Kunze, John A. "Nonbibliographic Applications of Z39.50." The Public-Access Computer Systems Review 3, no. 5 (1992): 4-30. (Refereed Article.) To retrieve this article, send the following e-mail message to LISTSERV@UHUPVM1 or LISTSERV@UHUPVM1.UH.EDU: GET KUNZE PRV3N5 F=MAIL.

1.0 Introduction

Although the Z39.50 information retrieval protocol is rapidly gaining acceptance as a standard for interoperability among networked library automation systems, [1] it has not been obvious how to make it work for nonbibliographic applications. This article describes how Z39.50 is being used as the basis for a networked campus information system called Infocal [2] at the University of California at Berkeley. [3]

The files we are making available via Infocal include class schedules, computer documentation, library OPACs, phone directories, public announcements, and software. Since Infocal is being developed by Berkeley's Information Systems and Technology department, the distribution of computer documentation and software was the original incentive for implementing the system. The majority of the information will originate on campus, though some will be purchased or licensed. Most of the nonproprietary information will be available over the Internet.

We avoid describing the project as a campus-wide information system (CWIS) because these systems often come with extra baggage such as bulletin boards, electronic mail, and student registration systems. Our system is a straightforward read-only, client/server-based campus information system with two extra goals: (1) accommodating nontextual data, and (2) interoperating with other information systems.

To help us achieve these goals, we decided to utilize the Z39.50 protocol. [4] The challenge was how to use Z39.50 to support all of our nonbibliographic requirements. This article addresses this challenge from the point of view of the designer of the client system program. This program, which sits between the user and the network, translates user commands into Z39.50 protocol requests for the server, and it translates server Z39.50 protocol responses into appropriate user displays.

2.0 Z39.50 Data and Retrieval Support

We need Z39.50 to support the following types of data:

- Bibliographic databases
- Data of unknown, but learnable, semantics
- Full-text documents
- Nonbibliographic databases
- Nontextual documents

We need it to support the following types of retrieval:

- Hierarchical browsing
- Hypermedia links

- Retrieval by object/document ID

The protocol, having come out of the library automation community, is straightforward to apply to the retrieval of bibliographic citations, where records have highly predictable structure and semantics. What about databases whose record structure and semantics are nonbibliographic? What about retrieval from databases whose record structure and semantics are known to the server system programmer, but are not known to the client system programmer? Is there a way for the protocol to transmit meta-information that would allow the client to retrieve data from such a database?

Full-text documents are also problematic. Should the protocol view a document as a database? Should it view a document as a record? As an element of a record? Or, as a set of records? The protocol's retrieval model requires that records have a fixed maximum size for the duration of a session; this means that the programmer, knowing that documents can have highly skewed size distributions, must decide whether to set the maximum record size ridiculously high--forgoing any optimization that might have otherwise been obtained--or to fragment documents across record boundaries.

The retrieval model also requires that records belong to the result set of a query: you are not allowed to retrieve a record unless it has satisfied some set of search criteria in advance. Therefore, if you are retrieving document fragments, you may have to invent a way to conduct a "fragment search."

The retrieval of nontextual data, such as images or video, is problematic due to the large size of these objects and, consequently, the need to support data compression. There are other more serious complications, such as the realtime protocol requirements of video data, that are not addressed here.

Hierarchical browsing, however unfashionable, is still an important access method for campus information system users. It seems that there will always be a very popular subset of information on a server that people prefer to get at by exploring a small, shallow tree of information, rather than by doing an index search. This is analogous to going to a restaurant and preferring to browse the menu instead of asking the waiter to list all dishes containing potatoes but not anchovies. To support hierarchies, the protocol will have to allow clients to discover tree node names inside menus and to retrieve node contents using the search facility. Hypermedia links would be improved by building them on top of hierarchical browsing capabilities.

It would be advantageous to deal with object (or document) ID retrieval issues at the same time as hierarchical browsing and hypermedia retrieval issues. If we had unique, server-qualified document IDs, users could reuse them to retrieve a known document without having to reconstruct the entire search leading to its original discovery. Document IDs could be valid across sessions, could be shared with other users, and could be submitted in relevance feedback queries without incurring the overhead of sending entire documents in the query. The Wide Area Information Server (WAIS) project [5] has made a relevant proposal in this area, and the Digital Object ID Project, under the auspices of the Coalition for Networked Information, also addresses this issue.

3.0 Z39.50 Applications

As a basis for later discussion, it is useful to outline how Z39.50 is applied to bibliographic information retrieval. The protocol has seven separate services, of which only three need be mentioned here: the Search, Present, and Explain facilities (Explain is currently under development [6] and will be discussed later). The Search facility defines how a query is represented, and Present defines how information and diagnostic records are represented. Although there are four kinds of queries, we need only consider the Type-1 query, since it is the only one currently being used for interoperability testing. It is a relatively full-featured query against multiple databases that allows nested Boolean combinations of qualified search terms and previous search results.

