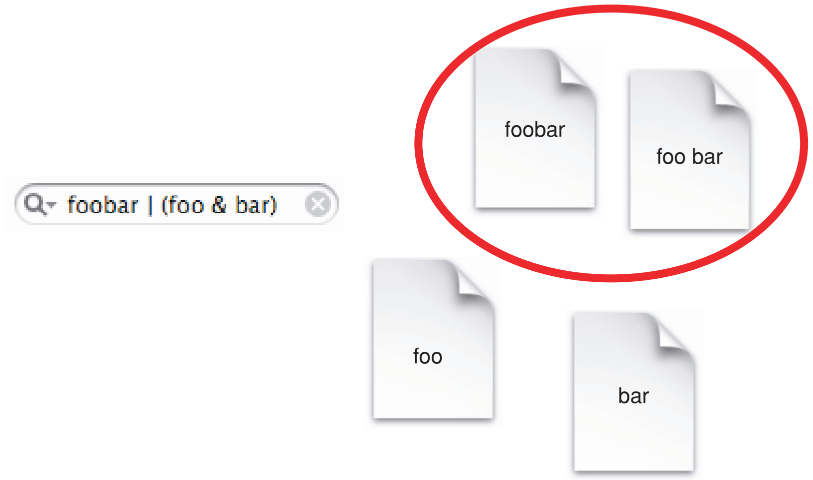
Here's another example of prefix searching. If you have mail messages from Billy Bob, Billy Joe, and Big Chief, you could enter "`Bo*`" to find all of Billy Bob's messages, or "`Jo*`" to find Billy Joe's messages. Entering "`Bi*`" would match messages from all three friends. "`Bi* OR Ch*`" would match Big Chief with the highest relevance but would also match Billy Bob and Billy Joe because they each contain one term matched by one of the query terms. The query "`illy*`" wouldn't match any of the mail messages.

Very similar to a prefix query is a **suffix search**. Suffix searching looks at each term in the query separately, `AND`ing multiple terms by default, and matches on the *ends* of terms in the currently targeted set of indexes. Again the wildcard character is the asterisk, but placed before the term as in "`*illy`". And again you can design your application so that users explicitly type the asterisk before terms, or you can add the asterisk implicitly before handing the query off to Search Kit.

A **Boolean search** offers full Boolean search functionality using the operators described in Table 2-1. You can design your application's interface so that users type the operators directly, or you can provide an alternate interface—converting the query to use the Search Kit standard operator syntax before handing off the query to the `SKSearchCreate` function.

The Boolean search in Figure 2-7 employs grouping operators as well as Boolean operators.

**Figure 2-7** A Boolean search



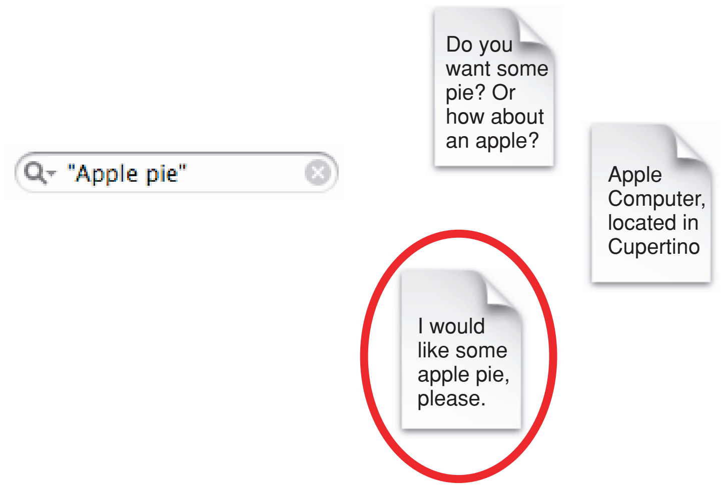
In the figure, the query in the search field indicates a request for documents that contain the exact term "foobar" as well as documents that contain both the term "foo" and the term "bar".

A **phrase search** works in inverted (and inverted-vector) indexes that were created with a true value for the `kSKPromimityIndexing` key in the text analysis properties dictionary. Such an index stores the position of each term in each document, along with the information otherwise stored in an inverted (or an inverted-vector) index.

Despite the name of the `kSKPromimityIndexing` text analysis properties key, Search Kit does not currently support arbitrary proximity searching. That is, you cannot search for documents in which two words are near each other but not adjacent. Search Kit supports only phrase searching.

When a user enters "Apple pie" as a query—including the surrounding quotation marks—Search Kit tries to find documents containing this exact phrase. See Figure 2-8 (page 33) for an illustration of phrase searching.

**Figure 2-8** A phrase-based search


"Apple pie"
Do you want some pie? Or how about an apple?
Apple Computer, located in Cupertino
I would like some apple pie, please.

## Search Results

Immediately upon invocation of a query—that is, upon creation of a search object—Search Kit asynchronous searching accumulates results into the search object. Using the `SKSearchFindMatches` function, your application retrieves results from the search object as they come in.

A search does not return documents per se. It returns document IDs. Your application, in turn, uses the document IDs to get document URL objects from the indicated indexes. The document URL objects, in turn, refer to the documents that satisfy the query.

The Search Kit framework does not provide display functionality. Instead, your application uses other Mac OS X frameworks to present the basic result data in the way you determine to be most useful. Tables are a popular way to display results, but you could just as well present search hits in outline form, as a graph, or as audible feedback.

# Search Kit Tasks

This chapter provides instructions and code samples for common Search Kit tasks including creating and using indexes of various types, performing searches, and displaying results.

