MARIAN Design

Robert K. France, Ben E. Cline, Edward A. Fox
Virginia Tech Computing Center
Information Systems Building
1700 Pratt Drive
Blacksburg VA 24061

CONTACT: Robert France: france@vt.edu

v.5 (MARIAN v.1.2)
14 Feb. 1995

NOTE:

This paper version is a linearization of a Diagram-2 hypermedia document. Thus it may be difficult to assemble the pieces. The order of pages is as follows:

Detailed Design Overview	1 page
Key	1 Page
Data Structures	2 Pages
Module descriptions	7 Pages
Call descriptions	13 Pages

To this have been appended design documents on the Feedback Query Synthesizer and underlying databases, and a task transition diagram.

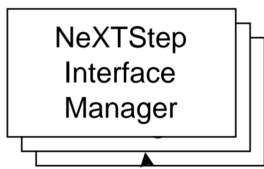
Introduction

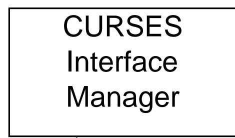
MARIAN (Multiple Access Retrieval of library Information with ANotations) is an on-loine library catalog information system. Intended for library end-users rather than catalogers, it provides the following features:

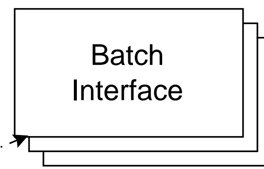
- Controlled search by author, subject entry, and imprint.
- Keyword search by title, subject, and other MARC text fields.
- Feedback, locating the closest books to a relevant book or books.
- User annotations of books.

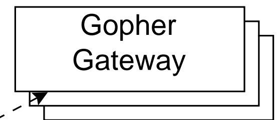
MARIAN: Detailed design

Interface Management Layer

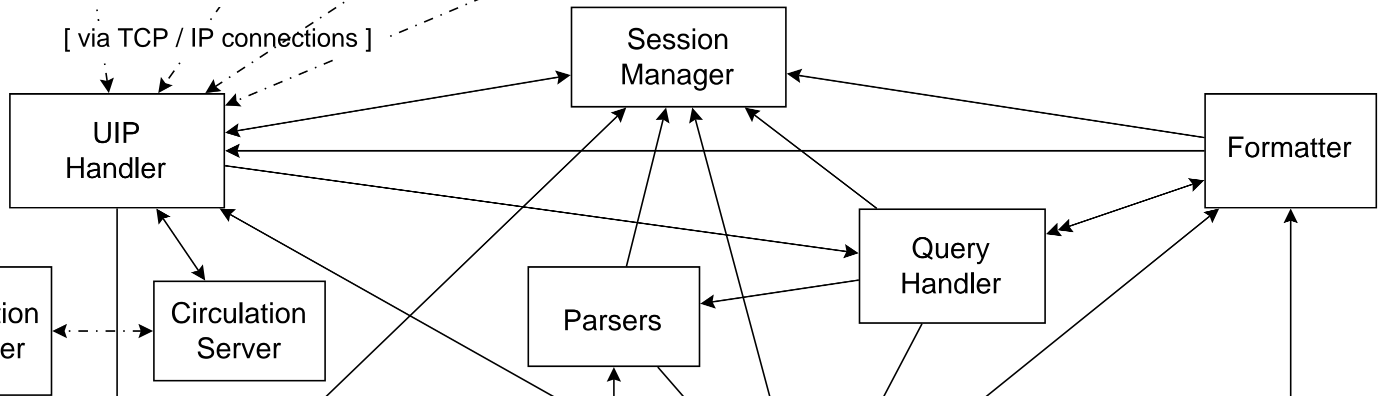




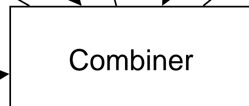


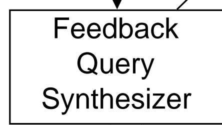


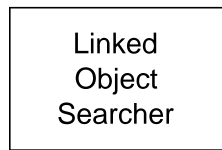
Session Management Layer

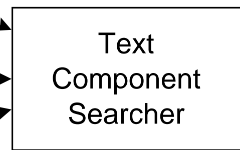


Access Method Layer

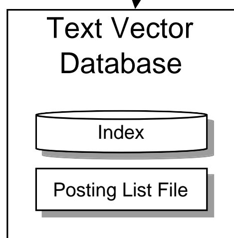


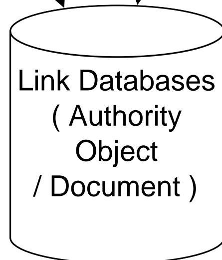


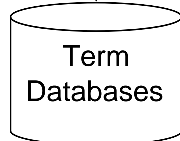


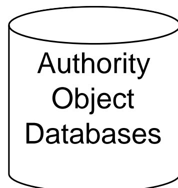


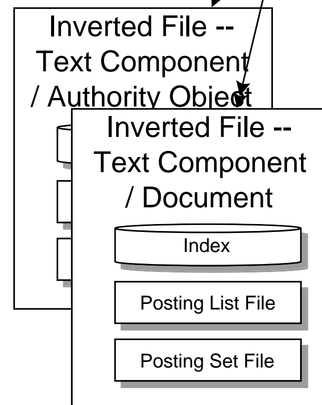
Data Management Layer

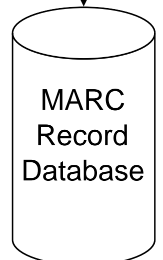








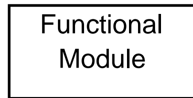




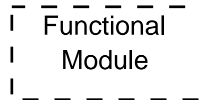
Key **Data Structures** **Database Classes**

Robert France
14 Feb. 95
v. 5 (MARIAN v.1.2)

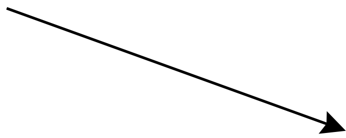
Key



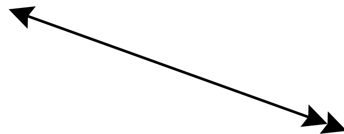
Computational object, with methods detailed by the messages that are sent to it.



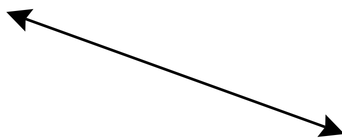
Computational object projected but not part of initial system.



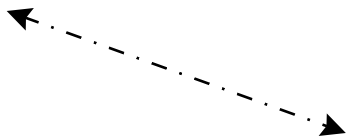
RPC (MiG) Message passed; no reply expected. (Asynchronous call. In CODER, this is called an inform() communication.)



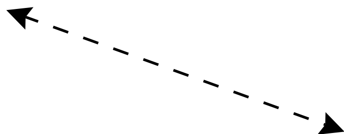
RPC (MiG) Message passed with reply. (Synchronous call. In CODER, this is called an ask() communication.) The module at the single-arrow end is the initiator.



RPC (MiG) Message passed without reply expected in one direction at one time, in the other direction at another time. (Two asynchonous calls. Sometimes this is a conversation, others not.)



Messages passed through asynchronous TCP / IP connection.



Messages passed through popen.

Diagram™ link. Links on message lines detail what messages are passed; links on computational objects detail their internal structure.

Data Structures

Any object in the system can be represented by a "Full" ID, composed of (the ID of) its class, plus an Instance ID distinguishing it uniquely among other instances of the class. This representation is particularly useful for documents, but is also used for term components, unique names, and so forth:

```
typedef struct
  {
   unsigned short ClID;
   unsigned long InstID;
  } FullIDType;
```

A "query vector" is a set of implicitly ANDed vectors, each of which are implicitly ORed sets of term IDs. "Term" in this context refers to any of: an author object, a subject object, and English root object, a number or string object, ...

```
typedef struct
  {
   int Size;            /* Number of elements in:                  */
   struct
     {
      int Size;
      struct
        {
         IDType     ID;
         int        Weight;
        } *Comp;
     } *Vect;
  } QueryVectType;
```

A "document vector" is a ranked list of document (ID)s. Each document is accompanied by a real number detailing its relevance to the query that found it, and a set of regions in the document that serve as evidence that it really is relevant to the query.

```
typedef struct
  {
   int Size;            /* Number of elements in:                  */
   struct
     {
      IDType       ID;
      float        Rel;
```

```
     F3ITermType Evidence;   /* What parts of the document are relevant.       */
    } *Vect;
  } RetVectType;
```

A "choice" is an object described two ways: an ID for internal system use together with a desciption fit to display to a human being.

```
typedef struct
  {
   IDType ID;
   char *Str;
  } ChoiceType
```

Authority Object Databases

We use the term "authority object" to refer to any object assigned by catalogers out of some controlled set of entities. Thus subject descriptors are authority objects since they are (mostly) drawn from the controlled *LC Subject Calegories,* and authors are controlled since they are supposed to be drawn form authority files for personal and corporate names.

At this point, no attempt is being made to analyze these objects in any way appropriate to their classes. They are being run through specialized text parsers, producing inverted files of their components, which provides a means of access to them by something other than exact string match. Then the exact strings are saved in the authority object databases. Some day something better should be done with such classes as dates and persons' names. Dates and names should be analyzed to reveal their proper parts, and at minimum LCSH headings should be differentiated from modifiers, which should be differentiated from non-LCSH subjects made up on the spot by catalogers. We will still, though, want the functions below even when more detail has been revealed.

The authority object databases are initialized and shut down with the calls:

```
int
initAuthDatabase(+IDType AuthClassID)

int
quitAuthDatabase(+IDType AuthClassID)
```

This is a typical string object class, and so the two typical retrieval calls are provided:

```
int
idToText(+IDType AuthClassID, +IDType InstID, -char** String)

int
textToID(+IDType AuthClassID, +char* String, -IDType* InstID)
```

If the first parameter is not in the database in either case, the functions should return NOT_FOUND.

-- RKF 8-26-92

Inverted File Databases

These are databases that link text components – words, names, various sorts of numbers, and unanalyzed srtings – to the objects that contain them. The MARC objects have two sorts of inverted file databases, one for the components of the text in the titles field(s) of a record, and one for the components of any sort of note field. In addition, there are inverted file databases for the text strings that identify each of the name objects that occur in author fields, and for the text of unique subject fields.

Each inverted file database has three parts. There is the inversion class proper, which is a real indexed class of objects, and there are two data files accessed by offset and length. The inversion class, as the only indexed class, is the only one that is stored in LEND. It maps text component IDs to one of three disjoint classes of objects, depending on the frequency with which the text component appears in the collection. For text components that only occur once, the inversion class returns the object ID that it occurs in and the weight with which it is associated. For text components that occur a few times, the inversion class returns the offset and length of a vector of <objID, weight> pairs in the posting list file. For components that occur many times, information is returned that allows a bit mask to be constructed. Combinations of these masks are then used to index the bit mask file.

The inversion databases is initialized and shut down with the calls:

```
int
initInvDatabase(+IDType TextClassID)

int quitInvDatabase(+IDType TextClassID)
```

The retrieval call is simply:

```
int
termIDToInvData(+IDType TextClassID, +F3lFullIDType FullID,
                     -int* InvType, -invDataType* InvData);
```

where InvType is one of the defined constants ONE, FEW, and MANY, and InvData is the union:

```
 typedef union          // Inversion data.  There are three types of records
   {                    //  that can be found in the inversion file database
```

```
 struct            //  ("dictionary"):
   {
    IDType ID;       // Records for terms that occur only
    weightType Wt;   //  ONE time in the collection have their posting
   } One;            //  written directly into the inversion database.
 struct
   {
    long Offset;     // Records for terms that occur only a FEW times
    int  Size;       //  in the collection have their postings recorded
   } Few;            //  explicitly in a file; Offset and Size show where.
 struct
   {
    int BitPos;      // Records for terms that occur MANY times are only
    int FirstBlock;  //  recorded in combinations.  The combinations are
   } Many;           //  accessed by bit vectors where each term is assigned
} invDataType;       //  a unique bit position; FirstBlock is where to start.
```

-- RKF 8-19-92

Link Database for hasAuthor and hasSubject Relations

These are databases holding binary relations that link MARC record objects to the person and subject objects that have been found to occur within them. In the current design, the relations are simply binary links, and may well be considered LEND-style arcs. It may prove useful to add to them at a later time, however, including such information as

- MARC field number and subfield label
- weight of association

that will need to be returned with the object linked to.

Each link databases is initialized and shut down with the calls:

```
int
initLinkDatabase(+IDType LinkClassID)

int
quitLinkDatabase(+IDType LinkClassID)
```

There are two retrieval calls: one to map an authority object to the MARC record containing it, and one to map a MARC record to the associated authority objects:

```
int
authIDToMarcIDVect(+IDType LinkClassID, +IDType AuthorityInstID,
                   -IDType** MarcInstIDVect, -int* NumMarcs)

int
marcIDToAuthIDVect(+IDType LinkClassID, +IDType MarcInstID,
                   -IDType** AuthInstIDVect, -int* NumAuths)
```

-- RKF 8-26-92

MARC Record Database

The basic unit for communicating library information is the MA chine-Readable Catlaoging record. In the abstract, a MARC record is a two-level record structure with some additional feature information. In other words, a MARC record has information on features like its language, level of cataloging, and so forth, and an aribtrary subset of a huge set of distinguished fields, each of which contains a subset of possible subfields. This standard is defined by the Libary of Congress, acting in concert with similar bodies form other nations. LC also defines a canonical way of linearizing MARC records, called the *MARC Tape Format*. All records interchanged between library systems are sent in MARC tape format. An early MARIAN design decision was to store the records held by MARIAN in tape format, and to reformat them at the time of presentation to the user. Since MARC records are rather large entities, it was further decided to store them in compresssed form.

The MARC record database stores these records, accessing them by a unique MARC object ID generated during document analysis, and used by the various linking databases to refer to the record. The records are stored in compressed form, and decompressed at the time they are asked for. One should note, though, that the methods of this clas make no mention of compression, so this profile would be valid if compression were not used.

The MARC record databases are initialized and shut down with the calls:

int
initMarcDatabase(+IDType ClassID)

int
quitMarcDatabase(+IDType ClassID)

A classname is provided for initialization even though at this point all MARC records are considered to form a single class. This allows for expansion to a MARIAN system with multiple card catalogs supported.

