Cobra: A new approach to IR system design

Tim Mills, Ken Moody, and Kerry Rodden
{Timothy.Mills, Ken.Moody, Kerry.Rodden}@cl.cam.ac.uk
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, United Kingdom

Abstract

This paper describes Cobra, an object-oriented system for building information retrieval applications. By founding an architecture for IR systems on a model for document content, we believe that the design process will benefit in the same way that database design is aided by data modelling. An object-oriented database has been extended with specific support for information retrieval data structures. Together with an application framework, this allows rapid construction of advanced IR systems.

A major goal has been to ensure that information about how a query proceeded can be delivered to the user interface to ensure that the user can build up an adequate model of the system's behaviour. We have created a Java interface to Cobra which actively supports query reformulation. This has been used to provide access to a Web-based collection of historical records. The whole system is implemented and working, and early experiments are encouraging.

1 Introduction

Recent years have seen the emergence of content-based indexing methods for multimedia documents. Although bespoke systems have addressed individual application areas, the effort expended in building such software suggests that a more general approach to these problems would be beneficial. In response we have developed Cobra, a content-based retrieval architecture which extends object-oriented database technology to allow rapid construction of complex IR systems.

The architecture is founded on a model for multimedia documents which allows use of different indexing features to build up a composite representation of the document content. Tools such as stemming and filtering may be applied to modify representations. Document types are described in an extended database schema which declares the representations and tools which support them. We will refer to the schema extensions as the *content model*, which supplements the traditional database schema.

By adopting this descriptive approach, much of the complexity of indexing, storage and retrieval of documents is handled by the system, allowing the application developer to concentrate on the specifics of the application rather than the full information retrieval problem.

Explicit in the design is support for informing users of the progress of their queries. Specifically, we wish to enable the users to make good use of relevance feedback and to understand how their queries relate to the system's responses.

Practical experience is deemed essential to evaluate the architecture. As a step towards this, we have used Cobra to build a text retrieval system to facilitate academic research into an important collection of historical documents. Our experiences with this first trial have been promising, and work is underway to build other retrieval systems to experiment with multimedia IR.

This paper begins with a review of related work in the area of IR system architectures and user interface design for retrieval systems; we then look at the IR design space and describe our Object Model and application framework. We conclude with a description of our first case study.

2 Related Work

Work at Xerox PARC presented an object-oriented architecture for text retrieval [CPH91] which aimed to facilitate functional modularisation, mix-and-match combination of module implementations and definition of inter-module protocols. This architecture is based on a process in which token streams representing document content are passed between tools in a pipeline.

ECLAIR [HW92] is an extensible object-oriented class library for information retrieval. The system is implemented in C++ together with an ODMS for persistent storage. As such, this represents an application development platform for IR systems and applications. In contrast to the Xerox architecture, ECLAIR uses document classes to encapsulate all the specifics of handling each document type. However, the design does not deal with tools as separate objects, but rather treats them as an integral part of the document.

FIRE [SF95] is a design for a reusable IR framework. The framework is a program skeleton that defines the basic concepts of the IR domain and how its components interrelate. The developers aim eventually to support integration of the application's data model into the FIRE object model and provide facilities for dealing with different media types.

Cobra builds on these approaches and aims to avoid their shortcomings. Cobra models tools as separate objects as in the Xerox architecture, while also encapsulating the document description as in ECLAIR. However, Cobra allows specification of an IR application down to the level of index features and retrieval statistics. This gives the application developer a much greater degree of flexibility in terms of time-space tradeoffs and choice of ranking methods.

Rather than relying on prebuilt class libraries which enforce a particular way of tackling an IR problem, Cobra emphasizes a database-like approach in which the specification of a solution is described. This description is then used by the Cobra system to generate appropriate code to achieve the task. As an extension of the database schema, the definition makes for tight integration of the application data model with the content model. Unlike FIRE, we explicitly specify content indices as objects within the schema, and do not require database indexing to facilitate content-based searching.

Relevance feedback is a query reformulation technique which has been shown experimentally to produce significant performance improvements in IR systems [HC93]. The user submits a query, browses the results, and then selects those results which she regards as particularly relevant to her query. This allows the system to refine the query by adding in new terms from the selected documents, and altering the weights of the original terms. This is more helpful than leaving the reformulation entirely to the user, who may already have made her best attempt at a query. It has been difficult, however, to create an interface which can persuade users to take advantage of relevance feedback. Koenemann and Belkin at Rutgers University [KB96] have conducted user studies of different interfaces to the INQUERY retrieval system. They concluded that making the relevance feedback process more visible to users and giving them control over the expansion of their queries can greatly increase the system's effectiveness.

Especially notable work in the area of user interfaces to IR systems is the ongoing series of Okapi projects at City University in London. Their most recent report [BDPJ97] describes the evolution of their interfaces through several different methods of query expansion. The current interface provides *incremental* query expansion, where the list of suggested extra terms is updated each time a document is marked as relevant. The user can then choose to delete any of the terms before resubmitting the query.

In constructing our interface, we have endeavoured to learn from these studies while experimenting with alternative approaches and additional functionality. Cobra can generate a report at each phase of query evaluation, which we can exploit to help the user understand how the query terms relate to the results received.

3 The IR Design Space

The diversity of techniques which have successfully been applied in IR systems at first makes the goal of creating a general system architecture appear intractable. There are a number of variables which must be finely balanced when designing a particular IR system.

Retrieval Style

At the extremes of retrieval style lie browsing and searching. Browsing enables users to move from document to document via static or dynamically generated links. Searching aims to pinpoint documents relevant to a particular information need as expressed through a query statement. Bridging the divide between these styles is relevance feedback, where a user marks those query results which are relevant and repeats the search.

Document Content Representation

The document content representation refers to the amount and sort of processing applied to a 'raw' document in order to distil a representation useful for retrieval.

Browsing often requires just the raw documents, where links are explicitly specified in the document format. In the case of searching, it is often essential to preprocess documents in an attempt to extract useful aspects of their content. Retrieval models often require that some statistics be gathered from the collection as a whole, so analysis of the entire set is necessary.

There is a great variety in the information we may select to characterise document content, and several feature types may be used in conjunction. Text documents are usually indexed by word features, however other tokens with special significance such as dates or names might additionally be used. Terms may be normalised, for example by case normalisation or morphological stemming.

User Interaction

The user interface must deal with the initial input of the user's information need, display of the results and reacting to any further interaction by the user such as making relevance judgements.

The most basic form of user interaction in common use requires the user to present a series of keywords, perhaps constrained by the operators of a query language. On submitting the query, a ranked list of results is returned. However, we should not limit the system to this form of interface; more exotic user interaction methods come from the field of information visualisation.

Unlike the case of database querying, no single query language has emerged to dominate the market. Thus the architecture must be flexible enough to allow a wide range of languages to be supported.

Multimedia retrieval presents more problems at the user interface, since it is more difficult for users to formulate their information need. Query by example is an often used alternative to query languages, where a user introduces candidate documents, and requests that the system find similar documents. Additionally, while text documents may be quickly skim-read, and pictures can often be judged for relevance at a glance, audio and video data takes more time to audition, and thus may require special consideration with respect to user interface design.