The code samples assume you are using Xcode and they illustrate how to use Search Kit's ANSI-C API within Objective-C methods.

## Using Indexes and Documents

To index a document your application first creates an empty index and then adds document content to it. Search Kit indexes have a variety of characteristics you can set at the time of index creation. These characteristics determine such things as which types of searches users can perform.

Your application can create file-based or memory-based indexes. Use file-based indexes for information you want to retain between invocations of your program. Use memory-based indexes for temporary processes such as searching on a subset of a file-based index. Because memory-based indexes can be opened as read-only as well as read/write, you can use a memory-based index when your application needs an index that is open only for searching.

When you create an index, either file-based or memory-based, the index is automatically open. The Search Kit functions for opening indexes let your application re-open an index that you've closed, or open a previously-created file-based index.

### Creating a File-Based Index

To create a persistent, file-based index your application need only supply a file-system location, in the form of a CFURL object, and an appropriate constant specifying the type of index you want. If you're developing in Cocoa, you can use an NSURL object for the index location and cast it to a CFURLRef object as shown in Listing 3-1.

To create an index in memory, see "Creating an Index in Memory" (page 36). To specify text analysis properties for an index, see "Specifying Text Analysis Properties" (page 39).

**Listing 3-1** Creating a file-based index

```
- (void) newIndexInFile
{
    NSString * path = [pathTextField stringValue];          // 1
    NSURL * url = [NSURL fileURLWithPath: path];            // 2

    NSString * name = [nameTextField stringValue];          // 3
    if ([name length] == 0) name = nil;

    SKIndexType type = kSKIndexInverted;                    // 4
    skIndex = SKIndexCreateWithURL (                        // 5
        (CFURLRef) url,
        (CFStringRef) name,
        (SKIndexType) type,
        (CFDictionaryRef) NULL
    );
}
```

Here is how this code works:

1. Gets the file-system path for a new file-based index and converts it to an NSString object.
2. Converts the NSString object to an NSURL object.
3. Gets the optional index name, if provided, and saves it in an NSString object. If there's no index name, uses `nil`.
4. Defines the index type, in this case specifying an inverted index.
5. Creates the new, empty, file-based index using the specified URL, name, and type. Returns a Search Kit index object. In this simple example, the code creates an index without specifying a text analysis properties dictionary. See "Specifying Text Analysis Properties" (page 39) for more information on using a text analysis properties dictionary.

## Creating an Index in Memory

To create an index in memory, your application supplies an NSMutableData object (or, equivalently, a CFMutableData object) to hold the index.

To create an index in a file, see "Creating a File-Based Index" (page 35). To specify text analysis properties for an index, see "Specifying Text Analysis Properties" (page 39).

**Listing 3-2** Creating an index in memory

```
- (void) newIndexInMemory
{
    NSString * name = [nameTextField stringValue];              // 1
    if ([name length] == 0) name = nil;

    SKIndexType type = kSKIndexInverted;                        // 2

    indexObject = [[NSMutableData dataWithCapacity: 2^16] retain];    // 3

    skIndex = SKIndexCreateWithMutableData (                    // 4
                 (CFMutableDataRef) indexObject,
                 (CFStringRef) name,
                 (SKIndexType) type,
                 (CFDictionaryRef) NULL
             );
}
```

Here is how this code works:

1. Gets the optional index name, if provided, and saves it in an NSString object. If there's no index name, uses `nil`.

2. Defines the index type, in this case specifying an inverted index.

3. Creates a mutable data object to hold the new memory-based index, specifying the capacity and retaining it.

4. Creates the new, empty, memory-based index using the supplied mutable data object, name, and type. Returns a Search Kit index object. In this simple example, the code creates an index without specifying a text analysis properties dictionary. See "Specifying Text Analysis Properties" (page 39) for more information on using a text analysis properties dictionary.

## Opening a File-Based Index For Searching or Updating

To work with an already-existing file-based index (one that existed before your application launched), or to work with one that you have explicitly closed, your application must first open it. Once open, a file-based Search Kit index can be searched or updated.

Memory-based indexes can be opened in a read-only mode as well as in a read/write mode. See "Opening a Memory-Based Index for Searching Only" (page 38) and "Opening a Memory-Based Index for Searching or Updating" (page 38).

**Listing 3-3** Opening a file-based index for searching or updating

```
- (void) openIndex
{
    NSString * path = [pathTextField stringValue];     // 1
    NSURL * url = [NSURL fileURLWithPath:path];        // 2

    NSString * name = [nameTextField stringValue];     // 3
    if ([name length] == 0) name = nil;

    // open the specified index
    skIndex = SKIndexOpenWithURL (                     // 4
                  (CFURLRef) url,
                  (CFStringRef) name,
                  true
              );
}
```

Here's how this code works:

1. Gets the file-system path for the existing file-based index and converts it to an NSString object.

2. Converts the path NSString object to an NSURL object.

3. Gets the index name, if provided, for the existing file-based index and saves it as an NSString object. If there's no index name, uses `nil`.

4. Opens the file-based index and returns a Search Kit index object.

## Opening a Memory-Based Index for Searching Only

To work with an already-existing memory-based index that your application has explicitly closed, your application must first open it.

This task describes how to open a memory-based index for searching only. To open a memory-based index for searching or updating, see "Opening a Memory-Based Index for Searching or Updating" (page 38).

**Listing 3-4** Opening a memory-based index for searching only

```
- (void) openIndexInMemoryReadOnly
{
    skIndex = SKIndexOpenWithData (
                 (CFDataRef) indexObject,
                 (CFStringRef) nil
             );
}
```

Here's how this code works:

Your application provides a previously-created mutable data object to the `SKIndexOpenWithData` function, which then returns a read-only Search Kit index object. The index name is optional and in this example is specified as `nil`. To open a memory-based index for searching and updating, see "Opening a Memory-Based Index for Searching or Updating" (page 38). For information on creating a mutable data object, see "Creating an Index in Memory" (page 36).