3.1 Semantic Modules

One of the claims of Z39.50 is that it does not restrict the kind of query you submit or the kind of data you get back. On the other hand, it is easy to see how an implementor might think, having just read the standard, that too few details are given to build an interoperable system. This is because Z39.50 is modularized in order to isolate those parts of the protocol that would require data-dependent assumptions. Modules are identified and can be replaced to accommodate new sets of assumptions.

Each module is assigned an unambiguous tag or identifier by the National Information Standards Organization (NISO), and the protocol guarantees that the client and server always understand what modules are currently assumed during a session. It accomplishes this by making sure that the module identifiers get tucked into protocol control messages whenever they are needed. Table 1 identifies the five protocol modules that completely define the structure and semantics for protocol control, queries, information, error messages, and resource reports.

Table 1. Z39.50 Semantic Modules

Module Name	Describes	Bibliographic Instance
Z39.50 kernel	Control	N/A
Query type/attribute set	Queries	Type-1/Bib-1
Record syntax	Information	MARC
Diagnostic set	Error messages	Bib-1
Resource set	Costs and time	Bib-1

The official Z39.50 terms for these modules are the query type/attribute set pair, record syntax, diagnostic set, and resource set. The attribute set applies only to the Type-1 query, which we are already assuming. It defines several concepts about the search term and assigns a number to each one.

A list of these numbers accompanies the search term so that the server can understand what element is to be searched, what relational operator to use, what position in the element, and so forth. The record syntax, often called transfer syntax, is much more than simple lexical structure; it embodies intimate structural and semantic knowledge about a sequence of bits. The diagnostic and resource sets are just tables of messages and message numbers that the two sides agree on.

The standard only goes so far to effectively define one set of modules (and a few close variants), which is the minimum required to build an implementation. This particular set of modules, listed under bib-1 and MARC, is designed for bibliographic applications. The specific bibliographic modules it defines or refers to are listed in Table 1: the Type-1 query with bib-1 attribute set, the MARC format [7], the bib-1 diagnostic set, and the bib-1 resource set. It also defines the module identifiers; using this information, the programmer keeps track of the current semantic context.

3.2 Client System Modules

The client system is hard coded to understand whatever combinations of modules it supports. The control module makes sure that neither client nor server ever has to deal with semantics that it did not agree to (through lower level negotiation).

3.2.1 Bibliographic Modules

For bibliographic data, the client system programmer can construct user interface menus that list searchable elements (e.g., author, date, and title) or valid relational operators (e.g., equals, less than, or greater than) because these features are all defined in the semantic module known as bib-1. The programmer can also write code that understands how to read a record as a sequence of bits in the MARC format and display it to the user.

3.2.2 Nonbibliographic Modules

What does the programmer do differently to support new, nonbibliographic semantics? The answer is:

(1) Define new semantics for search, display, and diagnostics.

(2) Get the new attribute set, record syntax, and diagnostic set registered with NISO.

(3) Write code to support searching and displaying these semantics.

(4) Plug in the new code modules to support the new semantic modules.

For example, consider supporting a student directory database. You need a new attribute set that might include element names such as NAME and EMAIL ADDRESS, but not PHONE NUMBER. Note that even though we might have record semantics that support the

display of phone numbers, we might be using an attribute set that does not support searching on them. In the example in Figure 1, the record syntax is the same one used in the Campus Wide Information System Protocol (CWISP), [8] which has a printable ASCII format. As for error messages, you might just choose to keep on using the bib-1 diagnostic set since it is fairly generic.

Figure 1. Example Student Directory Database

```
+ record syntax = CWISP
     name: Kunze, John A.
     email: jak@violet.berkeley.edu
     phone: 510-642-1530
+ attribute set = student-1
+ diagnostic set = bib-1
```

This approach to nonbibliographic data is problematic. There is a lot of work involved in writing code to support each new set of semantics. Imagine defining a new set of semantics for each one of your database formats, going through the registration process with NISO, then writing code to support each. If a database format or user search requirement changes, you need to register again with NISO and stop interoperating with the systems you used to work with until the new format gets approved. If you want to support more than one set of semantic modules, your code will have to link in all your corresponding program modules. With code hardwired to each attribute set and record format, there is nothing to prevent one user interface to ten database formats from looking like ten different user interfaces, which is exactly the situation that Z39.50 was supposed to avoid.

Berkeley's Infocal system will have at least a half dozen database formats and they will probably each change once a year, so this scheme will not work for us. Fortunately, there is another approach to nonbibliographic information that addresses the problem of retrieving information from databases whose format is unknown until the database is first accessed. What is needed is an attribute set, a record syntax, and a diagnostic set, but they must be general purpose and dynamic.

