IBM Cúram Social Program Management
Version 6.0.5

Cúram Generic Search Server

Note
Before using this information and the product it supports, read the information in "Notices" on page 41

Revised: March 2014

This edition applies to IBM Cúram Social Program Management v6.0.5 and to all subsequent releases unless otherwise indicated in new editions.

Contents

Figures

Tables

Developing with the Generic Search Server

Use this information to develop performant and scalable searches in the Cúram application with the Cúram Generic Search Server. Searches can be implemented with Cúram Generic Search Server and database searches. Database and Generic Search Server searches can be enabled on a per-search basis with application properties.

Introduction

Cúram Generic Search Server Guide

The Cúram Generic Search Server is a tool provided by IBM Corporation that can be used to develop performant and scalable searches for your application solution.

This document describes the Cúram Generic Search Server and provides an overview of its architecture. It is also a reference for the configuration of the Generic Search Server and its database tables. Finally, it provides an end-to-end example of how to implement a search using the Cúram Generic Search Server.

Prerequisites

Readers of the Cúram Generic Search Server Guide should be familiar with the Cúram architecture, in addition to being familiar with Cúram modeling and development constructs and processes.

Audience

This document is intended to be read by architects, designers and developers interested in using the Cúram Generic Search server to implement search pages.

Concepts and Definitions

Introduction

This chapter introduces several important searching and indexing concepts, in addition to definitions related to the Cúram Generic Search Server which are used throughout this document.

The Generic Search Server

The Cúram Generic Search Server is a standalone application which supports performant searching of application data via a number of APIs. Behind the scenes, the Generic Search Server is implemented using the Apache Lucene API. Those implementing GSS searches should use only the APIs exposed by GSS.

The Generic Search Server can be deployed as a plain Java™ Application (to ease development-time testing) as well as a Java Platform, Enterprise Edition application.

Indices

At the heart of the Generic Search Server is the concept of searching an Index, which is a performant, non-database representation of a set of related searchable

data. A Generic Search Server Index is an “inverted index” that maps words to database records that they appear in.

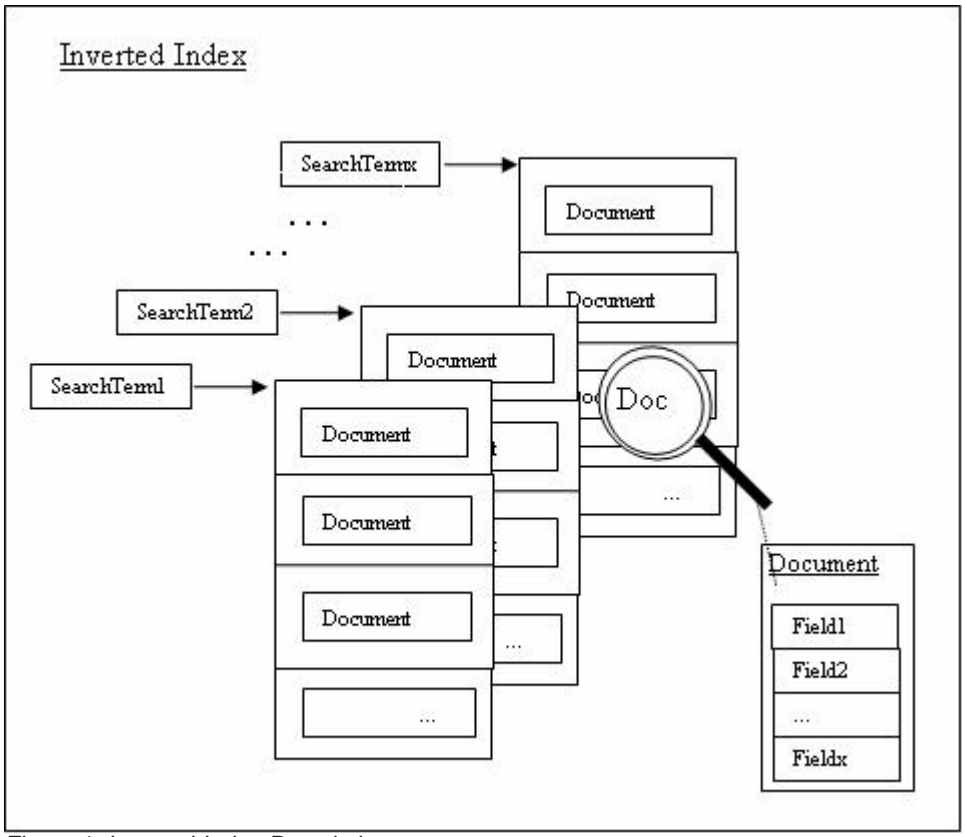

Figure 1. Inverted Index Description

When searching an Index for a word, all matching records are retrieved without having to search large datasets. As a result, such Indices scale well, and for large systems it will be possible to run multiple Indices in parallel, allowing for excellent search performance if the right deployment configuration and Index tuning parameters are chosen.

Developers creating application searches do not manipulate or maintain Indices directly - all of this is handled for them behind the scenes by the Generic Search Server.

Search Service

A Search Service describes:-

1. Information relating to fields being searched
2. Analyzers used on each field, field datatypes
3. Entity information to populate a run-time index
4. Status of Search Service (whether up to date or requires synchronization)

When seen in this way a Search Service is simply meta-data, however this document also uses the term to describe the run-time populated index.

There should be one Search Service defined for each discrete set of data to be searched upon (e.g. Person Search, Payment Search,etc.). Each search performed must specify which Search Service it is to operate on.