## Opening a Memory-Based Index for Searching or Updating

To work with an already-existing memory-based index that your application has explicitly closed, your application must first open it.

This task describes how to open a memory-based index for searching or updating. To open a memory-based index for searching only, see "Opening a Memory-Based Index for Searching Only" (page 38).

**Listing 3-5** Opening a memory-based index for searching or updating

```
- (void) openIndexInMemoryReadWrite
{
    skIndex = SKIndexOpenWithMutableData (
                 (CFMutableDataRef) indexObject,
                 (CFStringRef) nil
              );
}
```

Here's how this code works:

Your application provides a previously-created mutable data object to the `SKIndexOpenWithMutableData` function, which then returns a read/write Search Kit index object. The index name is optional and in this example is specified as `nil`. To open a memory-based index for searching only, see "Opening a Memory-Based Index for Searching Only" (page 38). For information on creating a mutable data object, see "Creating an Index in Memory" (page 36).

## Closing an Index

A Search Kit index, as a Core Foundation object, can be closed by passing the index object to the `CFRelease` function. Alternatively, your application can use the `SKIndexClose` function as shown here.

**Listing 3-6** Closing an index

```
-(void) closeIndex
{
    if (skIndex) {
        SKIndexClose (skIndex);
        skIndex = nil;
    }
}
```

## Specifying Text Analysis Properties

Your application specifies an index's text analysis properties at the time of index creation. This task illustrates setting some of the available properties including minimum term length, stopwords, and customized term characters while creating a new file-based index.

**Listing 3-7** Specifying text analysis properties

```
- (void) newIndexWithPropertiesInFile
{
    NSString * path = [pathTextField stringValue];                      // 1
    NSURL * url = [NSURL fileURLWithPath: path];                        // 2

    NSString * name = [nameTextField stringValue];                      // 3
    if ([name length] == 0) name = nil;

    SKIndexType type = kSKIndexInverted;                                // 4

    NSNumber * minTermLength = [NSNumber numberWithInt: (int) 3];      // 5

    NSSet * stopwords = [NSSet setWithObjects:                          // 6
                            @"all",
                            @"and",
                            @"its",
                            @"it's",
                            @"the",
                            nil
                        ];

        NSDictionary * properties =                                     // 7
        [NSDictionary dictionaryWithObjectsAndKeys:
            @"", @"kSKStartTermChars",   // additional starting-characters for
 terms
            @"-_@.'", @"kSKTermChars",   // additional characters within terms
            @"", @"kSKEndTermChars",     // additional ending-characters for
terms
            minTermLength, @"kSKMinTermLength",
            stopwords, @"kSKStopWords",
            nil
        ];

    skIndex = SKIndexCreateWithURL(                                  // 8
        (CFURLRef) url,
        (CFStringRef) name,
        (SKIndexType) type,
        (CFDictionaryRef) properties
    );
}
```

Here's how this code works:

1. Gets the file-system path for a new file-based index and converts it to an NSString object.
2. Converts the NSString object to an NSURL object.
3. Gets the optional index name, if provided, and saves it in an NSString object. If there's no index name, uses `nil`.
4. Defines the index type, in this case specifying an inverted index.
5. Specifies the minimum term length as a NSNumber object.
6. Specifies a list of stopwords as an NSSet object.
7. Specifies the text analysis properties dictionary, including minimum term length, stopwords, and additional term character specifications.
8. Creates the new, empty, file-based index using the specified URL, name, type, and text analysis properties. Returns a Search Kit index object.

## Loading The Spotlight Text Importers

Loading the Spotlight text importers is a single-step process that applications typically perform at launch time.

**Listing 3-8** Loading the Spotlight text importers

```
- (void) loadImporters
{
    SKLoadDefaultExtractorPlugIns ();
}
```

## Adding a File-Based Document to an Index

To add a file-based document to an index your application simply supplies the index object and a document URL object. You can optionally provide a MIME type hint, as described in *Search Kit Reference*. Finally, you specify the document as replaceable or not.

**Listing 3-9** Adding a file-based document to an index

```
- (void) addDoc
```

```
{
    NSString * path = [filePathTextField stringValue];    // 1
    NSURL * url = [NSURL fileURLWithPath: path];          // 2
    SKDocumentRef doc = SKDocumentCreateWithURL (         // 3
                            (CFURLRef) url
                        );
    [(id) doc autorelease];                               // 4

    Boolean added = SKIndexAddDocument (                  // 5
        (SKIndexRef) skIndex,
        (SKDocumentRef) doc,
        (CFStringRef) NULL,       // optional MIME type hint
        (Boolean) true            // replaceable
    );
}
```

How this code works:

1. Gets the file-system path for the document to be added to an index. Converts the path string to an NSString object.

2. Converts the NSString object to an NSURL object.

3. Creates a document URL object from the NSURL object.

4. Specifies that the document URL object should be auto-released.

5. Adds the document identified by the document URL object, along with its text, to the specified Search Kit index.

## Adding a Folder of File-Based Documents to an Index

To index a folder of documents, your application makes use of other Carbon or Cocoa frameworks. This task illustrates how to do this in Cocoa, descending recursively into a folder structure.