4.0 The Dynamic General-Purpose Attribute Set Info-1

For the purposes of discussion, it will be useful to define some terms: Use attributes, elements, Explain facility, and objects.

4.1 Use Attributes and Elements

Server information records are structured to allow searching on and retrieval of different combinations of individual record components. These processes, completely defined by a server, are mapped onto canonical Z39.50 processes so that clients have a common language to access information. In order to give servers adequate flexibility, Z39.50 distinguishes between searchable and retrievable record components. Searchable components are called "Use attributes" and retrievable components are called

"elements." A server is free, for example, to map a Use attribute called "name" onto five different record components.

+ Page 11 +

4.2 Explain Facility

The Explain facility is a set of mechanisms integrated with the Z39.50 Search and Present facilities that retrieve information, both human- and machine-readable, about an information server. For example, searches against the reserved database name, "IR-Explain-1," can return a schedule of server availability, a list of database names, and a list of browsable hierarchies. It can be searched by database name to return a list of valid attribute combinations, various levels of human-readable record component descriptions, and element set names. When the client encounters an unknown database, it uses the Explain facility to build menus for searching and to provide help text for explaining query semantics.

4.3 Objects

In this paper, an object is anything (e.g., book, painting, person, digitized image, or electronic document) having an associated electronic information record that may be accessed through a unique identifier, called the object identifier (objectID), about which more will be said later. This record may be thought of as an object citation, the elements of which describe aspects of the object, such as its name, what kind of thing it is, where it comes from, when it came into being, where it is currently located, and a brief description. An object citation easily maps onto the Z39.50 record model and is a natural access point for many, if not all, aspects of the object. Therefore, if the object exists primarily in electronic form, a citation element may contain a copy of the object itself.

In cases where Explain is not used and where sophisticated processing is not required, a server may map these record components to a small set of generic, previously defined Use attribute values and element names for search and retrieval. To be useful, most of these elements would be printable ASCII text suitable for human consumption. This provides a kind of base-level service from databases of otherwise unknown semantics without requiring the Explain facility. The Table 2 below shows the generic names (given symbolically here, although the protocol uses integer tags) in the first column, and examples of how four different databases might be mapped to them. The same tags can be used to identify returned elements.

+ Page 12 +

Table 2. Generic Use Attributes (Elements) and Example Mappings

Tag	Bibliographic Database	Personnel Database
Type	"Book"	"Employee"
Name	Title	Employee name
By	Author	Organization
Location	Publisher	Phone/address
Date	Publication date	Hire date

Abstract	Contents	Job description
Object	Text/(N/A)	N/A

Tag	Course Database	Art Database
Type	"Class"	"Artifact"
Name	Class title	Title
By	Instructor/department	Artist
Location	Room/building	Owner
Date	Meeting time	Creation date
Abstract	Course description	Description
Object	N/A	Bitmap/(N/A)

The attribute set info-1 (registered with NISO as 1.2.840.10003.3.1000.2.1) was created by first making a copy of the bib-1 attribute set, complete with its broad categories (or types) of attributes and erasing all the pre-defined bib-1 Use attribute names. A small set of Use attribute names was then reserved as shown below in Table 3. The generic tags for concept and conceptId are useful for relevance feedback searches that look for objects similar to an object specified with the search term; a concept term contains the object, and a conceptId term contains a pointer to the object.

Table 3. Dynamic Attribute Set Info-1

Attribute Types

Type	Example
Use	(Statically undefined except for generic tags)
Relation	=, !=, >, >=, <, <=
Position	First in element, last in element, etc.
Structure	Date, name, word, etc.
Truncation	Right, left, etc.
Completeness	N/A

Generic and Predefined Use Attribute (Element) Tags

	Generic Tags	Predefined Tags
1	Type	Kind
2	Name	What
3	By	Who
4	Location	Where
5	Date	When
6	Abstract	What+
7	Object	What++
8	Concept	What
9	ConceptId	*What
10	Satisfier	Which~
11	Userinfo	Miscellaneous
12	Any	Any elements server wants

13	Keywords	Extra index terms
14	Record size	Estimate in bytes
15	Record update	Last update of record
16	Provider	Who maintains record
17	ObjectID	For fast object access
18	Explain	Bootstraps tags
19	Default	Any single element
20	PobjectKey	Physical object id
21	IrecordKey	Internal record id

This sketches out a minimal generic query interface. What about the problem of displaying server responses to the user? A generic diagnostic set is not hard to define, and the bib-1 diagnostic set with a few additions would be an adequate start. Coming up with a generic record syntax is a little more complicated.