Field

As mentioned above, Search Services are made up of sets of Fields. These can be thought of as somewhat analogous to column definitions in database tables. A Field has a name and a type, and if being returned from a search it will also have a value, which is the result.

Fields may be marked as being 'Stored'. Fields marked in this way will cause the Index to physically contain relevant values extracted (see “Extractor” on page 4) from the database. This means that their values can be retrieved directly from the Index after a search and returned to the caller without the need to access the related record on the application database table. Note however that this does increase the Index size and may impact the performance of the search.

Fields may also be marked as 'Indexed' or not. Fields marked as such are searchable, and Fields not marked as such are not searchable. This feature is useful for fields such as unique IDs that may be desirable to store in the Index but not searched upon.

Note that Fields do not have to be marked as 'Stored' to be searchable.

Document

A Document is a record in an Index. A Document is in turn made up of a set of Fields. Search results are returned from the Generic Search Server as sets of Documents which can then be converted to Cúram struct objects. For example, a Person search Document might consist of Firstname, Surname, Address, Gender, etc. Fields, and performing a Person search/Query (see “Query” on page 4) based on a number of input criteria will return zero or more such Documents.

Lucene

Lucene is an open-source project created by the Apache Software Foundation. Behind the scenes, the Cúram Generic Search Server uses Lucene for its indexing and searching functionality.

Note: Note that information on indexing and Lucene is provided purely for background purposes - developers creating searches using the Generic Search Server do not need to manipulate Indices or Lucene objects directly. These are all wrapped by the Generic Search Server API.

Staging Database

The Generic Search Server staging database consists of a set of database tables used for the following purposes:

- To store Search Service definitions - information about which Search Services are available together with their structure
- To store values extracted from the operational database which will be used to populate Indices corresponding to the Search Service Definitions.

The fundamental design rationales for using database tables as an intermediary are as follows:

- They offload the searches from the main database which means that searches do not impact on live system performance
- They persist appropriately for the search service - Data is persisted in a form that is suitable for the purposes of building the search indices. The Application data is transformed, scrubbed and consolidated before being stored in the

staging database. Therefore, batch jobs will not have to be continually rerun to re-extract the data each time a Generic Search Server instance is started.

Query

A Query is an object (a struct, to be precise) that is passed to the Generic Search Server when a search is being performed.

Term

A Term is a part of a Query object. Currently, there are three different types of Term - Standard terms for searching on regular text fields, Date terms for searching on Date fields, and DateRange terms for specifying a range of dates on which to search.

Analyzer

An Analyzer is a Lucene concept, representing a class that implements the Lucene `org.apache.lucene.analysis.Analyzer` abstract class.

Analyzers prepare text for indexing and searching. For example, it doesn't make sense that every word of a text field is indexed - stop words such as “and”, “of” and “a” may be irrelevant during a search. If these are to be ignored during a field search then the field is tokenized, ie. passed through an analyzer before writing the field to the index and likewise for a term value being searched.

Analyzers are language-specific - what defines a word is not the same in all languages. Some can be configured to ignore common stop-words (an, the, if, etc), to ignore numbers, and so on. Analyzers used by the Generic Search Server are configurable on a per-Search Service basis.

Mapper

A Mapper is a class which has to be written by developers of application searches for each Search Service. Its function is to transform data from the application into a format which can be written onto the staging database and imported into a Index. The transformation involves identifying relevant Entity properties of interest to the Search Service, constructing a list of these values and mapping them to a single consolidated text value. This value, stored in the staging database, is used later in the construction of a single search index Document. Every Search Service that is written must provide its own Mapper implementation.

Extractor

The Extractor uses the Search Service metadata to obtain the relevant application data necessary to populate the search indices. The extractor interrogates the relevant Application Entities identified via the metadata and the required Entity properties are mapped(with the mapper) to the staging database for indexing upon Search Service startup.

Generic Search Server Overview

The Generic Search Server and Lucene

The concepts behind indexing and the Lucene API have already been introduced. So why not just use Lucene directly in Cúram application?

Whereas Lucene is an excellent API for indexing and searching, it does not address all of the requirements of a Cúram searching product:

- It does not address deployment issues - how to run multiple search servers, how the application should communicate with the search servers, etc.
- It does not address the issue of how to import data into Indices
- It does not address the issue of keeping Index data synchronized with source data in the running application.
- It does not address the issue of interpreting data returned from an Index search as Cúram datatypes and structs.
- It does not address the more overarching application requirement of protecting the Application Developer from in-depth knowledge of specific third-party products; given that Lucene is only one potential searching solution, it would seem to make more sense to provide a more generic searching API.

The Cúram Generic Search Server was developed to deal with these requirements.

Importing Data from Cúram

One implication of using an indexing technology is that, before being able to search an Index, it must first be created. Because a lot of the hard work of searching is essentially done up-front in Index construction, runtime searches become fast; however, it is worth noting that the indexing process itself may take some time, and this time increases proportionally with the amount of data to be indexed.

Initialization of the Generic Search Server is done in two phases.

In the first phase, existing application data is exported from the application into a set of database tables used by the Generic Search Server - the staging tables. This export has been implemented as a batch process, called the Database Search Extractor, and is provided as part of the Generic Search Server distribution. The export only needs to be performed once, when the Generic Search Server is first being used. Special helper classes called Mappers are needed for each Search Service; these assist the extractor in preparing the data to be imported into the Staging Tables.