There is good evidence that providing more information accounting for the behaviour of the system allows users to build a better cognitive model of the search engine [BDPJ97]. This should enable them to understand the effects of keywords in their queries on the search results, and thus understand how to modify the query to improve the search.

These facilities, while perhaps too varied to form a fundamental part of the system, must be *supported* by the architecture. There are two requirements here: the architecture should not dictate the choice of query language and interaction style, and it must assist user interface design by making available the information used internally in the retrieval process for use at the user interface.

Retrieval Model

Most retrieval models fit into the categories of Boolean, Vector Space, Probabilistic or Logical. To choose an appropriate model for an application, it would be useful if the retrieval model could be replaced to allow experimentation. This goal is somewhat difficult to achieve, since each retrieval model requires its own particular set of statistics to be available for evaluation.

In addition to these statistics, proximity information and similar incidence data may be required. While it may be possible to store a large number of these possible statistics in case they are needed, this would lead to significant overheads in terms of space and time. Thus we require that the statistics required for an application can be flexibly specified.

Efficiency

Once a representation of the documents has been constructed, it is necessary to provide access to these representations in a form suitable for querying. The storage requirements are dependent on time/space trade-offs and the types of queries which are to be handled.

At one extreme, serial file searching uses the document material directly and thus requires no extra storage overhead for the document representation and no additional cost for update of the document collection. However for other than small document collections this technique becomes impractical because of its high computational expense.

At the other, the inverted file aims to maximise the speed of retrieval by considering only those terms that are specified in the query. This structure associates with each term or concept of the document representations a list of locations at which the concept occurs within the document collection. The index maintenance cost and storage overhead can be reduced by efficient implementations and use of compression techniques.

4 An Object Model for IR

Most indexing and retrieval systems consider documents as a collection of words to be indexed, that is, a set of information items of a single type. However, the content of a document is often much richer than this simple representation can express. For example, while a date within a document may simply be considered as a set of characters like any other word, if it is instead treated as having the semantics of a date then better use may be made of the information, such as allowing queries against date ranges.

Once we wish to cater for content-based indexing of non-textual documents there may not be any words to index, and so we must look for other content-bearing features on which to base subsequent retrieval. To provide adequate retrieval performance, a combination of a number of features may be required.

While databases commonly provide a wide array of types for modelling data, there is little support for management of unstructured data such as documents other than binary large objects (BLOBs). The content of these unstructured objects is of primary interest in IR at query time. As Van Rijsbergen notes in [vR79] we require to select objects based on the possession of some proportion of a set of attributes, as opposed to only selecting those which satisfy an exact match on all the attributes. As they stand, traditional databases fall short of the requirements for information retrieval.

Despite these differences, databases do provide many attractive features such as storage management and transactions. If it is possible to augment a database to bridge the gap between data retrieval and IR then the benefits of each approach can be combined.

In order to develop a system capable of building a range of retrieval applications, we take a descriptive approach. In a similar manner to databases, we aimed to allow a specific application to be described with a schema. As our starting point we took the ODMG Object Model [Cat96] which is used to define object-oriented database schemas.

4.1 The Object-Oriented Data Model

The ODMG-93 standard introduces the Object Definition Language (ODL) as a programming-language independent data description language. This is used to define the database schema in terms of the ODMG Object Model.

The Object Model provides typed *objects* and *literals* as primitive data modelling units. Objects are uniquely identified by *Object Identifiers* (OIDs), and may be independently stored in and retrieved from the database. Literals do not have OIDs and therefore cannot stand alone but must be embedded within objects. Object state is described by a set of *properties*, which are either *attributes* or *relationships* between objects. A set of *operations* which may be invoked on the object defines object behaviour. The Object Model supports inheritance, defining subtype and supertype relationships between types. Collection objects and literals allow sets, bags, lists and arrays to be modelled.

An OODB may be queried either programmatically, using the supporting programming language, or via the language-independent Object Query Language (OQL). OQL is a superset of the part of the standard SQL that deals with database queries. It is a declarative, typed, functional language which provides high-level primitives to deal with the structures and collections of the Object Model. Update of the database is handled by invoking object operations which are defined for that purpose.

4.2 OODBs and IR

To investigate the suitability of an OODB for information retrieval, consider how a document might be represented within the Object Model. Using the terminology of the ODMG-93 standard, the body of the document would form one attribute, either as a string or an array of octets. Alternatively the document might not itself be stored within the database, and so instead a reference to the document on a local filesystem or remote server could be used. For convenience, a title might form another attribute together with a document abstract. Since these are derived from the document itself, we could instead generate these from the document source as they are needed at query time by defining a read-only attribute. Other information such as file size and access permissions might also be stored.

We might choose to give a definition in ODL for a simple document as:

```
interface Document
{ }
{
    attribute          String title;
    readonly attribute String extract;
    attribute          String body;
};
```

From this, it is easy to see how document attributes may be displayed and compared with exact-match database queries, however it is not obvious how the queries typical of IR systems may be formulated. If the database provided string matching, it would be possible to locate matching words within the document body attribute, effectively giving serial-file searching. Even with this facility, the representation lacks the collection information required by information retrieval models to compute the relevance of a document to a query. Standard database systems provide no concept of "best match" and no notion of the probability of relevance. This of course means that relevance feedback has no meaning at the database level. Such facilities would have to provided at the application level.

To tackle these problems we must look at the indexing process which is carried out by conventional IR systems and decide what information must be represented to allow documents to be judged for relevance. As mentioned earlier, some researchers have shown that relational and object-oriented databases can be used to model all the data needed by an IR application [Put91] [SF95]. While this may be the case, it is doubtful whether this provides a desirable abstraction of the indexing and querying process, which involves more than just storage and retrieval of data. By making the IR system just another database application, we have two data models in use at once, namely the IR 'application' and the user's application. We propose an alternative approach in which the database is minimally extended to integrate the application data model with the data structures required for information retrieval.

While databases often provide indexing of attributes to aid query execution, those indices are not inherently part of the database schema—indeed indices may be created and deleted on the fly. We could store for each document the set of indexing features, and form an index based on these sets. However, this requires that we duplicate information—the terms are both an attribute and a key in an index. This contrasts with IR system design, where only the index need be stored. The following sections explore the advantages of a more suitable abstraction for integrating document content descriptions into a database schema which can more pleasingly model the processes involved in IR.

We also look at the use of query languages. The database's Object Query Language is intended for searching structured data. Documents have a much looser structure than database objects, and thus it may be unsuitable for some aspects of information retrieval queries (in respects other than its lack of best match). While we may wish to describe some aspects of documents in a structured manner, the document content is likely to be semi-structured or completely unstructured. Therefore it is also necessary to determine how querying should proceed in the context of information retrieval.

In summary, we wish to:

- integrate content model descriptions into the database schema.
- provide data structures to support information retrieval.
- support query languages for best-match retrieval on loosely structured data.

