Document Indexer Persistence Component Specification

1. Design

The *Document Indexer Persistence* component implements the persistence layer as required by the Document Indexer component. The pluggable framework allows different persistence mechanisms to be used. For the initial version, two mechanisms (XML and database) are provided. The primary emphasis of this design is on a fast retrieval solution, since we are dealing with an indexed document, which could consist of thousands of indexed words any gains in speed of retrieval will be compounded when collections of documents are loaded since a collection could be made of a multitude if indexed documents.

1.1.1 Anatomy of the proposed design

The design itself is very simple. We have two implementations:

1. Database driven persistence
2. XML file driven persistence

1.1.1.1 Database considerations

The main aspect of the design here will be the fact that the designer decided after some research to use a compact structure of a CLOB (or byte data) to store all the words and their positions for a given document. This will speed up retrieval in an order of about 40x (i.e. more than 40000% which is tremendous boost for the component's usability - please look at the provided benchmark exposition in Appendix B of this document, you could confirm the results for yourself) but there is a caveat to this approach in the sense that it hides the word data in a single record and thus makes some data mining more difficult. Here is a better expose of this.

There are two ways to tackle the issue of table design:

- We simply normalize the data and create four tables. First table will hold the document id, document usage count, and all the `WordSourceID` information except the delimiter list, the second table will hold all the delimiters that were used in indexing this document, the third table will hold all the words related by the document id, and finally the fourth table would contain for each word the positions where the word appears in the document.

- If we replace the third and fourth tables mentioned above with a single table, which holds a CLOB of all the words and their positions, we will obtain a much more compact storage than the previous one. While the other storage is a bit more 'friendly' since we can easily query the database for some additional relationships such as show me all the documents that contain the word "Hello", the speed retrieval benefits far outweigh the 'friendliness' of the first approach.

Thus we have speed vs. simplicity of design issue here. Since the component is very simple the designer has decided to provide two implementations of data base persistence:

- A **fast implementation** that uses CLOB data storage type.
- A **simple implementation** that is done in a more traditional manner.

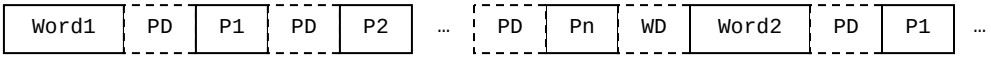
The user will then have a simple pluggable choice of what they would like to use.

NOTE: in Informix A CLOB column can be up to 4 terabytes in length. This is so large that we do not have to anticipate using multiple CLOB records to store a single index.

There is another consideration to be taken care of when dealing with the fast implementation. Since we are creating a stream of sequential data we need to know how to parse the words and their positions from the Unicode char stream. Here we could have a small problem: if we wanted to use delimiters to specify when one words begins and another ends we have to be extra careful not to use delimiters that could actually be part of a legitimate word (since then we would split a word into two words) The good news is that the design already provides us with the solution. Since we persist the delimiter list used for this document index we will use the delimiters from this list to delimit the CLOB data. This way there will be no chance of any conflicts.

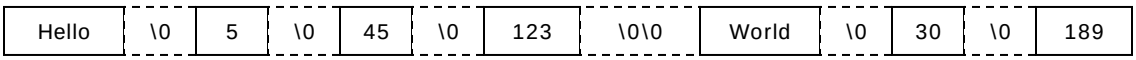
1.1.1.1.1 Proposed CLOB structure

We simply take the first delimiter in that list and use it as a delimiter in our CLOB. As an example assume that the delimiter we will use is the null character **'\0'**, we will simply use a single version of this delimiter to delimit positions and a double occurrence of this delimiter to delimit words. Here is the CLOB structure to be used and an example with the delimiter being a '\0'



Where PD is a position delimiter and WD is a word delimiter, and where Px is position x for the word in the document.

Here is an example:



Note that we use the \0 character as a position delimiter and a twice-repeated \0 as a word delimiter. This means that all that we ever need is a single delimiter and the design guarantees that we will never get an empty list of delimiters.

1.1.1.2 XML considerations

Given the requirements that we store indexed data in separate files for each document we already have a simple way of dealing with document index: we simply store each document with a specific xml file and to keep the structure simple we create a document node which then has word sub nodes where each word sub-node will store all the word positions again as sub nodes. Here is a simplified overview:

```
<doc xml:lang="en-US"> // note that for locale we will also use xml specific tag.
   <document-index document-id="16527">
        <word-source-data>
            <source-identity> identity object as a base64 encoded object
            </source-identity>
            <class-name> class name </class-name>
            <locale language="en" country="US" variant="POSIX"/>
            <delimiters>
                <entry delimiter=" "/>
                <entry delimiter="\n"/>
                <entry delimiter="\r"/>
                <entry delimiter="\t"/>
            </delimiters>
        </word-source-data>
        <word-index>
            <word> Hello </word>
                <pos> 24 </pos>
                 <pos> 56 </pos>
                        . . .
        </word-index>
                . . .
    </document-index>
</doc>
```

Collections will be quite simple since all we need to collection is the document ids as follows:

```
<document-collection collection-id="37684">
        <documents>
                <document-id>16572</document-id>
                <document-id>46537</document-id>
                        . . .
        </ documents>
</document-collection>
```

Another thing to consider is how do we relate the usage of documents? The designer has decided to create a relating (basically we are modeling a relationship of usage to document and document to all the collections that it belongs to) xml as follows:

```
<documents>
    <document-usage document-id="16572" usage-count=5>
            <collection-inclusion>
                    <collection collection-id="37684"/>
                    <collection collection-id="11111"/>
            </collection-inclusion>
    </document-usage>
    <document-usage document-id="46537" usage-count=0>
    </document-usage>
                        . . .
</documents>
```

The reason for this extra document is that this document will be updated possibly often and since xml updates basically mean file rewrites we need to ensure that the file is reasonably small (as compared to rewriting an document index file) and thus we guarantee a relatively fast update. This file will only be overwritten when data in a collection changes, either as a result of a document addition or document deletion. This file will always be named `document_usage.xml`

1.1.1.2.1 How do we use all these files?

The basic idea is that for each document we will have an xml file, for each collection we will have an xml file, and we will exactly one xml file for document usage.