5.0 The Dynamic General-Purpose Record Syntax Info-1

Referring back to the list of features in section 2.0 helps motivate some of the design decisions for a dynamic general-purpose record syntax. The client still needs a way to discover what is being returned from a database of dynamically defined or unknown semantics. It also needs to support documents, images, hierarchies, and hypertext. There is no obvious way to do all this within the existing protocol control kernel, so the next place to look is in the semantic modules for queries and information.

The semantics of information returned by a server is given in three ways. In classical Z39.50, a registered record syntax, such as MARC, informs the client that a stream of bits is structured into specific elements containing well-defined types of data. In a second scheme (the flip side of the generic Use attribute tags in Tables 2 and 3), generic element tags accompany returned data elements. This involves using the record syntax info-1 to hold elements that, for the most part, contain visible ASCII. The third scheme uses the info-1 syntax for records with semantics dynamically defined with Explain and the element set name parameter. This allows for tagged data or, when efficient volume transfer is called for, positional, untagged data.

5.1 Documents

As mentioned earlier, an object citation record provides a natural way to access the object itself, provided it is stored in an electronic form. In considering online documents, several issues come up. How does a client request a document citation minus the full text? How does a full-text data element fragment across records? How do citations relate to links needed for hierarchical browsing and hypertext? How does a client select the form of a document that varies in several dimensions, such as word processing format, language, compression technique, and version? How does it even find out what these variant forms are? These questions generalize easily to other electronic objects such as digitized images.

5.2 Variants

It is not always feasible to index a group of closely related objects separately. This means that there will be cases when a single Z39.50 result set record is the only access point for multiple underlying variant objects. Objects can vary in four dimensions: composition, encoding, language, and version. For example, a pamphlet may be available that has: (1) composition using TEX, Postscript, and Troff; (2) compression encodings of ZIP and "compress"; and (3) text written in French, Spanish, and English.

One citation for this pamphlet would cover 18 variant pamphlets (note that some variants, and objects for that matter, may not need to be stored but are generated on demand). Registered tags known as qualifiers identify variants in client requests using the element set name parameter described below. With the info-1 record syntax, server responses can include qualifiers with returned data. The version qualifier together with a variant message allows a server to define a variant dimension of its own. Clients may find out what variants exist for an element by using the element set name parameter; the server responds with a record, each element of which is empty except for a combination of variant qualifiers. The current qualifiers for info-1 appear in Table 4.

Table 4. Currently Defined Info-1 Qualifier and Value Tags

1 Composition	2 Encoding	3 Language	4 Version
1 Text	1 UNIX compress	1 English	(Server-defined)
2 Hytext	2 UNIX tar	2 French	
3 PostScript	3 JPEG	3 Spanish	
4 TIFF	4 G3 FAX		
5 MARC	5 G4 FAX		
6 SGML+DTD			
7 SGML-DTD			

The composition "Text" refers to an ASCII text variant that would display reasonably in an 80-column window, containing lines terminated by ASCII new line and possibly a carriage return. The composition "Hytext" refers to a hypertext variant similar to Text that may contain short or long objectID references of either the form "@(shortref@ objectID @)shortref@" or "@(longref@ objectID @)longref@."

Although the generality afforded by these variant qualifiers is indispensable, it is unlikely that any given Z39.50 dialogue will make use of more than a handful of them. Instead of including them in each Present, in the long term it makes sense to let them default to values determined when the dialogue is first established; however, at the moment, the protocol does not support this.

5.3 ObjectIDs

Central to any system that supports hypermedia and hierarchical browsing (the second being a special case of the first) is a robust, generalized way to reference a networked object. As pointers to objects, objectIDs are efficient to exchange and remain valid as the underlying objects are updated. If individual records in a search result set have objectIDs, they provide a short cut to accessing those records next time.

Earlier an objectID was defined as a unique identifier used to access an electronic information record associated with an object. As long as each has a unique identifier, objects may have multiple associated information records. Identical copies of an object (e.g., for redistribution) may have different objectIDs for routine access, even though this poses problems for clients that collect objects from disparate servers and need to know when they have more than one copy of the same object. For this reason, the original objectID is carried within the objectIDs of copies.

Each element of an information record contains an aspect of an object (e.g., an electronic instance of it). In order to represent this with the info-1 record syntax, we need to define a field as an info-1 component containing either an entire element or a fragment of it. In order to access an element fragment, a minimal objectID must contain the server name, internal object control number, and fragment address. When objectIDs are used for Z39.50 retrieval, many parameters have to be supplied by convention and much of the generality of Search and Present goes unused. For example, a more generalized Z39.50 objectID might contain server, port, database, attributes, term, element, and fragment.

Beyond the Z39.50 context, a proposal for Universal Document Identifiers (not restricted to text) [9] has made a compelling argument for UDIs (objectIDs) encoded in visible ASCII that transcend protocol (e.g., FTP, WAIS, and Z39.50). The visible ASCII requirement allows objectIDs to be exchanged in e-mail messages and to appear in printed publications.