In the second phase, an Index is constructed for every defined Search Service. When the Generic Search Server is started up, a process is run to read the appropriate data from the staging database tables and construct the Indices and other data structures to be used to perform searches. Once the Indices are constructed, the server will be in a position to respond to search requests. Information on optimizing this performance is available in “Performance” on page 31

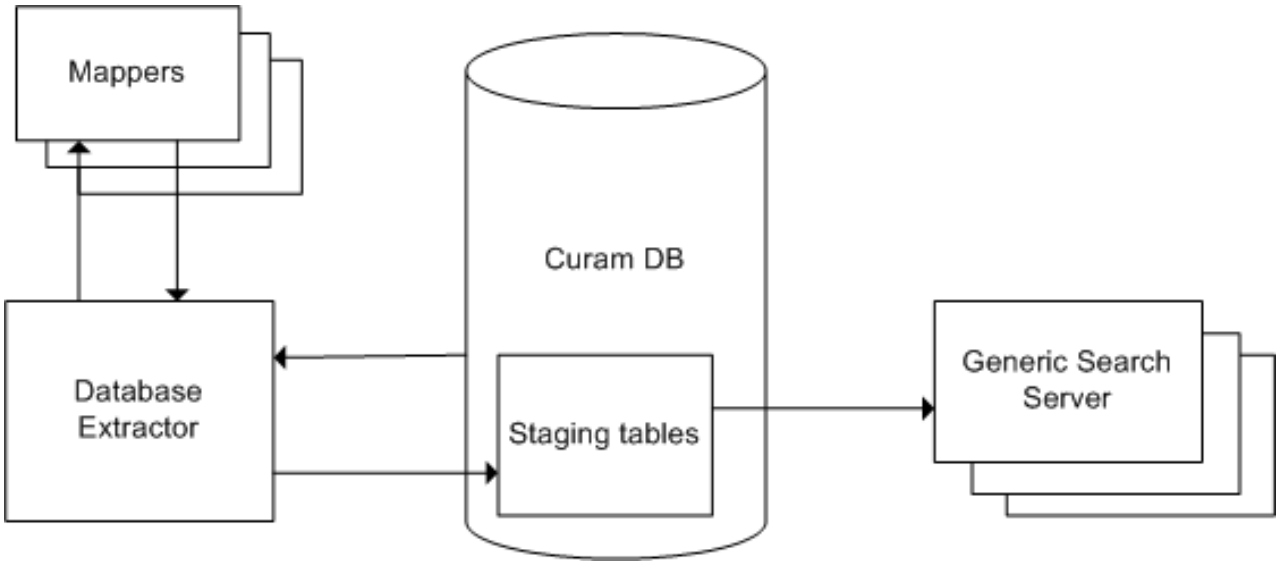

Figure 2. Database Extractor and Generic Search Server Startup Process

Search Server Synchronization

Because the Generic Search Server searches not on the live data itself but on an Index that is built from that data, updates to application data need to be replicated on the Index. In Cúram implementations, it is essential that updates to searchable data be reflected in the relevant Indices in a timely and predictable fashion. With the Generic Search Server, the time lag is short (and configurable).

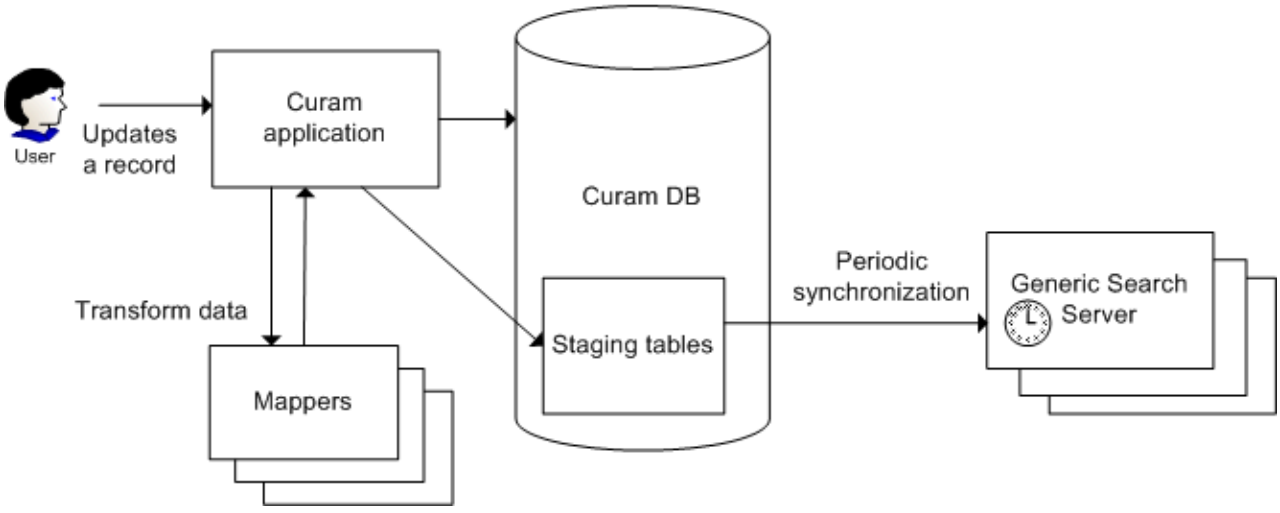

Figure 3. Data Synchronization

Similar to the initial import of data described above, there are two steps to the synchronization process.

The first step in the process occurs when the application data (which is used in an Index) changes, typically as a result of an insert, update or logical delete. When

this occurs, the application must write information about this data change to the Generic Search Server staging tables. All new and updated items are marked with a timestamp.