But we still have a very important issue unanswered: how do we identify which xml file belongs to which collection/document? Also there is the issue of storage limit. Since files have a limited capacity (such as 2G for example), how do we ensure that we can store files larger than that if need be?

First we will tackle the general file naming which is very simple.

- Each collection xml file name will have the following format: `doc_collection_collectionId.xml`
- Each document index file name will have the following format: `doc_index_docId_sequenceid.xml`

For example here is a simple document collection xml file for a collection with id of 175658:

```
doc_collection_175658_.xml
```

Here is an example of a document index xml file for a document with id of 16527:

```
doc_index_16527_001.xml
```

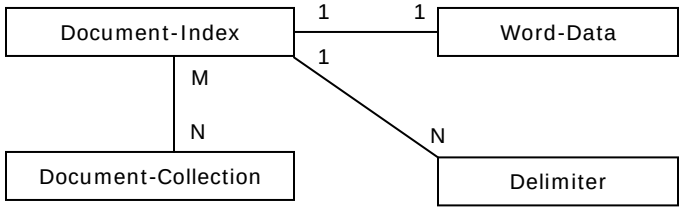
The *sequenceid* is used to 'link' files with data that is larger than say a limit of 2G (this can be configured) this means that if we have indexed data that is larger than say 2G we split the file into 2 files with the first having the sequence id of 001 and the second one being obviously 002. This gives the flexibility for persistence of very large content.

1.1.2 Data base diagram, ERD, and DDL

Here is a simplified data base diagram, which is followed, by an ERD and then a DDL for Informix.

Data base diagram

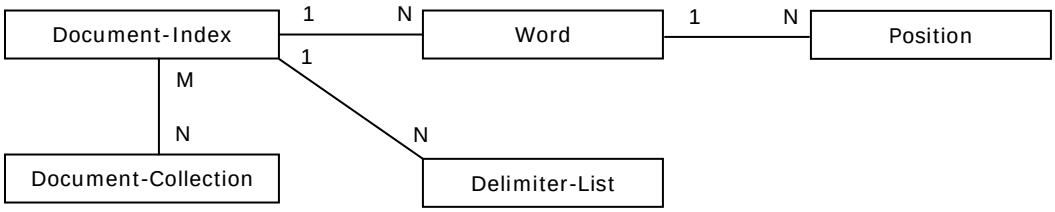
Fast access solution



The above basically states the following

1. A document has one record (a CLOB) associated with it which contains all the word data
2. A document can have one or more delimiters associated with it
3. A document can be associated to many collections. Here we have actually 0 or more. It is possible for a document to belong to no collection.
4. A collection can hold many documents. Here we have actually 0 or more. It is possible for a document collection to contain no documents (i.e. an empty collection)

Simple solution



The above basically states the following

1. A document has many words
2. Each word can have many positions
3. A document can have one or more delimiters associated with it
4. A document can be associated with many collections.
5. A collection can hold many documents

ERD

Two ERD diagrams have been placed in the Appendix (Appendix A) to this document as well as in the \docs directory as JPEG files.

DDL

Simple Access Solution:

```
CREATE TABLE DOCUMENT_COLLECTION_DOCUMENT_XREF
(
	COLLECTION_ID VARCHAR(40),
	DOCUMENT_ID VARCHAR(40),
	FOREIGN KEY (COLLECTION_ID) REFERENCES DOCUMENT_ COLLECTION,
	FOREIGN KEY (DOCUMENT_ID) REFERENCES DOCUMENT
)

CREATE TABLE DOCUMENT_ COLLECTION
(
	COLLECTION_ID VARCHAR(40) PRIMARY KEY
)

CREATE TABLE DOCUMENT
(
	DOCUMENT_ID VARCHAR(40) PRIMARY KEY,
	LOCALE VARCHAR(30),
	SOURCE_IDENTITY BLOB,
	CLASS_NAME VARCHAR(255),
	USE_COUNT INTEGER
)

CREATE TABLE DELIMITER
(
	DOCUMENT_ID VARCHAR(40),
	DELIMITER_ENTRY VARCHAR(10),
	FOREIGN KEY (DOCUMENT_ID) REFERENCES DOCUMENT
)
```

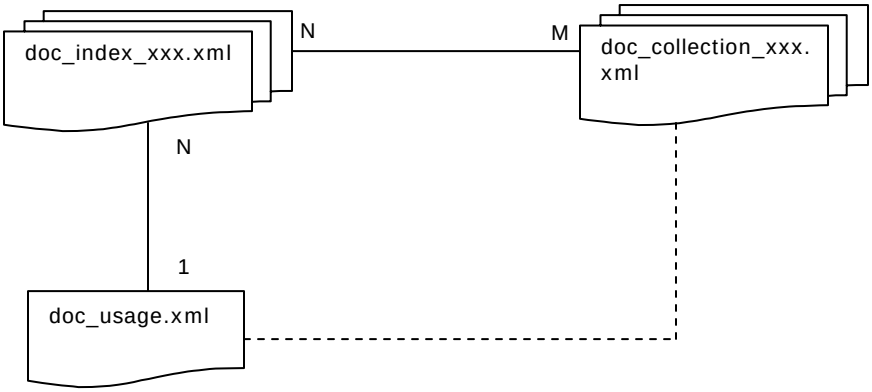
```
CREATE TABLE WORD
(
       WORD_ID INTEGER PRIMARY KEY,
       DOCUMENT_ID VARCHAR(40),
       WORD_ENTRY VARCHAR(255),
       FOREIGN KEY (DOCUMENT_ID) REFERENCES DOCUMENT
)

CREATE TABLE WORD_POSITION
(
       DOCUMENT_ID VARCHAR(40),
       WORD_ID INTEGER,
       WORD_POSITION INTEGER,
       FOREIGN KEY (DOCUMENT_ID) REFERENCES DOCUMENT,
       FOREIGN KEY (WORD_ID) REFERENCES WORD,
)
```

