Architecture of a Content Management Server
for XML Document Applications

Tim Arnold-Moore, Michael Fuller, Alan Kent, Ron Sacks-Davis, and Neil Sharman

RMIT Multimedia Database Systems, Royal Melbourne Institute of Technology,
GPO Box 2476V, Melbourne 3000, Victoria, Australia.

{tja,msf,ajk,rsd,neil}@mds.rmit.edu.au

Abstract

The paper describes the data model used to implement the SIM Content Management Server, an SGML/XML-native content server designed to support extremely fast data access and dynamic update of 100Gb collections under high load. The paper describes requirements for supporting text intensive applications and for building XML/SGML document management solutions. The SIM CMS employs a data model designed to directly support SGML and XML; this model is described, and a comparison to other models based on general purpose data base management systems is made.

1. Introduction

Structured documents, which can be seen to be a particular instance of semi-structured data, possess characteristics that arguably make the relational and object-oriented approaches less than appropriate choices as the basis for structured document database systems (also known as content management systems) [5]. One possible alternative is to use a data model directly based on a document representation format such as SGML or XML. This approach lends itself naturally to efficient implementation of several important areas of content management, namely, document validation; efficient content and structure indexing; structure, content, and hypertext querying; scalability; document versioning; and document management. This paper describes an implemented system based on such an approach.

The SIM content management system is built around a content management server, the SIM CMS. The SIM CMS is based on the Z39.50 protocol [1], a client/server protocol commonly used over TCP/IP for information retrieval applications. An overview of the SIM CMS data model is presented in Figure 1. Physical record structures describe how records are physically stored. Logical record structures define a public view of the stored data. Mappings define the relationship between the logical and the physical record structures. To facilitate searching of collections, indexes can be defined on fields in logical records. Both simple and complex search requests can be directly resolved from these indexes, without needing to examine the stored physical records. When requested, those physical records matching the search conditions are retrieved from the database for presentation.

![Figure 1. Overview of the data model used in the SIM Content Management Server](image)

The CMS stores SGML and XML documents natively, parsing documents for validation, index generation, and result set preparation. The structure of documents inserted into the database can be validated against the SGML
or XML DTD. Treating SGML and XML documents as native database objects has many advantages. No translation to relational or object oriented database formats is required, and documents do not have to be fragmented across multiple relational tables. Storing the document data directly as SGML or XML permits highly customized indexing and flexible data manipulation and delivery.

User access to the SIM system may be indirect over HTTP via a custom web server designed to handle high access loads and to provide stateful HTTP session management, or direct to the CMS itself through a Z39.50 interface.

2. Data model

The CMS data model has four components: the physical record structure, which defines the internal, physical representation of data in the repository; the logical record structure, which defines a public, abstract representation for stored data; a set of mappings, which define the relationship between the logical view of the database and the physical record structure; and a set of indexes, based on elements of the logical record structure and which also define access points into the repository. These four components are used to resolve search requests and retrieve data. Searches may be made against indexed elements of logical records. Resolution of search requests uses the indexes appropriate to the nominated logical elements to identify a result set of matching records. Present requests may be made for elements of a result set. Resolution of a present request uses the mappings between physical data and logical structure to identify or derive the data corresponding to the requested logical elements from the stored physical data.

2.1. Physical records

Data is stored intact in the CMS as individual physical records. A record may contain multiple fields. A field may contain raw data, simple data of a basic type, or may contain complex data, such as SGML or XML. SGML and XML data is parsed on insertion as part of the storage and indexing process. For example, a physical record may have a field of type XML containing a complete XML document and several other fields containing metadata associated with the XML document. Using Z39.50, XML documents and associated metadata can easily be managed together as a single record.

A field may also be defined as a nested structure. Fields may be optional, and may be repeating. Fields may be stored in compressed form or may be marked as deferred - to be accessed from disk only when required. For text databases, this is particularly important as individual documents can be very large.