In the second step (which happens on a periodic basis), the Generic Search Server synchronizes its Indices against the current contents of the staging database. To do this, it reads all newly changed items since the last time it synchronized, and imports these into the Indices; specifically, this is achieved by comparing timestamps associated with each changed item to the latest timestamp used during the last synchronization step.

Note: When writing unit tests that include calls to Generic Search Server searches, it is important to bear in mind the delay in synchronizing data. In addition, as a result of the fact that the Generic Search Server instance will be running in a separate process to the unit tests, it will not be part of the same transaction. Consequently, Generic Search Server synchronizations will not pick up any data that has changed in the test transaction, unless it is explicitly committed.

Search Controller

The Search Controller is an important component of the synchronization mechanism. It maintains a list of all the entities associated with each Search Service.

When an entity changes, the Search Controller can be checked to see if that entity is used by one or more Search Services. If it is used, the data in the staging database should be updated in the same transaction as the entity update. The Search Controller also provides an API for updating the staging database.

Note: A number of Cúram Platform entities (which appear in some Cúram Platform searches) have been modified to allow for the implementation of such synchronization updates in the future release. These modifications have taken the form of the creation of pre- or post-operation exit points which contain stubbed-out implementations; these pre- and post- exit points are reserved for future implementation and should not be changed directly by customers.

The Search Process

The search process can be broken down into three phases.

In the first phase, the Cúram application constructs a valid Query to present to the Generic Search Server. It populates this Query using search criteria entered by the user.

In the second phase, the Cúram application contacts a running Generic Search Server instance and performs the search as defined by the Query object.

In the final phase, the Cúram application interprets the results it receives back from the Generic Search Server as Cúram datatypes, performs its usual security checks regarding the sensitivity of the data, and displays them to the user.

References

Lucene website: http://lucene.apache.org/.

Generic Search Server enabled searches

Introduction

IBM Corporation has introduced the Generic Search Server as an optional searching mechanism for Platform and Solution Module searches. Several searches have been implemented using both the Cúram Generic Search Server and database searching, and some are available only as GSS searches. For the searches that are available either as database or GSS searches customers may enable or disable performant search on a per-search basis via setting application properties.

Generic Search Server related properties in the Cúram application

These properties are the application system properties and can be administered in the usual way via the property administration in the application. All of the relevant properties are available under the Category called "Application - Lucene enhanced search parameters". A full list of these properties may be found in "Configuration Properties" on page 35

Table 1. Cúram Generic Search Server Related Properties

Property Name	Description
curam.lucene.luceneEnhancedSearchEnabled	Default: "NO". By default, all Generic Search Server functionality is disabled. In order to enable it, you must set this property to "YES" to turn on enhanced search. Unless this is set to "YES", no enhanced searches will be available.
curam.lucene.luceneOnlineSynchronizationEnabled	Default: "NO". To enable the event publishing mechanism that makes changes in searchable data available to the Search Server you must set this property to "YES". Unless this is done, inserts and updates to searchable data will not be propagated to the Search Server.
curam.lucene.externalUpdateEventsEnabled	Default: "NO". To ensure that if any search service related data is updated externally, then the external system receives related update synchronization events to synchronization the searchable data, in case if property "curam.lucene.luceneOnlineSynchronizationEnabled" is not enabled. Enabling this property has same impact as enabling "curam.lucene.luceneOnlineSynchronizationEnabled" on the application. To enable property "curam.lucene.externalUpdateEventsEnabled" set this property to "YES".

Finally, each search that supports Enhanced Search has a property that determines whether it uses the Generic Search Server or the database. This allows each organisation to choose on a per-search basis which enhanced searches to use.

Keeping Cúram data and search data synchronized

It is necessary to keep the live application data and the search index synchronized if search results are to be accurate. The infrastructure that the GSS provides in order to accomplish this has been described elsewhere (see "Search Controller" on page 13).

However, there is also an onus on application developers to add calls to the SearchController when relevant data changes in the application. This section describes for information purposes the event-based approach used, and which we recommend to customers implementing their own GSS-based searches.

As well as the event mechanism we also provide the Pull Mapper synchronization, which is described in its own chapter in this guide, see “Pull Mapper” on page 23.

Event-based synchronization

Cúram provides events to allow loosely coupled parts of the application to provide information to each other about changes of state. They are documented in the *Cúram Server Developer's Guide.*.

Each entity that contributes to a search service should have events raised when it is created, deleted, or modified. The event handler then calls the `SearchController` class to update the search server with the change.

Any entity that contributes to a search service must have postmodify, postinsert and postremove operations added that raise the events.

Staging Database Tables

Introduction

The staging database tables are database tables on the operational database that are used by the Generic Search Server. There are four such tables: SearchService, SearchServiceField, SearchServiceRow, and SearchSrvcRowExt.

This chapter details the purpose and structure of the SearchService and SearchServiceField tables. Developers creating search services do not need to access the SearchServiceRow or SearchSrvcRowExt tables directly, nor write DMX files for them.

The SearchService table defines Search Services known to the Generic Search Server (see “Search Service” on page 2 for introduction to Search Services). As an administration API for managing Search Services has not been provided, Search Service records must currently be created and maintained by either accessing the database table directly or by editing DMX files and rebuilding the application database.

The SearchServiceField table defines a single Field of a Search Service - its name, its data type, and several other attributes that are explained fully below. Each SearchServiceField database row is associated with a single SearchService row. As with Search Services, Search Service Field records must currently be created and maintained by either accessing the database table directly or by editing DMX files and rebuilding the application database.