Fast Solution

```
CREATE TABLE DOCUMENT_COLLECTION_DOCUMENT_XREF
(
       COLLECTION_ID VARCHAR(40),
       DOCUMENT_ID VARCHAR(40),
       FOREIGN KEY (COLLECTION_ID) REFERENCES DOCUMENT_ COLLECTION,
       FOREIGN KEY (DOCUMENT_ID) REFERENCES DOCUMENT
)

CREATE TABLE DOCUMENT_COLLECTION
(
       COLLECTION_ID VARCHAR(40) PRIMARY KEY
)

CREATE TABLE DOCUMENT
(
       DOCUMENT_ID VARCHAR(40) PRIMARY KEY,
       LOCALE VARCHAR(30),
       SOURCE_IDENTITY BLOB,
       CLASS_NAME VARCHAR(255),
       USE_COUNT INTEGER
)

CREATE TABLE DELIMITER
(
       DOCUMENT_ID VARCHAR(40),
       DELIMITER_ENTRY VARCHAR(10),
       FOREIGN KEY (DOCUMENT_ID) REFERENCES DOCUMENT
)

CREATE TABLE WORD_DATA
(
       DOCUMENT_ID VARCHAR(40),
       WORD_ENTRY CLOB,
       FOREIGN KEY (DOCUMENT_ID) REFERENCES DOCUMENT
)
```

1.1.3 XML diagram and XSD schema

Simple file break-down



The above diagram is really a simple depiction of the following:

1. A single document index might have a number of sequences
2. There can be many documents referred in the document usage file.
3. A single document collection might have a number of sequences
4. A Collection may consist of many documents (0 or more)
5. A document can belong to many Collections (0 or more)
6. There is only one usage document, which tracks all document index usage in the Collections. This is very important when we would like to ask a question such as which Collections does this document index belong to?

XSDs for all the xml files
Those are located in the \docs directory.

1.2 Design Patterns

No specific pattern shave been utilized since this is an implementation of existing interfaces with plug-in capability (i.e. Strategy) and manager abstraction (i.e. Façade) already in place. This means that the classes provided by this design participate in an existing Strategy and Façade patterns.

1.3 Industry Standards

XML, JDBC 3.0

1.4 Required Algorithms

Here we will deal with some pseudo code for dealing with database and xml. Some algorithms will not be discussed since they are either very simple or should be common knowledge to a developer. Aspects of serializing an object to a database will not be described since this is a basic java programming skill.

Very important note: there is a flaw in the original Document Indexer v 1.0 design, which assumes that a document index can be uniquely identified by WordSourceId instance. This is unfortunately a design mistake, which has the following implication for a data base implementation:

1. An object can not be used as a key in a database
   Even if we break the object down into a composite key (such as locale, className, etc) this will still not work since delimiters are of variable length

(i.e. an array) and sourceIdentity is a serializable object (which puts us back in square one)

2. We could simply not use a primary key, dump everything into a BLOB (or some combination of keys) but then this solution becomes quite ridiculous since we can't use SQL to fetch anything (again how do we target a BLOB or a byte array object in our query?) and iterative approach would kill the efficiency of this component.

Either way, these issues have been raised in the forums but have not been addressed.

I propose to make the following assumption: I will use the `hashCode()` of the `WordSourceId` as an actual DOCUMENT_ID in the database, and this will be used as a primary key used to identify the document uniquely. This is not a perfect solution and a better one would have been to explicitly force the WordSourceId to actually have a simple String or int based id. This would involve a change to the API, which at this point this designer didn't want to attempt since possibly many changes would have to be affected.

1.4.1 CRUD for Document index data from simple access database

Here we will deal with the aspects of CRUD with reference to the simple non-access optimized data base structure.

1.4.1.1 Reading a document index

Step 1. For the given document index ID Fetch the record with the specified DOCUMENT_ID.

Step 1a. read in LOCALE, CLASS_NAME, and SOURCE_IDENTITY that are stored in the fetched record.

- Reading LOCALE is done in the following manner. Since the format of the string is `Language_Country_Version` we need to parse the string to get the constituent parts. We should watch out for the following variations: `Language_Country`, `Language`. There should be no other variants present.
- CLASS_NAME is just a simple string
- SOURCE_IDENTITY is also just a String.

Step 1b. Fetch all the DELIMITER records matching the document index ID.

- Create a String array based on the size of returned record set

For each record put the DELIMITER_ENTRY string into the array.

Create a new object of the type WordSourceID and initialize it with the above data.

Step 2. Fetch all the WORD records where DOCUMENT_ID is the index document id that we are looking for.

- create a collator with the previously read in Locale

```
Collator myCollator = Collator.getInstance(locale);
```

- Create a Map where we will put all the words and

```
Map myMap = new HashMap();
```

For each record fetch all the WORD_POSITION records where DOCUMENT_ID is the index we are looking for and WORD_ID is

the id of the current word in the WORD record.
- Create a CollationKey object initialized with the current word.

```
CollationKey key = myCollator.getCollationKey(currWord);
```

- create a List in which we will keep all the positions for the current word.

```
List myList = new ArrayList();
```

For each WORD_POSITION record add the position to the list wrapped into an Integer instance:

```
list.add(new Integer(currpos));
```

- add the list to the Map

```
myMap.put(key, mylist);
```

Step 3. Create a new DocumentIndex object and initialize with the fetched data and return it to the caller.

```
DocumentIndex document = new DocumentIndex(WordSourceID, myMap);
return document;
```

1.4.1.2 Deleting an indexed document

- START_TRANSACTION

Step 1. For the input index id we delete the DOCUMENT record

Step 2. Delete all the WORD records matching the input document index id on the DOCUMENT_ID field.

Step 3. Delete all the WORD_POSITION records matching the input document index id on the DOCUMENT_ID field.