Record fields may be virtual and defined using an embedded SGML/XML enabled programming language. Virtual fields can be derived from other physical fields at run time, or may be cached. Cached virtual fields are pre-calculated and stored on disk; this is sometimes important for efficiency, for example when calculation of a virtual field requires parsing a large SGML record.

The native data types supported by SIM include SGML, XML, MARC, BINARY, STRING, INTEGER, FLOAT, BOOLEAN, DATETIME, DURATION, and STRUCTURED.

2.2. Logical records

The CMS logical record structure provides a public, conceptual view of data, defining logical elements and sets of elements. Logical elements form the basis for indexing and searching; element sets group elements for presentation. Logical elements are defined in terms of mappings from a physical element. Mappings may be direct, or may be derived.

2.3. Internal SGML/XML support

The CMS embeds two standard SGML/XML parsers that are used during document insertion, conversion, indexing, and retrieval. To reduce the number of API's, a single scripting language, called Ace, is used within all layers of the architecture. ACE provides full SGML/XML functionality such as document parsing, tree walking, and tree manipulation and transformation [19]. In particular, this language allows automated custom conversion of documents on insertion and retrieval, and the creation of specialized indexes. A fully featured, multi-threaded,
object-oriented programming language, it is therefore closer to DSSSL’s SDQL [11] in terms of functionality than to, say, Omnigraph [9] or XSLT [24]. The CMS also contains a full XPath [21] engine that can be used during index creation and during document manipulation. This engine is also the basis for an XSLT processing engine that can serve as a simpler but standards-based alternative to the ACE language.

3. Example: A Memo Database

In this the following we give an illustrative example of how physical and logical records might be defined, and provide some further insights into the features that are provided.

In this example, a database of memos will be constructed where the memos will be marked up using SGML. The DTD, named memo.dtd in this example, is as follows:

```xml
<!ELEMENT memo  (to?, p*, from?)>
<!ELEMENT to   (#PCDATA) >
<!ELEMENT from (#PCDATA) >
<!ELEMENT p    (#PCDATA|bold) >
<!ELEMENT bold (#PCDATA) >
```

An example memo conforming to this DTD is:

```xml
<!DOCTYPE memo SYSTEM "memo.dtd">
<memo>
<to>Sally</to>
<p>Dear Sally</p>
<p>I really wanted to thank you for all your help. The demonstration was a success. See you when I get back from overseas!</p>
<from>Harry</from>
</memo>
```

When SGML data is retrieved from a database, not all of the document needs to be retrieved, or the information retrieved may be a transformation of the stored physical document. In our example, clients might ask for only the sender or recipient. This means that all portions of memos that can be left out must be made optional in a DTD or else the returned text will not conform to the DTD. Rather than change the original DTD, a new DTD is created called a retrieval DTD. For the memo database, the only changes to be made to the original DTD are to make the ‘to’ element optional.

A common requirement in applications is that the search terms present in retrieved documents should be highlighted. SIM handles this by embedding SGML processing instructions in the retrieved document that identify the terms to be highlighted. This can alternatively be addressed by using SGML inclusions in the retrieval DTD that specify highlighting elements that can be inserted into retrieved documents.

For this database, indexes will be set up to enable searching on the sender and recipient memos, as well as the main body of the memo. For the sake of example, all of the bold keywords will be indexed separately, a cached virtual field will be created summarizing the memo by returning the first 30 characters of each paragraph concatenated as a string, and the time the memo was created will be retrieved with memos.