The retrieval call maps the compressed string into a stream of ASCII/ANSEL characters – the MARC record in tape format – stored in a buffer:

int
marcInstIDToString(+IDType ClassID, +IDType InstID, -char** Buffer)

-- RKF 8-26-92

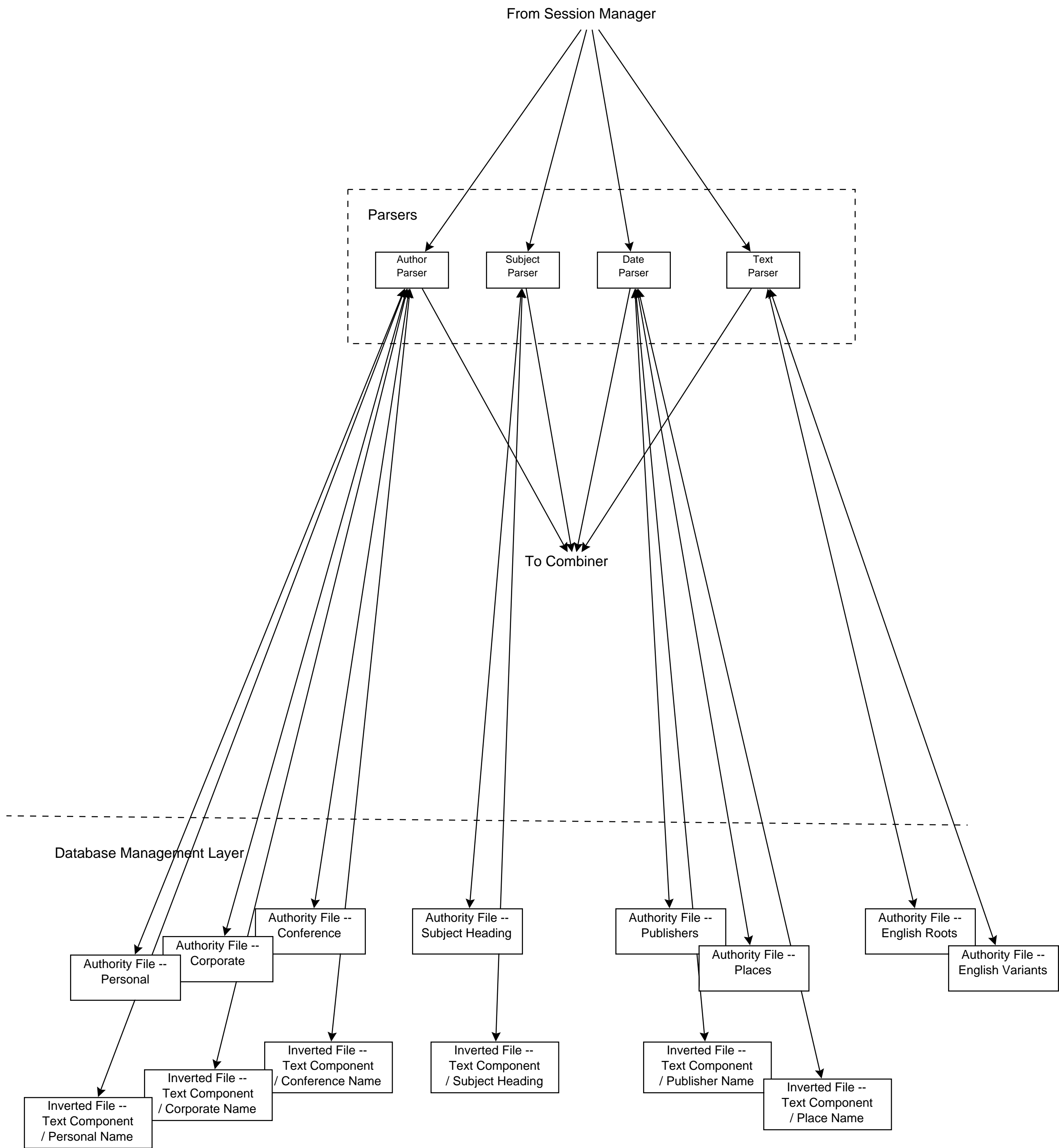
General Notes on the Parser Complex

The job of a parser in MARIAN is to translate the user's representation of an object to a representation that will be useful by the MARIAN searchers. Thus far, we assume that users always represent the objects they are searching for by linear sequences of characters, usually making up sequences of word and number tokens. On the other side, MARIAN searchers always expect objects of the QueryVectType (q.v.). Gettting from one to the other, however, can be a complex task.

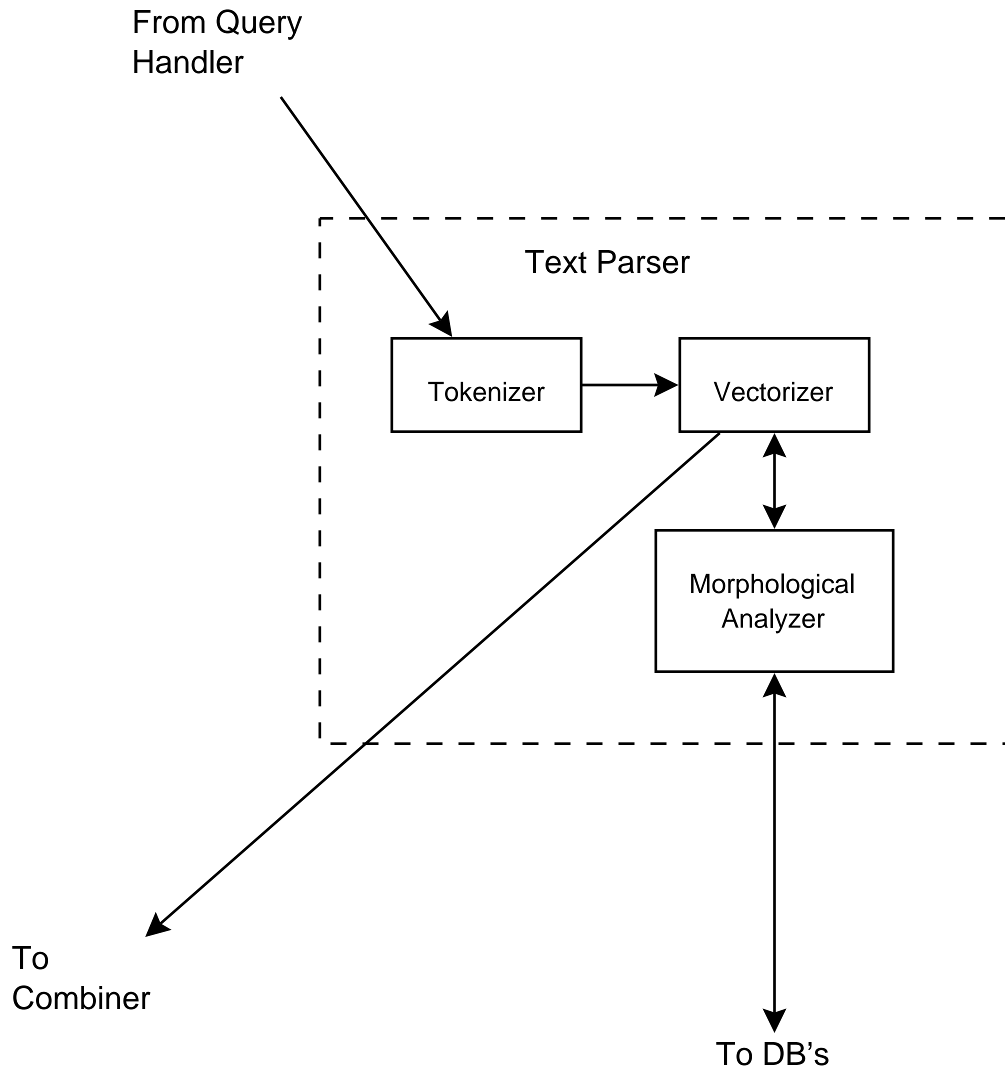
MARIAN uses different parsers to treat fields which are known to contain text representations of objects of different semantics. The most obvious example is the **date** field of a bibliographic query, which is expected to contain (the character string representation of) either a single date or a date range. The Date Parser is specialized to extract these objects from user representations. Similarly, the Subject and Author parsers treat **subject** fields and **author** fields appropriately to their type. The default is the Text Parser, which treats **title**, **note**, and any other undifferentiated field as English text.

The Text Parser produces the simplest QueryVectType object: a flat vector of termIDs, each with an associated weight. The other parsers make more subtle uses of this type. The Date Parser produces a single sequence of termIDs for a single date and a pair (beginning and end) for a date range. the Subject Parser also produces a pair of objects, the first of which is a vector representation of the subject description and the second of which details the exact order of the components. The Author Parser produces a set of pairs, each of which is in form like those produced by the Subject Parser, and each of whcihc corresponds to one name in the **author** field.

-- RKF 7-24-92


From Session Manager
Parsers
Author Parser
Subject Parser
Date Parser
Text Parser
To Combiner
Database Management Layer
Authority File -- Personal
Authority File -- Corporate
Authority File -- Conference
Authority File -- Subject Heading
Authority File -- Publishers
Authority File -- Places
Authority File -- English Roots
Authority File -- English Variants
Inverted File -- Text Component / Personal Name
Inverted File -- Text Component / Corporate Name
Inverted File -- Text Component / Conference Name
Inverted File -- Text Component / Subject Heading
Inverted File -- Text Component / Publisher Name
Inverted File -- Text Component / Place Name

Text Parser



Query Handler

The Query Handler brokers the creation and modification of queries. Informed by the UIP handler whenever the user requests a new query, it has the responsibility to choose an interaction object to carry that query, label it and send it to the user via the UIP handler. When the user has filled the query object in to hi/r own satisfaction, the object is shipped back to the Query Handler, which then divides up the object into component pieces and passes the pieces to appropriate parsers. Similarly, when the user initiates a browse request, the Query Handler chooses the point at which the browse should begin, and mediates the user's navigation as the browse progresses.

When the Query Handler receives a completed form query such as a bibliographic form query object from the UIP Handler, it sends several messages simultaneously to several different recipients. For each field that the user has filled in the query, a queryToken() message is sent to the appropriate parser or parsers. For instance, if the user makes use of the "Author(s)" field of a biblioFormQuery, the text that the user has created is sent to the author parser. If the user has filled in a text field with coverage "Title + Subject", the text is sent to both title and subject parsers. At the same time, a newQuery() message is sent to the Combiner, so that it can know when it has received all the parsed fields. The number of fields reported in the newQuery() call is the total nuhmber of messages sent, not the number of unique text strings that hte user has created. When this is completed, it also sends a newQuery() message to the Session Manager.

When the Query Handler receives a browse point query, it passes the text to the appropriate parser. Again, it notifies the session manager of its action.

The Query Handler keeps track of the query history of each session. As each new query is received, it is saved in the form in which it was created. The user is kept informed of h/ir past queries through a choiceList object. Thus, the user has the ability to call back an earlier query by choosing an item from the list. When that happens, the Query Handler creates a dublicate of the query, labels it so as to reflect its derivation, and returns it to the user, who can then either re-submit it or edit it to create a related form.

Similarly, feedback queries pass through the Query Handler on their way to the Feedback Query Synthesizer, even though the Query Handler does not need to break them into components. As they pass through, the Query Handler labels them and includes them in the query history for the user's later use.

Thus, each query of each session had two unique attirbutes: it's Query ID number, which is passed throughout the MARIAN back end, and its label, which is a character string that we hope means something to the user. The label is

available to the remainder of the MARIAN backend through the final call provided by the Query Handler, findTitleByQID().

-- RKF 7-24-92

The Session Manager

The Session Manager oversees the progress of all currently active sessions, and of all major activities within each session. It maintains information on which sessions are currently active, and handles graceful termination of all internal processing associated with the session when it is terminated.

Much error handling passes through the Session Manager. Specifically, the Session Manager watches for any failure of the lower layers of MARIAN to function properly, and handles recovery where possible and/or reporting to the user when desirable. Conversely, when a MARIAN User Interface Manager dies or loses contact with MARIAN proper, the UIP Handler calls the Session Manager, and the Session Manager resolves any processing associated with the lost session.

The Session Manager maintains a representation of each search, browse, or other major task associated with an active session, detailing the progress of the task. It responds to user requests for status by explaining the progress of each of the (active) tasks in the session. It may monitor this progress, either continually or whenever the user requests status, to detect excessive holdups in processing that may be indicative of unreported failure in one of the other modules.

In order that this information be gathered, all MARIAN modules report back to the Session Manager at key points during processing. This reporting is done through a common set of calls on the Session Manager used by all other MARIAN modules:

```
int
start_processing(+int SessionID, +int ModuleID, +int FuncCode,
                 +int QID)

int
end_processing(+int SessionID, +int ModuleID, +int FuncCode,
               +int QID)

int
error_for_sm(+int SessionID, +int ModuleID, +int ErrorCode,
             +char* Message)
```

In all the above, ModuleID is a code for the task making the call, defined by a set of constants in "marian.h". FuncCode, defined similarly, identifies the type of major task that the call is part of.

All user identifiaction and accountatbility tracing is performed by the Session

Manager. Thus it is the Session Manager that handles the login process, that performs the new user registration dialog, and that brokers any permissions that the user may require for certain operations (e.g., remote printing, publishing or reading annotations, sending messages to Library Circulation, and so forth). It is also the Session Manager that stores user preferences and communicates them to those modules requiring them.

All **Help** requests pass through the Session Manager. It is expected that almost all of these will evoke canned help, but upon occassion, some help information may need to be synthesized dynamically. The Session Manager is the unique MARIAN module with the information to do this.

It is possible that future versions of MARIAN may include some degree of adaptive or deductive user modelling. If this happens, much of this functionality will be localized in the Session Manager.

-- RKF 7-27-92

Text Component Databases

There are several of these. Some are simple static string databases for different types of enitites, such as English word roots or name components. One is an extensible string database for uncategorizable strings. And one is a relational database mapping English word variants to their associated word senses, together with their variant type and part of speech.

The string databases are initialized and shut down with the calls:

```
int
initStringDatabase(+char* ClassName)
```

```
int
quitStringDatabase()
```

They all implement the two retrieval calls:

```
int
idToText(+IDType InstID, -char** String)
```

```
int
textToID(+char* String, -IDType* InstID)
```

If the first parameter is not in the database in either case, the functions should return NOT_FOUND. The dynamic "newString" database also implments:

```
int
addText(+char* String, -IDType* InstID)
```

The variant database is initialized and shut down with:

```
int
initVarDatabase()
```

```
int
quitVarDatabase()
```

Only one direction of retrieval is supported:

```
int
```

variantToSense(+char* String, -IDType* SenseID, -char* Transform,
-char** PartOfSpeech)

(Transform is actually a single character: if these specifications were in C++, the argument would be declared char& Transform. PartOfSpeech might well be changed to an int*, but it is currently stored as an atom, so we're keeping it as it is.)

-- RKF 8-19-92

User Interaction Protocol (UIP) Handler

The UIP Handler is the interface between the various user interface managers and the rest of the system. All of the user interface programs, whether they are single-terminal processes like the X-Windows or VM Interface Mangers or whether they are local terminal servers like the CURSES Handler, communicate with the rest of the system through the CODER/MARIAN User Interaction Protocol. The UIP, documented in *User Interaction Objects for CODER, INCARD, and MARIAN*, v.2.4 or higher, is a set of classes of "interaction objects" such as text display objects and choice prompt objects defined by standard calls both from the UIP Handler to the interface managers and from the interfaces to the UIP Handler.

UIP calls are classes by the type of interaction and the type of data object involved. However, a single class of UIP objects (for instance, choice list display objects) may be used for several purposes in the system. It is the job of the UIP Handler to differentiate among these different uses and to pass user actions (for instance, making a choice from a list) to the appropriate module in the remainder of MARIAN. In most cases, the UIP Handler passes user actions back to the module that originated the interaction object. Other user actions may be re-directed to a spceific destination, as in when the user selects an action from the main menu.