**Listing 3-10** Adding a folder of file-based documents to an index

```
- (void) addDocsInFolder
{
    NSString * path = [filePathTextField stringValue];    // 1

    NSDirectoryEnumerator * dirEnumerator =               // 2
        [[NSFileManager defaultManager] enumeratorAtPath: path];

    NSString * item;
    while ((item = [dirEnumerator nextObject]) != nil) {  // 3
        NSString * fullPath =                             // 4
            [path stringByAppendingPathComponent: item];
        if (!fullPath) continue;

        NSURL * url =                                     // 5
            [NSURL fileURLWithPath: fullPath];
        if (!url) continue;

        SKDocumentRef doc = SKDocumentCreateWithURL (     // 6
                                (CFURLRef) url
                            );
        if (!doc) continue;
        [(id) doc autorelease];                         // 7

        Boolean added = SKIndexAddDocument (            // 8
            (SKIndexRef) skIndex,
            (SKDocumentRef) doc,
            (CFStringRef) NULL,   // optional MIME type hint
            (Boolean) true        // replaceable
        );
    }
}
```

Here's how this code works:

1. Gets the file-system path for a folder containing file-based documents. Converts the path string to an NSString object.
2. Creates an NSDirectoryEnumerator object for recursively finding all the file-based documents within the folder.
3. Steps through the enumerator, getting the path to each file-based document in the folder.
4. Converts each path to an NSString object.
5. Converts each NSString object to an NSURL object.
6. Creates a document URL object from each NSURL object.
7. Specifies that the document URL object should be auto-released.
8. Adds the document identified by the document URL object, along with its text, to the specified Search Kit index.

## Adding Text Explicitly to an Index

To add text explicitly to an index, your application gets the text, parses it as necessary, and then hands it off along with an index object to Search Kit. This example illustrates indexing text from a text field in the user interface of an application. You could just as well take text from a database record or from a remote URL, for example.

**Listing 3-11** Adding text explicitly to an index

```
- (void) addDocWithText
{
    NSString * path = [filePathTextField stringValue];    // 1
    NSURL * url = [NSURL fileURLWithPath:path];           // 2

    SKDocumentRef doc = SKDocumentCreateWithURL (         // 3
                            (CFURLRef) url
                        );
    [(id) doc autorelease];                               // 4
    NSString * contents = [fileContentsTextView string];  // 5
    Boolean added = SKIndexAddDocumentWithText (          // 6
                        skIndex,
                        doc,
                        (CFStringRef) contents,
                        (Boolean) true     // replaceable
                    );
}
```

Here's how this code works:

1. Gets the file-system path for the document to be added to an index. Converts the path string to an NSString object.

2. Converts the NSString object to an NSURL object.

3. Creates a document URL object from the NSURL object.

4. Specifies that the document URL object should be auto-released.

5. Gets textual content from a user-interface text field and converts it to an NSString object.

6. Adds the document URL object along with the specified text to the specified Search Kit index. The text is associated in the index to the document URL object such that queries that match terms in the text will return the document URL object.

## Updating an Index When a Document Changes

To reindex a document that has changed, simply replace it in the index. Do this by calling the `SKIndexAddDocumentWithText` or `SKIndexAddDocument` function, as appropriate, with the *`inCanReplace`* parameter set to a `true` value. This example assumes that the document involved is a local, file-based document and so uses the `SKIndexAddDocument` function.

**Listing 3-12** Updating an index when a document changes

```
- (void) replaceChangedDoc: (id) sender
{
    NSString * path = [filePathTextField stringValue];   // 1
    NSURL * url = [NSURL fileURLWithPath:path];
    SKDocumentRef doc = SKDocumentCreateWithURL (
                            (CFURLRef) url
                        );
    [(id) doc autorelease];
    Boolean replaced = SKIndexAddDocument (              // 2
                        skIndex,
                        doc,
                        NULL,
                        true
                    );
}
```

1. Gets the file-system path for the replacement document. Converts the path string to an NSString object.]

2. Replaces the document identified by the document URL object, along with its text, in the specified Search Kit index.

## Updating an Index When a Document Moves or Moves and Changes

To reindex a document that has moved, or moved and changed, perform these three steps:

1. Remove the old document with the `SKIndexRemoveDocument` function.
2. Remove the old text (if desired, or if the index is significantly fragmented) with the `SKIndexCompact` function.
3. Add the changed document and its text to the index.

Compacting the index during this process removes any orphaned terms. However, the `SKIndexCompact` function can be expensive in terms of performance. Apple recommends that you do not call it every time a document is modified or deleted, but only when an index is significantly fragmented (bloated with unused text).

To check for bloat you can take advantage of the way Search Kit allocates document IDs. It does so starting at 1 and without reusing previously allocated IDs for an index. Simply compare the highest document ID, found with the `SKIndexGetMaximumDocumentID` function, with the current document count, found with the `SKIndexGetDocumentCount` function.

The following simple example illustrates compacting without checking for index bloat.

**Listing 3-13** Updating an index when a document moves or moves and changes

```
- (void) replaceDoc: (id) sender
{
//......................................................................
// remove a specified document from an index
    NSString * path = [filePathTextField stringValue];   // 1
    NSURL * url = [NSURL fileURLWithPath: path];         // 2

    SKDocumentRef doc = SKDocumentCreateWithURL (        // 3
                            (CFURLRef) url
                        );
    [(id) doc autorelease];                              // 4

    Boolean removed = SKIndexRemoveDocument (            // 5
                          skIndex,
                          doc
                      );
//......................................................................
// compact the index to remove the terms associated with the removed document
    SKIndexCompact (skIndex);                            // 6
//......................................................................
// add the document and its terms back to the index
    NSString * path = [filePathTextField stringValue];   // 7
    NSURL * url = [NSURL fileURLWithPath:path];
    SKDocumentRef doc = SKDocumentCreateWithURL (
                            (CFURLRef) url
```

```
                    );
    [(id) doc autorelease];
    Boolean added = SKIndexAddDocument (                // 8
                        skIndex,
                        doc,
                        NULL,
                        true
                    );
}
```

Here's how this code works:

1. Gets the file-system path for the document to be removed from an index. Converts the path string to an NSString object.
2. Converts the NSString object to an NSURL object.
3. Creates a document URL object from the NSURL object.
4. Specifies that the document URL object should be auto-released.
5. Removes the document URL object from the specified index.
6. Compacts the index to remove the terms associated with the removed document.
7. Gets the file-system path for the replacement document. Converts the path string to an NSString object.
8. Adds the replacement document identified by the document URL object, along with its text, to the specified Search Kit index.