SearchServiceRow is a table used to store searchable data from the application for use in building Indexes. The Generic Search Server provides an API (see “Getting Started with the Generic Search Server API” on page 12 and “Implementing a Search with the Generic Search Server” on page 18) that is used to manipulate SearchServiceRows - developers should interact with this database table only via this API rather than accessing it directly.

There are two other GSS database tables: GSSMapperType and GSSEntity. These are used only with the Pull Mapper feature - otherwise they can be ignored. These tables are described in “Pull Mapper” on page 23.

SearchService Table

Each Search Service must contain a record on the SearchService table. Together with its SearchServiceField child rows, the SearchService table defines the schema for each Search Service. A description of each column of the SearchService table is provided below:

searchServiceId

The Search Service Identifier; a string used to uniquely identify a Search Service.

extKeyName

The name of a Search Service Field that will uniquely identify each record in an Index created from this Search Service definition. It is essential that values in the Index corresponding to this Search Service Field be unique, as when searchable data is updated in the application database, the value of this field will be used to identify the appropriate Document to be updated in the Index.

analyzer

The Search Service analyzer to be used when converting from the application database text terms to Index terms. The contents of this column should denote one of the predefined analyzer names provided by the Generic Search Server (see the list below) or a fully qualified Java classname of a class that implements the abstract class `org.apache.lucene.analysis.Analyzer`. This may be either a standard Lucene analyzer or a third-party or custom implementation. Note that the class must be available on the Generic Search Server classpath if it is not a standard Lucene analyzer.

For a list of the analyzers supplied with GSS and a more in-depth discussion of how to choose an analyzer, see "Analyzers in Depth" on page 27.

frcdReidxTimeStmp

Used by the Extractor to force the Generic Search Server to rebuild its Indices after an extract has been run. When creating Search Service records, this should be initially set to null.

mapperName

The name of the mapper implementation (see "Implement Mapper Operations" on page 19). A Mapper implementation is a class that converts a set of application entity data to a format suitable for indexing. The value of this column should be the fully qualified classname of the Mapper class, and as with the Analyzer implementation, this should be on the Generic Search Server runtime classpath (If the Mapper is developed as part of the application it will be on the classpath by default).

dbLastWritten

This is used in synchronization. It should not be initialized or updated by application code or administrators.

prstBlobSize

This specifies the size of the blob associated with the table used to persist this search service index. If not specified, the blob size defaults to 50M. The property type is a String and the value should conform to the size specifier syntax of the concerned database.

SearchServiceField Table

Each Field of a Search Service must contain a record on the SearchServiceField table. Each Search Service Field represents a SearchService element that can be

either searched upon, returned from a search, or both. Search Service Field are used in a number of places throughout the Generic Search Server - in Terms, in Queries, in Documents. A description of each column of the SearchServiceField table is provided below:

srchServiceFldId
The Unique Identifier of the Search Service Field.

searchServiceId
searchServiceId of the parent Search Service record.

name
The name associated with the Search Service Field. This is the name that is used to reference the Field when performing searches or retrieving results. It does not need to correspond exactly to Field names in Cúram entities and structs, although it simplifies development if it does so.

type
The Cúram datatype of this field. The set of acceptable values is described in the table below.

The process of exporting and synchronizing data to the Search Service involves some conversion of operational data to strings and vice-versa, so it is important that an accurate data type be defined for each Field. See the following table for reference on this. If incorrect values are presented to the Generic Search Server, it will throw an exception.

Table 2. Mappings from basic Cúram Domain Definitions to GSS Field data types

Domain Definition	GSS Field data type
SRV_BOOLEAN	boolean
SRV_DATE	Date
SRV_DATETIME	DateTime
SRV_INT8	byte
SRV_INT16	short
SRV_INT32	int
SRV_INT64	long
SRV_FLOAT	float
SRV_DOUBLE	double
SRV_MONEY	Money
SRV_CHAR	char
SRV_STRING	String
SRV_UNBOUNDED_STRING	String

Note: The type field is case sensitive, so ensure you use the type name exactly as laid out above.

indexed
Indicates whether this Field is searchable. Sometimes it may be desirable to store a value for a record in the Search Service but not to search on it (an example would be the unique ID of a record, or perhaps it's sensitivity level). Not indexing values that don't need to be indexed will minimize Index size and help performance, so it is good practice to index only the fields your searches will use.

stored
Indicates whether this field may be returned in a search result or not, i.e. whether the value itself is stored in the Index. Note that stored fields will still only be returned if the Query object passed to the Generic Search Server indicates that they should be returned. Every field should be either indexed or stored or both - if a field is neither then it is of no relevance to the Search Service. Again, not storing values that your searches will not use will minimize index size and help performance, so only store the fields your searches will use.

entityName
The name of the application entity associated with this Field, or to be more specific, the name of the application Entity containing an attribute corresponding to this Field which will be used to populate the Index based on the parent Search Service definition. This information is needed for synchronization of application data with the Generic Search Server - all entities that are listed as being related to Search Service Fields will be registered with the SearchController (see "Search Controller" on page 7) and monitored for inserts, updates, and deletions. It is vitally important that the entityName attribute be populated with the appropriate values; omitted or invalid entityName attributes may result in invalid Index updates over time.

untokenized
This property indicates whether a field is to be tokenized and passed through the analyzer or not. It is a boolean value. If set to true, no tokenizing will be done and analysis will not be performed on this field before indexing or while searching.

analyzerName
This property specifies the analyzer to be used when tokenizing this field. The contents of this field may be set to LUCENESTANDARD, STANDARD, SIMPLE, STOP, WHITESPACE, KEYBOARD. (see analyzer in "SearchService Table" on page 10) If this field is not set then the default analyzer used will be that taken from the analyzer field of the associated SearchService.