The calls that UIP provides to the remainder of the MARIAN back end are precisely the UIP calls that back end modules use with two changes: the call name has an "M_" prepended to it, and an "int SessionID" parameter is added at the beginning of each parameter list. At the date below, they are:

```
M_showInformation(int session, char *Message);
M_solicitBiblioQuery(int session, char *Title, char *AuthorStr,
                     int Text1Coverage, char *Text1Str,
                     int Text2Coverage, char *Text2Str,
                     int Text3Coverage, char *Text3Str, char *DateStr);
M_showRetrievalColl(int session,char *Title,
                    UserRetrVect x[], int NumRetr, Bool MoreToCome);
M_addToRetrievalColl(int session,char *Title,
                     UserRetrVect x[], int NumRetr, Bool MoreToCome);
M_showChoiceList(int session, char *Title, char *Choices[], int numChoices,
                 int selected);
M_addToChoiceList(int session, char *Title, char *Choices[], int numChoices);
M_clear(int session,char *Title);
M_goAway(int session,char *Title);
```

RKF 7-22-92

Calls Combiner --> Formatter

The combination of all the query parts results in a vector of document IDs. the top K elements of this vector (those with the highest relevance to the query) are passed to the formatter using the method:

```
int
topKDocs(+int SessionID, +int QID, + IDVectType DocIDVect,
              +Bool MoreToCome)
```

additional batches of elements from the vector are passed using the corresponding:

```
int
nextKDocs(+int SessionID, +int QID, + IDVectType DocIDVect
               +Bool MoreToCome)
```

In either case, MoreTocome is TRUE iff there are additional elements in the vector that have not yet been passed.

In some cases only a single document may match the user’s query. In that case, a specific call can save much of the overhead of topKDocs():

```
int
docChoice(+int SessionID, +int QID, +FullIDType DocID)
```

-- RKF 7-27-92

Calls Combiner -->Authority Object Searchers

The calls made on the various Authority Object Searchers by the Combiner are syntactically identical to those made on the text searcher. They differ significantly in the semantics and form of the QVectType objects passed in, and the use to which they are put. This discussion should therefore be considered as a generalization of the discussion in the Combiner <--> Text Searcher link.

The Combiner calls an Authority Object Searcher to identify the object, using the RETURNING call:

int
queryVectToID(+QueryVectType* QVect, -int QVectID, -int LengthExplored)

Once obtained, the QVectID can be used to obtain a portion of the document vector:

int
findDocVectSegment(+int QVectID, +int NumToSkip, +int NumToReturn)

This call starts the process that results in a docVectSegment() call being passed back to the Combiner (see below).

An additional call quickly estimates the total size of the document vector that would be retrieved by a query:

int
findApproxDocVectSize(+int QVectID)

This results in a docVectSize() return call (see below).

Calls Authority Object Searchers --> Combiner

Each Authority Object Searcher only replies to the Combiner when spoken to. For each findDocVectSegment() call, it provides one response:

int
docVectSegment(+int QVectID, +RetrVectType* RetVect)

and for each findApproxDocVectSize() call, it provides one response:

```
int
approxDocVectSize(+int QVectID, +int Size)
```

Finally, when a QueryVector object is released, the Text Searcher informs the Combiner that a given QVectID is no longer valid:

```
int
releaseQVectID(+int QVectID)
```

-- RKF 7-27-92

Calls Combiner --> Text Searcher

The Combiner calls the Text Searcher in order to find some portion of the list of documents matching the vector representation of a piece of text. These lists are always returned in decreasing order of similarity, so it is possible to get, for instance, the first 20, followed by the next 20, and so forth, without documents being either duplicated or omitted. Each vector representation is treated as an object independent of the session or query that it comes from. The first step for the Combiner, therefore, is to identify the object, using the RETURNING call:

```
int
queryVectToID(+int TextClassID, +QueryVectType* QVect,
                +int ApproxNumNeeded, +weightType MinSimNeeded,
                 -int* QVectID, -int* ApproxNumMatching, -int* NumReady)
```

The Combiner provides guesses as to how many postings it will require and/or the minimum similarity it is interested in. The Text Searcher in turn reports the number postings currently available and a guess at the total number of postings that have non-zero similarity to the query. NumReady is a measure of how much work has been done in determining the document vector: specifically, it is the number of top documents that have been established with certainty. The first time the Text Searcher sees a unique QVect, NumReady will necessarily be 0. If several queries make use of the same texts in close succession, however, NumReady may be progressively higher for later queries.

> NOTE: This call is the exact analog of the call sent by the Combiner to the Authority Searcher(s). The Text Searcher differs from the Authority Searchers, though, in searching only flat vectors. Thus, it expects QVect to always have exactly one component. Given this, it may be reasonable at some point to adapt the call to take a QueryVect, rather than a QueryVectType.

Once obtained, the QVectID can be used to obtain a portion of the document vector:

```
int
getRetrVectSegment(+int QVectID, +int NumToSkip, +int NumToReturn,
                      -RetrVectType* Segment)
```

An additional call quickly estimates the total size of the document vector that would be retrieved by a query:

```
int
```

findApproxRetrVectSize(+int QVectID, -int* Size)

Finally, the combiner reports to the Text Searcher whenever it ceases to be interested in a QueryVector:

int
releaseQVectID(+int QVectID)

Calls Text Searcher --> Combiner

Should a QueryVector object need to be released by the Text Searcher before the Combiner is done with it (e.g., in case of cache overflow), the Text Searcher informs the Combiner that its QVectID is no longer valid:

int
releasingQVectID(+IDType ClassID, +int QVectID)

The ClassID here is intended to uniquely determine the Searcher that produced the ID. We have yet to see how well this works in the Combiner.

Note

In MARIAN v.0.75-1.0, getRetrVectSample() is implemented as a synchornous call with return value. It would increase interleaving and possibly speed up processing to implement it as a pair of asynchrous call and call back. Doing this would complicate the current design of the Combiner beyond measure, so it should not be done unless necessary, but if so, it might be done this way: Two Combiner calls on the Text Searcher would be re-written:

int
getRetrVectSegment(+int QVectID, +int NumToSkip, +int NumToReturn)

int
findApproxDocVectSize(+int QVectID, -int* Size)

The Text Searcher would then reply to the Combiner only when spoken to. For each findDocVectSegment() call, it would provide one response:

```
int
retrVectSegment(+int QVectID, +int NumToSkip, +RetrVectType* Segment)
```

and for each findApproxDocVectSize() call, it provides one response:

```
int
approxDocVectSize(+int QVectID, +int Size)
```

-- RKF 10-9-92

Calls Formatter --> Query Handler

When the Formatter builds a user-undetstandable title for a retrSetDisplay window, it needs to translate the abstract query number into an ASCII form that has meaning to the user. For that it uses the RETURNING calls:

```
 char*
M_findTitleByQID(+int SessionID, +int QID).

 char*
M_findLongTitleByQID(+int SessionID, +int QID).
```

The first call returns only the query history number (e.g., "Query 1.2.1"). The second returns the number plus a small amount of identifying text ("Query 1.2.1: Title: cat dog, Author: Piermanetti").

-- RKF 7-27-92
updated 2-14-95

Calls Formatter --> UIP Handler

The calls used by the Formatter are just those required to manipulate retrSetDisplay objects:

```
int
M_showRetrievalColl(+int SessionID, +char* Title, +UserRetrVect DocList[],
                    +int DocListSize, +Bool MoreToCome)

int
M_addToRetrievalColl(+int SessionID, +char* Title, +UserRetrVect DocList[],
                     +int DocListSize, +Bool MoreToCome)
```

SessionID and MoreToCome are set from the motivating calls; Title is built from the QueryID number. DocList is built out of the DocID lists passed in by the Combiner.

-- RKF 7-27-92

Calls Parsers --> Combiner

```
int
queryTerms(+int SessionID, +int QID, +int FieldID,
           +QueryVectType TermIDVect)
```

The first three arguments are inherited directly from the queryToken() call that motivates this one; the final argument is a (parsed) representation of the field in FieldID for <SessionID, QueryID>. The internal structure of TemrIDVect varies among different types of fields (see general notes on Parsers).

-- RKF 7-24-92

Calls Query Handler --> Combiner

Simultaneously with sending the query fields to the parsers, the Query Handler sends a message to the Combiner to tell it how many fields to expect. This message also carries the maximum number of matches to send the user in the first batch.

int
newQuery(+int SessionID, +int QID, +int NumFields, +int NumToReturn)

The Query Handler also passes along requests to extend existing retrieval sets:

int
nextKDocs(+int SessionID, +int QID, +int NumToReturn)

> NOTE: this is just the M_nextKDocs() call with object title mapped to <SessionID, QID>.

-- RKF 7-24-92

Calls Query Handler --> Parsers

All calls from the Query Handler to the various parsers are in the form of messages without reply:

```
int
queryToken(+int SessionID, +int QID, +int FieldID, +char *String)
```

-- RKF 7-24-92

Calls Combiner --> UIP Handler

The Combiner has occasion to talk (more or less) directly to the user in two sorts fo circumstances. First, when a query produces absolutely no retrievals at all, it may send messages to the user:

```
kern_return_t M_showInformation(int SessionID, char *Message);
```

Second, when a (partial description of an) authority object in a user's query matches too many objects in the authority object file, and these in turn match too many works, the combiner may use the choiceListDisplay interaction object to solicit a smaller subset from the user:

```
kern_return_t M_showChoiceList(int session,char *Title,char *Choices[],
                        int numChoices,int selected);
kern_return_t M_goAway(int session,char *Title);
```

Calls UIP Handler --> Combiner

The UIP Handler has no need to reply to an informationDisplay interaction. In the second case above, though, it must pass back the choice(s) that the user has made from the objects offered by the Combiner:

```
  kern_return_t M_userChoice(int SessionID, char *Title,
                                    char *ChoiceVect[], int NumChoices);
```

-- RKF 7-24-92

Calls UIP Handler --> Query Handler

The UIP Handler sends the Query Handler requests to originate new queries:

```
  kern_return_t M_searchCollection(int SessionID, char *CollectionName);
  kern_return_t M_searchText(int SessionID, char *Title, char *Text);
```

These stimulate a dialog that includes the opening and maintenance of user interaction objects for the queries themselves, and for the query history (see below). Replies to these queries are motivated by further user actions:

```
  kern_return_t M_biblioQueryText(int SessionID, char *Title, char *AuthorStr,
                                  int Coverage1, char* Text1,
                                  int Coverage2, char* Text2,
                                  int Coverage3, char* Text3,
                                  char* DateStr, int MaxNumToFind);
  kern_return_t M_userChoice(int SessionID, char *Title, char *Choice);
```

Finally, faute de mieux, the UIP handler passes requests for query continuations through the Query Handler:

```
  kern_return_t M_showNextK(int SessionID, char *Title, int MaxNumToAdd);
```

Calls Query Handler --> UIP Handler

As part of the dialog above, the Query Handler calls for the UIP handler to display query objects:

```
kern_return_t M_solicitBiblioQuery(int session,char *Title,char *AuthorStr,
                          int Text1Coverage,char *Text1Str,
                          int Text2Coverage,char *Text2Str,
                          int Text3Coverage,char *Text3Str,
                          char *DateStr);
```

and to create and maintain the query history object:

```
kern_return_t M_showChoiceList(int session,char *Title,char *Choices[],
                        int numChoices,int selected);
kern_return_t M_addToChoiceList(int session,char *Title, char *Choices[],
                        int numChoices);
kern_return_t M_clear(int session, char *Title);
```

The clear() function can also be used, together with

```
kern_return_t M_goAway(int session, char *Title);
```

to control the lifetime and contents of query objects.

-- RKF 7-24-92

Calls UIP Handler --> Session Manager

The UIP Handler informs the Session Manger whenever a new user connects to or disconnects from MARIAN proper:

int
M_newSession(+int SessionID, +int Version)

int
M_endSession(+int SessionID, +int Condition)

It passes along requests for help or status initiated by users:

int M_sendHelp(+int SessionID, +char* ObjectName, char* Subject)

> As documented in *Interface Objects* v.2.

int M_status(+int SessionID)

> The result of a contextless request for status.

int error(+int SessionID, +char* Condition)

> I'm not quite sure how this comes up, but I'm sure it will.

It may also make use of the common reporting calls described in the Session Manager text.

Calls Session Manager --> UIP Handler

The Session Manager uses text objects to communicate **Help** and **Status** information:

int
M_showTextObj(+int SessionID, +char* Title, +char* Text)

It uses the marianLogin object to identify users:

int
M_getLogin(+int SessionID, +char* Message)

It also uses other UIP commands, including the modal information and question interactions,and perhaps the choiceListDisplay interaction, to engage in dialogs during registration and at other times.

```
int
M_showInformation(+int SessionID, +char* Message)
```

[[etc.]]

-- RKF 7-27-92

Calls UIP Handler <--> Interface Servers

All interfaces from the User Interface Protocol handler to the various interface servers use a variant on the set of calls documented in "User Interface Objects for CODER, INCARD, and MARIAN", v.2.4 or higher.

RKF 7-22-92

Feedback Query Synthesizer

Draft 1
Robert France
22 March 1994

Introduction

Feedback in information retrieval is a process where documents from one retrieval set are used to generate another retrieval set. In the MARIAN system, feedback requests can be generated from any result set display object by an action called "Use as query." In the Envision system, the action is called "Find more like." In both systems, the user has several ways of choosing a germinal set of documents for the feedback search. These documents – or rather, their full IDs – are the starting point for a new cycle of search and display. The process can then be repeated from the new retrieval set as often as the user chooses, or the user may go back to the original starting point and generate a new set of documents to begin a new feedback cycle. The entire process is often called "relevance feedback," since it is presumed that the set chosen by the user is relevant to (one of) her information need(s). We will use this phrase here, but cautiously, since it is rarely clear to the system which need the set is relevant *to.* In particular, it is not always relevant to the need expressed in the original query, and it does not usually have any commonality with the "relevance" weights assigned by the system to the documents in a retrieval set.

The function of the Feedback Query Synthesizer is to take the set of documents chosen by the user and construct from it a query object: a description of a document that is calculated to fit the chosen documents well. In the vector space model of document retrieval, this query forms the centroid of the document set. In an information-theoretic model, it approximates the common information of the documents while ignoring the noise that differentiates them. In any case, it is a description that captures as much as possible of their common aspects while ignoring the aspects where they disagree.

Several refinements of relevance feedback are ignored in the Feedback Query Synthesizer. First, many feedback retrieval systems use the "original" query as part of the synthesized query. This can either be the query that produced the last retrieval set or the initial query authored by the user that began the whole process. Second, it has been found that feedback queries can often be improved by removing terms found in the highest-ranked document in the retrieval set that was not chosen by the user. Using both of these refinements results in the formula known as "Ide dec hi." Finally, some retrieval systems remove the generating documents from the result set of a feedback query. The

reasoning behind doing this is that the user has already seen those documents and so will regard them as distractions in this new set.

