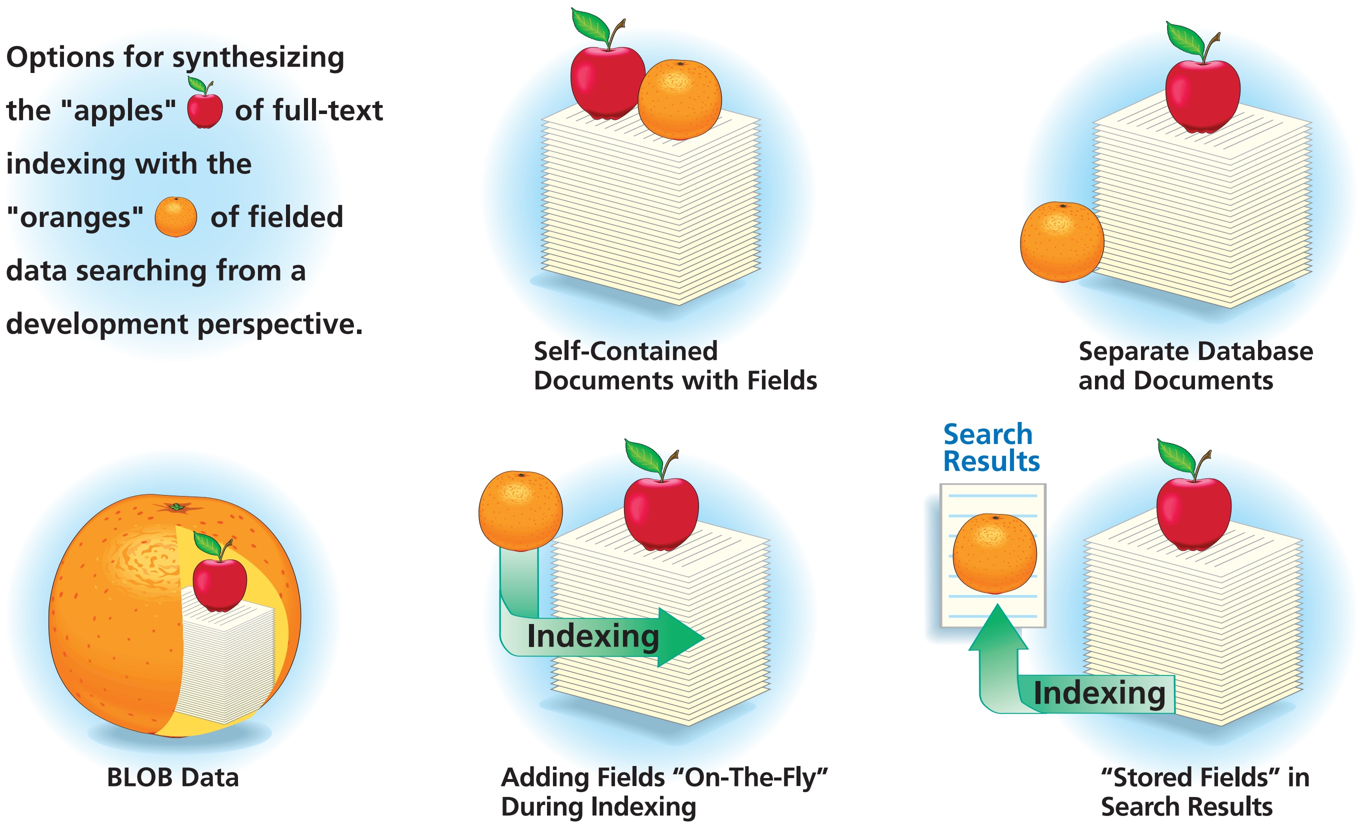
Crossing the Full-Text Search / Fielded Data Divide
from a Development Perspective

This article discusses methods for synthesizing the “apples” of full-text searching with the “oranges” of fielded data, using the dtSearch® Text Retrieval Engine as an example.

Where individual PCs can store gigabytes of data, and enterprise Intranets and public sites terabytes of data, finding the correct document (or Web page) requires a complete arsenal of full-text indexed and fielded data search tools. While this combination makes sense for the end-user, from a development perspective, these two approaches to data are very different — the equivalent of “apples and oranges.” This article discusses methods for synthesizing the “apples” of full-text searching with the “oranges” of fielded data, using the dtSearch® Text Retrieval Engine as an example.

Apples and Oranges: Full-Text and Fielded Data Searching

A full-text search index is the democracy of data structures, holding every word in a document collection, along with its location in the document. A full-text indexed search can instantly retrieve a reference to the *rarest of rare birds*, and display this reference as a highlighted hit in the document. The retrieval process in a full-



Options for synthesizing the "apples" of full-text indexing with the "oranges" of fielded data searching from a development perspective.