Getting Started with the Generic Search Server API

Introduction
This chapter is not intended to be an exhaustive description of the entire Generic Search Server API - a full set of Javadoc is available as part of the installation. The purpose of this chapter is to provide a short introduction to the most important classes and operations in the API in order to allow Generic Search Server-based searches to be rapidly developed.

Mappers
Mappers are classes which define how Search Service data is mapped from the application database tables to the staging database tables. Each Search Service has its own Mapper - the Mapper to use is specified in the SearchService database table. For more details see "mapperName" on page 10.

This Mapper functionality is used in two processes:

1. When the Database Extractor is run, each Search Service Field is iterated over for a particular Search Service. For each Field, the corresponding Entity Attribute data is retrieved from the application database and populated into the SearchServiceRow staging database table

2. When a create, update or remove operation is called for an entity that is used in a Search Service, the relevant SearchServiceRow rows are updated with the related entity modifications

In both of these processes, the relevant Mapper for each Search Service is invoked to map data from the application database tables to the staging database tables.

On initialization of the Generic Search Server, the staging database information is read and used to construct the Indices from the Search Service metadata. The Search Server will periodically check the staging database for updates and keep the service data up to date.

The following Mapper API methods require implementation by search developers on a per-Search Service Basis:

```
SearchServiceRowDtlsList mapToStagingDb(
  final SearchServiceKey id) throws AppException,
    InformationalException;

List getObjectList(final SearchServiceKey serviceId,
  final Object obj) throws AppException, InformationalException;

String getExtKey(final SearchServiceKey serviceId, List objList);

void remove(final SearchServiceKey serviceId, final Object objKey)
    throws AppException, InformationalException;

Object getFieldValue(final SearchServiceKey serviceId,
    final List objList, final SearchServiceFieldDtls field);
```

For more details see “Implement Mapper Operations” on page 19

Search Controller

The Search Controller is a singleton object available for use in the application. It is responsible for keeping track of which entities are referenced in which Search Services. In addition, it provides an API for synchronizing changes made to application data with the relevant Indices on the Generic Search Server. Note that from a Client-Server perspective, the Search Controller lives on the 'Client' (in this case, the Cúram Application Server), not the 'Server' (in this case, the Generic Search Server).

The SearchController API is composed of three methods which can be invoked if any entity involved in populating an Index is modified. The search developer must be aware of which application entity operations will result in such modifications and invoke the appropriate methods on the SearchController. The methods exposed in this API are:

```
void SearchController.insert(final Object objectDtls,
  String entityName);
void SearchController.modify(final Object objectDtls,
  String entityName)
void SearchController.remove(final Object objKey, final String entityName);
```

For more details see “Add Synchronization to each Search Entity” on page 22

Search Service Connector

The SearchServiceConnector is a utility class that allows searches to be performed. The 'search' operation on this class is the only supported way for search developers to invoke a search on a Generic Search Server Index.

Behind the scenes, this class handles the details of connecting from the running application to an instance of the Generic Search Server, wherever it may be deployed.

Searches may be performed with the SearchServiceConnector using the method:

```
static SearchServerResults search(CuramQuery query)
```

Note: If the search index does not contain any data it will throw an IndexEmptyException. Developers implementing searches should handle this exception gracefully.

User credentials are required to connect to the Generic Search Server. The connector picks up the details of the current user and uses those to communicate with the Generic Search Server.

Note: Do not attempt to use the DoSearch method (or any Generic Search Server method) directly - it will not work as it is running in the context of the Cúram application, and not the context of a running Generic Search Server application

Queries

In order to do a search, a CuramQuery object must be constructed. The CuramQuery class consists of:

- The searchServiceId of the SearchService whose Index you wish to search. See “Search Service” on page 2 for more information on the concept of Search Services and “searchServiceId” on page 10 for details of how the searchServiceId is defined
- A list of CuramTerm objects or a Text attribute representing a Lucene query string- these represent the search criteria. See below for more information on Cúram Terms and the Text attribute
- A list of CuramField objects - values for these Fields will be returned as part of the search results, but only if the fields have been marked as 'Stored' in the SearchServiceField definition (see “stored” on page 12)
- An integer attribute maxHits indicating the maximum number of hits to be returned for this query.
- A boolean flag maxHitsUnbounded indicating that the maximum number of hits is not limited. If this flag is set the maxHits attribute value is ignored.

CuramTerm

CuramTerms are the part of the CuramQuery structure that represents search criteria.

There are three types of Terms: StandardTerm, a DateTerm, or a DateRange term. The CuramTerm object contains one of each of these types of these types, and has termType attribute specifying which of the term subtypes should be used. Only of one of the aggregated term subtypes is valid for each CuramTerm object.

For all term types, the 'field' attribute specifies the name of the Field in the Search Service to be searched (see “Field” on page 3 and “name” on page 11). The 'value' attribute is the search criterion to be used - the meaning of this varies for the different types of terms and is described below.

Query Structure

Each term has a field called *occurs*. How this is set determines the structure of the query - whether all the search terms must exist, only one, or some other combination. The possible values for *occurs* are `MUST`, `SHOULD`, `MUST_NOT`, and `MUST_FIELD`.