These goals require solutions at two levels. At the lower level, extensions to the database Object Model may be necessary, while we also require an application framework above the database. In the following sections we concentrate on the Object Model and discuss the application framework.

4.3 Extending the ODMG Object Model

Here we propose extensions to the ODMG Object Model for explicit support of information retrieval, and justify their inclusion. The extensions are intended to provide a natural integration of IR with the database and to enable efficient implementations to be devised. The data structures support the content representation for a single document, representation of a collection of documents and the collection of features for query expansion. Their intended use is more fully explained in the following section.

Literals

The fixed-length literal Word is a fixed-length form of a string, with additional methods to perform operations such as case normalisation. It is included to improve execution performance when dealing with large numbers of textual indexing features.

A proximity list literal ProximityList is used to store monotonically increasing integers representing word positions. Since this restriction is made, an implementation can exploit this to compress the data at storage time. The datatype allows appending of a new position onto the end of the list, and is thus much more restrictive than the collection type, allowing a more efficient implementation for this common operation.

Document Content Representation

During indexing, documents are parsed to extract features which are the semantic units of information used to express a concept. The features are associated with statistical and locational information required by the chosen retrieval model(s). These features may be of a variety of types, but since a content representation models only one aspect of the content, we may use several content representations to model each characteristic which is of interest.

Here we make a case for inbuilt support of term vectors and inverted indices, the data structures of choice for most retrieval systems. Both have been extensively studied and efficient implementation techniques have been developed for each.

We omit formal interface specification of the proposed additions; full details can be found in [Mil97].

Feature Vectors One way to represent the content of a single document would be to use a feature vector. While the simplest of feature vectors might contain only occurrence information (meaning that it was simply a set of features), it is usual to associate each feature with statistics such as a frequency count.

We can implement feature vectors using an associative container. Since we do not want to restrict the type of features or statistics which are to be collected, we should not limit the data structure to a particular type. This is entirely consistent with the ODMG collection type generators, in which a container is parameterised by a type. In the case of the feature vector, we need to parameterise it by the feature and statistics types.

In terms of the features alone, the data structure bears many similarities to sets, however, we must also take account of the statistics which are to be maintained. When a feature is added to the vector which is already contained in it, updates to the statistics must be carried out. In the case where we only need to record the existence of a feature, the data structure reduces to a set.

We must also consider the operations which may need to be carried out on the data structure to implement typical IR functions. Processes such as stopword filtering necessitate the removal of a feature-statistics pair. Stemming involves the replacement of a word by its stem. If two words have identical stems, the associated statistics must be merged. Index-time thesaurus expansion would require a feature-statistics pair to be copied with the replacement of a term by its synonym. Therefore after a representation has been created (through a series of insertions), we wish to provide a limited number of methods to modify it.

We have a need to build up representations cumulatively. For example, we may wish to count the number of occurrences of a keyword as we come across them during parsing. This requires that we update the mapping between the keyword and its statistics. Because the ODMG Object Model states that literals are immutable, we do not directly update the vector, but instead read an old value of the mapping and replace it with a new mapping. The alternative would be to force all statistics to be modelled by objects. There are two reasons for avoiding this. Objects are independently retrievable entities, however statistics are only meaningful when detached from their corresponding feature. Secondly, the implementation of objects will introduce space and time overheads which can be avoided by using literals. Note that in the implementation we would wish to optimise this lookup-then-update for efficiency purposes.

Content Indices Although the feature vectors, as objects, can be stored and retrieved independently, it is often more important to consider the content representations of a collection of documents rather than of one document in isolation. Thus we provide the traditional inverted index for this purpose. This is another associative data structure, which is analogous to the extent of the type in that it contains references to all the instances of that type. It differs in that these references are associated with indexing features and statistics.

The functionality and performance of these indices are critical to the operation of the IR system. It is likely that a full-scale implementation would provide a number of implementations to allow tuning to specific applications. This specialisation may well include different implementations depending on the feature type. In addition to one-dimensional indices such as B*-Trees, an implementation would be expected to provide multidimensional structures, such as R*-Trees, which are often required to store indices of multimedia object features.

This object is used in a significantly different way to the standard ODMG collections. We wish to insert both references to documents and the information which associates them, and allow removal of references irrespective of this information. The approach taken here is to use the object in a session-like way. A reference is inserted and then all the feature-statistics are inserted, before notifying completion. Queries on the index return lists of locations which pair document references to their associated statistics.

To make the indexing vocabulary available, an iterator may be created to walk over the index. The utility of this depends on the ability to interpret the feature information—for example, while whole words are meaningful to users, algorithmically constructed word stems may be less so.

4.4 Best Match Results

For some applications we may just wish to return, say, the best ten results regardless of the scores assigned to them by the retrieval algorithm, often we wish to have access to these scores. This is of course essential when applying operators to combine the evidence from each query term.

We therefore define a Result which has methods confidence which returns a value representing some measure of the system's confidence in the referred object being relevant to the query, and reference gives access to the object.

5 Content Modelling

Given the basic data structures of the extended Object Model, we now discuss how they would be used to build up models of document content.

We define *content representations* as a mechanism for managing the complexity and diversity of document content. These are objects which contain the data structures for storing feature-statistics pairs, together with methods which aid their use. This bridges the semantic gap between the data structures supported by the database and the representation used by the application.

Even though we may specify an index to store string features, there is still a great variety in the way that the string is used. For example, all the strings may be case-normalised single words. The representation methods perform such operations, and thus standardise the usage of the set; any document represented by a particular representation will have its content represented in the same way. The content representation serves to reduce variability, group functionality and allow easy identification of the way in which content is modelled. Additionally, once a content representation object has been defined, it can be reused, thus building up a library of objects for content modelling.

For example, consider a content representation for plain text, that is, text without any markup information or other embellishments. A suitable semantic unit for indexing plain text is to use words as features. The most simple location information we could provide is a document identifier, that is, the document identifier of the document from which the feature was extracted. As has been mentioned previously, retrieval models require various statistics to be at hand when computing term weights at query time, so we choose to store the maximum term frequency, together with the list of positions at which the term occurs. This will allow the content representation to be used in conjunction with certain retrieval models, while making it impossible for use with others. It is thus envisaged that a content representation must be chosen for its suitability to a particular task. It moves control of document representation from the internals of an IR system into the hands of the designer of the document class. It is this shift of emphasis which reduces the complexity of the problem by a representation with exactly the right facilities for a particular job. Of course, this does require a designer to understand how the choice of representation affects the application retrieval performance. This representation object supplies methods to aid insertion into the set, creating the appropriate statistics.

It is also necessary to define a function which can create an instance of a feature. While the ODMG-93 standard does not support free-standing operations, it is stated that future revisions may support this facility.

5.1 Transforming Representations

So far we have seen how collections of features extracted from a document are modelled. However, the 'raw' initial set of features may not be the best method of indexing the document. Text retrieval systems typically apply processes such as stopword filtering and morphological conflation to the extracted terms in order to refine the content description.