## Searching

This section describes each of the sub-tasks important to query-based searching and then assembles them in the task titled "A Complete Search Method" (page 49). Briefly, the steps are:

- Specify the maximum number of hits to return; this is simply defining a constant.
- Set up the search options.
- Get the user's query.
- Create an asynchronous search object.
- Request hits from the search object.
- Convert the hits into document locations and displays the results.

Your application can use similarity-based searching instead of query-based searching by turning on the `kSKSearchOptionFindSimilar` flag when creating a search object. In this case, use the content of an example document, or a portion of an example document, as the query string.

## Setting Up Search Options

When your application creates a search object it can specify a variety of search options. Each option is simply a binary flag for the *inSearchOptions* parameter of the SKSearchCreate function.

**Listing 3-14** Setting up search options

```
SKSearchOptions options = kSKSearchOptionDefault;        // 1

if ([searchOptionNoRelevance intValue])                  // 2
    options |= kSKSearchOptionNoRelevanceScores;
if ([searchOptionSpaceIsOR intValue])                    // 3
    options |= kSKSearchOptionSpaceMeansOR;
if ([searchOptionSpaceFindSimilar intValue])             // 4
    options |= kSKSearchOptionFindSimilar;
```

Here's how this code works:

1. Specifies use of the default set of search options.
2. If the user has specified that searches should not consider relevance, adds that option.
3. If the user has specified that spaces should indicate a logical OR in a query, adds that option.
4. If the user has specified similarity searching instead of query-based searching, adds that option.

## Getting a User's Query

This task illustrates the simple step of creating an NSString object from a user's query.

**Listing 3-15** Getting a user's query

```
NSString * query = [searchField stringValue];
```

## Creating an Asynchronous Search Object

Your application creates a search object by supplying an index object, an NSString object representing the user's query, and a set of option flags that determine the searching behavior.

**Listing 3-16** Creating an asynchronous search object

```
SKSearchRef search = SKSearchCreate (            // 1
                         skIndex,
                         (CFStringRef) query,
                         options
                     );
[(id) search autorelease];                       // 2
```

Here's how this code works:

1. Creates a search object based on the user's query and the specified search options, targeting the specified Search Kit index.

2. Specifies that the search object should be auto-released.

## Getting Matches From a Search Object

To retrieve matches from an asynchronous search object, your application sets up arrays to hold the results and then requests the results. You can use a loop to keep requesting additional hits while there are more to be had, as illustrated in this task and in "A Complete Search Method" (page 49).

The result of a hit is a lightweight document identifier. The next task, "Getting Document Locations and Displaying Results" (page 48), illustrates converting the document identifiers to document locations as represented by document URL objects.

**Listing 3-17** Getting matches from a search object

```
while (more) {

    SKDocumentID    foundDocIDs [kSearchMax];          // 1
    float           foundScores [kSearchMax];          // 2
    float * scores;                                    // 3
    Boolean unranked =                                 // 4
        options & kSKSearchOptionNoRelevanceScores;

    if (unranked) {                                    // 5
        scores = NULL;
    } else {
        scores = foundScores;
    }

    CFIndex foundCount = 0;                            // 6
    more = SKSearchFindMatches (                       // 7
              search,
              kSearchMax,
              foundDocIDs,
              scores,
              1,    // maximum time before function returns, in seconds
              &foundCount
          );
    // display or accumulate results here

    totalCount += foundCount;                          // 8
}
```

Here's how this code works:

1. Sets up an array to hold document identifiers resulting from hits during the search.

2. Sets up an array to hold relevance scores.

3. Sets up a pointer to the relevance scores array.

4. Creates a Boolean flag specifying whether or not relevance scores should be reported.

5. Uses the relevance flag to define the *outScoresArray* parameter for the SKSearchFindMatches function.
6. Initializes the found count to 0.
7. Queries the search object. The SKSearchFindMatches function places hits in the *foundDocIDs* array, relevance scores in the *scores* array, and a Boolean value indicating whether there are more results to be had (TRUE) or not (FALSE).
8. Accumulates the number of new matches into the total number of matches.

## Getting Document Locations and Displaying Results

To obtain results that are meaningful to a user, you convert each of the document identifiers, provided by the search object as hits, to a document location in the form of a document URL object. This task illustrates a simple logging of results—including document ID, document URL, and relevance score if applicable—to a text field.

In this code excerpt, and in the complete example shown in "A Complete Search Method" (page 49), the code simply displays results as they're found. Using other Carbon or Cocoa frameworks, you may want to, instead, accumulate results and present them in a single list sorted by relevance and perhaps dynamically updated.