First a logical record structure will be defined, with fields: sender, recipient, text, keywords, summary, and created. In addition two element sets F (Full) and B (Brief) are defined. Element sets allow clients to specify a number of primitive elements with a single reference. An element set is a named definition that lists the logical record structure elements to be used when the element set name is given. This allows another level of abstraction to be used, such as the full view of a record and the brief view of a record. SIM requires that a full and brief element set definition exist for all databases so that clients are guaranteed to be able to ask for the full or summary versions of any record from any database. This is a common requirement in Z39.50 servers that simplifies the development of standard Z39.50 clients. Within SIM, it also simplifies the creation of applications that perform searches across multiple databases in parallel.
Next, we create the physical record structure. Each physical records will contain a memo, an SGML document conforming to the "memo.dtd" DTD and with an associated retrieval DTD, "memo-rtr.dtd". An additional field containing some related metadata, namely the time when the record was created; and a virtual field containing a summary of the contents of a memo that will be derived from the SGML data.
some of the SGML from the MEMO document instance.

```
DOLL CREATE MAP RECSSTRUCT MemoStruct

ELEMENT sender IS
  { rec.sgmlDocument("memo")
    .root()
    .all("FROM")
  }

ELEMENT recipient IS
  { rec.sgmlDocument("memo")
    .root()
    .all("TO")
  }

ELEMENT text IS
  { rec.sgmlDocument("memo")
    .root()
    .all("P")
  }

ELEMENT keywords IS
  { rec.sgmlDocument("memo")
    .root()
    .all("BOLD")
  }

ELEMENT summary IS summary
ELEMENT created IS created
END;
```

4. Indexing and querying

Indexes are created with respect to logical elements, and can be defined at the word, sentence, or paragraph level using default or custom term parsers, custom finite state machines, custom code, or XPath index proseing. Indexes provide access points, based on logical elements and elements sets. Storing full word, sentence, and paragraph positional information, this approach allows Boolean and ranked queries, phrase and proximity queries, structure and field queries, and index scanning queries to be fully evaluated directly from indexes; query resolution never requires inspecting stored data.

The index layout uses compressed, inverted files with buffered update, and is designed for the support of high performance querying concurrent with interactive updates (insertions, deletions and modifications) for terabyte document collections. Buffers are flushed when necessary, with a write-ahead log for error recovery. Storing full term position information, indexes are compressed using run-length encoding and implement skipping, avoiding the need to decompress an entire index to retrieve information for a single term. Index sizes are typically between 10% to 50% of the size of the source data; the larger index sizes allow for the storage of full word, sentence and paragraph positional information and for statistical information about the frequencies of words within each document and within the whole database, required for document ranking.

Given a block of text to be indexed, it is necessary to extract the terms to be indexed (typically words or phrases, but possibly n-grams or other tokens). ACE scripts can be defined to extract these terms from records. However, scripts are not generally used for performance reasons. Because extracting terms from text is such a common operation, special purpose code is provided; several predefined term parsers exist for word indexing (the W1 parser) and for phrase indexing (the PHRASE1 parser). Different parsers split text into words or phrases, map lower case to upper case to allow efficient case insensitive searches. Database designers are also able to write their own parsing rules when the default rules for punctuation or term boundaries are not appropriate; for example, parsers can be defined that perform stopping or stemming (although because the compressed indexes are most efficient for common words, the need to use stopping and stemming to reduce indexing overhead is minimal, in which case stopping and stemming may in fact unnecessarily impair query functionality—instead, these operations can simply be applied to those queries that may benefit). Custom parsers for indexing can either be defined as a finite state machine, or written as dynamically loadable C libraries.

To create an index, a number of data structures, including inverted files, B-Trees, and hash files can be specified. In the example below, a word level index is created for a number of fields within the memo records using an inverted index. The SGML mark up is filtered out using ACE scripts within the index definitions.

101
5. Requirements for text intensive applications

Applications involving large document collections are text intensive: large volume of structured text must be stored, retrieved and manipulated. Text intensive applications include web portals, technical documentation systems, legislation systems and digital libraries. These applications differ fundamentally from transaction intensive business-to-business applications. In this section we describe some of the requirements for content management systems designed to support text intensive applications.

5.1. Searching

Efficient support of queries incorporating Boolean operators (AND, OR, NOT, and XOR), term proximity (word distance, same sentence, and same paragraph) and term patterns (truncation, wild cards, stemming, and fuzzy matching) is required. The Z39.50 protocol allows a history of previous searches to be available for incorporation in later queries. Thus, previous results can be combined to refine queries. Web based sessions also must be stateful so it is possible to revisit previous states.