Stopword filtering discards high frequency features which, if used as search terms, would result in retrieving almost every document in a collection. In the context of English text, words such as 'and', 'or' and 'the' are common examples. These words are considered to be of little use in indexing because of their low discrimination value. Of course, it is possible to argue against stopword filtering, since phrases containing stopwords may well be useful for discrimination (e.g. "to be or not to be"). Nevertheless, stopword filtering may be important from a practical perspective since it reduces the size of the ancillary indices.

Stemming reduces morphological variants of words to their common root. Thus 'fishing' and 'fishes' would be reduced to the stem 'fish'. The effects on retrieval performance are usually positive, so stemming is considered to be a useful thing to do.

It is possible to think of analogies of such tools in non-textual media, such as the clustering of similar colours for image retrieval or the removal of 'ums' and 'ers' in speech.

We propose the *tool* as the means of representing such transformations. A tool is defined as a binary relation between two representations, and is modelled as a database object. While a tool may not change the physical type of a content representation, it does have an effect on the semantics. For example, a stemming tool may map between English words and English stems.

5.2 Queries

The ODMG-93 standard defines the Object Query Language (OQL) as the primary means of querying object-oriented databases. OQL was designed for querying of databases, and as such it will quite understandably have some deficiencies with respect to information retrieval queries. Firstly, it is a structured query language, and queries are constructed with a knowledge of the database structure. Documents rarely exhibit the record-like structure of database objects, and we would not expect the user to have to deal with any structure imposed upon the collection. Secondly, OQL provides only exact match queries; a record either satisfies or fails to satisfy a query. IR requires querying based on degrees of similarity. Often only the best ten results are required, rather than an exhaustive list of every possible answer, no matter how weak the similarity. Thirdly, OQL has no support for the operators of a query model. Indeed, if we intend to build a generic IR system, we cannot say exactly what the operators will be, as new retrieval models may introduce new operators.

Clearly OQL on its own is inadequate. At the very least we need some mapping between a language suitable for information retrieval and the underlying query language. At the same time, we do not want to lose the benefits of the structure that is available to the user, for example, allowing the user to filter results by author.

A query language suitable for one task may be unsuitable for another, due to the expected sophistication of the user, the application area or the media types involved. Thus we want to enable a document class to be queried by a variety of languages. A query language may generate features which do not directly match those stored in the document class representations, and so will require converting with tools. Given these facts of life, we must enable queries to be interpreted in the context of document classes.

The context for a database query is usually a set of objects, either the result of a previous query or the extent of that type. For IR, the context is even more important, since this determines some of the parameters of the query model. We take as the context of an IR query the indices associated with the document class. Thus the indices are analogous to the extent of the type in the standard ODMG Object Model. The indices contain the content representations of each object of that type, and can thus hold the collection statistics required by the retrieval model.

Most query languages allow the user to specify not only key features, but also relationships between those features. This gives the query language both a vocabulary and a grammar. For example, a query language may allow the user to specify that two keywords must be adjacent, or that an object in an image must be located in the top left hand corner. This translates to constraints on the sets of retrieved location information, creating a dependency between query language and document class representations. Taking the proximity constraint as an example, if the underlying document class representation has no way of obtaining proximity information (because it was not recorded at index time and there is no way of obtaining such data at query time) the constraint cannot be satisfied. In this case, perhaps some weaker constraint can be substituted, or alternatively a different query language should be used.

6 Document Classes and Retrieval Models

By modelling a document as an object, we have encapsulated its behaviour and the way in which it is indexed. This means that we have a gap between the way in which a query language represents a query and the way a document represents those same concepts. Therefore we need a mapping between the two. This need for a mapping is a key strength of the system design, since it means that a query language can be used with a wide range of document classes without imposing restrictions on the representations used by the document class. Thus it is possible for queries to range over heterogeneous document collections.

Users will want an easy-to-use language in which to formulate their queries. However, the query language must map onto the underlying retrieval model. Query languages allow the user to specify terms which are to be matched, but often operators are provided to constrain the search. For example, INQUERY provides a wide range of operators which mimic Boolean expressions, while AltaVista allows HTML-specific features such as hyperlinks to be searched. Obviously the language cannot provide functions which the retrieval model does not support. The basic probabilistic and vector space models do not allow such constraints to be specified, whereas the inference model and boolean models do.

By separating out the query language from the retrieval model, it is possible to allow one to be replaced without affecting the other. Thus the query language should construct queries in terms of the operators of the retrieval model. Typically this would proceed by building a data structure from retrieval model operators.

It may be useful to use the same query language together with several retrieval models, combining the results in some way appropriate to the application. Thus we approach the problem of bridging the gap between the document class and the query language by implementing document-specific converters to perform the mapping. A document class may support any number of converters to handle a variety of query language and retrieval models. While the retrieval model supplies the model operators, the converters provide the operands.

Figure 1 illustrates the abstraction. A query language constructs a query from the operators of one or more retrieval models. Each document class exports a converter which implements the operands of the query language. Note that query languages and retrieval models are part of the application framework and not the Object Model. A discussion of their behaviour is therefore left until section 8.6.

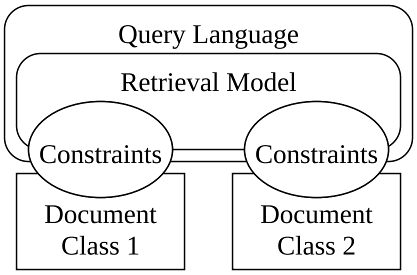

Figure 1: The document-query interface.

7 Performing Queries

While there have been several attempts at integrating operators suitable for information retrieval into structured query languages, we do not investigate that approach here. This is partly because SQL and OQL are both standards with which it is unwise to tamper, but primarily because it there is no finite set of operators which can be added to ensure that any retrieval model can be supported. Because new retrieval models are still emerging, it is inappropriate to limit the system to some arbitrary set of current models. Thus we treat query languages as entities separate from the database query language, although they should be compatible in that the results of one type of query may be used as arguments for another.

Here we illustrate how queries are carried out on both single document classes and multiple classes.

7.1 Queries on a Single Document Class

There are usually two parts to ranking. Firstly index terms and query terms are weighted, and there are a great many formulae using a variety of statistics to accomplish this. Secondly, these weighted documents and queries must be compared to compute a single value used to rank documents for relevance to the query. The former task requires knowledge of the index information available. The latter stage can remain ignorant of this information, for it need only work with those computed values.

Consider a query language which can specify keyword and proximity matches on documents. The query language expresses these constraints via an object interface which defines methods to return result lists corresponding to occurrences of words and phrases.

To support this language, the document class must provide an implementation of this object. This implementation supplies code to convert the operation arguments to arguments for query operations on the class indices. The results of these operations are then used to compute term weighting, and these results are passed back as results.

This gives great flexibility, since it does not impose a single way of representing information on the document class. For example, if proximity information is not directly available for the indices, it could be computed by locating the objects themselves. This means we can easily design systems which trade off index space against query execution time. In addition to the operations performed on indices, we can also perform query operations on document attributes and relationships at this stage.