**Listing 3-18** Getting document locations and displaying results

```
SKDocumentRef   foundDocRefs [kSearchMax];          // 1
SKIndexCopyDocumentRefsForDocumentIDs (             // 2
    skIndex,
    (CFIndex) foundCount,
    foundDocIDs,
    foundDocRefs
);

CFIndex pos;
for (pos = 0; pos < foundCount; pos++) {            // 3
    SKDocumentRef doc =                             // 4
        (SKDocumentRef) [(id) foundDocRefs [pos] autorelease];
    NSURL * url =                                   // 5
        [(id) SKDocumentCopyURL (doc) autorelease];
    NSString * urlStr = [url absoluteString];       // 6

    NSString * desc;
    if (unranked) {                                 // 7
        desc = [NSString stringWithFormat:
            @"---\nDocID: %d,
            URL: %@",
            (int) foundDocIDs [pos],
            urlStr];
    } else {
        desc = [NSString stringWithFormat:
            @"---\nDocID: %d,
            Score: %f,
            URL: %@",
            (int) foundDocIDs[ pos],
            foundScores [pos],
            urlStr];
    }
    [self log: desc];                               // 8
```

Here's how this code works:

1. Sets up an array to hold document URL objects derived from the document identifiers collected in "Getting Matches From a Search Object" (page 47).
2. Converts the document identifiers to document URL objects. The found count and the document identifiers are from "Getting Matches From a Search Object" (page 47).
3. Iterates through the `foundDocRefs` array to convert document URL objects to NSString objects for display.
4. Gets the next document URL object.
5. For the current document URL object, gets the URL.
6. Converts the URL to an NSString object.
7. Formats the result information for the current hit. The information to be displayed depends on whether the search was ranked or unranked.
8. Displays the results using the applications `log:` method (not illustrated here).

## A Complete Search Method

This task collects the preceding code excerpts in this section and presents a complete search method. As described in the introduction to this section, your application:

- Specifies the maximum number of hits to return.
- Sets up the search options.
- Gets the user's query.
- Creates an asynchronous search object.
- Requests hits from the search object.
- Converts the hits into document locations and displays the results.

For descriptions of how this code works, refer to the preceding subtasks in this section.

**Listing 3-19** A complete search method

```
//.........................................................................
// specify the maximum number of hits
#define kSearchMax 1000

- (void) search
{
//.........................................................................
// set up search options
```

```
    SKSearchOptions options = kSKSearchOptionDefault;

    if ([searchOptionNoRelevance intValue]) options |=
kSKSearchOptionNoRelevanceScores;
   if ([searchOptionSpaceIsOR intValue]) options |= kSKSearchOptionSpaceMeansOR;
    if ([searchOptionSpaceFindSimilar intValue]) options |=
kSKSearchOptionFindSimilar;

//....................................................................
// get the user's query

    NSString * query = [searchField stringValue];

//....................................................................
// create an asynchronous search object

    SKSearchRef search = SKSearchCreate (
                             skIndex,
                             (CFStringRef) query,
                             options
                         );
    [(id) search autorelease];

//....................................................................
// get matches from a search object

    Boolean more = true;
    UInt32 totalCount = 0;

    while (more) {

        SKDocumentID    foundDocIDs [kSearchMax];
        float            foundScores [kSearchMax];
        SKDocumentRef    foundDocRefs [kSearchMax];

        float * scores;
        Boolean unranked = options & kSKSearchOptionNoRelevanceScores;

        if (unranked) {
            scores = NULL;
        } else {
            scores = foundScores;
        }

        CFIndex foundCount = 0;
        CFIndex pos;

        more =    SKSearchFindMatches (
                    search,
                    kSearchMax,
                    foundDocIDs,
                    scores,
                    1, // maximum time before func returns, in seconds
                    &foundCount
                );

        totalCount += foundCount;
//......................................................................
// get document locations for matches and display results.
//     alternatively, you can collect results over iterations of this loop
//     for display later.

        SKIndexCopyDocumentRefsForDocumentIDs (
            (SKIndexRef) skIndex,
            (CFIndex) foundCount,
            (SKDocumentID *) foundDocIDs,
            (SKDocumentRef *) foundDocRefs
        );

        for (pos = 0; pos < foundCount; pos++) {
            SKDocumentRef doc = (SKDocumentRef) [(id) foundDocRefs [pos]
autorelease];
            NSURL * url = [(id) SKDocumentCopyURL (doc) autorelease];
            NSString * urlStr = [url absoluteString];

            NSString * desc;

            if (unranked) {
                desc = [NSString stringWithFormat: @"---\nDocID: %d, URL: %@",
 (int) foundDocIDs [pos], urlStr];
            } else {
               desc = [NSString stringWithFormat: @"---\nDocID: %d, Score: %f,
 URL: %@", (int) foundDocIDs[ pos], foundScores [pos], urlStr];
            }
            [self log: desc];
        }
    }

   NSString * desc = [NSString stringWithFormat: @"\"%@\" - %d matches", query,
 (int) totalCount];
    [self log: desc];
}
```

For descriptions of how this code works, refer to the preceding subtasks in this section.

## Using Timeout to Search an Index Group in Parallel

To search an index group in parallel you can use a separate thread for querying each search object. Alternatively, as illustrated here, you can repeatedly rotate through a set of search objects, querying one and then moving on to the next, by using the timeout option in the `SKSearchFindMatches` function.