Searching must also support ranking of documents as is commonly available in Internet search engines. SIM supports both ranked queries (treating all the query terms as a conjunction) and also ranking of the results of previous queries. This allows mixed Boolean and relevance searching. For example, a query about the policies of President Clinton with respect to social security might be expressed first as a Boolean query to retrieve all records containing the term “Clinton” followed by a relevance query incorporating terms such as social security and welfare.

Other ranking techniques that we have found useful and are supported by the SIM CMS include ranking using logical structure of records, user term weighting, and weighting based on the logical structure of records (so that, for example, a word appearing in a title has greater weight than the same word in the body of a document).

5.2. Term Highlighting

A seemingly innocuous requirement is to highlight matching terms with respect to a query. However documents are constantly being transformed and modified, so tracking the position of words within a document is non-trivial. It is also necessary to only highlight those terms in a document that contribute to the document matching the query. Thus there may be many occurrences of the terms “to”, “be”, “and”, and “not” in a matching document, but if the query is “to be or not to be” only those occurrences that form that phrase should be highlighted. Term hits must be mapped from the logical to the physical layer and term highlighting must also work when scripts are used to define virtual fields.

In SIM, term highlighting is SGML/XML aware. This means that if a Z39.50 element is defined by selecting text within a particular XML element, only occurrences of the query term in that element will be highlighted. By default, processing instructions are inserted around the matching terms. Matching records containing highlighted terms are SGML/XML conformant, and correspond to the retrieval DTD defined in the physical layer. SIM can be configured to insert user-defined markers for term highlighting.
5.3. Selective Dissemination of Information

For many information retrieval applications it is desirable to allow users to define standing expressions of interest to be lodged with the database server. These queries are then automatically invoked periodically and the results of any matching records are then sent to the user. For databases of news feeds, for example, users often want to be informed when any new relevant news item is inserted into the database. Selective dissemination of information is supported by SIM and the full power of SIM’s query language can be used to define an expression of interest. The action that is taken if matching records are found can be defined using Ace. This might include transforming the matching records into an appropriate HTML summary and e-mailing them to the user who set up the expression of interest.

5.4. Scanning Indexes

As well as enabling and accelerating advanced forms of text queries, the indexes of full text databases hold much valuable information. Indexes can be scanned to determine the frequencies of words, to help in the formulation of queries, to locate spelling errors, and to discover more about the usage of a term in a database. More complicated uses of scanning indexes involve analysis of the term distribution in a database. Typically, indexes are maintained so that terms can be returned to a user in sorted order. Because SGML/XML documents directly correspond to database objects, it is possible to support the scanning of indexes. In contrast, scanning is not generally supported in relational databases where documents are fragmented over multiple tables.

5.5. Database Slices

For applications such as technical documentation or legislation systems, it is extremely useful to be able to recreate an old version of a database or a complete subset of a database. We refer to these views of a database as a database slice. In a database slice, not only should retrieved documents reflect the correct database version or subset, but also all retrieved hypertext links, vocabulary terms and term frequencies should accurately reflect the correct database version.

Consider, for example, a legislation database containing law that is constantly being amended and changing over time. The support of database slices means that the correct state of the law at a previous point in time can be created automatically, critical for many legal applications. There are many other examples where database slices are extremely useful, including point-in-time searching, querying on record types and querying on geographic location.

SIM provides support for database slices. While similar to relational views, the requirements that result from the nature of the text data, the support of vocabularies and for scanned indexes, provide a different set of requirements to the support of relational views.

6. Building XML/SGML Applications

Standard architectures for document management are based on the use of relational or object oriented general-purpose database management system. These systems are not designed to manage XML or SGML data. Complex table definitions are usually required to map XML/SGML data onto relational tables. Many relational implementations break XML/SGML data into very small objects. Documents are not stored in their native format but are fragmented over numerous tables. To access these tables, indexes created are indexes to relations not to documents.