Step 4. Delete all the DELIMITER records matching the input document index id on the DOCUMENT_ID field.

Step 5. Delete all the DOCUMENT_COLLECTION_DOCUMENT_XREF records matching the input document index id on the DOCUMENT_ID field. This needs to be done to keep the collections consistent so that they do not point to non-existent data.

- END_TRANSACTION

If there were any issues during any step we rollback the transaction and throw an exception.

1.4.1.3 Creating a new document

Preconditions:

1. Contains a unique document index id.
2. Contains a valid Locale
3. Contains a non-empty list of delimiters with at least one delimiter
4. Contains at least one word which contains at least one position

- START_TRANSACTION

Step 1. Create a temporary idCounter, which we will use for word ids. Initialize it to 1.

```
int idCounter = 1;
```

Step 2. Create a new DOCUMENT record.

- Fill in the information for CLASS_NAME, SOURCE_IDENTITY.
- When filling in locale simply insert `locale.toString()`
- We also set the USE_COUNT to 0 since this document is not currently part of any collections.

Step 3. Create a DELIMITER record for each member of the delimiters array in the `wordSourceId` variable.

Step 4. For each entry in the words Map we obtain the key (which is a `CollationKey` type and we extract the word from it)
For each key we extract the word:

```
key.getSourceString()
```

and we create a new WORD record initialized with it. We use the current `idCounter` as the word id.

Step 4a. For the current key we extract from the DocumentIndex.words map the list of positions. For each position we create a new WORD_POSITION record and we initialize it with the integer position as well as the current document id and the word id.

Step 4b. increment the counter to generate a new word id.

```
idCounter++;
```

- END_TRANSACTION

If there were any issues during any step we rollback the transaction and throw an exception.

1.4.1.4 Update a document index

The API currently has no direct update capabilities.

1.4.2 CRUD for Document index data from fast access database

Since this is basically the same structure as the simple access variant, the discussion here will be specifically around the structure of the CLOB and the means of reading and writing it.

1.4.2.1 Creating CLOB word data

Given the CLOB structure as depicted on page 2 we need to go through the following steps:

Step 1. Create the word and position delimiters to be used
- We take the first delimiter from the delimiter list supplied with the document to be persisted:

```
DocumentIndex.wordSourceID.delimiters[0]
```

- To create a position delimiter (`posDelimiter`) we will simply take the above delimiter
- To create a word delimiter (`wordDelimiter`) we simply concatenate the posDelimiter twice.

Step 3. We create a new WORD_DATA record and initialize it with the proper document index id.

Step 2. We now simply write out all the words and their positions, sequentially into the CLOB (make sure that they are written as

UNICODE rather than ASCII) and delimit the positions and words with the proper delimiters.

1.4.2.2 Reading word data out of a CLOB

This is a simple, reverse process of the writing. The only concern would be how do we parse the words and positions with using delimiters where one delimiter is a subset of the other. This can be very easily done with a primitive state machine where we start in the state of read-word and then read characters until we hit a possible delimiter, a delimiter is only decided when we hit a non delimiting character (by this time we know that, we have a fully formed delimiter in our hands) if the fully formed delimiter we have is `posDelimiter` then we are in state of read-position, and of it is a `wordDelimiter` then we are in the state of read-word. We then create the entries of the Map to be used to construct the DocumentIndex instance as in Step 2. of the 1.4.1.1 section.

1.4.3 CRUD for Collection Index data persistence in a data base

Since the actual structure used for document collection persistence is exactly the same for both access modes (i.e. fast and simple) we present only one set of algorithms.

1.4.3.1 Create a new Collection Index (document collection)

This is a very simple process that mostly relates existing document index records with a collection.

- START_TRANSACTION

Step 1. Using the provided id create a new DOCUMENT_COLLECTION record.

Step 2. If the collection is not empty (i.e. it has actual indexed documents) then for each document we check if this document has been added to this collection.
For each WordSourceID instance in the `Collectionindex.allDocumentsIds` we check if the document has been related to this collection. This can be done by querying for an entry in the DOCUMENT_COLLECTION_DOCUMENT_XREF table that has the DOCUMENT_ID of the current document index (as identified by the `hashCode()` and the provided `CollectionIndex.id`). If the entry exists then we do nothing but if it doesn't exist then we execute these steps:

Step 2a. We create a new DOCUMENT_COLLECTION_DOCUMENT_XREF record with the proper relating Ids.

Step 2b. We update the document count for the document index identified. This is simply done by reading the current count for the Document Index (as given by USE_COUNT) and then updating the record with an increment of this count.

- END_TRANSACTION

1.4.3.2 Read an existing collection index (document collection)

This is a relatively simple process that mostly delegates to document index-reading routines.

Step 1. We create an empty map to hold words and empty Set to hold all the document Id WordSourceId instances.

```
Map myMap = new HashMap();
Set mySet = new HashSet();
```

Step 2. For the given CollectionIndex.id we fetch all the DOCUMENT_COLLECTION_DOCUMENT_XREF records matching this id in the COLLECTION_ID field.

Step 3 For each such record we extract the DOCUMENT_ID and we fetch the index document from persistence according to either section 1.4.1.1 and section 1.4.2.1 if we are doing this as part of fast access.

Step 3a. for the current document we walk through all the indexed words.
For each word we add the word (after wrapping it in `CollationKey`) to `myMap` as follows:
- If the word doesn't exist in the map we add it to the map (as the key) with the value being a new `HashSet()` to which we add the current document's WordSourceId instance.
- if the word already exists we fetch the value (which is a Set) and we add to this set the current document's WordSourceId instance. We then re-insert the updated `Set` into `myMap` under the original key.

Step 3b. To `mySet` we add the current document's WordSourceId instance.

1.4.3.3 Update a collection index (document collection)

Updating a collection index is really a simple walkthrough the DOCUMENT_COLLECTION_DOCUMENT_XREF table to see if

Step 1. Are there are any document index entries there that are not in the input CollectionIndex object?. If this is so we need to remove the document index from the DOCUMENT_COLLECTION_DOCUMENT_XREF table and additionally we need to decrement the USE_COUNT in the DOCUMENT table for the specific document index.