By this design, we have managed to keep encapsulated all knowledge of how the document class represents its content and what statistics are available. This has avoided 'contaminating' either the query language or the retrieval model with any document-specific information.

7.2 Queries on Multiple Document Classes

If a set of document classes each export the same language constraints objects, the same query may be run against each class. Because the language constraints are specialised to a type, running a query over several types interprets the query in the context of each document class. This allows the same query to be used to search heterogeneous collections. However, combining the results in a meaningful way is non-trivial. Since each class may use a different term weighting scheme, it is not guaranteed that the rankings are commensurable. This pushes the problem up to the application level.

Alternatively, if the document classes are subtypes of a common ancestor, a query can be specified on that type, ensuring that a common term weighting scheme is being used.

7.3 Post-filtering Queries

There are many occasions which require filtering of a document collection based on non-probabilistic measures. Since we have an IR system coupled with a database, we have a natural ability to implement such filtering as database queries over the result sets of an IR query. If the implementation of the system provided a query optimiser, it would be possible to order the computation to achieve better execution times, either performing the relevance ranking before or after the filtering. Such techniques are beyond the scope of this work.

7.4 Query Expansion

As part of relevance feedback, query expansion requires that we should be able to extract features from a document. If the content-bearing features used to represent a document are meaningful to the user, we may use a standard collection type such as a set to collect together the terms. However, processes like stemming produce terms which may be disconcerting to a user. We therefore introduce a final data structure which acts rather like an inverted index, except that it pairs up indexing features with human-readable features. For example, the data structure could be used to group words by their stem. By exporting a mutator, the data structure can be used with the standard document representation tools so as to reflect the processes which have been carried out at indexing time.

8 An Architecture for IR

Figure 2 illustrates the layered architecture. Common to both indexing and searching are the low-level storage service and the object-oriented database layers. The database implements the Object Model discussed in 4.3. This is implemented on top of a storage service providing abstractions for file structures such as buddy files and disc block managers; these handle allocation, deallocation and growth of store. While simple, the low-level storage abstractions made it much easier to build more complex data structures, and avoided a great deal of code duplication.

The application data and content models are built on top of the database. An object definition language compiler is used to aid this process, generating code to handle storage and retrieval of application objects. At index time, framework modules may be used in the application program to handle document accession, that is, the input of data from external sources into the database. The application program will make use of query language modules, which in turn use retrieval model components. Finally user interface components help the application to communicate either with client software or with the user directly via user interface modules.

8.1 Application Data and Content Models

The application data and content models are defined in terms of the extended Object Model, and

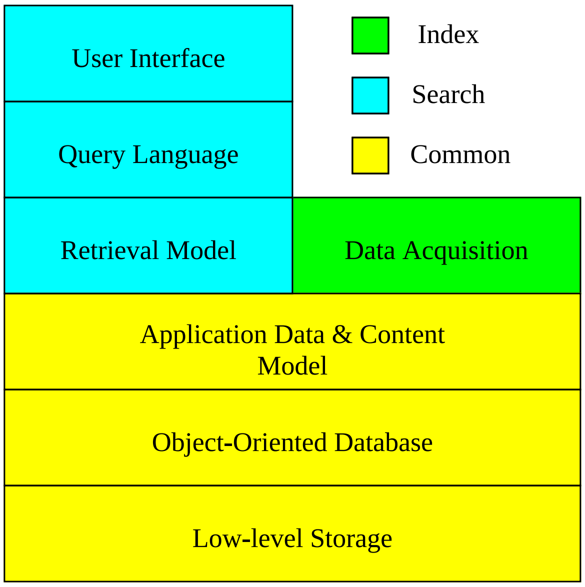

Figure 2: Architecture layers.

encode the semantics of the application. For IR applications, we wish to handle retrievable objects which we shall refer to as documents. These may be conventional textual documents or complex multimedia entities. Document objects represent the units of retrievable data, whose content model is described by the associated representations, tools and indices using the Object Definition Language.

8.2 Representations

Representations are built out of collection types (usually the feature vector), and supply methods to maintain the collection in a way which reflects its semantics. For example, a text representation might provide a method to insert a word with a given position into the collection.

If representations are to persist, they are declared as attributes of the document. Transient representations may be used internally to collect document feature statistics which are then placed into the type's indices. A number of representations may be used to reflect different aspects of the content of a document.

The Cobra implementation provides a number of ready built representations which further eases the content modelling process. These include text, numeric, field, date and colour representations. By taking advantage of the built-in representations, it is as easy to manipulate text in Cobra as in a database that supports text as a primitive datatype. Then if the designer wishes to refine the way in which the content is managed or introduce representations for new feature types, further representations can simply be added.

8.3 Tools

Tools are provided to refine representations. The set of tools is extensible to allow new functionality to be incorporated. Tools, as objects, exist independently of the document classes which reference them. This means that a tool can be shared between a number of document classes, which avoids the overhead of maintaining identical objects. When these objects contain a lot of state, such as a stopword list, this efficiency measure is a great benefit.

Cobra currently provides tools to perform stopword filtering, stemming (via the Porter algorithm) and vocabulary limitation (to restrict features to words from a fixed vocabulary), and also a WordNet tool (which provides an interface to WordNet [Mil95] with facilities to find the synonyms of a word). Since tools are independent, a tool can be replaced by an improved implementation exporting the same interface; any object which uses this tool will automatically benefit.

8.4 Indices

The Cobra implementation currently provides B*-Tree and R*-Tree index structures, implemented on top of the lower-level storage abstractions. The postings file associated with each structure closely follows the implementation in [BCC94].

Inverted indices present a serious design problem. We want to satisfy two conflicting goals, namely high speed indexing and instant availability of documents as they are added to the collection. In the database world, it is usual to expect that an update on a data structure will be available immediately within the context of that transaction. However, efficient implementation of an inverted index gains from batching updates. There is no simple solution, but we may argue that the typical patterns of access to inverted indices in information retrieval applications mean that we can afford to batch updates, providing that after commitment all updates are visible in order to maintain atomicity. Of course, in some applications (such as indexing a continuous news feed) this may not be a reasonable assumption. Ideally, multiple implementations of the inverted indices would be available, selectable at compile time, to balance both needs in accordance with the application.

8.5 Document Objects

Document objects represent retrievable information units within the database. Typically these have a one-to-one correspondence with the documents they represent, although there is no restriction enforcing this. For example, a book could be represented as a collection of document objects representing each chapter.

Document objects are defined using the attributes, relationships and operations of the ODMG Object Model. In addition, the tools, representations and indices used by the objects are also defined in the schema. Finally, the constraints which form the interface between the query language and the document class must be provided.

Tools, indices and language constraints are defined as properties of the type rather than of type instances. This means that all type instances share the same tools, indices and language constraints in the same way that they share extents in the standard ODMG model. Thus an index acts as the context for queries over the type.