**Listing 3-20** Using timeout to search an index group in parallel

```
completeCount = indexCount;                   // 1
while (completeCount) {                       // 2
    for (i = 0; i < indexCount; i++) {
        if (more [i]) {
            more [i] = SKSearchFindMatches (  // 3
                           searchObjects [i],
                           kSearchMax,
                           foundDocIDs,
                           scores,
                            timeout,
                            &foundCount
                        );

            if (!more [i]) completeCount--;   // 4
            ProcessHits (                     // 5
                searchObjects [i],
                foundDocIDs,
                scores,
                foundCount
            );
        }
    }
}
```

Here's how this code works:

1. Initializes the `completeCount` variable to the number of indexes in the group. An application using this code would have previously defined one search object per index in the group. It also would have initialized the `more` array with `true` values for each element. The `completeCount` variable holds the diminishing number of search objects that still have results available.
2. Iterates through the list of search objects, getting hits from each in turn. After the period specified by the *`timeout`* parameter, moves on to the next search object. As long as hits from at least one search object are not exhausted, repeats the iteration.
3. Gets the next set of search hits for the current search object, accumulating the results in `foundDocIDs`, `scores`, and `foundCount`.
4. If no new search hits were found for the current search object, decrements the number of active search objects.
5. Calls the application-defined `ProcessHits` function to work with the new search hits.

# Document Revision History

This table describes the changes to *Search Kit Programming Guide*.

| Date | Notes |
|---|---|
| 2005-12-06 | Major update for Mac OS X v10.4. Added "Search Kit Tasks" (page 35) chapter. Renamed document from *Adding Search to Your Application*. |
| 2004-06-28 | Made minor corrections to figures. |
| 2004-05-20 | Corrected some typographical errors. Changed descriptions of the `SKDocumentRef` opaque data type from "document reference" to "document URL object." |
| 2004-04-22 | First publication of "Adding Search to Your Application." |

# Glossary

**AIAT** See Apple Information Access Toolkit (AIAT).

**Apple Information Access Toolkit (AIAT)** In Classic Mac OS, an object-oriented information access engine that contained a collection of tools for indexing, searching, and analyzing large volumes of documents. Search Kit is the Mac OS X implementation of the AIAT. AIAT was formerly known by its code name V-Twin.

**Boolean searching** Matching of a query string to indexed terms using Boolean (logical) operators such as `AND` and `OR` between query terms, optionally employing grouping for precedence using parentheses. The entire query expression is matched. See also search.

**compact** To make an index smaller by removing unused bits. Over time, as documents get added to and removed from an index, the index's disk or memory footprint may grow due to fragmentation. Search Kit includes APIs to check for fragmentation and to compact an index. See also fragmentation .

**corpora** Plural form of corpus.

**corpus** A collection of one or more documents, typically related, and available to an information retrieval system. Plural: corpora.

**document** In general, a specifically locatable information object of useful granularity and arbitrary structure. In Search Kit, anything that contains text and that the Search Kit client application addresses as a document—an RTF document, a PDF file, a Mail message, an Address Book entry, the contents at an Internet URL, the result of a database query, and so on. See also document URL object.

**document collection** See corpus.

**document object hierarchy** A collection of documents in which each document exists at a location relative to a root document. The locations may may be real, as in a file system, or virtual.

**document URL object** A URL to a document. In Search Kit, a document URL object comprises a scheme, a parent document URL object, and a name, with the format of each component defined by the client application. Search Kit document URL objects may be converted to or from CFURL objects. See also document, parent document URL object, scheme.

**fragmentation** In Search Kit, an unwanted increase in index size due to accumulation of unused capacity. Over time, as documents get added to and removed from an index, the index may become fragmented—its constituent documents and terms may become arranged in a manner that includes a significant amount of unused disk or memory space. See also compact.

**inclusion/exclusion result** See inclusion/exclusion searching.

**inclusion/exclusion searching** Unranked searching where the result simply includes documents that match the query and excludes documents that don't. Inclusion/exclusion searches tend to be faster than ranked searches. Search Kit supports inclusion/exclusion searches. See also relevance-based result.

**index** A memory- or file-based sequential collection of the terms in one or more documents. In addition to terms, Search Kit indexes contain context information that specifies which documents each term belongs to, along with term and document metadata useful during display of search results. Search Kit performs its searching and analysis on indexes. See also inverted index; inverted-vector index; vector index

**index group** A short-lived collection of one or more indexes; the target of a search. An index group corresponds to one or more aspects of the corpus of documents you want to search. For example, one index in a group might contain document titles, while another contains the body text of those same documents. An index group can also comprise indexes of multiple corpora. See also corpus; document.

**information retrieval (IR)** The process of locating information based on a well-defined information need. An information retrieval system consists of a corpus, one or more indexes of its content, a query interface, a search system, and a results interface. See also corpus; search.

**inverted index** An index containing terms, as keys, mapped to references to the documents they appear in. The index is sorted by its keys. "Inverted" means that the documents are found by matching on terms, rather than the other way around. See also index; inverted-vector index; vector index

**inverted-vector index** An index containing terms mapped to document URL objects representing the documents that the terms appear in, as well as document URL objects mapped to the terms that each document contains. See also index; inverted index; vector index.

**IR** See information retrieval (IR).

**MIME type hint** Advisory metainformation suggesting the likely content type for a URL. MIME is an acronym for Multipurpose Internet Mail Extensions. In Search Kit, common MIME type hints include `text/plain`, `text/rtf`, `text/html`, `text/pdf`, and `application/msword`.

**minimum term frequency** The fewest number of times a term can appear in a document and still be indexed. This functionality is not currently supported by Search Kit indexes.

**minimum term length** The shortest-length term to index. When Search Kit adds terms from a document to an index, it skips over words whose length is shorter than the minimum term length.

**name** In Search Kit, a document name as represented in a document URL object. For documents that are on-disk files, the name should correspond to the actual filename. For other types of documents, your application can assign any name to a document. See also document URL object

**operator** A character or word that has a special meaning when used in a query. Operators in Search Kit include `AND`, `OR`, `NOT`, parentheses, quoation marks, and several others. Search Kit interprets operators and determines the user's intended search type according to the operators' meanings.