An important idea missing from the UDI proposal is optional descriptive information. Normally stored as elements of the associated info-1 record, this information needs to be bound closely to objectIDs that will appear in menus, since a separate server access will be required to return elements that the user needs to see (e.g., document title) before a selection will be possible. The close binding makes it easy to update remote menus through a kind of re-linking process. The proposal also requires that no whitespace characters appear in a UDI, but given the length of Z39.50 objectIDs, nonsignificant whitespace needs to be allowed to assist readability. To restrict even one of the many objectID components to, say, the number of characters that fit on one text line is not feasible. It is worth noting the similarity between this objectID format and a text-based query language.

Equipped with hybrid UDI-style objectIDs, the client system programmer has a powerful tool for building hypermedia systems. A few operations on objectIDs would be useful, though not required for interoperability:

(1) Open: begin accessing object.

(2) Read: sequential or random access to object.

(3) Close: end accessing object.

(4) Sync: get fresh copy of objectID.

(5) Compare: client test for identical objectIDs.

In an object-oriented sense, these operations will depend heavily on the server and object's type. Open and Close are merely advice giving operations; for example, they might be ignored on stateless servers that do not keep track of whether an object is open. The Read operation will likely resemble file I/O for document and image objects, with a mix of sequential and random access capability depending on the server. On the other hand, if an object is a database, a server, or a person, even sequential access is unlikely. The Sync operation returns an updated objectID, providing a mechanism for replacing stale descriptive information and obtaining a new forwarding address. One reason for not putting copyright disposition inside an objectID is that even with the Sync operation, an objectID could not normally be trusted to be either current or authoritative. Operations are specified using the element set name parameter.

ObjectID's are hierarchical in that they consist of a sequence of increasingly specific components. The shorter the sequence, the higher level the object (e.g., a fragment has a long sequence). Some of the high-level components may be inferred, so that, for example, every object in a database need not be stored with its full objectID. Not only servers but also clients must be able to parse Z39.50 objectIDs in order to understand what level of object (e.g., server or fragment) is implied. A received objectID could then be modified to imply lower or higher level objects. Here is an example of a Z39.50 objectID identifying the first 4096 bytes of a simple ASCII text version of this article:

```
{ir infocal.berkeley.edu 210
        DB_docid objectID kunz92 object_text 1-4096 {}
        {"Nonbibliographic Applications of Z39.50" 10/16/92}}
```

Rather than use the special purpose notation of the UDI proposal, this example uses a different structure (offered without further explanation at this time as it has not been finalized) expressed with Tcl. [10] This was an easy choice since Tcl expresses hierarchical lists with quoting and non-ASCII capability, and the Tcl software to read and build such lists is freely available for UNIX, Macintosh, and PC platforms.

In generalizing objectIDs to this extent, care must be taken to define what happens when access errors occur or when the object is not a monolithic element fragment. If, for example, the object is a query that returns multiple records from a result set, a reference to the object may cause the client either to start up an interactive dialogue with that result set or to simply display the records noninteractively.

5.4 Satisfying Sections

Sometimes a search locates a record based on criteria not

immediately apparent to the user. For example, a server may match on synonyms generated from the user's term or use a relevance feedback mechanism. Particularly in cases where an element that triggered a match is large and matches at multiple locations, there is a need to return a list of satisfying sections or "hits" within an element.

While a satisfying section may indicate a simple segment of bytes, for some objects a server may disallow byte addressing and offer section addressing instead, where an element (e.g., a document) is divided at the server into a sequence of variable-length sections (e.g., physical page frames or SGML elements). In this case, each section has its own objectID, called a sectionID, that a client submits as the "current" section, relieving the server of having to maintain state information needed for the client to retrieve the next, previous, or current section.

5.5 Element Set Names

The element set name parameter is used by clients to request that returned records be composed of a particular combination of elements. It has been provisionally extended to a composite of several hierarchically arranged parameters that currently do not have a well-defined role within Z39.50 and use the element set name parameter as a temporary home. They specify element set, fragment, variant, and more, as listed in Table 5. Until better solutions are found through experience, a Tcl-based list format for this parameter is suggested, with the provisional format selected (instead of the classical format) whenever the first character of the parameter is the "{" character.

At the top level, clients can request an element set name understood through the Explain facility to refer to a particular combination of elements. By default, info-1 elements are returned in a tagged format, but an efficient untagged (positional) format may be requested. An ordered sequence of field parameters can be used to request a particular record makeup. Each field parameter identifies an element and optional information about operations, variants, and fragments.