Step 2. Are there any document index entries in the CollectionIndex object that are not in the DOCUMENT_COLLECTION_DOCUMENT_XREF table? If so we need to add the proper relation record by creating a new DOCUMENT_COLLECTION_DOCUMENT_XREF record with the CollectionIndex.id and the document index's id (given by `WordSourceId.hashCode()`). We also need to increment the USE_COUNT in the DOCUMENT table for the specific document index.

This should be done as part of a transaction.

1.4.3.4 Delete a collection index (document collection)

Deletion simply means that we remove all the DOCUMENT_COLLECTION_DOCUMENT_XREF references to the COLLECTION_ID that matches the current CollectionIndex.id. It also means the proper decrementing of the appropriate DOCUMENT.USE_COUNT entries for the documents that were contained in the collection. The final step is to remove the specific DOCUMENT_COLLECTION record.

All of this should be done in a transaction.

1.4.4 CRUD for xml based persistence

1.4.4.1 Reading a document index

We will be using a SAX based parser for this.

Step 1. For the given document index id (docId) we look for all files that begin with the following name:
`"doc_index_" + docId + "_"`
Every such file that we find will use (in sequence as specified in the section 1.1.1.2.1.

Step 2. For each file we do the following:
- Since each file will contain the same document information and will only differ in extended word and position content we only read `locale`, `className`, `sourceIdentity` (this is base64 encoded so we will need to decode this element first using the Base64 encoding algorithm is section 1.4.5.1 and then deserialize the decoded stream), and delimiters from the first file in the sequence.
- We read all the word and position information into an appropriate Map as specified in the data base algorithms.

1.4.4.2 Deleting an indexed document

Here we will need to use DOM to update all the collections that the document belongs to since it will no longer load. We will use SAX to quickly parse `document_usage.xml` file to find out which collections the document being deleted belongs to.

Step 1. We need to find all the collections that this document index belongs to. This is very simple. Look for a document-id argument in a <document-usage> node.

Step 2. For each collection id that we read that this document index belongs to we open the appropriate collection file which we simply
construct as follows:
`"doc_collection_" + collectionId + ".xml"`
and we create a DOM tree and we remove the specific node (i.e. <document-id>)from the DOM and the we overwrite the original file with the new DOM

Step 3. We delete all the indexed document's xml files (there could be a couple of them so we might have to remove a couple files)

1.4.4.3 Creating a new indexed document

The only difficult think about persisting a new indexed document revolves around how to split the file if the data is bigger than the prescribed size limit. Please note that here we do not have to use any type of an xml specific parser as we can create the document by simply packing a StringBuffer with proper xml tags. The only thing to note here is how do we know when we need to split a file and start writing in a new one. This is easily achieved by the following fashion.

Step 1. Generate the header and all the non-word related tags and data and place it into an output stream (buffered)

Step 2. Use a temporary StringBuffer (`tempBuffer`) for word data. Fill in this buffer with all the necessary data for the current word.
- If (currentFileLength + tempBuffer()+length >= limit – threshold) threshold is safety of perhaps 1000 chars so that we can actually properly close the file, then we know that we have to finish up this file and create a new file to write the rest of the word data. If this is the case we write all the necessary closing tags into the file and close the file. We then create a new file with the same name by with an incremented sequence number and go to step 1.
- If on the other hand we have not reached the limit we then simply add the contents of the tempBuffer to the current output stream and write it out (we also update the currentFileLength with e the number of bytes that we have just written)

1.4.4.4 Creating a new Collection index

This is relatively simple and depends on properly coordinating the different document index files. This can be created directly by using the appropriate tags from the xml definition in section 1.1.1.2.. Please also consult the document index creation section 1.4.4.1, index deletion section 1.4.4.2, as well as the data base section of 1.4.3.1. DOM will be used if we need to update the document usage information in the document_usage.xml file.

1.4.4.5 Reading a collection index

This is quite straightforward and depends on properly coordinating the different document index files. SAX parser should be utilized. Please consult the xml for collections.

1.4.4.6 Updating a collection index

This is quite straightforward and depends on properly coordinating the different document index files. DOM parser should be utilized for the collection and data-usage xml files since we will be overwriting. Please consult the xml for collections and for document usage.

1.4.4.7 Deleting a collection index

Please follow the general algorithm for the data base since the steps there will apply to here as well. We use SAX to read information that will be needed for deletion.

1.4.5 Base46 Encoding algorithm

This algorithm is very simple, as we will rely on the TopCoder *Base64 Codec* component to do the work of encoding and decoding for us.

1.4.5.1 The Base 64 encoding/decoding algorithm

To encode a byte stream into a base64 encoded string we do the following:

Step 1. Obtain a byte representation of an object. This step is done through java serialization.
Let us assume that the resulting byte data resides in `rawObjectBytes[]`.

Step 2. To encode the stream we do the following:

```
// create an encoder
// we just have one long line of output using the standard
// alphabet
Base64Encoder encoder = new Base64Encoder(0
        , null
        , Base64Codec.STANDARD_ALPHABET)

// set encode input
encoder.setInput(rawObjectBytes);

// get encoded bytes
// We create a buffer to hold the encoded data. Since we do
// not know the exact size but we know that roughly 1.3
// times will be needed we allocate 2*original string just
// in case.
byte[] encodedObjectBytes =
        new byte[rawObjectBytes.length * 2];
int encodedByteCount = encoder.deflate(encodedObjectBytes);

// now the 0..encodedByteCount bytes hold our encoded data
// to get the String representation we simply do the
// following
String encodedString = new String(encodedObjectBytes, "UTF-8");
```

The reverse decoding process is quite simple and can be deduced from the code above. One thing that the developer has to be aware of is any character set used in representing the string. It should be safe to assume a UTF-8 encoding for both base64 encoding and decoding as shown above.