Much of the coding effort for a specific IR system is concentrated on the implementation of the document class and language constraints. Firstly, parsing and/or lexical analysis of the document format is required. There are a number of processes which may require use of the document parser, including the document indexer, term extraction for relevance feedback, and perhaps a display parser which highlights query words matched in the document. Since the parser is a major piece of code, it is useful to reuse it where possible; so the following procedure was followed. The document parser is constructed as a skeleton. The skeleton has abstract methods which correspond to the actions of the parser. For example when a word is found, the parser calls a corresponding method. Concrete subclasses of the skeleton provide method bodies to perform the appropriate tasks.

Typically, the document indexer collects terms, placing them in type-specific representations. The representations are then transformed by the tools as required, before finally being inserted into the appropriate indices. Note that we have the freedom to apply a tool to an indexing feature *before* we insert it into the representation. This means that we can obtain the best performance possible. If a tool is expensive, it is best applied after the representation has been constructed, since it will be applied only once for each distinct feature that occurs in some document. The possible extra cost is that of having to merge the statistics of two or more features should they become identical under transformation. If the tool is cheap, we can apply it to each feature before insertion avoiding the overheads of merging feature statistics.

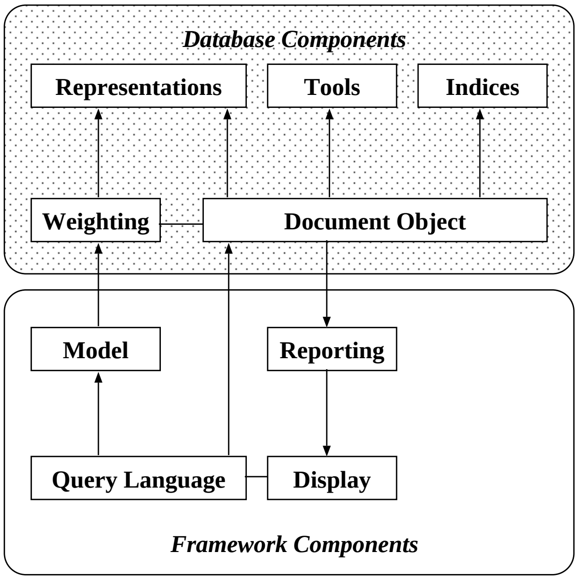

Figure 3: The application framework.

The other major coding effort is on the language constraints interface. This object contains the term query methods, performs term weighting and records a report about how the query proceeded. These are discussed later in section 8.13.

8.6 The Application Framework

The application framework aims to guide implementation effort and ensure that the components can be easily reused. This guidance is necessary because of the complex relationships between application components. At index time, we need to manage document accession and the document classes. Figure 3 illustrates the components used at query time, and shows the interfaces between the components.

The document object makes use of representations, tools and indices. This means that information about the content model is encapsulated in that object. Choice of a representation imposes restrictions on the term weighting functions which can be used. The weighting functions will require certain values, for example, the maximum term frequency. If that information is not available from the content representation, that function cannot be applied.

We have therefore chosen to place term weighting functions within the document object language constraints class. As part of the document object, it is privy to the knowledge about the internal representation. A number of common weighting functions are provided as standard. The constraints class thus provides functions to evaluate query language operands. For example, the Boolean query statement `(fish AND chips)` would have the operands `fish` and `chips` evaluated in the constraints class, while the `AND` operation is handled by the retrieval model module.

Another reason for handling operands in the constraints class is to ensure that a wide range of queries can be answered. For example, finding phrases may require the use of proximity information. This information is again located with the document representations. If proximity information is not available for a class, the operand could instead be evaluated as a strict Boolean `AND` operation. This flexibility would not otherwise be possible. A further example is field searching, for instance, matching against the document title. There are a number of ways this could be implemented. We could store `START` and `END` field tags, and so search for the words between 'title' tags. Another option would be to have a specific index for words found in the title. This variation can be managed as appropriate to each document type, rather than enforcing one particular solution.

Since the weighting function is often an integral part of the retrieval model, it is up to the application designer to ensure that the two are matched. Since they are separated, the designer is also free to experiment with substitution of one weighting function for another under the same retrieval model, or use of a new retrieval model with the same term weighting function.

The query language is to some extent governed by the retrieval model. If a desired language construct cannot be supported by the underlying retrieval model, we must select another retrieval model. Query languages are thus closely tied to a model, but one retrieval model may support several query languages so we make the distinction between them in the implementation. For example, simple and advanced query languages may map onto the same model, the former using a restricted set of model operators. Also, one query language may have instances which map onto distinct models.

The query language also has some influence over the user interface. If the query language expects to receive a string as input, a user interface which allows the user to traverse a virtual space filled with documents will be of little use. It is worth noting that query languages may have to be used in combination to elicit multiple query representations from the user. For example, an image retrieval system may rely on both a textual description (for matching against text associated with the images) as well as image data extracted from a sketchpad interface.

8.7 Document Accession

Document accession refers to the process of introducing new documents to the system. This functionality is somewhat akin to the bulk-load facilities of conventional databases. However, while bulk-load file formats are fixed by the database implementors, in the case of an IR system we should be able to deal with the file formats of those documents we wish to index. It would be unacceptable and impractical to insist that all documents are converted to some single intermediate language which Cobra is able to consume. Therefore we move the problem into the application framework and document classes. A document class is responsible for indexing the content of a particular document format, but this information must be located from external sources such as file systems or online document repositories. Given that Cobra may be used to supplement existing systems, such as a World Wide Web server, necessarily we must be able to obtain documents from these sources. Of course, *because* we have to be able to integrate with existing systems, we cannot provide a finite set of document acquisition modules. It is useful, however, to identify the roles that acquisition modules must perform, and to supply some common document acquisition facilities in an implementation.

Because we may not actually store documents in their entirety but rather keep references to external files, it will be necessary to ensure that the index and the external files are consistent. Thus the document accession modules are responsible for locating new documents and finding documents which have been updated. While the local file system is the obvious location for information to index, news feeds and World Wide Web sites are other common sources. The accession modules are responsible for keeping the index consistent with external sources when necessary.

A further possibility for file selection is that of file references within documents. For example, a web site manager may wish to index only those documents reachable from a given root. The root would be selected and parsed causing both indexing and extraction of links to further documents. These links can be fed back to the accession module as new roots.

8.8 Weighting Functions

Weighting functions are applied in the language constraints class for each document type. A variety of weighting functions are supplied as classes. For example, the class to provide the Combined Iterative Weight [RSJ94] gives a function `initial` which is used to weight terms in the absence of relevance information, `feedback` when it is available, and also a function `offer` to compute the Offer Weight, for use when selecting terms for relevance feedback. Additionally, any tuning parameters for the ranking function are encapsulated within these classes.

8.9 Retrieval Models

An essential feature of this architecture is that it supports the general IR paradigm without being committed to any particular retrieval model. This is because retrieval models are independent of the document storage model and are built within the application framework.

While term weighting is carried out by the weighting functions, the retrieval model components specify how the results of term weighting are combined. A strict Boolean retrieval model, for example, would supply the Boolean operators, ignoring the weights associated with the terms. Models such as the inference model [Tur91] provide a great many operators, which combine the term weights in a variety of ways.