Table 5. Element Set Name Parameters

Top-Level Element Set Name Parameters

Unit	Meaning
Esname	Set name discovered via Explain
Zsname	Predefined classical Z39.50 name ("F" or "B")
Untagged	If nonzero, do not tag returned elements
Field	Per field specifiers and qualifiers

Field Specifiers and Qualifiers

Unit	Meaning
Name	Element (tag) requested
ObjectID	Element or fragment identifier
Operation	Open, Close, Read, and Sync

Fragment	Fragment specifier
Variants	Get list of variants if nonzero
Composition	Format (text, hytext, TIFF, MIDI, etc.)
Encoding	Archive/compression (tar, JPEG, MPEG, etc.)
Language	For text (French, English, Spanish, etc.)
Version	Server-defined variant (VMS, Ultrix, DOS, etc.)

Fragment Specifiers

Unit	Meaning
Start	Offset of beginning of fragment (bytes)
End	Offset of end of fragment (bytes)
Length	Length of fragment (bytes)
Section	Next, Previous, First, Last, Current, and Best
Units	Bytes, Lines, Paragraphs, Pages, and Frames

A fragment may be requested by relative section or offset, depending on the element variant. The default unit for numeric fragment specifiers is bytes, but alternate units may be requested. Info-1 field fragments come with flags indicating if the beginning or ending of an element was returned. An element or fragment objectID (once the format is finalized) is capable of expressing all the per field requests and can be used instead of the per field parameters.

+ Page 21 +

It is worth mentioning that the "Best" section specification above makes most sense when the server is returning a record from a result set built according to user-supplied criteria; otherwise, it may mean anything that the server likes. Also, "Pages" specifies a publisher-defined unit.

In Figure 2 are two example element set name parameters. The first one specifies generic elements "object" and "by," with only the first 8192 bytes of a JPEG-compressed TIFF image variant requested. In the second example, an efficient positional sequence of elements is requested.

Figure 2. Element Set Name Parameter Examples

Example 1

```
{field {name "object"
     composition "TIFF"
     encoding "JPEG"
     fragment {start 1 end 8192}}
     field {name "by"}}
```

Example 2

```
{field {name "name"}
     field {name "date"}
     field {name "by"}
     untagged 1}
```

5.6 Extensions to the Present Capability

Some extensions to the Present facility to support object retrieval would be extremely desirable, but formal extensions may benefit by a few short-term conventions. Current proposals for document retrieval call for an internal control number search with a piggy-backed Present of what is expected to be a single record result set. This method was partly chosen to allow for stateless servers that do not keep result sets. In terms of objects, this sort of search is so common and so specialized (in that the result count must be zero or one), that it really belongs as a special kind of Present. Using the objectID element set name parameter in a Present against the reserved result set name ("_NoResultSet_") a server could be truly stateless and not have to support Search at all, let alone the piggy-backed document Present kludge.

Another urgently required extension is batched Present Responses. A request for an entire object (e.g., an image) fragmented into multiple Present Responses at the server would allow for much more efficient data transfer than having the client take receipt of each fragment before being able to request the next one. In the short term, a client Present against the reserved result set name ("_BatchPresent_") could authorize a server to fragment the requested element and send the pieces in multiple responses. A new temporary fragment flag could indicate when the batched responses are done or if the server refuses to batch them.

Those clients that need to retrieve scattered records (e.g., in result set browsing) or those that need the server to send a sampling from a result set also need protocol extensions. Currently record numbers are not returned nor is it possible to request noncontiguous records or records from multiple result sets in a single request. Temporary solutions using new element set name parameters and a new reserved result set name may be called for.

5.7 Copyright Statements

When the copyright component of an info-1 field is present, it contains an objectID that can be used to obtain a copyright statement. Once the client has retrieved and displayed the statement, future occurrences of that copyright identifier during the same session may obviate the need to display it again, depending on the legal obligations.

5.8 Formal Info-1 Structure

In Figure 3 is the formal ASN.1 [11] structure of the general-purpose record format being described. Its main job is to contain a sequence of data fields. A substantial number of diagnostics still need to be added to the bib-1 diagnostic set to support the new functionality that info-1 promises.