1.5 Component Class Overview

1.5.1 Main classes and interfaces

1.5.1.1 com.topcoder.document.index.persistence.impl.db

This package will hold all the persistence aspect dealing with db-based persistence.

AbstractDBIndexPersistence <<abstract>>
This is an abstraction of a typical data base driven persistence contract. This provides for getting a connection from the `DBConnectionFactory` as well as specifying transaction control. This hides nicely the intricacies of configuration that would be used by all derived classes.

SimpleAccessDBIndexPersistence <<concrete>>
This is an implementation of the `AbstractDBIndexPersistence` that deals with a simple and straightforward, normalized data based schema. This is

provided for a user who is not as concerned about speed of the execution but is more concerned about simple data mining capabilities for 3rd party API. Since all the data is in plain view (nothing except for a serialized BLOB java object) this can be easily utilized.

FastAccessDBIndexPersistence <<concrete>>
This is an implementation of the `AbstractDBIndexPersistence` that deals with a fast, CLOB based, normalized data based schema. This is provided for a user how is concerned about speed of the execution. By storing all the word and word position data in a single CLOB structure we are achieving a 4000% speed increase over a comparable solution that uses thousands of records to store words and related positions.

1.5.1.2 com.topcoder.document.index.persistence.impl.xml

This package will hold all the persistence aspect dealing with xml-based persistence.

XmlIndexPersistence<<concrete class>>
This is a concrete implementation of the `IndexPersistence` contract that uses xml files as storage medium. This implementation uses a dual strategy for parsing and utilized SAX parsing for reading data in and DOM parsing for data updates (such as CollectionIndex updates and document use count updates)
This implementation allows for creation of arbitrarily large document index files by sequencing (splitting) large files into chunks of file based data. Since it is required that binary data is stored in the xml files for document index a serialized Base64 encoded data will be both encoded and decoded by this implementation.

1.6 Component Exception Definitions

This design is fully dependent on the contracts specified in the Document Indexer 2.0 component and as such reuses its persistence based exceptions. But since we will be dependent on configuration to instantiate the different persistence implementations we will also create a `PersistenceConfigurationException` which will be derived from the base `IndexPersistenceException` provided.

1.6.1 Custom Exceptions

1.6.1.1 Persistence related exceptions

PersistenceConfigurationException
This is an exception that signals persistence configuration issues when instantiating or initializing persistence instances. Ideally this exception should go into the com.topcoder.document.index.persistence package but currently it will be placed in the common com.topcoder.document.index.persistence.impl package.

1.6.2 System and general java exceptions

IllegalArgumentException
This will be thrown in situations where the input is considered illegal (this includes illegal null pointer input) . Some examples follow:

- Empty String (i.e. a string which after it is trimmed has a length of 0)

1.7 Thread Safety

The current design doesn't make the component thread-safe, Here are two points about thread-safety for this component:

1.7.1 Thread-safety

There is no requirement to make this thread-safe, but we could argue that this should be made thread safe. As far as persistence instances are concerned there are no shared resources that are used internally within the instance itself. Thus for db persistence a connection is never exposed and is created for each new persistence request. The same idea goes with the xml persistence where SAX or DOM parser instances are created on demand within the actual persistence call. Thus we can argue that persistence acts almost like a utility.

In other words none of the current persistence implementations have a state that is shared.

1.7.2 Race condition from external resources possible issues

If we look at the actual aspect of external resources (such as an xml file or a database table) then we have a possibility of a race condition (akin a thread-safety) this would call for an ACID based solution for DB and a file locking solution for xml based persistence. It was deemed though after some though that since indexing (i.e. writing) will be done usually from one source and group creation will be also probably done from one source (again writing) and since this component's main purpose will be to read the indexed data from persistence we will not have to burden the solution with involved process safety solutions and as such no process safety solution has been implemented.

2. Environment Requirements

1.1Environment

- Development language: Java1.4
- Compile target: Java1.4, Java1.5

1.2 TopCoder Software Components

1. **DB Connection Factory Version 1.0**
   This will used to load up pre-configured connections.

2. **Configuration Manager 2.1.4**
   This is being used indirectly for purposes of configuring Db Connection factory component as well as for specific xml persistence configuration.

3. **Document Indexer 2.0**
   This is actually the plug-in contract for this component and the persistence defined here will plug into Document Indexer 2.0 component.

4. **Base64 Codec 1.0**
   This is used for base 64 encoding and decoding process.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location

1.3 Third Party Components

None used.

3. Installation and Configuration

1.4Package Name

com.topcoder.document.index.persistence.impl
com.topcoder.document.index.persistence.impl.db
com.topcoder.document.index.persistence.impl.xml

1.5 Configuration Parameters

Currently we need to configure the DB persistence entries as well as some specific xml persistence information.

DB persistence implementation parameters:

Parameter	Description	Values
ConnectionName	The name of the connection to be used ***Optional.***	Any valid name will do. Will use default connection if no name given.
ConnectionFactoryClassName	This is the class name of the connection factory ***Optional***	Example: com.topcoder.db.connectionfactory.DBConnectionFactoryImpl
ConnectionFactoryNamespace	This is the namespace to pass to the connection factory. ***Optional***	Any valid namespace.

XML persistence implementation parameters:

Parameter	Description	Values
XmlPersistencePath	This is the place where all the xml files that make up the persistence are stored for this session. This must be a valid fully qualified path. ***Required***	Example: "c:\xmldata"
FileSizeLimit	This is a file size limit for saving a chunk of indexed data. It is measured in Mbytes ***Required***	Example: 2000 This would represent a limit of 2 Gb, which is a limit on a window file system for example.
FileSizeThreshold	This is a safety net size in Kbytes that is applied before the fileSizeLimit is reached. ***Required***	Example: 10 This would represent a safety net of 10 Kb to be applied before the size limit is reached. In effect it subtracts from the filesizeThreshold value.