A typical architecture for such an approach is presented below. Because general-purpose systems are used, a number of different layers must be supported and each layer implies, a possibly different product and vendor. XML-aware operations are handled by a parsing processing engine that is typically separate from the underlying components.

Also implied by the number of layers required in such a system is a large number of APIs (Application Programming Interfaces) that must be understood and maintained by the system administrators and system programmers. For example, these interfaces
might include HTTP/HTML and CGI for the Web layer, a document manipulation language such as Perl, a language such as XSL or DSSSL for the electronic publishing component.

In contrast to the RDBMS approach, the XML-native approach, adopted by SIM, is presented in Figure 3. The architecture consists of the content server and a customizable web server. In this approach XML or SGML documents are stored directly in the repository within tables or documents whose schemas are described by their XML or SGML DTD (Data Type Definition). For long documents, it may be desirable to split these into smaller fragments; in this case, fragments, like whole documents, are stored directly in the repository. No translation from the SGML or XML DTD to a relational schema and no fragmentation of documents into relations is required. In an XML-direct system, the parser is incorporated directly within the processing engine, closely coupled with the workflow, versioning, document management, and querying systems.

Several major benefits result from the XML-direct approach. First queries are more efficiently processed as indexes directly access documents rather than numerous tables, and documents that are retrieved as a result of a query do not have to be reconstructed from multiple fragments stored across many tables. Another significant benefit of the XML-direct approach is that the number of layers in the architecture that must be supported is greatly reduced. The architecture described below is fully supported by SIM so the technology is single sourced. Users do not have to deal with multiple vendors and implement a large number of interfaces and APIs. In the architecture below, a single programming language, ACE, can be utilized throughout the system. A smaller number of layers means much lower support costs to maintain the application over time.
7. Extended XML support

The data model and architecture described here allow XML and SGML documents to be flexibly and efficiently stored in large numbers and volume, their textual content indexed, and allows dynamic update and retrieval under high demand. These abilities by themselves enable much of the functionality current applications demand of a content management system. The next generation of XML systems, however, introduces new requirements including structural querying, quasi-relational operations, standardized document transformation and manipulation, integrated hypertext linking, extended schema and metadata support, and distributed document management.

Unlike the flat text handled by traditional information retrieval systems, the binary objects with metadata stored by existing document management systems, or the tabular data of relational database systems, XML data is explicitly structured, with each element positioned within its context of parent and ancestors, siblings, and children and descendants. Indeed, the cross-relation structures modeled in relational databases by the use of keys, can be directly represented as hierarchical XML structures. Examples of new query languages that address (to varying degrees) the needs of XML include PAT [10], SGQL [3], HyQ [12], SDQL [11], UnQL [5], Lorel [2], XQL [13], XML-QL [8], YATL [7], and Quilt [14].

What is needed to support such languages? Previously, we have seen that XML data can be directly stored, unchanged, in the SIM CMS, and that individual logical fields—programmatically derived from the source document—can be custom indexed at the word, sentence, and paragraph level. This, when used with Boolean, proximity, and wildcard operators, gives efficient ranked querying, and support for simple structural queries (such as containment within a specific element, that perhaps has a specific context). These queries can be resolved directly from indexes. However, the set of structural queries that can currently be supported is restricted to those known at the time of database (specifically, index) creation.