The MARIAN search system can implement any or all of these refinements, but none of them are part of the Feedback Query Synthesizer. All of them make use of information that is already resident elsewhere in MARIAN: to be specific, in the MARIAN Query Combiner, which is the module responsible for mapping queries to retrieval sets. The new query produces by the Feedback Query Synthesizer must be passed to the Combiner in any case, so that it can be mapped to a retrieval set. Any refinement that we choose to use can beset be implemented as a filter or set operation on the synthesized query once it is received.

There is another reason to avoid implementing these refinements in the Feedback Query Synthesizer. All of them are controversial in the context of the MARIAN search system, and none of them have been tested with our intended user community. Query synthesis itself, however, has received good marks as a desirable feature. Thus we have chosen to bring up this simplest version independently of any refinements.

One additional refinement should be mentioned here, as it is most effectively implemented as part of the Synthesizer proper. During synthesis of a text query – that is, during combination of terms occurring in a text field of several documents – it is not uncommon for very large sets of candidate terms to emerge. These sets must then be pruned so that the resulting query is kept at a manageable size. There are several techniques for doing this, but one attractive one is to simply present the set to the user and ask her to select which terms to include and which to exclude. If we chose this method, it should be part of the Synthesizer proper.

Context

The Feedback Query Synthesizer fits into the MARIAN system as part of the Access Method layer (Fig. 1). It receives its initiating call feedback() from the Query Handler, and passes the synthesizer query off to the Combiner. During processing, it makes use of the Text Vector Databases, the Link Databases, and during user interaction, the various String databases. If it interacts with the user, it passes messages through the UIP Handler and receives its replies directly from the UIP Handler.

For the Feedback Query Synthesizer to be integrated into MARIAN several small additions are required, to the UIP Handler, the Query handler, and the Combiner. The UIP Handler must now react to the feedbackQueryRequest() message from the User Interface Manager. (Currently, this message is ignored.) It should package the message into an M_feedbackQueryRequest() message as per usual and send it to the Query Handler. The UIP Handler must also be prepared for M_getUserChoices() and M_yesNoQuestion() messages from the Feedback Query Synthesizer, and be prepared to direct the responses back to the Synthesizer when they arrive.

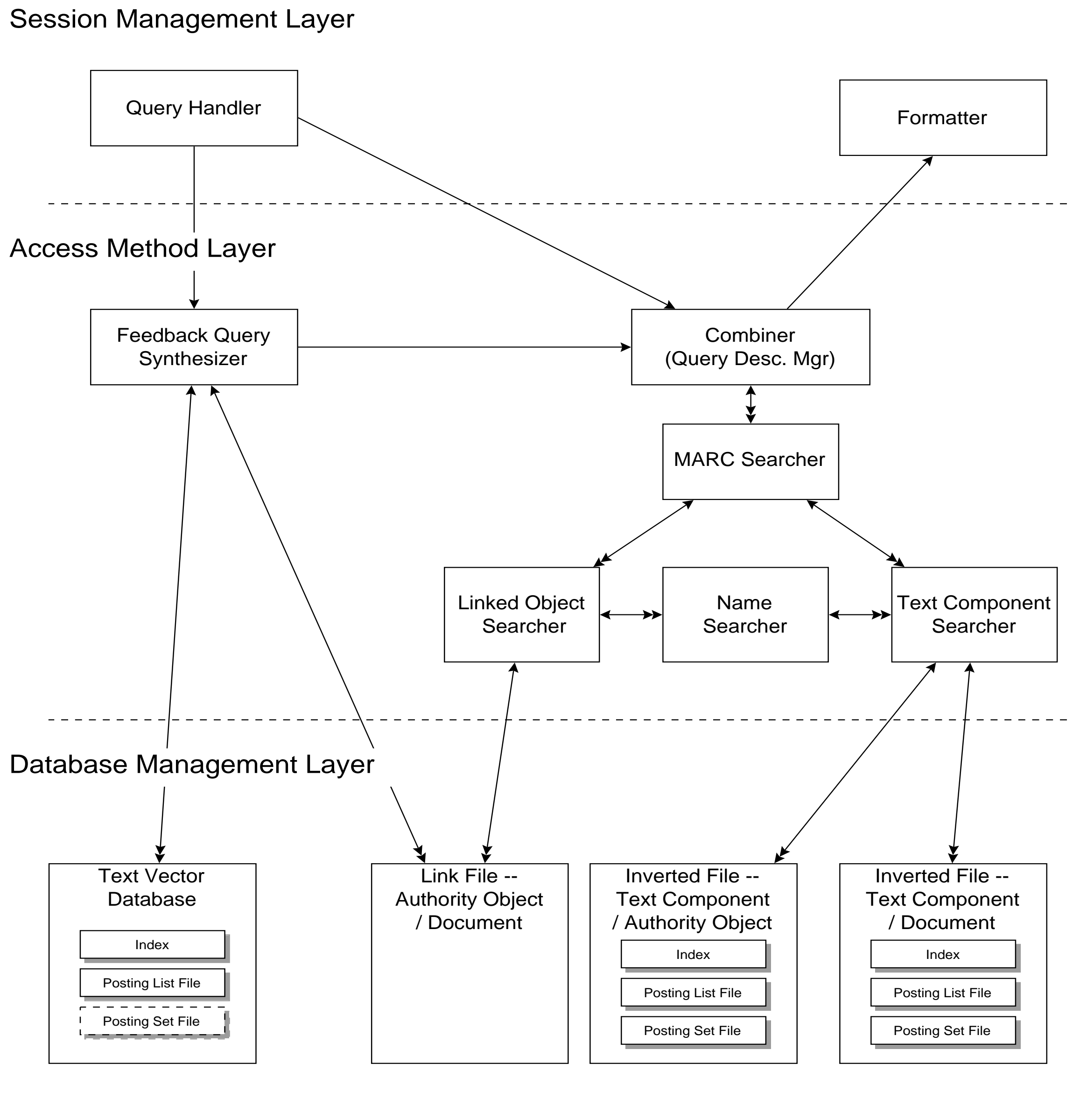


Fig. 1: Context for the Feedback Query Synthesizer within the MARIAN Search System.

When the Query Handler receives a feedbackQueryRequest() message, it needs to assign it a local QID, enter it into the table of queries, generate a natural-language surrogate for the query, and update the user's "Queries" window. The QH should then send two messages to the Access Layer: to the Feedback Query Synthesizer, it should send the sessionID and new QID, together with the set of chosen documents. To the Combiner, it should send the SessionID, the new QID, the old QID, and (redundantly) the set of chosen documents.

The only part of this that is not covered by current query processing is the natural-language representation. I suggest the following: The feedbackQueryRequest() message includes that title of the retrieval set display object from which it was generated. From this title, the QH can deduce the QID and natural language surrogates for the parent query. Say that query is designated “Query 1.3.2,” and say that this is the first feedback request generated from that retrieval set. Then the new query should be designated “Query 1.3.2.f1” (for “feedback 1.”) A second feedback query from that same window would be “Query 1.3.2.f2,” and labels like “Query 1.3.f2.f1.f5” would not be unlikely. The long description is much less obvious; there certainly is not space in a long description string for any sort of natural-language representations of the documents that make up the feedback request, and the document IDs will mean less than nothing to a user. So I suggest the long description simply say “<Feedback: use %d documents as query>” where the number of documents is substituted for “%d”.

The Combiner needs to react to the new message form the Query Handler and to the outgoing message from the Feedback Synthesizer. This situation is directly analogous to the currently implemented processing for new queries, where it must integrate a message from the QH with one or more from the Parsers. In the first version, the combiner will ignore the old QID and chosen document set; these can be used later to implement one of the Ide systems if desired, or to filter the new retrieval set to remove the documents in the chosen set. As remarked above, though, these refinements will come later. Currently, the Combiner need only await both messages, then pass the synthesized query along to the MARC Searcher for processing.

Design

The purpose of the Feedback Query Synthesizer is to produce a query object from a set of document IDs. In the current version of MARIAn, the documents selected by those IDs are always MARC records, and the query objects will always be a four-part MARC object description. Thus the task of the Synthesizer becomes to assemble each of the four parts and combine them into a single query.

The four parts of a MARC record (in the relatively impoverished form currently supported by MARIAN) are: author set, subject set, title, and notes. The first two are represented as sets of controlled objects linked to the MARC record; the third and fourth by single text fields. the text fields are themselves represented as *term vectors:* weighted object sets of the terms that occur is the particular texts. These four parts are independent of each other, and can be synthesized separately by two calls each to two types of processors (sse Fig. 2).

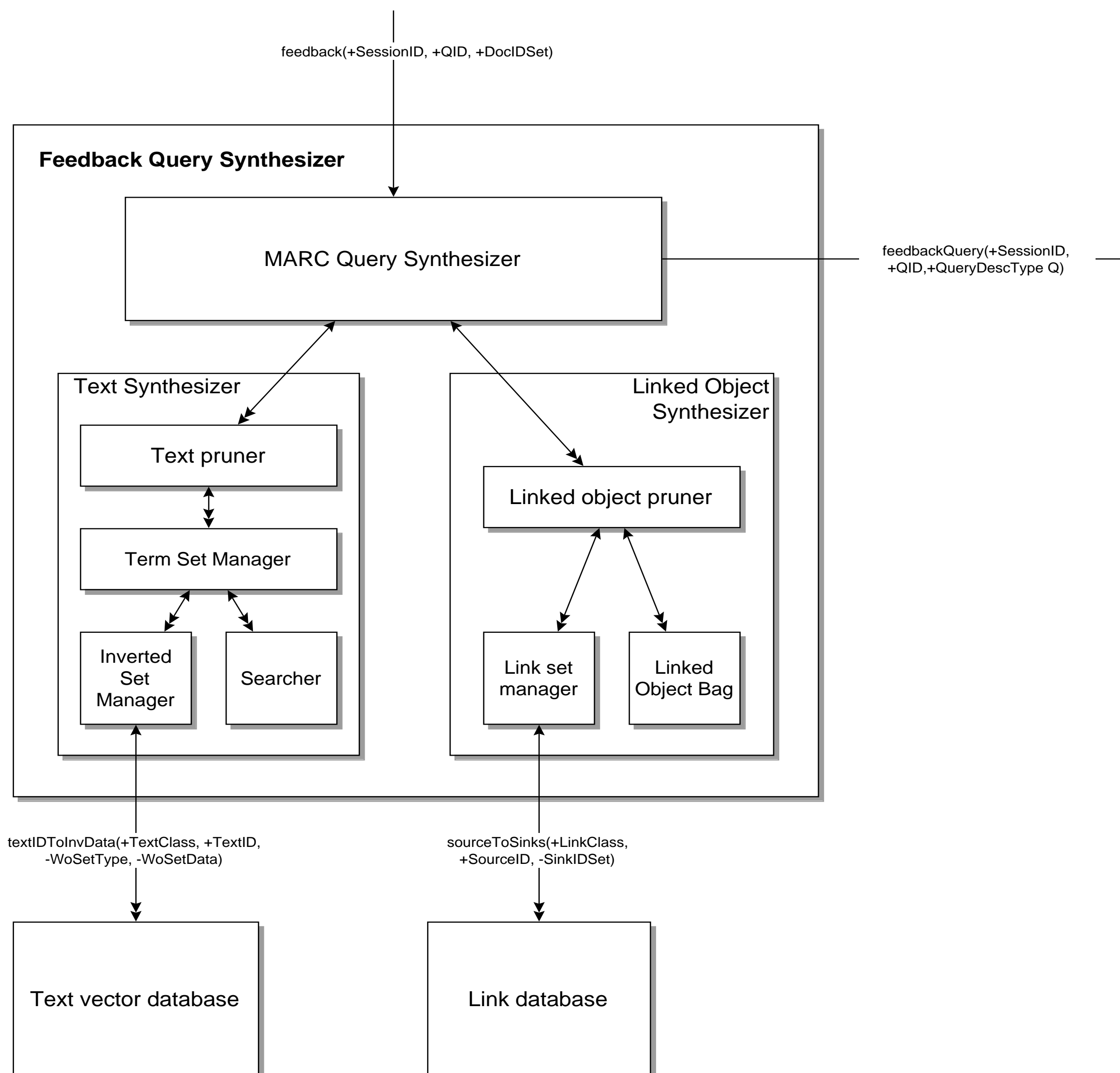


Fig. 2: Internal structure of the Feedback Synthesizer. Creating a MARC description object requires two sorts of processing: text processing in the left-hand branch and linked object processing on the right.

In the case of the text fields title and note, we can rely on constructs already in MARIAN to map the incoming document IDs to a single weighted object set of term IDs. These constructs are the inverted set and the set union operator implemented as a Searcher. the inverted set is a category of weighted object sets currently used to represent all the texts that include a given term. the Feedback Query Synthesizer can make use of this construct on a complementary data set to represent all the terms included in a given text. The Searcher then performs a weighted union operation on the weighted object sets associated

with each text to produce a single weighted object set: the set of terms occurring in any of the texts, with those terms given higher weights which occur more frequently in the text set than in the collection as a whole.

(Note: the MARIAN searchers currently implement an inner product weighting function that owes parentage to both Harmon and Croft. There is some question whether this weighting function is the best for text feedback synthesis, but is not clearly inappropriate and it is readily available at the moment. Many other formulae can be tried, simply by altering the function of the searcher. In any case, the weighting formula is localized to the searcher; changing the function will not change the operation of the Feedback Query Synthesizer as a whole.)

In fact, the Term Set Manager, as we have dubbed the combination of the inverted set manager with a (set of) searcher(s), should be exactly analogous to the Text Searcher. As with the Text Searcher, the Term Set Manager needs to determine whether it has been passed a single document ID or a set of them. If the feedback request is limited to a single document, it can return the weighted set produced by the Inverted Set Manager without change. Otherwise, it needs to create a new searcher, add to it the inverted sets (text vectors) produced for each document ID in the set, and return the resulting object.

Especially in the case where a set of document IDs have been combined to produce a weighted object set of terms occurring in any or all, the resulting set will need to be reduced to a size small enough to search efficiently. This is the job of the Text Pruner. The Text Pruner may use several methods for limiting the size of the term set. Future versions, for instance, may use the getUserChoices() method to enlist the user's aid. For the initial version, however, the Text Pruner should be limited to a pair of heuristics: only terms above a certain minimum weight should be returned; and of those, no more than a certain maximum number. These parameters are given by the MARC Query Synthesizer, so that they may be adjusted to the type of text field being pruned, or to a larger context. At the moment, though, they should be implemented as defined constants. Unless the implementer has a better guess, we will let the minimum weight be .50 and the minimum number be 32.

The process of creating a set of linked objects (authors or subjects) to use in MARC query synthesis is similar to the case for text, but the problems are different. The process again involves generating possible sets and pruning them, but how this is done differs with the character of the data. In particular, where text fields typically contain many terms, each with a known differentiating ability based on its statistical profile in the collection as a whole, typically only a few authors or subjects are linked to any given MARC record, and the statistics that determine their interest in query construction are their local distributions in the chosen document set.

