5 - LexEVS 6.x Query Service Extension

Contents of this Page

LexEVS 6.x Programmers Links

- Programmer's Guide Main Page
  - LexEVS API
  - LexEVS 6.0 CTS2 API
  - LexEVS 6.x CTS2 API Quick Start
- Value Set and Pick List Guide
- LexEVS 6.0 Main Page
- LexEVS Current Release

Introduction

This document is a section of the LexEVS 6.x Programmer's Guide.

LexEVS 6.0 implements the following performance and behavior enhancements in the Query Services Extension:

- Lucene lazy loading for improved query retrieval performance
- Search interface for plugging in custom search algorithms
- Enhanced and new search algorithms for improved accuracy and performance
- Sort interface for plugging in custom sort algorithms
- SQL optimization for improved performance in large scale query retrievals

Lucene Lazy Loading

After the Lucene search is complete, the system stores only the Document id of documents that match the search criteria. Then, when information from the document is needed, it is retrieved from the document. This is helpful in iterator-type scenarios, where retrieval can be done one at a time.

Background - Lucene Documents

Lucene stores information in documents, and these documents have fields that are used to hold information. Each document has a unique id. For example, an index of people may be indexed in Lucene as:

```
Document: id 1
First Name: John
Last Name: Doe
Sex: Male
Age: 45

Document: id 2
First Name: Jane
Last Name: Doe
Sex: Female
Age: 40

... etc.
```

LexEVS stores information about entities in this way. Property names and values, as well as qualifiers, language, and various other information about the entity are held in Lucene indexes.

Background - Querying Lucene

Lucene provides a query mechanism to search through the indexed documents. Given a search query, Lucene will provide the document id and the score of the match. (Lucene assigns every match a score, depending on the strength of the match given the query.)

So, if the above index is queried for "First Name = Jane AND Last Name = Doe", the result will be the document id of the match (2), and the score of the match (a float number, usually between 1 and 10).

Notice that none of the other information is returned, such as sex or age. It is useful for that extra information to be there, because if it exists in the Lucene indexes we do not have to make a database query for it. But, retrieving data from Lucene documents is expensive, just as retrieving data from a database would be.

Lazy Retrieval

Lazy retrieval can be leveraged to increase performance in LexEVS. Consider this simplified LexEVS entity index:

```
Document: id 1
Code: C12345
Name: Heart

Document: id 2
Code: C67890
Name: Foot

Document: id 3
Code: C98765
Name: Heart Attack
```

If a user constructs a query (Name = Heart*), the query will return with the matching Document ids (1 and 2). Previously, LexEVS would immediately retrieve the **Code** and **Name** fields from the matches, and use them to construct the results that would be ultimately returned to the user. This does not scale well, especially for general queries in large ontologies. In a large ontology, a query of (Name = Heart*) may match tens of thousands of documents. Retrieving the information from all these documents is a significant performance concern.

Instead of retrieving the information up front, LexEVS will simply store the document id for later use. When this information is actually needed by the user (for example, the information needs to be displayed), it is retrieved on demand.

Searching

The org.LexGrid.LexBIG.Extensions.Extendable.Search Interface

This interface enables the user to plug in custom search algorithms. Users can construct any type of query given search text. The query can include wildcards, it can group search terms, etc.

Class:	`org.LexGrid.LexBIG.Extensions.Extendable.Search`
Method:	`public org.apache.lucene.search.Query buildQuery(String searchText)`
Description:	Given a String search string, build a query object to match indexed Lucene documents

Default AND

Previously, for most search algorithms Lucene applied an OR to the terms if multiple terms were input as search text. For example, a search of 'heart attack' would match all documents containing 'heart' OR 'attack'. This lead to non-intuitive query results being returned. In LexEVS 6.0, the Lucene default is changed to AND. Consequently, search precision is increased and returned results are more intuitive. In most cases the AND shrinks the number of results returned for a given query, which in turn increases overall performance.

Algorithms

More Precise DoubleMetaphoneQuery

DoubleMetaphoneQueries enable the user to input incorrectly spelled search text, while still returning results. Because this is a 'fuzzy' search, it is important to structure the Query in a way that the most appropriate results are returned to the user first. For example, the Metaphone computed value for "Breast" and "Prostrate" is the same. Given the search term "Breast", both "Breast" and "Prostrate" will match with exactly the same score. Technically, this is correct behavior, but to the end user this is not desirable. To overcome this, LexEVS 6.0 has introduced a new query, WeightedDoubleMetaphoneQuery.

WeightedDoubleMetaphoneQuery

This algorithm does not automatically assume that the user has spelled the terms incorrectly. Searches are also based on the actual text that the user has input, along with the Metaphone value. Again, if the user input "Breast", the query will still match "Breast" and "Prostrate", but "Breast" will have a higher match score, because the actual user text is considered. This algorithm adds a greater precision to this fuzzy-type query.

Algorithm:

```
get: user text input
2: total score = 0
3: metaphone score = 0
4: actual score = 0
5: metaphone value = lucene.computeMetaphoneValue(user text input)
6: metaphone score = lucene.scoreMetaphoneValue(metaphone value)
7: actual score = lucene.score(user text input)
8: total score = metaphone score + actual score
9: halt
```

Case-insensitive Substring

The **SubStringSearch** algorithm is intended to find substrings within a large string.