8.10 Query Languages

Query languages are implemented as classes within the application framework. The language makes use of a retrieval model, but must also specify an abstract constraints class, concrete implementations of which will provide the interface between the query language and the document types. Cobra provides a number of general purpose query languages, including AltaVista and INQUERY clones.

8.11 Reporting

An important design feature of the Cobra retrieval system is the ability to report to the user about how a query proceeded. This is intended to enable the user to build up a mental model of the processes being carried out at query time.

At query time, the document query functions are passed a report data structure. Each query operand function can then update the report with information about that query term. For example, we might wish to record that a term was ignored because it was in the stopword file, or note the number of occurrences of the term in the collection.

The report structure is implemented as a collection of type-specific reports, that is, each document class uses its own report type, which serves to group reports by document class, and allows collection of class-specific information.

The final report is returned as the result of the query language execute function, and so the information is available for display by the application.

8.12 User Interface Management

The user interface is one area which may require a great deal of variation. For example, the search engine and search interface may be integrated into a single piece of software, and thus the user interface will be handled in this code. Alternatively, a client-server architecture might separate the user interface from the server; in that case we have two interfaces—the human-client interface and the client-server interface. It is this second interface for which we provide explicit software components, but we also support the first by ensuring that information suitable for display to the user can be amassed.

To aid the user's understanding of the system, we need to keep information about how the query proceeded. This information is available via the query report, and application code can present this information to the user.

The Cobra implementation provides a number of classes for handling the user interface, including interfaces to the command-line, the terminal, and the World Wide Web (both via the Common Gateway Interface and as an HTTP daemon). Finally a multi-threaded server serving a proprietary protocol allows new client interfaces to drive the search engine.

Each of these display modules has to manage input of a user's initial query, output of result lists and perhaps also output of the documents themselves. For retrieval engines exhibiting relevance feedback, the display module also needs to manage the selection of relevant documents and possibly the display of relevant terms.

Of course, there is no one set of methods which are enough to implement any user interface. More exotic user interfaces may require further information to be extracted or recomputed from the data, such as similarity coefficients between all pairs of documents.

8.13 The Query Process

A query formulated by the user is parsed to produce a data structure to perform the evaluation. This consists of operands which invoke methods on the document class language constraints object, and the operators of the query model which combine that information. Within the document class, the query features are transformed via the document class tools into features of the document representation(s), during which time a report is generated about the effects of tools on the features. The feature statistics are then retrieved from the indices and the term weighting function is applied. This generates a results list for that feature which is then combined with the other such lists by the query model operators to provide a final ranked list. The portion of this list required by the application (for example the first ten documents) is then made available as a list of document references ordered by rank.

If the query is further refined by relevance feedback a similar process is followed. Feedback may result in query expansion, in which terms from those documents judged relevant are added to the query, and query model reweighting, in which parameters of the query model and weights applied to terms may be modified.

At requery, the query parser is invoked with extra arguments, namely the list of relevant documents and the list of terms to be added to the query. The query language module is responsible for incorporating the new terms, perhaps using operators of the query model to weight feedback terms differently from user terms. The language constraints class is responsible for utilising the non-empty relevant document set by its choice of term weighting function.

9 Applying Cobra to the Web

Our first large-scale use of Cobra has been to provide web-based access to a historical document collection [MMR97]. In 1972 the Social Science Research Council initiated a project in which three communities would be examined with the aim of reconstructing as fully as possible the lives of individuals who lived in these selected areas [Mac89]. One of the places chosen was Earls Colne in Essex, England. The rich collection of historical records describing the village between 1400 and 1750 were transcribed, numbered and manually annotated, identifying each instance of an entity from a number of predefined categories [MHJ77]. These annotations included delimitation of dates, names, kinship relations, places and land transfers. Of particular importance to the users of these records are instances of people, places and dates.

The resulting transcriptions and indices derived from them were published in microfiche form. The plain text of the records comprises around thirty megabytes, while the manual annotation adds several megabytes more. Although in contemporary information retrieval terms this is not a large amount of data, for those who wish to study the material in detail the collection is of an intimidating size.

Although the data could have been indexed by an existing Web search engine, an approach that exploited the existing markup format was chosen so that stronger semantics could be expressed at search time. In addition to normal keyword matching, dates, people and land areas can be specified explicitly. This means, for example, that all references to a woman can be retrieved even if that woman has changed her name after marriage, a facility which is considered essential for this collection.

9.1 Browsing with Hypertext

The original form of the data was inconvenient for processing, since it contained duplicate information and the results of nominal record linkage were separated from the main text. We have combined the sources, marking them up according an SGML markup scheme. This unified format is stored as records in the OODB, and HTML is generated automatically at query time.

The data itself presents some interesting problems. Firstly, some of the original documents were damaged or partially illegible, so the meaning of some of the words has been lost. A great deal of work has been carried out by experts to determine the set of individuals referred to in the collection. Manual indexing of the references to these individuals has allowed this domain specific information to be encoded in the file format and exploited during hypertext and index generation. Problems alleviated by this include those of distinct individuals sharing the same name, and land names changing over time.

Hypertext links are defined as references between documents in the database. These are automatically generated from the markup information. Browsing is possible in sequential order, via references to individuals and land names, and chronologically.

9.2 Searching the Earls Colne Database

The search engine was constructed to take account of the various types of information represented in the document collection by composing standard Cobra components to define a new document class. Attributes hold the document title, record catalogue identifer and the document body. Text representations handle free text, individuals and plots of land while a date representation deals with dates. The tools which are applied to these representations include a stemmer and stopword filter for the free text, and a synonym tool for the name and land representations. As the records are added to the database, the document parser extracts the content-bearing features from the annotated text. The resulting search engine is a probabilistic system based on the inference model [TC91].

Since users would find it unacceptable to refer to individuals by identifiers rather than by name, the user interface is responsible for refining the query so that the assignment of names is made apparent to the user. During the parsing of a query, if the synonym tool finds an ambiguous name, the reporting mechanism is used to terminate the query processing and to present the possible options for the user to select. This is important, since the problem of deciding how to map names to individuals has no self-evident solution. Thus, if a user doubts the assignment of identifiers to individuals made by the (human) indexers, system behaviour can be adjusted during query refinement.

9.3 The Search Interface

Previous query interfaces to Cobra have been either through a command-line or based on HTML forms. These interface styles are inappropriate for an interactive retrieval system supporting advanced facilities like query reformulation. While HTML forms have been successfully used in Web search engines such as AltaVista, they are static and restricted by the stateless nature of HTTP: to change the display in any way the page has to be reloaded. We have therefore constructed a graphical user interface to the system with Java, giving us the flexibility to make it available as a normal application, or deliver it as a platform-independent applet over the World Wide Web. The interface uses a simple protocol to communicate with a Cobra server, which runs separately from the main Web server but on the same machine.