Figure 3. Formal Info-1 Record Structure

```
BEGIN

-- Note that lots of things are VisibleString because the
-- client may need to resubmit them in a Present as an element
-- set name parameter.

GenericRecord ::= SEQUENCE {

     elementSetName      ElementSetName OPTIONAL,
     fieldCount          [0] IMPLICIT INTEGER OPTIONAL,

-- These fields allow client shortcuts in record parsing.

     userMessage         [1] IMPLICIT VisibleString OPTIONAL,

-- Anything not covered elsewhere.

     rank                [2] IMPLICIT INTEGER OPTIONAL,

-- Used for weighted result sets.
-- Large data (fields and records) go at the end so clients
-- doing partial parsing of records see headers first.

     positionalFields    [3] IMPLICIT SEQUENCE OF OCTET STRING
                         OPT.,

-- Lightweight for efficient transfer (e.g. tables and files).
-- Positional semantics from Explain and elementSetName.

     taggedFields        [4] IMPLICIT SEQUENCE OF TaggedField
                         OPT.,

-- Generalized fields.

     records             [5] IMPLICIT SEQUENCE OF GenericRecord
                         OPTIONAL

-- Composite/hierarchical record (e.g., holdings records).
-- Records at end allow tail recursion elimination.
}
```



```
TaggedField ::= SEQUENCE {

     tag                 [6] IMPLICIT INTEGER,

-- Tagged fields for variable or unExplained element sets.
-- The same tag may occur in more than one field (e.g., for
-- element variants or multiple abstracting levels).

     value               [7] IMPLICIT OCTET STRING OPTIONAL,

-- Data element fragment.

     objectID            ObjectID OPTIONAL,
```

```
-- Identifies current fragment if value present so that the
-- server need not maintain client's location within an element;
-- identifies "qualified" element if value absent.

     hits                [8] IMPLICIT SEQUENCE OF
                         SatisfyingSection OPT.,

-- Byte offsets and lengths of hits within this element are
-- not necessarily in this fragment.

     copyrightID         ObjectID OPTIONAL,

-- Means element is copyrighted.  This is a special objectID
-- used to retrieve actual copyright on demand; we don't
-- want to ship a legal document with each element.

     flags               [9] IMPLICIT BIT STRING { endOfElement
                         (0), beginningOfElement (1)},
     qualifiers          [10] IMPLICIT SEQUENCE OF Qualifier
                         OPTIONAL,
     variantSize         [11] IMPLICIT INTEGER OPTIONAL,
     variantMessage      [12] IMPLICIT VisibleString OPTIONAL
}

SatisfyingSection ::= SEQUENCE {

     fragmentID          ObjectID OPTIONAL,
     offset              [13] IMPLICIT INTEGER,
     length              [14] IMPLICIT INTEGER
}
```



```
Qualifier ::= SEQUENCE {

     qualifierType       [15] IMPLICIT INTEGER,

-- Composition, Encoding, Language, and Version.

     qualifierValue      [16] IMPLICIT INTEGER

-- Composition: TEX, TIFF, MIDI, etc.
-- Encoding: Compress, tar, JPEG, MPEG, etc.
-- Language: French, English, etc.
-- Version: (Statically undefined).

}

ObjectID ::=             [17] IMPLICIT VisibleString

ElementSetName ::=       [103] IMPLICIT VisibleString
END
```

5.9 Two Examples

It may be useful to walk through an example of a relevance feedback search followed by a retrieval of an electronic document without using the Explain facility. A relevance feedback search involves specifying a special Use attribute called "concept" for a search term containing a segment of text; the server tries to

find documents that are somehow similar to the text and constructs a result set of citation records, each of which identifies a document together with a ranking to indicate the degree of similarity. If the server orders the result set with highest ranked documents first, the Search Response can carry citations (not including the full text) for the most similar documents back to the user.

The client may elect to see the full text of a document in one of two ways. An older way involves doing another search--this time giving a term containing the objectID (from the citation), an "objectID" Use attribute, and a database name that is somehow well-known to the client. This is not a normal search because it must have a single-valued result that is piggy-backed onto the Search Response and is never accessed again as a result set.

A cleaner way to return the text treats the operation as a special retrieval on an objectID, but without a result set. This suggests that retrieval could take place without a prior search, and, in fact, the protocol supports sessions that allow Present while disallowing Search; a number of single-purpose document servers would likely choose such a configuration. Currently, this kind of retrieval would be accomplished by using the Present parameters of element set name for the objectID and result set name for the reserved name "_NoResultSet_." This is illustrated below in Figure 4.

Figure 4. Example of Document Retrieval

---

(1) Send document citation query based on known text segment.

```
{attributeType = 1(use)
attributeValue = 8(concept)
attributeType = 2(relation)
attributeValue = 3(equals)
term = <bytes of text segment go here>}
```

Note: normal search, no special element set name needed since well-behaved servers don't return large objects with citations, but indicate which variants are available for the object field by repeating the field without the optional value component.