Self-Contained Documents with Fields
Separate Database and Documents
BLOB Data
Indexing
Adding Fields “On-The-Fly” During Indexing
Search Results
Indexing
“Stored Fields” in Search Results

Reprinted with permission of PC AI Online Magazine V. 16 #5
For more information about PC AI Online Magazine, visit www.pcai.com.

To take full advantage of XML as a hierarchical data structure, dtSearch supports nested field searching.

text indexed search is independent of whether the match appears in a document title, or a single footnote buried in a collection of Web-based files.

By contrast, information in a fielded database structure is more hierarchical than democratic. In an XML database, for instance, the *rarest of rare birds* might reside in a category tree many levels deep. In an SQL database, the *rarest of rare birds* might reside in a multi-level table matrix.

Precision searching of these databases means not only finding the *rarest of rare birds* anywhere in the database, but also finding the phrase (and highlighting the match) only when it appears in a highly specific field structure. In an XML database, the *rarest of rare birds* might appear in two separate North American and South American branches of a tree structure. Precision searching could find the *rarest of rare birds* in the North American tree branch, and not, for example, in the South American tree branch.

In summary, while precision full-text searching treats all hits as equal, precision fielded data searching of a structured database must make distinctions among the various places in the database where a hit appears. This leads to two questions: from a searching perspective, why is it important to bridge these two very different approaches to data retrieval; and from a development perspective, how is it possible to bridge them efficiently.

Limits of Full-Text Searching

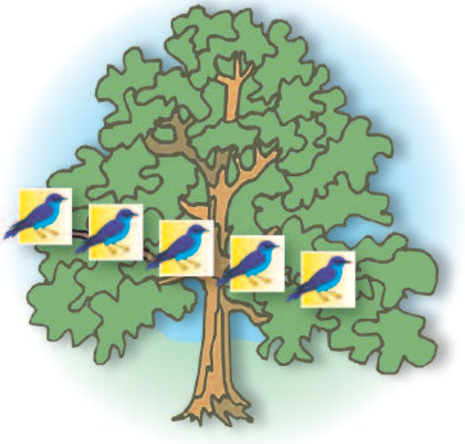
A bird watcher wanting to search a document collection for the *effect of pesticides on the mating habits of North American hummingbirds* might use the Boolean / proximity search:

(effect w/9 pesticides) and mating and North American and hummingbirds

This search would look for a document containing all of the following: the word *effect* within nine words of the word *pesticides*, the word *mating*, the phrase *North American* and the word *hummingbirds*. However, even this level of precision searching might generate a number of false hits, or documents that contain terms that match the full-text search but are not really on topic. For example, the search could retrieve a document that simply mentions the *effect on South American hummingbirds' food supply of pesticides* and *North American partridges' mating habits*.

Additional Boolean logic, term weighting and other search techniques can further refine the full-text search to eliminate some false hits. A search, for example, could exclude documents containing *South American hummingbirds* by excluding entire documents that contain the phrase *South American hummingbirds*, or less drastically, by giving documents containing *South American hummingbirds* a negative term weighting.

However, if an author's name happened to be *Hummingbird Q. Pesticide* and he enjoys writing articles on the *mating habits of North American gulls*, these articles would still result in a large quantity of false hits. Yet, even if a searcher were aware of this author, avoiding



A full-text search index treats all hits as equal.

Precision searching of a database means finding a hit only when it appears in a highly specific field structure.

these false hits through full-text queries alone would not be an easy task.

An alternative to Boolean logic is natural language "clustering." With this approach, once a search finds an applicable document, a follow-up search effectively enters the text of the entire document to look for other documents of that type or cluster. Natural language relevancy-ranked searching looks for all words in a search request or document, and ranks by hit term density and rarity retrieved documents with matching terms.

For example, if *hummingbird* appears in thousands of documents, but *pesticides* only appears in dozens, the latter receives a much higher relevancy ranking in looking for matching cluster documents. Documents that contain both words would rank even higher. Unfortunately, this also includes articles by *Hummingbird Q. Pesticide*.

While false hits, or over-inclusiveness in full-text searching, is annoying, under-inclusiveness, or false misses, because of spelling variants, phrase variants, and the like is also a concern. Certain techniques can find word variants: stemming can find variants such as *hummingbirding*; fuzziness can sift through misspellings, such as *hummingsbird*; and thesaurus searching can find a Native American *hummingbird* synonym. However, at a certain point, extending the list of retrieved documents to encompass word variants will itself start resulting in false hits.

Adding in fielded data components to encompass key search criteria assists in avoiding both false hits and false misses. For example, *mating habits* might reside in a

> One approach in dtSearch for linking this structure to full-text index data is through a database access library such as Microsoft's ADO.NET. An integrating ADO.NET application iterates over every row of every table or field in an SQL database, associating each field with the relevant document identifier.