In particular, consider a feedback query request of five to thirty documents. Without any hard evidence, we will postulate that this is a normal size. Given the distributions of authors and subjects, such a request would be likely to include between seven and forty authors and between ten and sixty subjects, although both lower and higher values are

possible. If among a set of twenty authors, no single author occurred more than once, we would be justified in believing that authorship was not a significant feature of this set. If an author occurred in ten out of twenty, or even five out of twenty works, on the other hand, we would believe that that author was relevant to the user's information need, and wish to include him or her. The situation for subjects is similar, although the cutoffs will be different. In a collection of forty subjects for twenty documents, a subject that only occurs once is unlikely to be relevant, but one that is linked to four or five documents may be worth considering, while one that is linked to ten is definitely important.

Given these set sizes, the Linked Object Synthesizer can get away with using relatively unsophisticated methods for merging the sets. Some sort of structure is required for detecting multiple occurrences of linked objects; in Fig. 2 this is labelled the "Linked Object Bag." The Bag can be any of several existing structures: a wtdObjTable, an accumulatorBank, or a wtdObjHeap are all possibilities. It does not. however, need to be a full-fledged searcher, which is probably overkill for a set of no more than 100 elements. On the other hand, it should not be an object that requires $O(n^2)$ time. Exact details are left to the implementer.

Constructing linked object fields is a three-step process. Each document ID is mapped to a set of linked objects by the same Link Set manager construct currently used in the Link Searcher. If there is more than one such set, the sets must then be merged, and the number of times each object occurs noted. Finally, those objects that occur often enough must be weighted and returned to the MARC Query Synthesizer. This last step is actually a combination of the weighting operation performed by a union Searcher and a pruning operation. Some efficiency can be gained in the observation that if there are more than a very few objects, those that occur only once can be ignored.

Objects returned by the linked object synthesizer should be weighted by how many times they occur. Weights, it will be recalled, run from 0.0 to 1.0. An object that is linked to all the documents in the chosen set should be assigned a weight of 1.0; any others, a weight less than 1.0 but greater than 0.0. There are several functions by which these weights may be computed: an obvious candidate is the average function:

(# of documents object is linked to) / (# of documents in chosen set)

The weighting function should be coded in such a way that it can be easily changed.

The MARC Query Synthesizer uses the text synthesizer path and the linked object path to produce a query of up to four fields. Some of the fields may produce nothing to search on, for instance when no documents have an author in common. In the initial implementation the MARC Query Synthesizer need do no more than put the fields returned by the lower synthesizers together and make the call feedbackQuery() on the Combiner. Later versions can be concerned with producing a balanced query by adjusting the size and weights of the query fields based on where good matches have been found between the chosen documents.

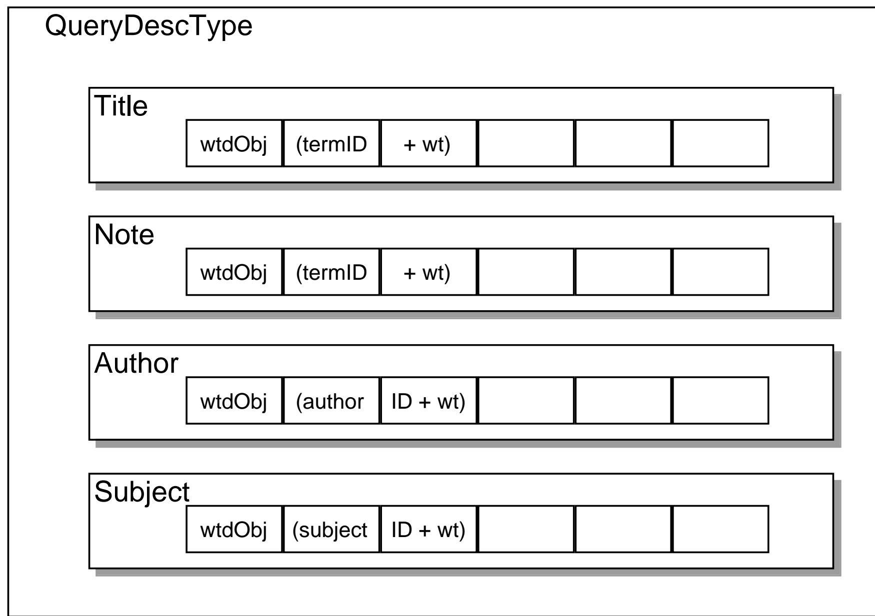
Functions

feedback(+int SessionID, +int QID, +int NumDocs, +fullID* DocIDSet)

DocIDSet is an array of NumDocs fullIDs, one for each document chosen by the user. SessionID and QID identify the session and query of this feedback request; they are to be passed along to the Combiner unchanged and otherwise ignored.

feedbackQuery(+int SessionID, +int QID, +QueryVectType Query)

Query is the synthesized MARC description.



Database Methods

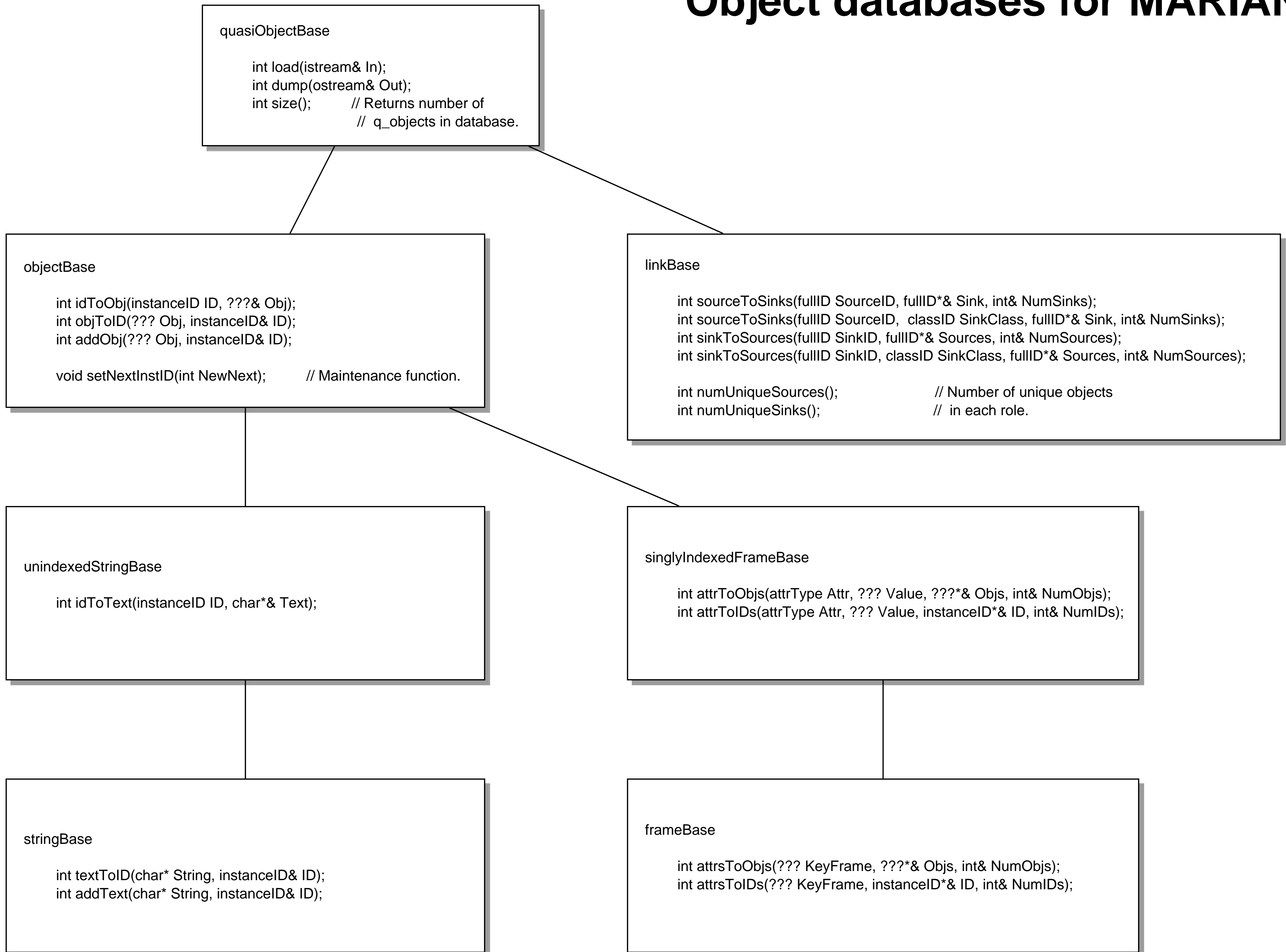
The database management layer is composed of one or more LEND processes per machine, each of which manage one or more of the database "files" shown in the main design document. Actually, each "file" may in fact be made up of more than one object class, each of which may require several files, but the breakup shown is convenient in that it captures a high-level division of the data.

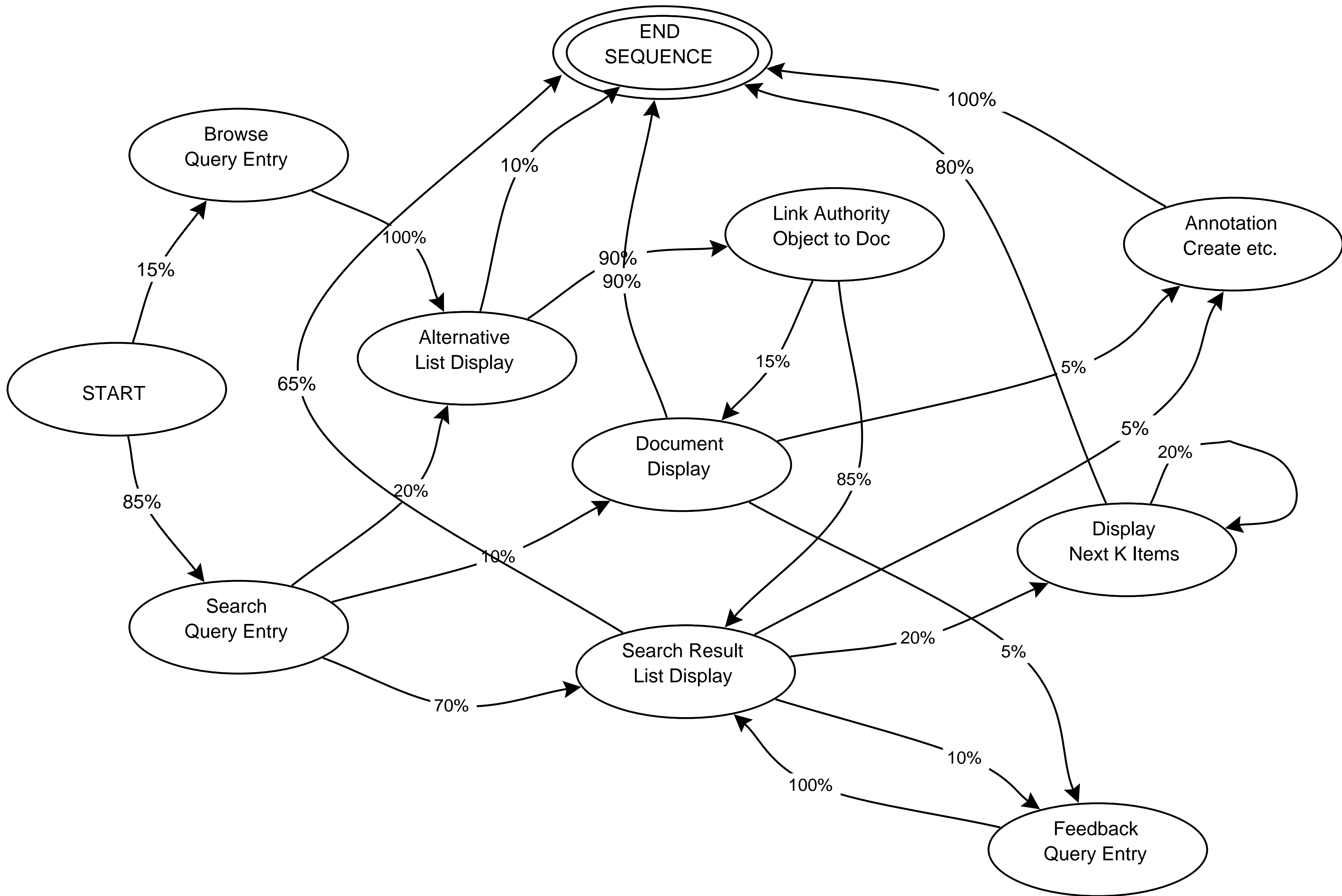
At production time, the "files" will be assembled into groups, each of which will be managed by a single LEND task. At the time of this wirting, LEND is not thread-safe, so each task will execute requests serially. Maximizing throughput in the database management layer may accordingly require several LEND tasks per machine; it may not be possible to determine how many until the system is up and running.

Accordingly, each "file" is being defined separately by its requirements at process invocation and by the methods it supports. These methods may then be bundled as needed in the production version.