If `MUST` is specified for the *occurs* attribute for set of terms then a result will be returned only if all of the terms are found. If `SHOULD` is specified for a set of terms then a result will be returned if one or more of the terms are found. However, mixing these in a single query will give an undefined result and should be avoided. If you need to construct complex queries with AND and OR sub-queries then you must use the *text* query attribute described in "Text" on page 16.

If `MUST_NOT` is specified for the *occurs* attribute then only documents that do not match the term will be returned. Terms specifying this value may be mixed with terms specifying other values for the *occurs* attribute.

Using the `MUST_FIELD` option allows you to construct a subquery testing a particular index field for one of a set of values, i.e. an OR subquery within your main query. You should set this as the *occurs* value for all the terms dealing with that field and add a term for each acceptable value. Terms using `MUST_FIELD` can be part of an overall query using either the `MUST` or `SHOULD` term options.

Standard Terms

A Standard term is used for all searches that do not involve Dates, so this is the term type that you will use most frequently.

The most basic way to use a standard term is to simply specify the field name and a single token as the value. The search server will return results where the field value matches the search term exactly.

Another way to use a standard term is to specify a value that contains multiple tokens, such as in address. Again, the search server will return results where the field value matches the search term exactly.

If the search term specified is a single token containing a wildcard character then the search server will return all matching results. Supported wildcard characters are '*' which matches any string of characters, and '?' which matches a single character. Example:- term = "Dub*"

A StandardTerm may be treated as a Prefix Search. This means that we are looking for search results that contain the search criteria at the start. You specify a Prefix Search by setting the isPrefixSearch attribute of the StandardTerm. It has the same effect as specifying a '*' multi-character wildcard at the end of your search value. A prefix search term may not contain any other wildcards.

Example 1: For a standard tokenized prefix term "abc" the underlying search is for term = "abc*", for tokenized and prefixed multi-term searches, for instance, a prefixed search term "abc def", the underlying search is for term = "abc* def*"

Example 2: For a standard tokenized non-prefix starting with abc the term value = "abc*" must be specified. For tokenized, non-prefixed, multi-term searches starting with "abc" and "def" the value "abc* def*" should be specified.

Date and Date Range Terms

A Date term is similar to a Standard Term except that it is used to search fields that are of type Date or DateTime.

A Date Range term can be used to search for values that are between a minimum date (beginDate) and a maximum date (endDate). The 'isExclusive' Boolean attribute determines if the begin and end dates are included in the search criteria. If 'isExclusive' is set to true, the search is performed exclusive of the begin and end dates. If 'isExclusive' is set to false, the search is performed inclusive of the begin and end dates.

Note: When a query contains more than one term, the returned results are those that match all search terms - there is currently no concept of OR or NOT in the Generic Search Server API

Note: Bear in mind when using Dates for searching that it is your responsibility to ensure that the Date in your search term refers to the same time zone as was used when exporting the data to the Search Service

Text

The text attribute of the **CuramQuery** class can be used as an alternative to a set of terms. Using the text attribute gives you more flexibility in specifying your search criteria. However, use this method only if required because it is easy to introduce bugs in your searches with this method. The format for specifying search criteria that use this attribute is described in the Lucene documentation. Review the **queryparser** documentation at http://lucene.apache.org/core/

You cannot combine **CuramTerms** and the use of the text attribute of the **CuramQuery** class. If the text query string is present, then any **CuramTerms** present in the query are ignored.

Generating Queries

The Generic Search Server API contains a utility class designed to allow you to construct CuramQuery objects easily. This class is: `curam.core.impl.util.QueryBuilder`.

Constructing a Query Builder

The `QueryBuilder` is not a static class, you must construct a new `QueryBuilder` instance for each query you produce.

Use the `setUnbounded(boolean unbounded)` and `setMaxHits(long maxHits)` methods to specify the number of hits your generated query should return.

Adding Search Criteria

The `QueryBuilder` provides a selection of methods of the form `addXXTerm(...parameters...)` to add different types of search terms to your generated query easily. These terms are AND-ed together to form a complex query. These methods will not be described fully here but full details are available in the GSS javadoc.

Generating Queries from a Struct

If you have a Cúram struct you wish to use to generate a query you can do so using this method: `setTerms(final Object key)`.

This expects a struct where each attribute *XX* has a corresponding boolean attribute called *searchByXX* which specified whether that attribute should be used to search. Each attribute *XX* will be assumed to correspond to a SearchServiceField in your SearchService.

If the names of the attributes of your struct do not correspond to the names of the Fields you have defined for your Search Service (see "Field" on page 3 and "name" on page 11), then you can define a mapping between them using a dictionary HashMap. The mapping is from the attribute names in the struct to the SearchServiceField names. Simply add the pairs of strings to the HashMap, with the name of the struct attribute as the key and the name of the Field as the value. The dictionary can be specified in the constructor when you create your QueryBuilder object or later using the `setDictionary(HashMap<String, String>)` method.

Specifying which search service fields to return

In your query you can specify which subset of the search service's fields you would like returned as results. Often you will want all of them returned, so you can use the following convenience methods:

- `includeAllFieldsInService()`
- `excludeField(String fieldName)`
- `excludeFields(String[] fieldNames)`

Obtaining the Query Object

Use the `getQuery()` method to get the generated CuramQuery object.

Dealing with Search Results

Similar to the requirement to convert Cúram key structs to `CuramQuery` objects, `CuramDocument` s returned from searches also need to be converted to Cúram structs to be used in the application.