In a relational database, structure is modeled by using keys to link relations, and querying structural relationships requires nested queries and complex join operations. In XML data, however, the structure is explicit and can be indexed directly. The majority of structural queries can be satisfied directly with such an index: some queries, the equivalent of relational joins, require that the results of multiple index lookups need to be combined. Structural queries can therefore be resolved far more efficiently. This requires a new type of index that summarizes structural information, such as either a position index or a path index, described by Sacks-Davis et al. [16]. A position index provides information about the word position of document terms and elements. Alternatively, a path index describes the structural context of document terms. An interesting class of structural queries is that involving path expressions, the structural equivalent of text data’s regular expressions. Path expressions are part of most of the recent XML or semi-structured data query languages, with XPath [21] the best-known definition. Interestingly, there is a direct correlation between path expressions and path indexes, suggesting the latter may be the appropriate choice to support XML querying.
It is normal for a relational query to present some variation of the data identified during the course of query processing. XML data, inherently, may be more complex than relational data; it is common that only a subset of a particular XML document, or a transformed variant of the document, be presented in response to a query. The SIM CMS already supports pre-computed sub-sets and transformations, embodied in the mapping from physical data to logical field. These transformations are pre-defined during database construction, using XSLT stylesheets. An alternative would be to use XSLT stylesheets to achieve the same end. While this increases the efficiency with which they can be carried out—for example, fields may be cached—it does not provide a general-purpose solution. Further, a transformation component is also proposed as part of new XML query languages [14]. Clearly, next-generation XML content management must have embedded support for generic transformation.

Just as SGML was the basis for the hypertext standard, HYTime [12], XML was designed for use in an online, hypertext environment. The W3C XLink [22] and XPointer [23] recommendations define syntax and semantics for generic hypertext linking in XML. XPointer defines how to locate specific points or ranges within nominated XML documents, using XPath path expressions. XLink builds on XPointer to define how one-to-one, one-to-many, and many-to-many inter- and intra-document relationships can be represented. As a pair, they provide a significant superset of the hypertext functionality already provided by HTML. However, rather more interesting than their ability to be used to implement browsable hypertexts is their use to describe generic inter-object relationships, to associate one node in an XML tree with another node, possibly in a separate tree. This extends the data modeling capabilities of XML, previously restricted to describing tree structures (ignoring the limited ID/IDREF functionality inherited from SGML), to modeling general graph structures. A new feature of the SIM CMS is the ability to define relationship indexes, which associate one specific node with another. This new index type can allow relationships to be efficiently resolved on demand.

The traditional mechanism for defining legal SGML and XML document structures is the document type definition, or DTD. DTDs are extremely limited in the structural and data constraints that they can describe. In contrast, XML Schemas allow detailed constraints to be represented. These constraints are interesting in that they can serve directly as the data definition in the CMS, in particular allowing data integrity to be aggressively enforced.

An interesting new application area is that of wireless devices. Such devices, in contrast to normal consumers of XML data, have extremely limited bandwidth with which to receive data, and limited computational resources with which to process it. The W3C Binary XML Format [25] addresses both issues, specifying a pre-parsed, stripped, and tokenized format for XML documents. This format need not be restricted to document delivery, however: XML documents can be efficiently stored in binary format within the CMS, potentially giving significant performance gains in document access, transformation, and manipulation.

8. Conclusions

The SIM Content Management Server is designed from the ground up to support XML and SGML. The ability to store XML in its native form avoids the need to recompute documents at retrieval time. The content server has been designed for text intensive operations; the CMS in multithreaded for performance, so that simple queries against a 100Gb document collection can be processed within a few seconds on standard workstation architectures and the CMS can support the thousands of ongoing, dynamic index updates that occur as a result of interactive document insertions and updates.

By comparison, general purpose database management systems such as relational database systems are designed for transaction based applications on highly structured data and XML/SGML awareness is normally layered on top of these systems. This results in complex indexing requirements and significant pre-processing of documents.
The SIM CMS data model is based on the Z39.50 standard. We have demonstrated how using this standard support for both physical and logical views is provided. An XML/SGML aware scripting language for text and data processing is used throughout the system and provides a means to translate between the logical and physical layers. The SIM CMS supports Boolean and ranking searches, term highlighting, selective dissemination of information, index scanning, database slices and other features highly desirable for text intensive applications. For document management applications a simple two-layered architecture is provided compared to the multi-layered architecture that results when a general-purpose relational database system is deployed and which typically requires many products, typically sourced by different vendors, and many APIs. The maintenance costs for the simplified architecture will be significantly lower as a result of the dependence on a smaller number products and APIs.

9. References