1.6 Dependencies Configuration

None at this point.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

1. Configure the necessary connections.
2. Create the specific persistence instance passing to it the proper initialization parameters and then pass this to an instance of IndexManager.

4.3 Demo

The demo will assume that specific sources for document indexing have been properly and that the database has been properly administered and is running.

4.3.1 Manager based persistence usage

```
// Create a xml based persistence
IndexPersistence xmlPersistence = new XmlIndexPersistence("some.name.space");
// pass it to the manager
IndexManager manager = new IndexManager(xmlPersistence);
// Create a fast db based persistence
IndexPersistence fastDBPersistence =
                new FastAccessDBIndexPersistence ("some.name.space");
// pass it to the manager
IndexManager manager = new IndexManager(fastDBPersistence);
// Create a simple db based persistence
IndexPersistence simpleDBPersistence =
                new SimpleAccessDBIndexPersistence("some.name.space");
// pass it to the manager
IndexManager manager = new IndexManager(simpleDBPersistence);
```

4.3.2 Direct persistence operations (persistence interface usage demo)

```
// Use the API to do some specific persistence. Here we showcase the direct persistence
// API on standalone basis rather than through the index manager. We will use the
// xmlPersistence instance

// Create a xml based persistence
IndexPersistence xmlPersistence = new XmlIndexPersistence("some.name.space");
// add a document to the persistence
xmlPersistence.addDocumentIndex(documentIndex)
// get it back
DocumentIndex documentIndex  =
        xmlPersistence.getDocumentIndex(documentIndex.getWordSourceId());
// remove it
xmlPersistence.removeDocumentIndex(documentIndex.getWordSourceId());
// create a collection index
xmlPersistence.addCollectionIndex(collectionIndex)
// get some persisted collection index
CollectionIndex someDocumentCollection = xmlPersistence.getCollectionIndex("265727");
// remove it
xmlPersistence.removeCollectionIndex("265727")
// change something in the collection
someDocumentCollection.add(documentIndex);
// update it
xmlPersistence.updateCollectionIndex(collectionIndex)
// increase the usage count
xmlPersistence.increaseDocumentUseCount(wordSourceId)
```

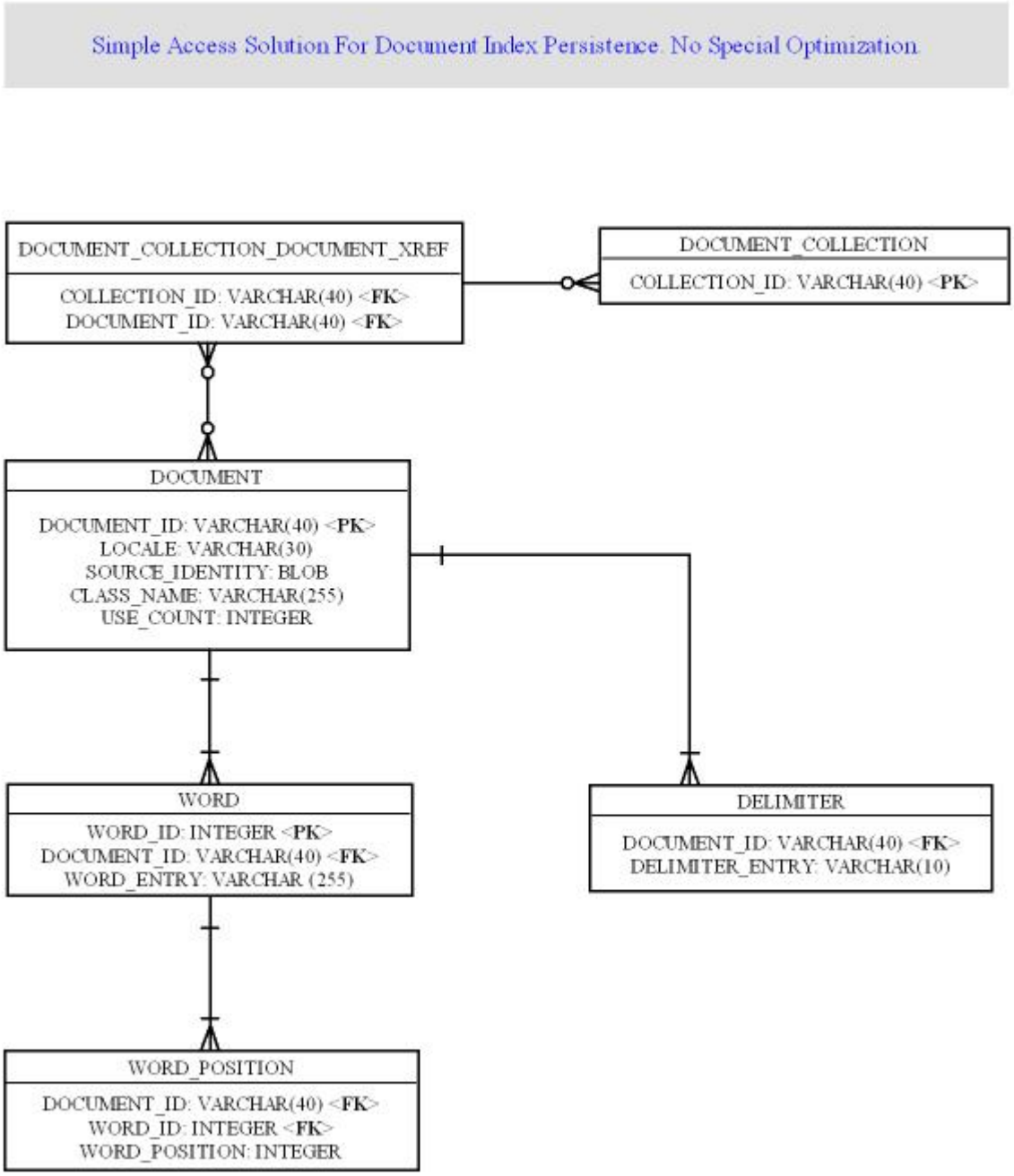
```
// decrease the usage count
xmlPersistence.decreaseDocumentUseCount(wordSourceId)
// get the usage count
int persistedCount = persistence.getDocumentUseCount(wordSourceId);
// get all persisted docs
Set docs= xmlPersistence.getIndexedDocuments();
```