The interface provides active support for *relevance feedback*. From previous studies, it is evident that users often do not take advantage of relevance feedback facilities when they are available. Two reasons may account for this. Firstly, users who do not need *every* relevant document may consider that they have received an adequate number of useful results after an initial search. We believe that when searching the Earls Colne collection, expert historians and anthropologists *will* want to find all information relevant to their research; discussion with users of similar collections has confirmed this.

The second problem is that users may not have an adequate mental model of what relevance feedback is and how it affects their query. In the past it has been common for query reformulation to happen "behind the scenes", so that the user does not get to see what is going on. Following the work of Koenemann and Belkin (see section 2), our user interface has been developed to make the effects of feedback explicit.

This has helped to facilitate a session-based model of interaction, in which state is maintained at the Java client. We can also have more control over adapting responses to suit the user, and keep detailed session logs. These can be used to gauge the effectiveness of the query interface, and examine user search patterns.

The query interface is illustrated in figure 4. The user enters an initial query and is presented with a list of results. She can view a result in her Web browser by clicking on its title; the open book icon shows which of the results is currently being viewed. After looking through the document, she may decide to indicate that it is relevant to her query, which she does by clicking on the result.

This triggers the incremental query expansion: whenever a result is marked or unmarked as relevant, the list of suggested extra terms is updated. Unlike Koenemann and Belkin's interface, ours is able to display the (more natural) unstemmed terms, using the facilities introduced in section 7.4. The user has the option to remove those terms she does not wish to have added to her query, and can then submit an expanded query.

To remind the user of the documents she has picked out as being relevant, we have adopted the metaphor of a bookshelf: for each document selected, a book icon appears on the shelf. The size of the book gives an indication of the length of the document, and the colour can be used to classify it into an application-specific category. A document may be deselected by clicking on the result again, or on the book. Moving the mouse over a book shows the title of the document it represents.

The interface currently supports full-text query-

Figure 4: The Java query interface.

ing, including phrases. Due to the special markup of the Earls Colne data, we can also allow the user to explicitly search for dates, names, or lands. Currently this involves specifying an attribute-value pair in the query, such as "`land:Chalkney`". We plan to experiment with the provision of separate fields in the interface for querying according to these attributes. As described in section 9.2 we can ask the user for clarification in the case of an ambiguous person name or land name.

The interface can take advantage of Cobra's reporting facilities, discussed in section 8.11, to provide the user with a indication of the progress of her search, for example the number of results returned, and details of any stopwords which were removed from the query.

Our interface has an experimental query history facility, enabling the user to re-issue any of the queries from the current session. This provides some degree of reversibility, as well as reducing the effort required on the part of the user. The "back" and "forward" buttons allow sequential movement through the history, and a separate menu option affords direct access to any query. As well as the user's terms, we retain the context of each query, namely the "bookshelf" of relevant documents, the list of extra terms, and any reports generated when the query was originally issued. Capturing of context is advocated by Shneiderman, Byrd, and Croft [SBC97] in their recently published set of guidelines for the design of interfaces to text retrieval systems. Unfortunately, Java's current security restrictions make it impossible to store the history on the client side between sessions.

10 Conclusions and Future Work

Our first experiences with Cobra have been promising. The Cobra system is currently being extended to handle more media types, and more representations of the supported media types. Further practical tests are underway to build systems to handle multimedia collections, including a combined text and audio collection, and an image database.

We believe that user interaction will play a key role in building a useful IR engine, and we are anxious to develop a robust system structure with text data before carrying out experiments with other media. If a search query is posed against a collection that contains documents of more than one media type, then the matching function must combine index terms which have different semantics. This general area (data fusion) is an active research topic: experiments with real users are required in order to validate a selected fusion technique. A quite separate and equally difficult problem is to design the user interface for multimedia queries, in particular for handling relevance feedback.

There are two main benefits of the Cobra architecture. Its modular design makes it easy to experiment with the structure of IR engines. Secondly, the flexible nature of content modelling allows us to carry out experiments with multimedia documents as well as textual sources. Of course, there are still many lessons to be learnt as we expose our system to real users.

11 Acknowledgements

We acknowledge the UK EPSRC for supporting this work both through studentships and under grant GR/J42007. We are grateful to ICL for general support of our research group. Our colleague Jean Bacon has made valuable comments on this paper.

We would also like to thank Alan Macfarlane and Sarah Harrison for their work on the Earls Colne document collection and Persimmon IT Inc. for their contribution of both equipment and time that will allow us to establish the Earls Colne site on the Internet.

References

[BCC94] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th VLDB Conference*, 1994.

[BDPJ97] M. Beaulieu, T. Do, A. Payne, and S. Jones. ENQUIRE Okapi project. Research and Innovation Report 17, British Library, January 1997.

[Cat96] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1996. Release 1.2.

[CPH91] D. R. Cutting, J. O. Pederson, and P. Halvorson. An object-oriented archiecture for text retrieval. In *Proceedings of RIAO'91*, 1991.

[HC93] D. Haines and W. B. Croft. Relevance feedback and inference networks. In *Pro-*

ceedings of SIGIR'93, pages 2–11. ACM, 1993.

[HW92] D. J. Harper and A. D. M. Walker. An extensible class library for information retrieval. *The Computer Journal*, 35(3):256–267, 1992.

[KB96] J. Koenemann and N. J. Belkin. A case for interaction: A study of interactive information retrieval behavior and effectiveness. In *Proceedings of CHI'96*, pages 205–212. ACM, April 1996.

[Mac89] A. Macfarlane, editor. *The Origins of English Individualism.* Basil Blackwell, 1989.

[MHJ77] A. Macfarlane, S. Harrison, and C. Jardine, editors. *Reconstructing Historical Communities.* Cambridge University Press, 1977.

[Mil95] G. A. Miller. WordNet: A lexical database for English. *Communications of the ACM*, pages 39–41, November 1995.

[Mil97] T. J. Mills. *Content Modelling in Multimedia Information Retrieval Systems.* PhD thesis, University of Cambridge, 1997. In preparation.

[MMR97] T. J. Mills, K. Moody, and K. Rodden. Providing world wide access to historical sources. In *Proceedings of the Sixth International World Wide Web Conference*, pages 347–356, April 1997.

[Put91] S. Putz. Using a relational database for an inverted text index. Technical Report SSL-91-20, Xerox PARC, January 1991.

[RSJ94] S. E. Robertson and K. Sparck-Jones. Simple, proven approaches to text retrieval. Technical Report 356, University of Cambridge, November 1994.

[SBC97] B. Shneiderman, D. Byrd, and W. B. Croft. Clarifying search: A user-interface framework for text searches. *D-Lib Magazine*, January 1997. `http://www.dlib.org/`.

[SF95] G. Sonnenberger and H. Frei. Design of a reusable IR framework. In *Proceedings of SIGIR'95*, pages 48–57, 1995.

[TC91] H. Turtle and W. B. Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, 9(3):187–222, 1991.

[Tur91] H. R. Turtle. *Inference Networks for Document Retrieval.* PhD thesis, University of Massachusetts, Amherst, February 1991.

[vR79] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, 2nd edition, 1979.