For example:

```
'with a heart attack'
```

...will match:

```
 'The patient _with a heart attack_ was seen today.'
```

Also, a leading and trailing wildcard will be added, so

```
'th a heart atta'
```

..will also match:

```
'The patient wi_th a heart atta_ck was seen today.'
```

Algorithm:

```
get: user text input
2: user text input = '*' + user text input + '*'
3: score = lucene.score(user text input)
4: halt
```

Sorting - The org.LexGrid.LexBIG.Extensions.Extendable.Sort Interface

This interface allows users to plug in customized Sort algorithms to sort query results:

Class:	`org.LexGrid.LexBIG.Extensions.Extendable.Sort`
Method:	`public <T> Comparator<T> getComparatorForSearchClass(Class<T> searchClass) throws LBParameterException`
Description:	Given a Class that this Sort is valid for, return the correct Comparator to compare the results and sort.
Method:	`public boolean isSortValidForClass(Class<?> clazz)`
Description:	Return whether or not this Sort is valid for Sorting on a given Class.

- **Sorting on Different Class types** A single Sort may be applicable for a variety of Class types. For instance, both an 'Association' and an 'Entity' may be sorted by 'Code', but the actual implementation of retrieving the Code and comparing it may be different between the two. It is the job of the Sort to implement a Comparator for each potential Class that it is eligible to sort.
- **Default Sorting** All result sets are sorted by default by Lucene Score, meaning that the best match according to Lucene will always be returned first by default. Note that if two or more result sets are being Unioned, Intersected, or Differenced, the user must explicitly call a 'matchToQuery' sort on the result set as a whole to order all of the results.
- **Sort Contexts** Sorts may be applicable in one or more 'Contexts.' (see: `org.LexGrid.LexBIG.DataModel.InterfaceElements.types.SortContext`). This means that a Sort may apply only to a CodedNodeSet, or only to a CodedNodeGraph, or some combination. Sorts will only be employed by the API if they match the Context in which the results are being sorted.
- **Performance Issues** Sorting is generally computationally expensive, because in order to correctly sort, the field to be sorted has to be fully retrieved for the entire result set. For very specific or refined queries, this may not be a problem, but for large ontologies or very general queries, performance may be a concern. To alleviate this, 'Post sort' has been introduced.
- **Post Sorting** In order to minimize the performance impact of sorting, users are encouraged to use a 'Post sort' where possible. A Post sort is done after the result set has been restricted, thus limiting the amount of information that must be retrieved in order to perform the sort. For instance, a query may match a set of Entities:

```
\{"Heart", "Heart Failure", "Heart Attack", "Arm", "Finger", ...\}
```

As described earlier, all results are by default sorted by Lucene score, so if we limit the result set to the top 3, the result is:

```
\{"Heart", "Heart Failure", "Heart Attack"\}
```

The restricted set can then be 'Post' sorted; and because the result set has been limited to a reasonable number of matches, sorting and retrieval time can be minimized.

Algorithm:

```
1: get: Sort requested by user
2: get: Context sort is being applied to
3: if: sort is not valid for Context
halt
4: else:
5: get: Class to be sorted on
6: if: sort is not valid for Class
halt
7: get: Comparator for Sort - given (Class to be sorted on)
8: sort results using Comparator for Sort
9: halt
```

SQL Optimizations

The n+1 SELECTS Problem

The n+1 SELECTS Problem refers to how information can optimally be retrieved from the database, preferably using as few queries as possible. This is desirable because query overhead is a concern. Every query must be packaged and sent to the database engine, processed, packaged again and transferred to the client. Although the overhead may be minimal (a few milliseconds), it does not scale. Although sometimes obvious, n+1 queries can remain in a system undetected until scaling problems are noticed.

To avoid this problem, a JOIN query can be used.

In LexEVS 6.0, there were three n+1 SELECT queries fixed:

- The EntryState while building the CodedEntry
- The EntityDescription on AssociatedConcepts
- AssociationQualifiers on AssociatedConcepts

The n+1 SELECTS Problem Example

Given two database tables, retrieve the Code, Name, and Qualifier for each Code.

Table Codes

Code	Name
C01234	Heart
C98765	Heart Attack

Table Qualifiers

Code	Qualifier
C01234	isAnOrgan
C98765	isADisease

```
SELECT * FROM Codes
```

Results in:

Code	Name
C01234	Heart
C98765	Heart Attack

To get the Qualifiers, separate SELECTs must be used for each.

```
SELECT * FROM Qualifiers where Code = C01234
And
SELECT * FROM Qualifiers where Code = C98765
```

This sequence results in 1 Query to retrieve the data from the Codes table, and then n Queries from the Qualifiers table. This results in n+1 total Queries.

The n+1 SELECTS Solution Example

Given two database tables, retrieve the Code, Name, and Qualifier for each Code.

Table Codes

Code	Name
C01234	Heart
C98765	Heart Attack

Table Qualifiers

Code	Qualifier
C01234	isAnOrgan
C98765	isADisease

```
SELECT * FROM Codes JOIN Qualifiers ON Code
```

Results in:

Code	Name	Qualifier
C01234	Heart	isAnOrgan
C98765	Heart Attack	isADisease

Because of the JOIN, only one Query is needed to retrieve all of the data from the database.