XML as a Database Format

To take full advantage of XML as a hierarchical data structure, dtSearch supports nested field searching. For example, sample dtSearch nested field searches over Shakespeare converted into an XML database might be:

- *persona contains Henry*
- *scene/stagedir contains exeunt citizens*
- *scene/speech/line contains publius*
- */play/title contains Henry the Fifth*
- *scene//line contains publius*
- *(henry the fifth) and (scene/speech/line contains publius)*

The first example looks for any field entitled *persona* that contains *Henry*. The second search, containing the / as a field separator, looks for a field called *stagedir* containing *exeunt citizens*, with the *stagedir* field directly nested in a field called *scene*.

The third example looks for a triple nested hierarchical *scene/speech/line* field sequence containing *publius*. The forth example, starting with the /, looks for the *play* field at the top of the hierarchy, with a *title* field just beneath it containing *Henry the Fifth*.

The fifth example, with the //, looks for a field called *line* containing *publius*. In contrast to the other examples, which specify precise hierarchical sequences, in this last example, the *line* field could be anywhere from directly beneath the *scene* field, to nested at multiple levels of depth.

Finally, the last example combines full-text searching with nested field searching. This example would combine a full-text search for *henry the fifth* and a nested field search for *scene/speech/line contains publius*.

Reprinted with permission of PC AI Online Magazine V. 16 #5
For more information about PC AI Online Magazine, visit www.pcai.com.

behavior field, *North American* might translate to a *geographic* field, and *hummingbirds* would be a *bird type* field, leaving *pesticides* for the full-text search component (assuming no *pesticide* field). The result is a much narrower margin of error.

Combining Full-Text and Fielded Data Searching

While combined hierarchical fielded data searching and "democratic" full-text searching can improve search precision, the question remains how to synthesize these two very different approaches to data from a development perspective.

Option 1: Self-Contained Documents with Fields

The easiest approach to bridging the full-text / fielded data search divide is to have each document in a collection contain its own fielded or meta data. For example, the bird document collection could contain HTML, PDF, or Microsoft Office files — word processor, spreadsheet, presentation, etc. Each document in the collection could contain a *bird type* field, a *behavior* field, and a *geographic* field, along with searchable full-text content.

For indexing efficiency, this solution requires a single pass over each document in the collection to pick up fields and full-text data. Another advantage to this approach is its organizational flexibility. Since each document contains its own fields, it represents its own self-contained unit. Even after removing the document from the group, its fielded information remains. This advantage is distinctly not present in most of the other options described below.

While storing fields inside the documents certainly has its advantages, getting to this result in a large document collection may not be an easy process. Simply because a file format supports meta data does not mean that all documents of that file type will already include such data. The size of a document collection as well as perhaps the diversity of document types can make editing each document in the collection to add fields prohibitively time consuming.

Finally, the fielded data itself may require a more complex structure than the underlying documents support. Classifying the fielded data may require a table structure or hierarchical data classification, which does not work in the limited fielded data options that, for example, the PDF format provides. In that case, a separate database structure may offer a more flexible option for fielded data storage.

Option 2: Separate Database and Documents

This option stores meta information, pertaining to each file, within a separate database structure such as SQL or XML. Database entries can include fielded data information for each document, such as *bird type*, *behavior*, and *geographic* area. Along with the fielded data, the database would also store pointers to associate each set of fields to the correct document.

One approach in dtSearch for linking this structure to full-text index data is through a database access library such as Microsoft's ADO.NET. An integrating ADO.NET application iterates over every row of every table or field in an SQL database, associating each field with the relevant document identifier. For example, specific fielded data entries for *bird type*, *behavior*, and *geographic* area would all correspond to specific document designations.

The full-text search index then incorporates the ADO.NET fielded data components along with the document pointers. In this manner, the document pointers act as the bridge between the full-text component and the fielded data component. Because an actual structured database holds the meta or fielded data, this approach supports a more complex relational structure to the field components.

The separate database and documents approach also preserves the original documents, and any existing fields inside of the original documents for indexing and searching. This original fields' preservation is an advantage this approach also shares with the following option.

Option 3: BLOB Data

The previous option stored meta information pertaining to each document in a database structure, along with pointers to that document. This option stores the full document copy, called BLOB data, in the database along with its fielded data. BLOB data can be anything from a raw text file to a structured file type such as a word processor, spreadsheet or presentation document.

The indexing mechanism for BLOB data is similar to indexing a standalone file. It also has the benefit of supporting any preexisting fields inside a BLOB document, along with the fields in the database. For example, if the database stored a word processor document as BLOB data, indexing supports the *title* and *subject* fields contained within that document, in addition to fielded data in the database.