**parent document URL object** In Search Kit, for file-based documents, the location of the enclosing folder for a document or for another parent document URL object. Search Kit manages documents using parent-child relationships, not paths. You can construct the path of any document by following its parent document links. See also document URL object.

**partial string searching** Matching of the terms in a query string to indexed terms, with implied wildcard characters at the start and end of each query term. Each term is matched separately. Search Kit does not currently support partial string searching as an option, but a client application can provide it by adding wildcard operators (asterisks) around each term before handing a query off to Search Kit. See also search.

**phrase searching** Matching of a query string to indexed terms, with the query string considered as a complete phrase. A match occurs when the exact query phrase appears in a document. Search Kit supports phrase searching in inverted and inverted-vector indexes. See also search.

**prefix searching** A specialized type of substring search. A prefix search involves matching of a term in a query string to indexed terms, with an explicit wildcard character at the end of the query term. A match occurs when the characters in the query term (minus the wildcard character) match the beginning of an indexed term. For example, the query string `car*` will match `car`, `carpet`, and `carnivore`. Search Kit supports prefix searching in inverted and inverted-vector indexes. See also search; substring searching; wildcard character.

**query** (n.) A text string, containing terms and operators, that represents a user's information retrieval request. Various types of query supported by Search Kit include simple, prefix/suffix/substring, Boolean, phrase, and similarity. (v.) To invoke a request for information in an information retrieval system. See also search.

**ranked searching** See relevance-based result.

**relevance-based result** See also relevance-based search.

**relevance-based search** A ranked search whose result includes a relevance rating for each document matching a query. In general, relevance ratings may be normalized to 100%, or nonnormalized. Search Kit supports only nonnormalized results. See also inclusion/exclusion searching; search.

**root word** See stem.

**scheme** A way to access a file-system or Internet resource, corresponding to an access protocol. Examples include `http`, `ftp`, and `file`. See also document URL object.

**search** In an information retrieval system, a process that attempts to locate documents that match a query, and that may assign relevance scores to the found documents. Upon a successful match, a search system returns references to the found documents. Search Kit supports a variety of search types, some of which can be combined. These types are simple, Boolean, ranked, unranked, phrase, similarity, prefix, suffix, and substring.

**search object** In Search Kit, an opaque data type representing an asynchronous search and containing its results, accumulated as they are found. A search object is of type `SKSearchRef`.

**similarity searching** Matching of a query string, typically consisting of a representative portion of a document, to indexed documents. A match occurs when Search Kit determines significant content similarity between the query and an indexed document. Search Kit supports similarity searching in vector and inverted-vector indexes. Similarity searching also works in inverted indexes in Search Kit, but performance is worse. See also search.

**simple search** Matching of the terms in a query string to indexed terms using exact, character-for-character matching. Each term is matched separately. In Search Kit, by default, spaces between terms behave like Boolean `AND` operators. See also search.

**stem** The root of a family of morphological or inflectional variants of a word. For example, "swim" is the stem of "swimmer," "swimming," and "swam."

**stemming** The algorithm-based removal of morphological and inflectional word components, typically endings. Language dependent. Stemming is sometimes referred to as suffix stripping, although some stemming algorithms perform prefix stripping as well. IR systems use stemming to improve search quality and to reduce index size. Search Kit does not support stemming; if needed, client applications implement it. Some stemming algorithms handle only regular variants, such as converting "swimming" to "swim," and do not handle irregular variants, such as converting "swam" to "swim."

**stopword** A word not to index. When Search Kit adds terms from a document to an index, it skips over words in its top-word list.

**substring searching** Matching of a term in a query string to indexed terms, with explicit wildcard characters at the start and end of the query term. A match occurs when the characters in the query term (minus the wildcard characters) match the beginning, ending, or middle of an indexed term. For example, the query string `*cat*` will match `cat`, `concatenate`, `tomcat`, and `cattle`. Search Kit supports substring searching in inverted and inverted-vector indexes. See also search.

**suffix searching** A specialized type of substring search. A suffix search involves matching of a term in a query string to indexed terms, with an explicit wildcard character at the start of the query term. A match occurs when the characters in the query term (minus the wildcard character) match the ending of an indexed term. For example, the query string `*ion` will match `ion`, `lion`, and `version`. Search Kit supports suffix searching in inverted and inverted-vector indexes. See also search; wildcard character.

**suffix stripping** See stemming.

**summarization object** In Search Kit, an opaque data type representing summarization information, including the summary text. A summarization object is of type `SKSummaryRef`.

**synonym** A term that an IR system considers to be equivalent to another term for both indexing and querying. For example, an IR system could define "car," "passenger vehicle," and "automobile" to be synonyms. See also information retrieval (IR); index; query.

**term** An atomic entry in a Search Kit index, typically corresponding to a word found in one of the index's documents.

**text extraction** Selective copying of terms from one or more documents into an index. See also stemming; stopword.

**unranked searching** See inclusion/exclusion searching.

**URL** Uniform Resource Locator. An Internet address, or a file-system path when formatted as a URL with a scheme. See also scheme.

**V-Twin** See Apple Information Access Toolkit (AIAT).

**vector index** An index containing document URL objects, as keys, mapped to the terms that each document contains. See also index; inverted index; inverted-vector index

**wildcard character** An operator used in a query that indicates matching on any character. In Search Kit, the wildcard character is the asterisk. Depending on usage, the wildcard character can indicate prefix, suffix, or substring searching. See also operator; query.