(2) Show returned info-1 records, each of the following form.

```
{rank = K
taggedFields = {
{tag = 2(name) value = <document title>}
{tag = 3(by) value = <document author>}
{tag = 4(location) value = <document location>}
{tag = 5(date) value = <publication date>}
{tag = 6(abstract) value = <abstract>}
{tag = 7(object) variantSize = S1 hits = {offset = M1}
qualifiers = {objectID = <documentID1>
qualifierType = 1(composition) qualifierValue = Q1}}
```

```
     {tag = 7(object) variantSize = S2 hits = {offset = M2}
     qualifiers = {objectID = <documentID2>
     qualifierType = 1(composition) qualifierValue = Q2}}
     {tag = 7(object) variantSize = S3  . . .
     }}}

(3) Ask for best section of first document using Present.

     elementSetName = {objectID = <documentID1>name = 6(object)
     fragment = {start = M1 end = M1+4096}}
     resultSetName = _NoResultSet_

(4) Show returned info-1 record, having the following form.

     {taggedFields = {tag = 6(object)  value = <section bytes>}
```

The second example illustrates retrieval against databases of unknown semantics, again without using the Explain facility. Consider a server that allows retrieval of course scheduling information for a university. The server maps the course catalog record components onto generic attribute and element tags and publicizes the database name (but, for this example, not using Explain). Figure 5 shows how the client can retrieve selected elements in an efficient untagged format for all courses taught by "Smith."

Figure 5. Example of Course Catalog Retrieval

```
(1) Send query with appropriate element set name parameters.

     query = {
          attributeType = 1(use) attributeValue = 3(by)
          attributeType = 2(relation) attributeValue = 3(equals)
          attributeType = 3(position) attributeValue = 3(any)
          term = "Smith"}}

     elementSetName = {
          untagged = 1
          {name = 3(by)} {name = 2(name)} {name = 6(abstract)}}

(2) Display positional fields in returned info-1 records.

     record1 = {untaggedFields = {<by1> <name1> <abstract1>}}
     record2 = {untaggedFields = {<by2> <name2> <abstract2>}}
     . . .
```

6.0 Conclusion

Z39.50 is a workable protocol for more than bibliographic information retrieval even though the client system programmer still has to work out some details. Steady growth in the number

of nonbibliographic implementors has flushed out some weaknesses in the protocol. Active development and interest from the computer industry and educational institutions are providing exactly the kind of cross-pollination the protocol needs to become more robust. Key to this evolutionary process will be the containment of a potential explosion in the number of semantic contexts while at the same time making sure that a few contexts are rich enough to build compelling general-purpose user interfaces from them.

References and Notes

1. For a more detailed description of the Z39.50 protocol see: Clifford A. Lynch, "Information Retrieval as a Network Application," Library Hi Tech 8, no. 4 (1990): 57-72.

2. John A. Kunze, "UCB Network Information Server--Project Overview" (Paper presented at the University of California Academic Computing Conference, 1989). (Computer file: help/dist/nis.txt, available via anonymous ftp from ftp.cc.berkeley.edu.)

3. This work was partially supported by Digital Equipment Corporation and Sun Microsystems, Inc.

4. ANSI/NISO Z39.50-199X, Proposed ANSI Information Retrieval Application Service Definition and Protocol Specification for OSI (Vienna, VA: Omnicom Information Service, 1991).

5. Brewster Kahle, "Document Identifiers, or International Standard Book Numbers for the Electronic Age" (n.p.: 1991). (Computer file: ZIG91-46, available via anonymous ftp from think.com.)

6. Clifford Lynch, "Extensions to ISO DP 10162/10163 to Support an Explain Service" (n.p.: 1989). (Computer file: ZIG90-9, available via anonymous ftp from think.com.)

7. Network Development and MARC Standards Office, USMARC Concise Formats for Bibliographic, Authority, and Holdings Data (Washington, DC: Cataloging Distribution Service, Library of Congress, 1988).

+ Page 30 +

8. CWISP Working Group, "Campus Wide Information System Protocol--Version 0.50 RFC, Draft 4" (n.p.: 1991).

9. Tim Berners-Lee et al., "Universal Document Identifiers on the Network" (n.p.: 1992). (Computer file: pub/www/doc/udi1.ps, available via anonymous ftp from info.cern.ch.)

10. John Ousterhout, "Tcl: An Embeddable Command Language," in Proceedings USENIX Winter Conference, January 1990 (Berkeley: The USENIX Association, 1990).

11. International Organization for Standardization, OSI Specification of Abstract Syntax Notation One (ASN.1) (Vienna, VA: Omnicom, Inc., 1987).

About the Author

John A. Kunze, Information Systems and Technology, 289 Evans Hall, UC Berkeley, Berkeley, CA 94720, (510) 642-1530. Internet: jak@violet.berkeley.edu.

The Public-Access Computer Systems Review is an electronic journal that is distributed on BITNET, Internet, and other computer networks. There is no subscription fee.

To subscribe, send an e-mail message to LISTSERV@UHUPVM1 (BITNET) or LISTSERV@UHUPVM1.UH.EDU (Internet) that says: SUBSCRIBE PACS-P First Name Last Name. PACS-P subscribers also receive two electronic newsletters: Current Cites and Public-Access Computer Systems News.