-- RKF 8-19-92

Object databases for MARIAN



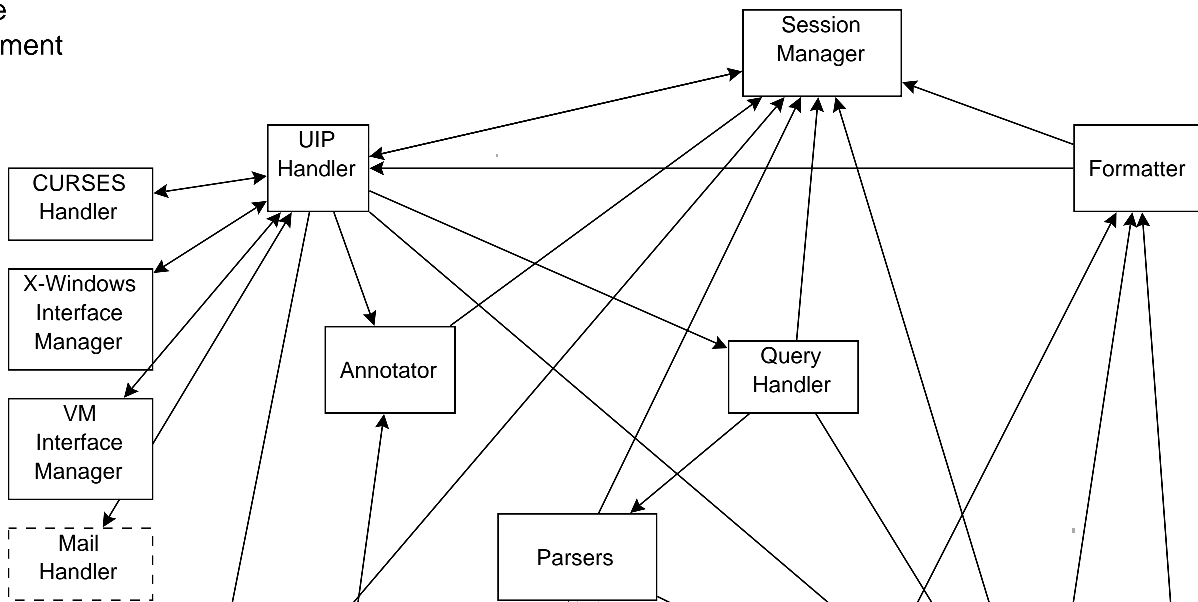


Transition Diagram Among System-Level Tasks

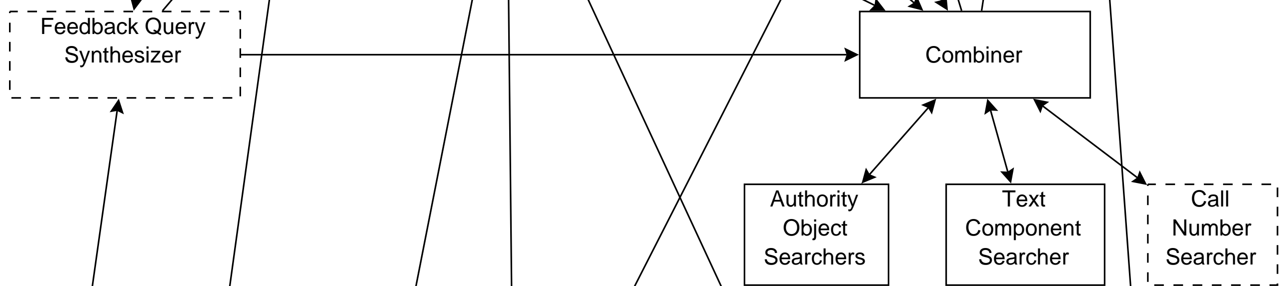
Robert France
Ben Cline
27 Aug 1991

MARIAN: Flow of Operations

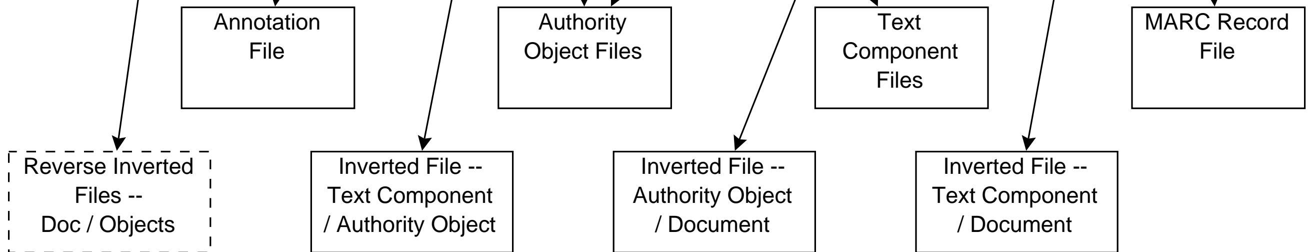
Interface
Management
Layer



Access
Method
Layer



Database
Management
Layer



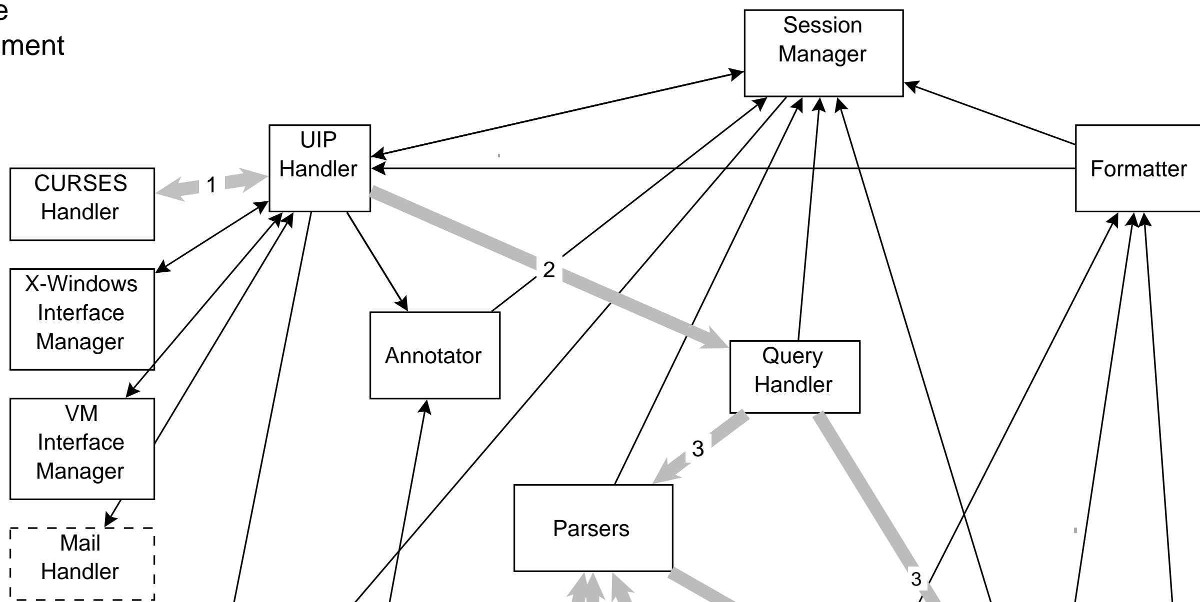
Note:

As each module in the system completes a task, it informs the Session Manager that it has completed the task. These messages are not mentioned in this docuemtn for the sake of simplicity. Any **status** or **help** user transaction s are passed to the Session Manager, which uses this information to determine the state of the user's searches and browses.

Robert France
6 Aug 91
v. 1

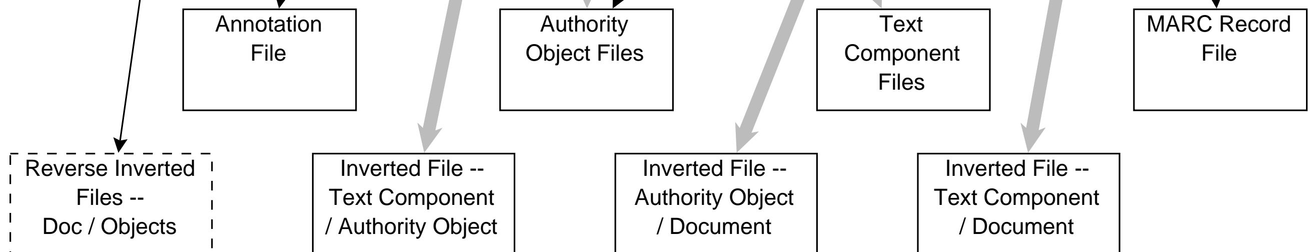
Search Query Entry

Interface Management Layer



Access Method Layer

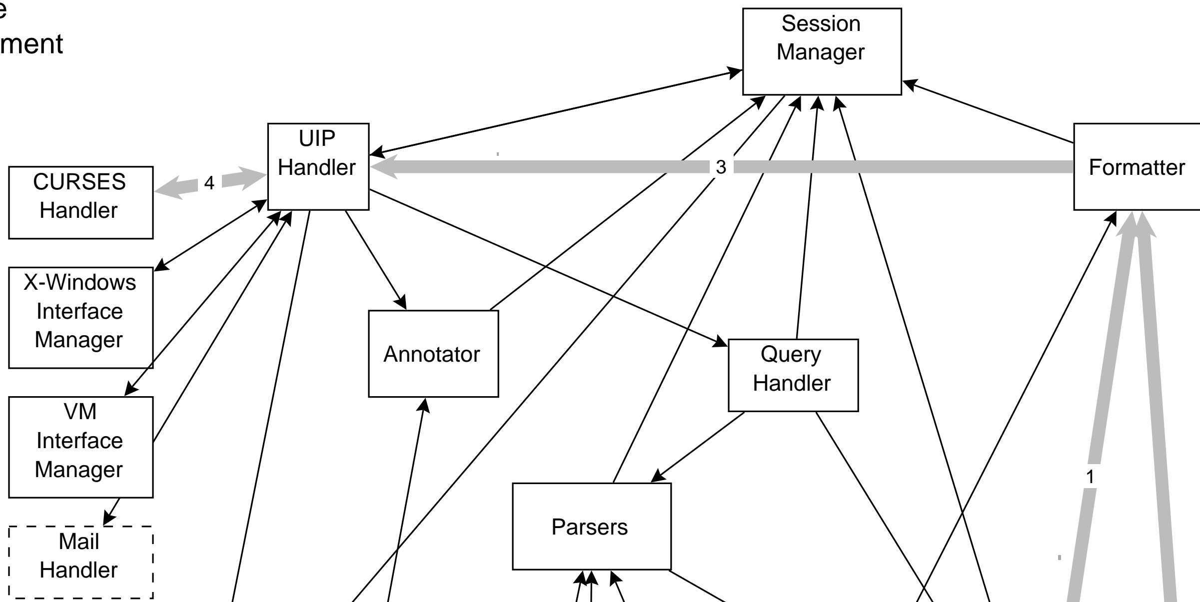
Database Management Layer



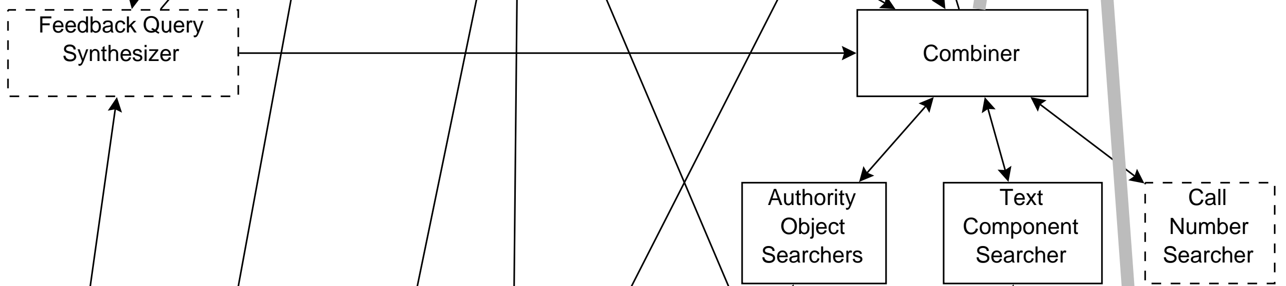
1. A query form filled in by the user is sent to the UIP handler.
2. The UIP hanlder assigns it a number and directs it to the Query Handler.
3. The Query Hanlder splits the query form into fields. It sends each field to the appropriate parser and sends the number of fields to the Combiner.
4. Ther various parsers access databases of text components, authority objects, and the text components that occur within the objects while parsing.
5. As each field is finished, it is passed to the Combiner.
6. The Combiner calls upon appropriate searchers, both for the number of matches a field produces and for the actual documents that match.
7. While searching, each searcher queries an inverted database.

Search Result List Display

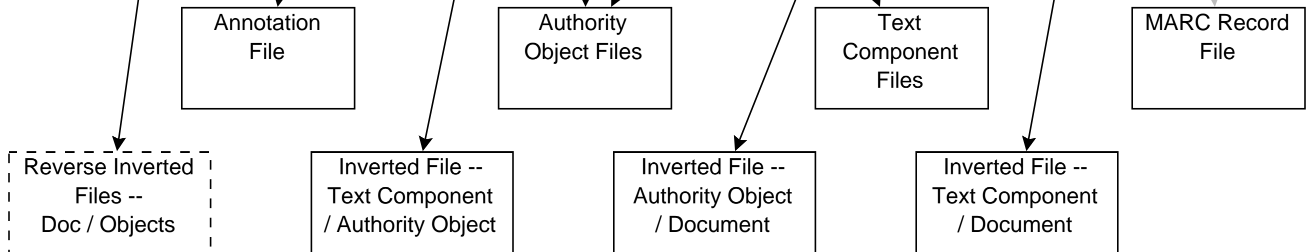
Interface Management Layer



Access Method Layer

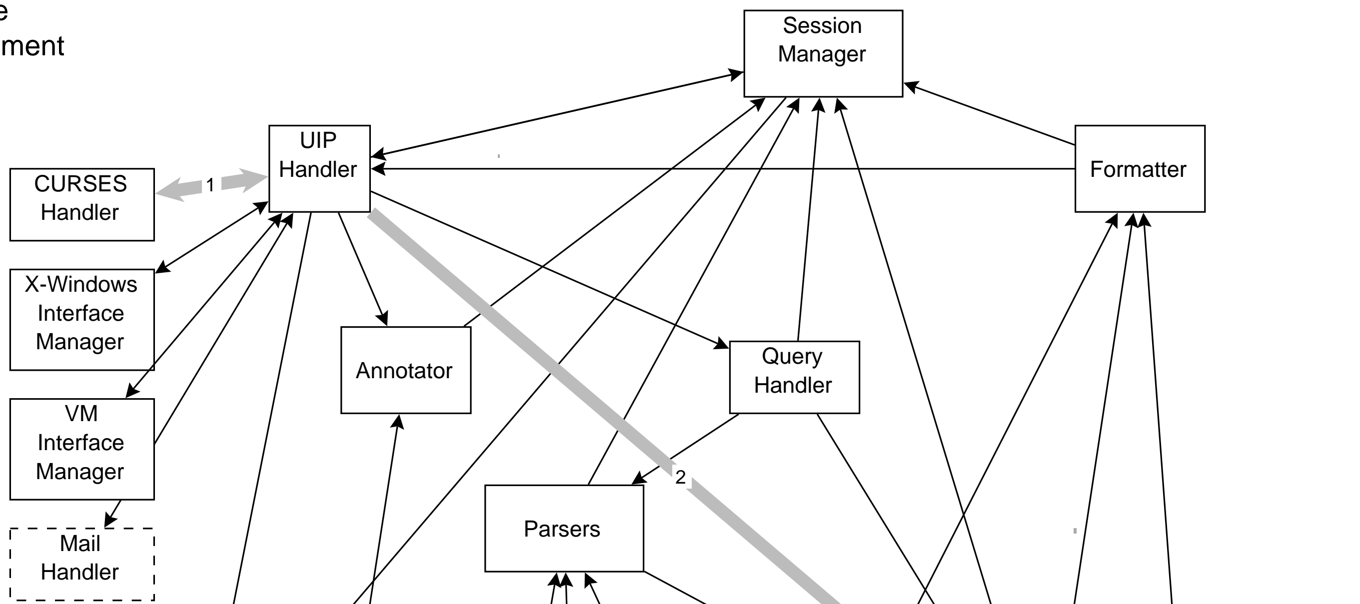


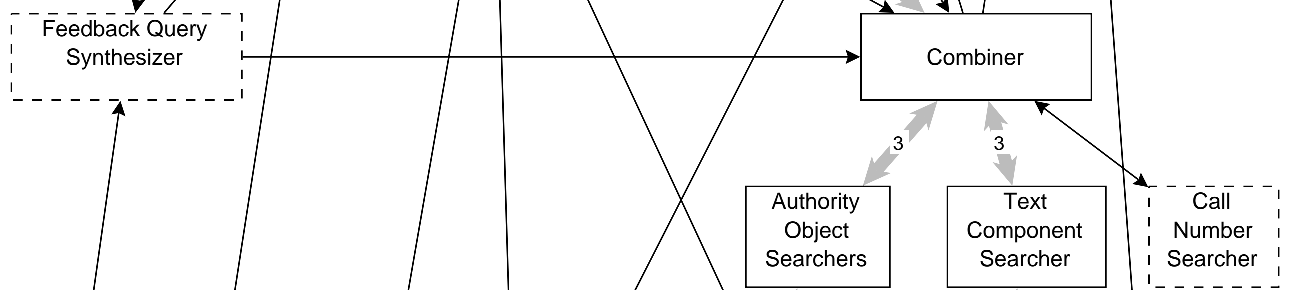
Database Management Layer

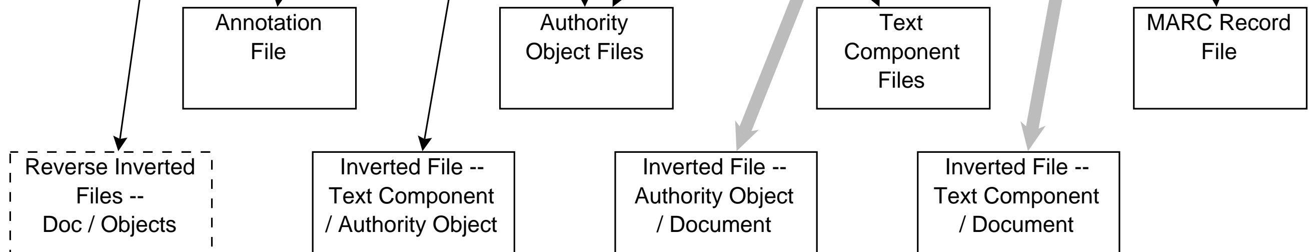


1. The Combiner forwards a list of at most K document IDs to the formatter.
2. The formtter draws on the MARC record file to prepare short descriptions of each docuemnt.
3. The formatter send the desciptions to the UIP handler.
4. The UIP Handler relays the text portion of the descriptions to the interface server for the appropriate session.