5. Future Enhancements

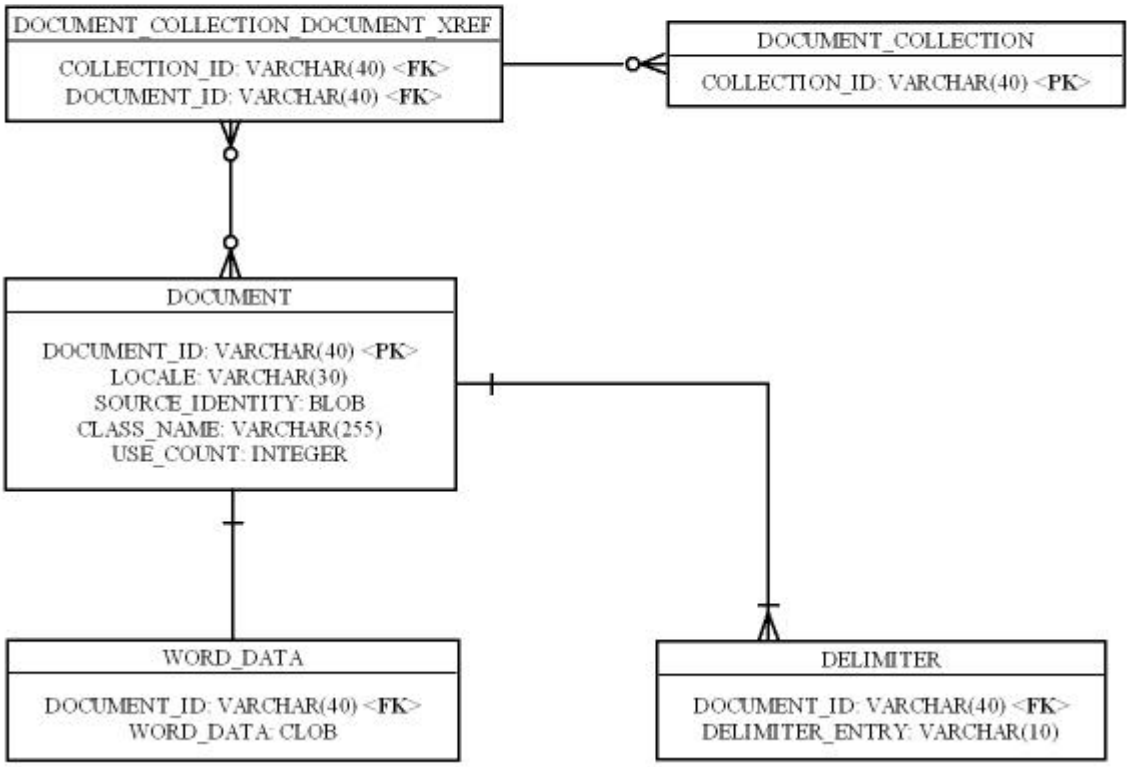
- None at this point

6. Appendix A – ERD diagrams

This appendix contains the two ERD diagrams for the data base solution.



Fast Access Solution For Document Index Persistence. Uses CLOB to store all word data.



7. Appendix B – sample benchmarks and source code

Here is some benchmark figures for fast data base access and the very simple code that was used to obtain these figures.

The test was run with 5 documents with approximately 10000 word records per document and 3 positions per word. The normal test has three tables, and the blob just one. The test includes running the query and processing the resultset and data. The test was repeated 100 times and the score averaged. The results are in milliseconds:

Run#	Simple Access Solution Run time in ms	Blob based Solution Run time in ms
1:	2000	109
2:	1704	47
3:	1734	31
4:	1719	31
5:	1750	47
6:	2141	46
7:	1750	31
8:	1656	125
9:	1656	47
10:	1734	47
11:	1656	47
12:	1750	31
13:	1734	47
14:	1656	47
15:	1766	31
16:	1656	31
17:	1734	47
18:	1671	47
19:	1766	31
20:	1657	47
21:	1766	31
22:	1672	31
23:	1734	32
24:	1656	32
25:	1766	31
26:	1672	47
27:	1750	31
28:	1641	47
29:	1765	31
30:	1672	31
31:	1750	32
32:	1657	46
33:	1750	31
34:	1656	31
35:	1781	47
36:	1656	47
37:	1750	47
38:	1656	47
39:	1750	47
40:	1703	31
41:	1781	31
42:	1656	31
43:	1750	46
44:	1641	47

simple access average: 1714.32 ms
blob average: 39.56 ms
performance improvement factor: 43.33

Run#	Simple Access Solution Run time in ms	Blob based Solution Run time in ms
45:	1765	47
46:	1672	46
47:	1750	31
48:	1640	31
49:	1735	32
50:	1656	32
51:	1766	31
52:	1656	32
53:	1765	32
54:	1641	46
55:	1766	31
56:	1640	47
57:	1734	47
58:	1656	31
59:	1750	47
60:	1672	32
61:	1750	47
62:	1640	47
63:	1750	47
64:	1672	32
65:	1734	47
66:	1641	31
67:	1734	32
68:	1687	31
69:	1734	47
70:	1656	31
71:	1750	32
72:	1672	31
73:	1766	47
74:	1672	31
75:	1750	31
76:	1625	31
77:	1750	31
78:	1656	31
79:	1765	32
80:	1656	32
81:	1781	47
82:	1657	31
83:	1750	47
84:	1656	47
85:	1750	32
86:	1672	31
87:	1750	47
88:	1687	31
89:	1734	47
90:	1657	32
91:	1765	32
92:	1672	47
93:	1750	47
94:	1672	31
95:	1750	31
96:	1656	47
97:	1750	47
98:	1656	32
99:	1766	31
100:	1656	31