Whether using document pointers or BLOB data, tools such as ADO.NET for accessing database fields are fairly advanced and easy to use. On the negative side, the database fields approach requires a separate database, resulting in substantial maintenance overhead. A separate database also eliminates the efficiency of single-pass integrated indexing of documents and fields.

Option 4: Adding Fields "On-the-Fly" During Indexing

Another alternative for combining full-text and fielded data searching is to add document attributes while indexing. In contrast to the above examples, these new fields are not part of the documents themselves, nor do they require storage in a separate database. Rather, these fields simply become a part of the full-text search index.

A special function stores these document attributes upon indexing, in addition to the full-text data. As with adding fields to the documents themselves, indexing returns to a seamless, one-pass operation. As with the BLOB and separate database approaches, the dynamic addition of fields upon indexing supports pre-existing fielded data already inside a document, in addition to newly added fields.

Adding attributes while indexing requires the matching of specific fields to existing documents. The easiest approach to obtaining the fielded data information for dynamic addition upon indexing is to require the entry of certain attributes when a document joins the collection. For example, a separate document management interface could require entry of fields such as *bird type*, *behavior*, and *geographic* area when checking in a document.

Dynamically adding fields to documents during indexing, similar to adding fields internally to documents, has the disadvantage of requiring potentially cumbersome individual attribute assignments to each document before indexing. However, this dynamic approach is more efficient for one key reason: it adds fields independent of the actual documents. Therefore, the dynamic indexing approach does not require opening, editing and then closing each individual document to add the fields, resulting in yet another major benefit relative to adding fields inside each file. Under the dynamic approach, the original documents remain untouched, preserving the original documents intact for archival purposes.

> In the dtSearch Engine, an "xfilter" can combine a full-text query with a filter for specific document attributes, such as file name, date, or size, or the presence in the document of a word or field. The field component can consist of a standard document attribute, or an attribute that dtSearch adds "on the fly" while indexing.

Option 5: Modifying Search Results Data to Add "Stored Fields"

The above options for combining fielded and full-text searching focus on the text retrieval engine's method for executing a query. Another alternative is to focus on the presentation of search results after a query. This alternative relies on fields and other methods to more efficiently sift through a data set following a search.

Drawing a sharp line between the presentation of search results and search techniques is, of course, impossible. For example, natural language

Sample Objects for Document Classification

In the dtSearch Engine, an "xfilter" can combine a full-text query with a filter for specific document attributes, such as file name, date, or size, or the presence in the document of a word or field. The field component can consist of a standard document attribute, or an attribute that dtSearch adds "on the fly" while indexing.

Search	Results
(user request) and xfilter(name "abc*.html")	This query would match any document that contains (_user request_) with a file name matching _abc*.html_
(user request) and xfilter(word "projectxyz")	This query would match any document that contains (_user request_) and that also contains the word _projectxyz_
(user request) and (xfilter (word "Type::projectx") and xfilter(word "classification::high"))	This final query adds two field restrictions to the _(user request)_: one for a named field called _type_ with an entry of _projectx_, and the second for a named field called _classification_ with an entry of _high_.

A dtSearch SearchFilter uses an in-memory object, consisting of a table of bit vectors, to achieve similar results to that of an xfilter.

relevancy ranking is both an integral part of the search process and an integral part of the retrieved document sorting that underlies the presentation of search results. However, while the execution of a search works on programmatic autopilot, after a search, sifting through a retrieved data set relies on the human factor.

Adding fielded data into this phase provides valuable information for the human search results browser to use in separating relevant items from false hits. “Stored fields” in search results, where a stored field information tag describes the contents of each document, is an example of this human-oriented approach.

If a search retrieved false hits in the form of articles by *Hummingbird Q. Pesticide*, this fact would become immediately apparent through a stored *author* fields tag. The user could then skip over the *Hummingbird Q. Pesticide* documents without even bothering to browse them. In this way, the stored fields approach effectively offers similar benefits to those of narrowing the scope of a search by entering specific fields.

Unlike the other approaches in this article, however, stored fields upon search results achieves its benefits without requiring specific fielded data elements in a text query. Because of its simplicity from an end-user perspective, stored fields is an ideal approach for experienced searchers and novices alike.

From a development perspective, the same function for adding dynamic document attributes upon indexing can also store document attributes as fields upon search results presentation. In other words, the same code that adds a *bird type* field, a *behavior* field, and a *geographic* field dynamically during indexing also serves to create the information tags in search results.

The option for adding fields on-the-fly dynamically while indexing thus serves double duty: in the automatic execution search-request phase and in the human-driven search results browsing phase. The end-result is a two-fold bridge of the gap between full-text and fielded data searching.

Please visit dtSearch online at www.dtsearch.com

Reprinted with permission of PC AI Online Magazine V. 16 #5
For more information about PC AI Online Magazine, visit www.pcai.com.