Display Next K Items

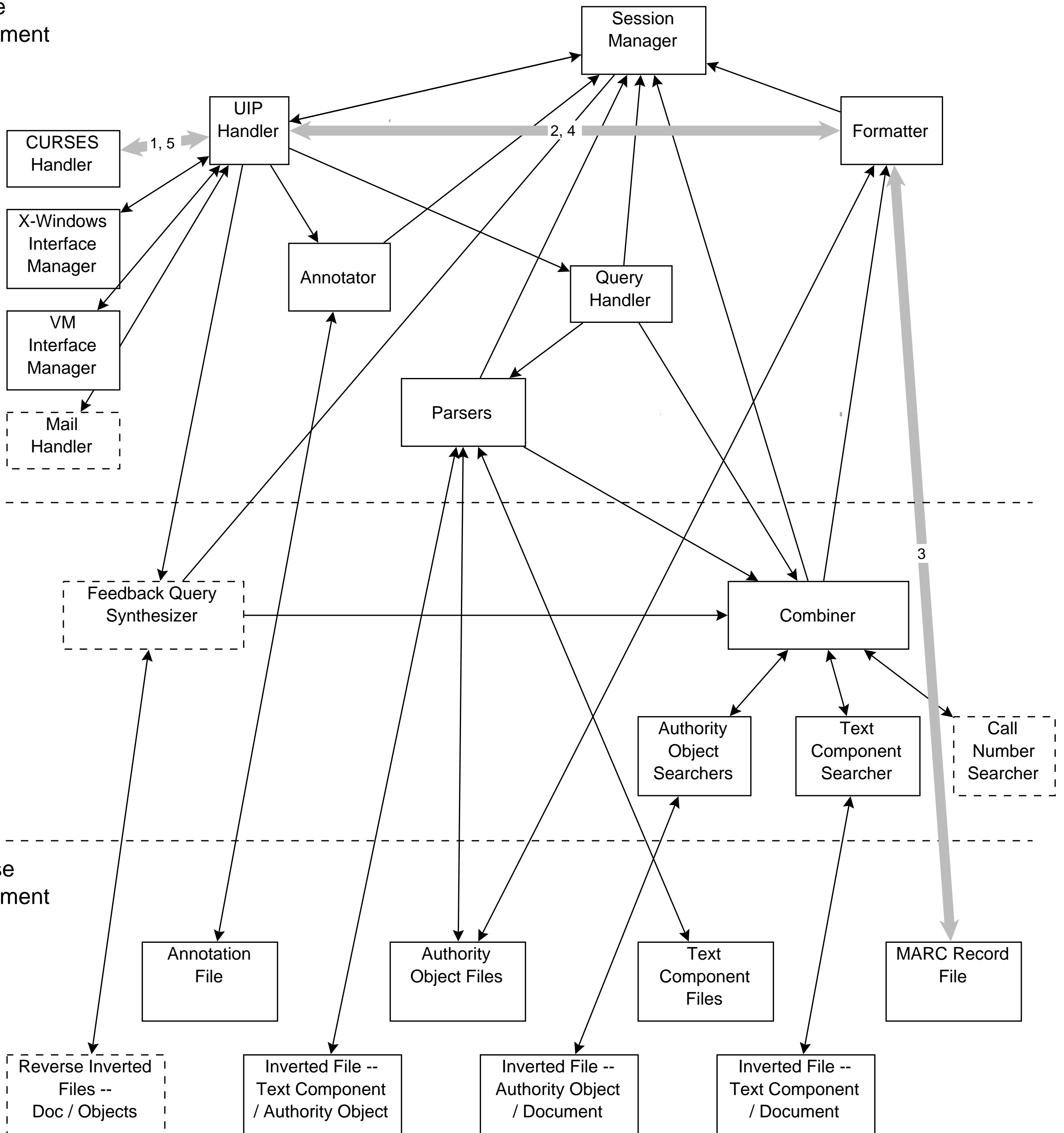






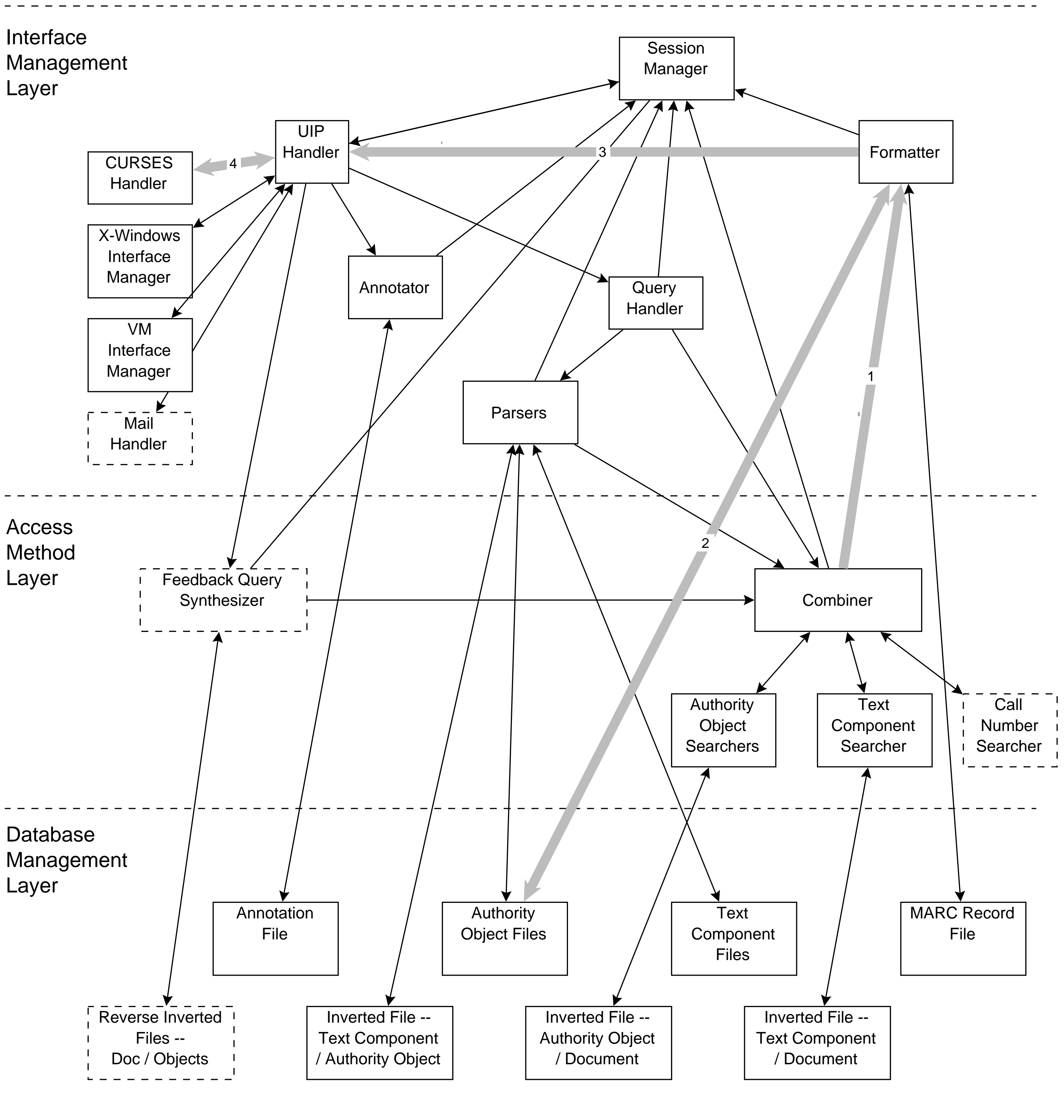
1. The user chooses tto see the next K items in a displayed list. The title of the list is sent to the UIP handler.
2. The UIP handler translates the title to a query ID and passes that ID to the combiner.
3&4. How far down the calling structure the combiner needs to go to determine the next K documents matching the query is not clear.

Document Display



1. The user selects a document from a displayed list. the user's interface manager relays the string selected to the UIP handler.
2. The UIP handler translated the string into the docID andp;assws that to the formatter.
3. The formatter pulls the MARC record from the record file and formats it for disply.
4. The text thus formed is passed back to the UIP handler.
5. The UIP handler relays trhe text to the appropriate iunterface.

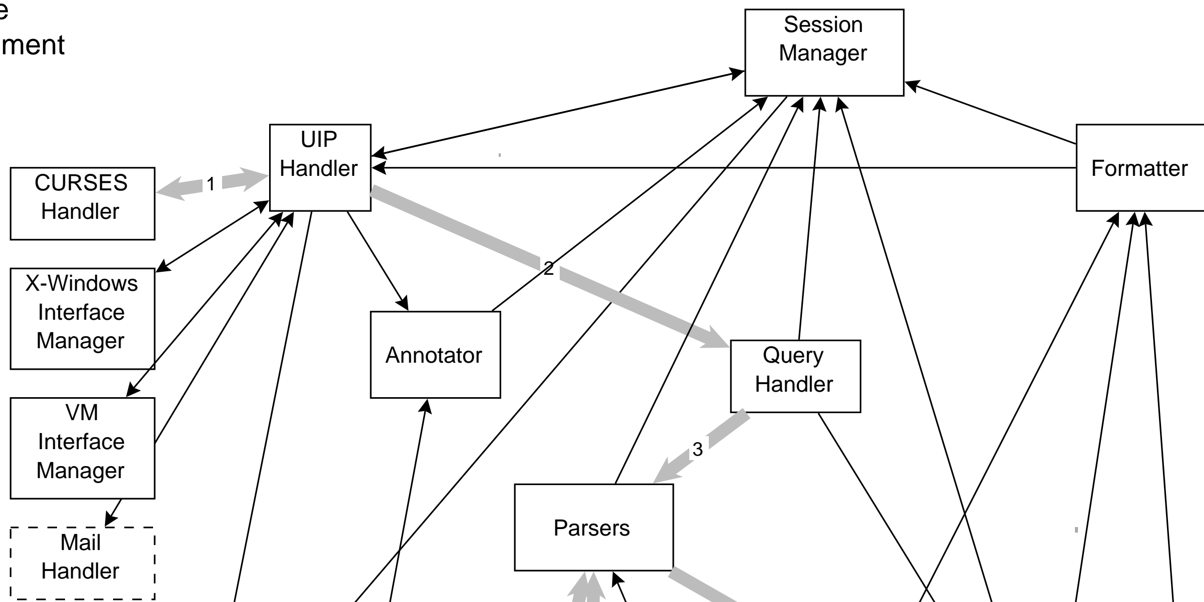
Alternative List Display



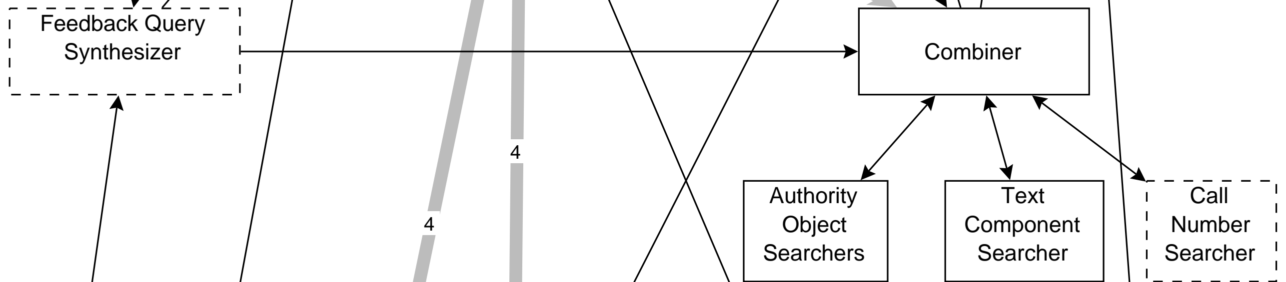
1. The combiner passes a list of alternate authority objects to the formatter.
2. The formatter uses the appropriate authority object file to construct description strings for each object.
3. The strings, together with object IDs for the objects, are forwarded to the UIP handler.
4. The UIP handler passes the strings to the appropriate interface server.

Browse Query Entry

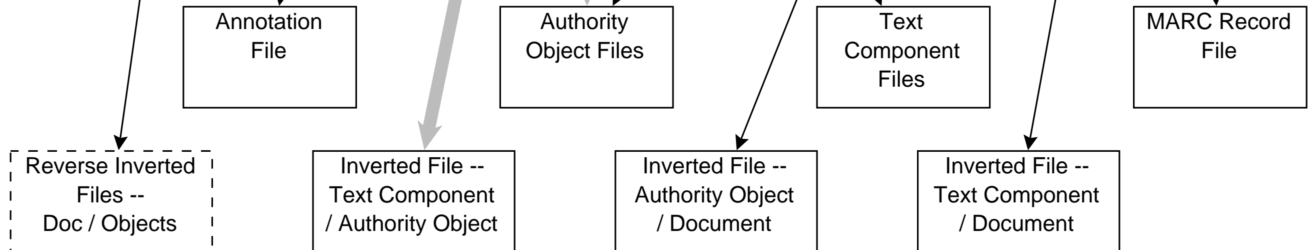
Interface
Management
Layer



Access
Method
Layer

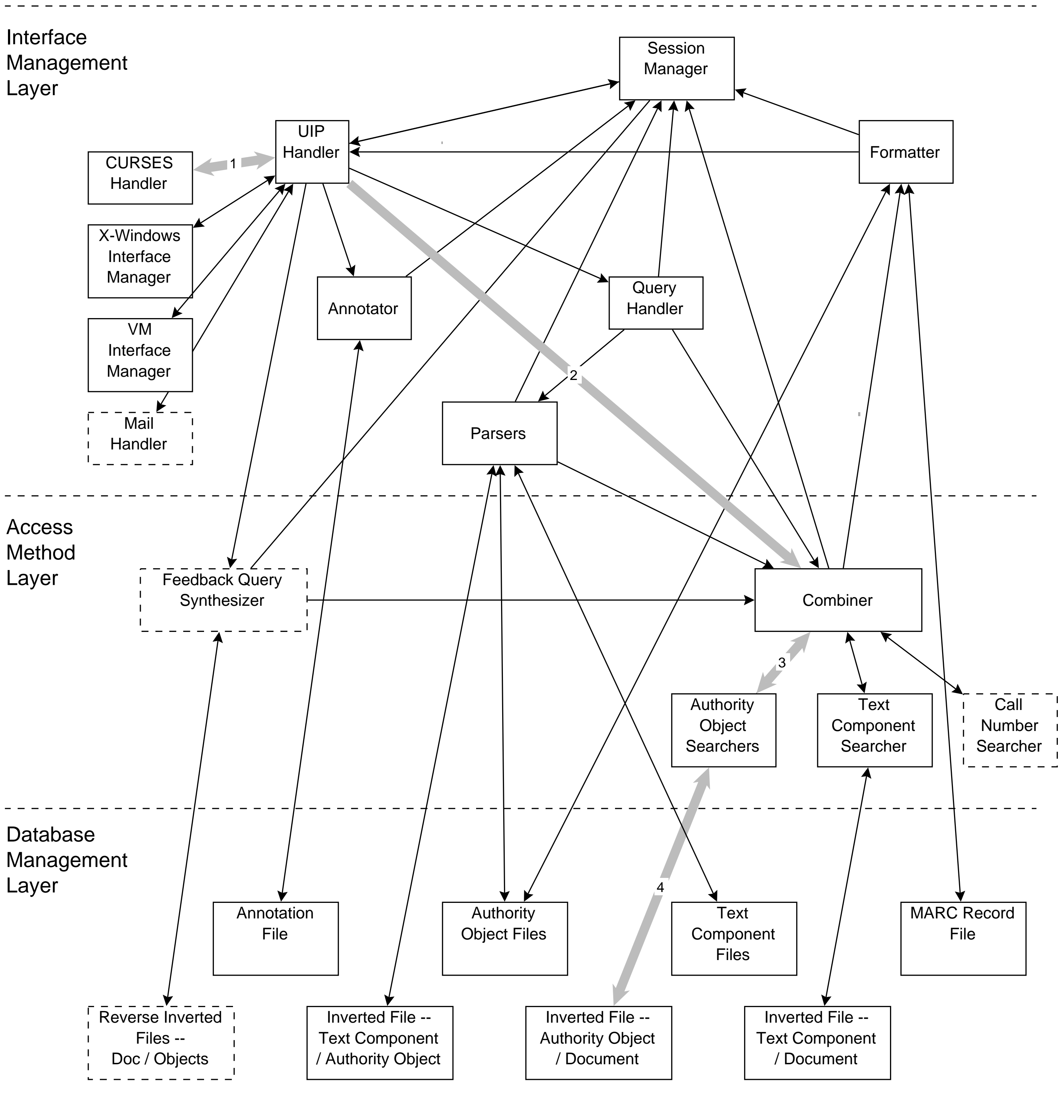


Database
Management
Layer



1. The user interface passes the point at which the user wishes to start browsing to the UIP handler in the form of a piece of text.
2. The UIP handler assigns it a query number and passes it to the Query handler.
3. The Query handler passes the text on to the appropriate parser.
4. The parser uses databases to find which authorities match the query most closely.
5. These are passed to the combiner.

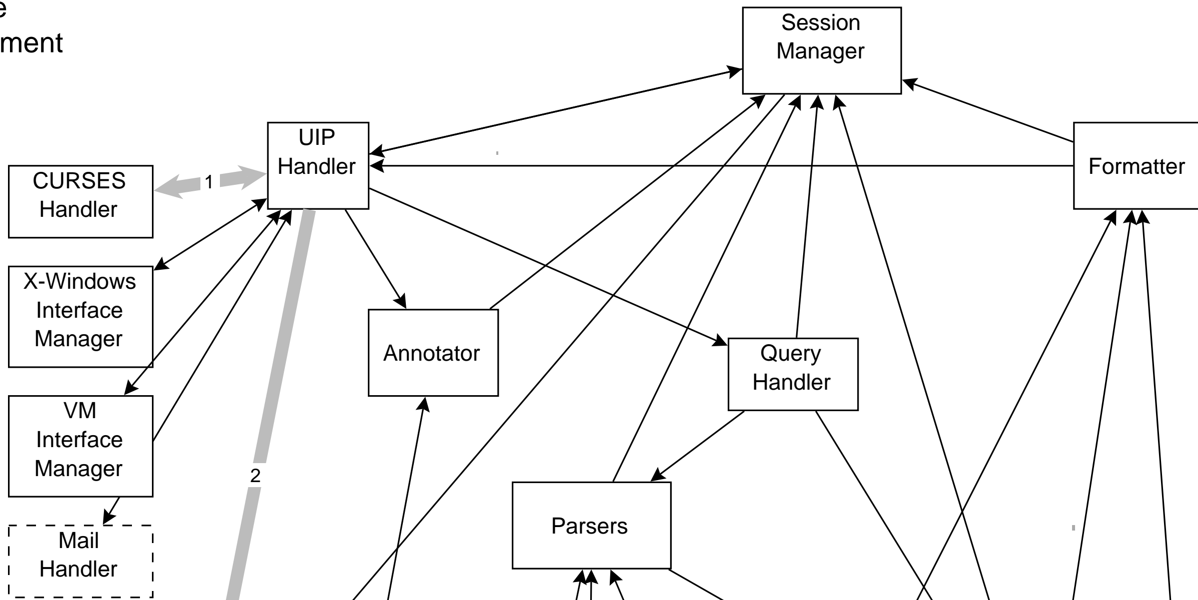
Link Authority Object to Doc



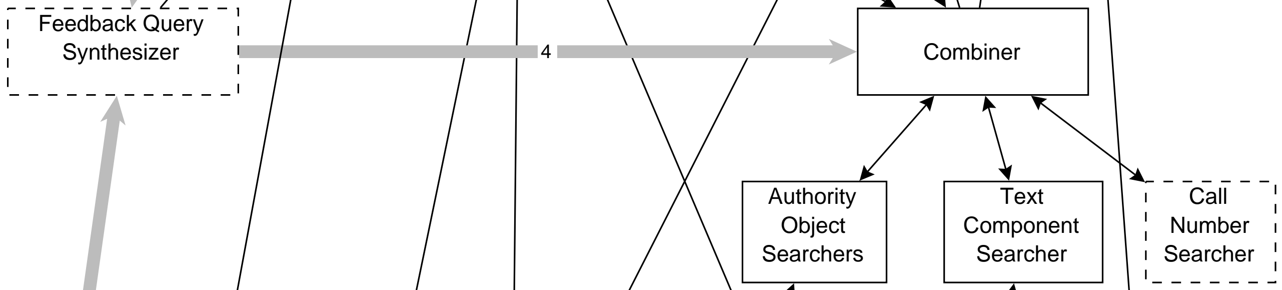
1. The user selects an authority object from a displayed list. The string describing that object is pased to the UIP handler.
2. The UIP handler converts the string into the ID of the object and passes that on to the combiner.
3. The combiner sends the ID to the approprite search engine.
4. During the search, the engine checks its related data file.

Feedback Query Entry

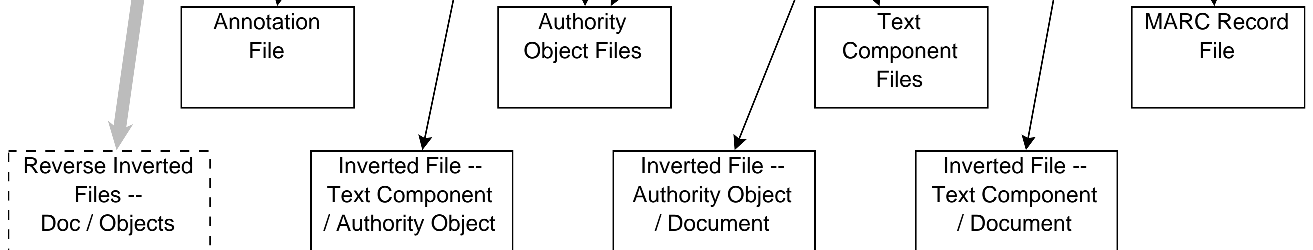
Interface Management Layer



Access Method Layer

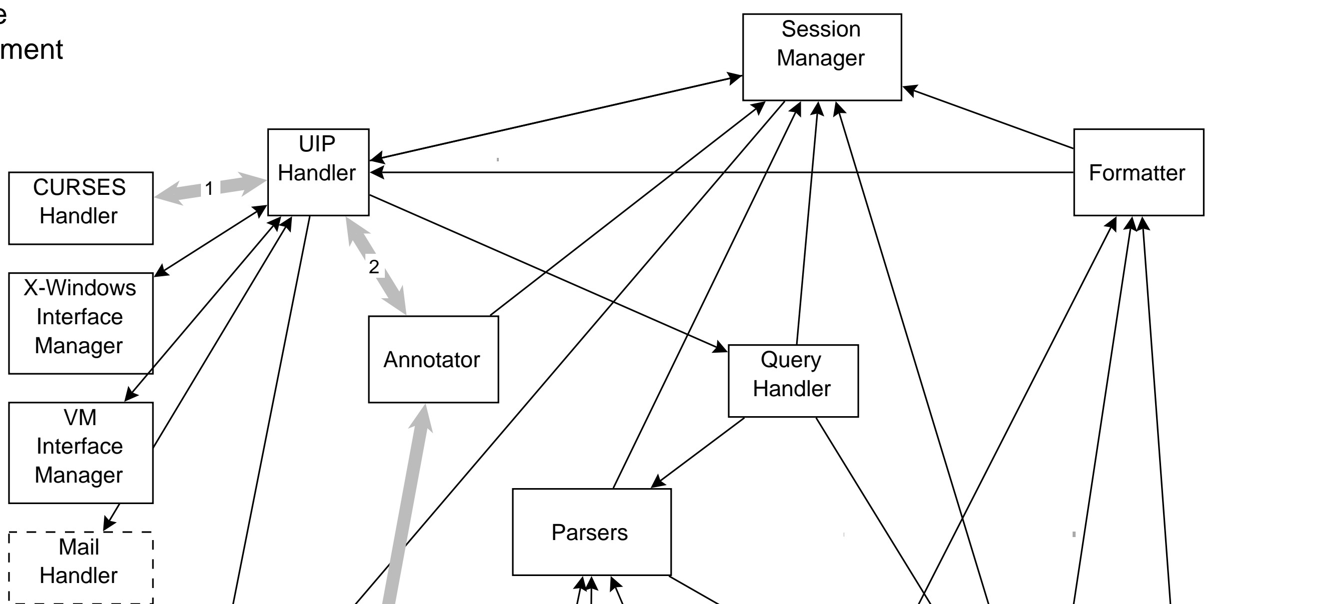


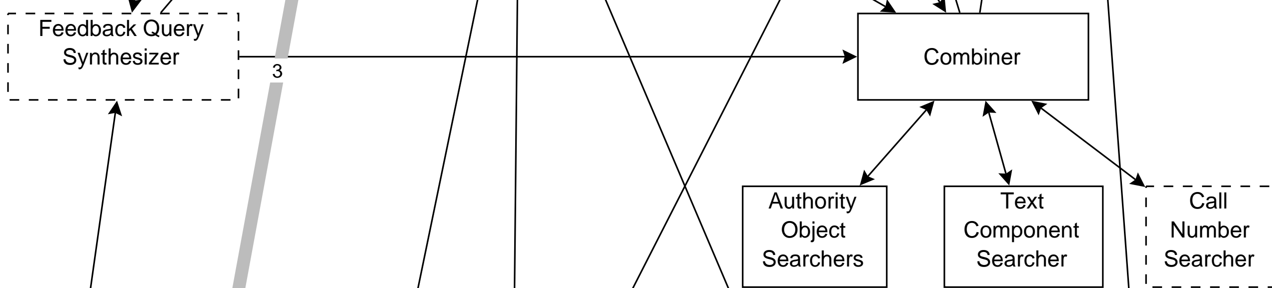
Database Management Layer

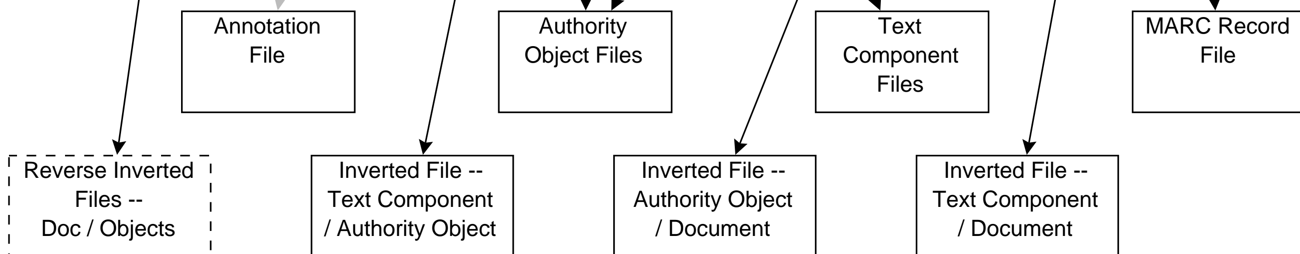


1. The user selects one or several documents to use as a feedback query. Somehow these are passed to thge UIP handler.
2. The UIP handler converts them to document IDs and passes these on to the feedback query synthesizer.
3. The synthesizer draws on a database relating documents to their components while constructing a single query from the set of documents.
4. As each field of the new query is constructed, it is relayed to the Combiner.

Annotation Create / Edit / Delete / Show Own / Show All







1. The operation selected by the user is communicated to the UIP handler.
2. The UIP Hanlder passes the request to the Annotator.
3. The Annotator then may:

- retrieve a note or a set of notes from the database,
- send an existing note or a blank text field to the user for editing,
- store an edited note in the database, or
- delete a note from the database.