The `SearchServiceConnector` search method returns results in the form of a `SearchServerResults` object. This consists of a list of `CuramDocument` s, and each `CuramDocument` consists of a list of `CuramField` s. A utility class called `curam.core.impl.util.CuramDocToResultStruct` is provided to convert between `CuramDocuments` and Cúram structs.

```
static java.lang.Object convert(CuramDocument document,
     java.lang.Object structObj,
     java.util.HashMap dictionary)
```

This method takes a `CuramDocument` and a struct instance (via the parameter structObj). For each Field in the `CuramDocument`, the method attempts to find an attribute in the struct of the same name and datatype. A struct containing all mapped values is returned, this should be cast to a struct of the correct type.

If the names of the attributes of your struct do not correspond to the names of the Fields you have defined for your Search Service (see "Field" on page 3 and "name" on page 11), then you can define a mapping between them using the dictionary parameter. The mapping is from the Field names in the Search Service to the attribute names in the struct - simply add the pairs of strings to the HashMap, with the name of the Field as the key and the name of the struct attribute as the value. The convert function will then match Field names to attribute names using this HashMap

Note: Note that the attributes in your results struct whose names correspond to Fields in your document must have simple Cúram types, and not be aggregated structs.

Data Types and String Conversion

The Generic Search Server contains an API for converting searchable Cúram datatypes to Strings and vice versa. These may need to be used occasionally in custom Mappers, or if parsing results directly rather than using the supplied utility class `curam.core.impl.util.CuramDocToResultStruct`.

The converter class is `curam.core.impl.search.datatypes.DataTypeConverter`. This class contains methods to convert Cúram datatypes to Strings and to convert Strings back to Cúram datatypes (by means of passing in a struct and specifying which attribute in the struct is to be set).

Implementing a Search with the Generic Search Server

Overview

This chapter provides a worked example of the implementation of a Generic Search Server-based search within the Cúram application. The example worked through here is a Person Search.

The implementation steps are as follows:

- Write the SearchService and SearchServiceField dmx files
- Implement Mapper interface
- Implement search routing and invocation functionality
- Add synchronization of application operations to search entities (or use the Pull Mapper approach, see "Pull Mapper" on page 23
- Create a user interface and facade for the search - this is normal application development.

Person Search Example - Overview

It is important to note that users of the Cúram Generic Search Server should notice no functional difference between their searches and server searches implemented using SQL; in addition, the screens and general user experience can remain the same. As such, the following example assumes that readers will develop such application functionality (along with the appropriate Facade classes, etc.) as normal.

In our Person Search example, users will navigate to the relevant UIM page to perform a Person Search. On this page, they will fill in one or more search criteria. When they hit the 'Search' button, the search will be performed. The results will consist of a list of records matching the search criteria.

In application searches, it is common for the search criteria and details returned in the results list to be collated from multiple related entities. For the Person Search the following entities and their attributes are either used as search criteria or returned as result fields:

- Person - primaryAlternateID, personBirthName, motherBirthSurname, dateOfBirth, gender
- ConcernRole - sensitivity, concernRoleID
- AlternateName - firstForeName, surname

- AddressElement - city, address.

Each of these entities is related by a foreign key association; concernRoleID is thus the external key of the SearchService attribute for the PersonSearch Search Service (see “SearchService Table” on page 10)

The following attributes will thus be used in the search - either as part of the search criteria, or as a displayable part of the results list:

- referenceNumber
- forename
- surname
- address
- city
- dateOfBirth
- sex
- birthSurname
- motherSurname

As such, these will be the Fields stored in the SearchServiceField table for the PersonSearch Search Service.

Develop SearchService DMX files

Setup SearchService Record

Please see “Search Service Record” on page 36 and “SearchService Table” on page 10

Setup SearchServiceField Record

Please see “Search Service Field Record” on page 38 and “SearchServiceField Table” on page 10

Implement Mapper Operations

See “Mapper” on page 4 and “Mappers” on page 12 for an introduction to Mappers.

The following sections describe the implementation of the Mapper interface methods for each Search Service. An example for PersonSearch Search Service is provided for each method of the interface. Comprehensive Javadoc is also available for the Mapper interface and this should be read by all developers implementing a Search Service.

Mapper.mapToStagingDb interface

```
/**
 * Maps information in the Application database to the search
 * service staging database for the specified search service id.
 *
 * @param id the identifier of the search service.
 * @return the list of all mapped rows for the specified search
 *   service.
 * @throws AppException application exception
 * @throws InformationalException information exception.
 */
SearchServiceRowDtlsList mapToStagingDb(
  final SearchServiceKey id) throws AppException,
    InformationalException;
```

This method is invoked during the Database Extraction batch process; for each Search Service, mapToStagingDb is called to retrieve information from the source entities and return them to the batch process.

A Cúram ReadmultiOperation needs to be written to process all records to be stored on the staging database for each Search Service. A Generic Search Server operation called ExtractReadMultiOperation needs to be invoked on each of these records. Internally, this operation works out what other entities are required to populate an entire SearchServiceRow based on this data, and also constructs a SearchServiceRow object.

The result of this whole process is simply a list of SearchServiceRows, constituting all initial data to be populated into the staging database. The Database Extraction batch process then takes care of inserting these rows onto the staging database.

Mapper.getObjectList interface

```
/**
 * Populates the list with all entity objects for the
 * Search Service given any one of the entity objects used.
 * @param searchServiceId. the search service identifier
 * @param obj. The entity object from which all other are
 *   retrieved
 * @return the list of all entity objects for the this search
 *   service given a specified object parameter.
 */
List getObjectList(final SearchServiceKey serviceId,
  final Object obj) throws AppException,
    InformationalException;
```

As mentioned earlier, it is possible for data in a Search Service to be gathered from a number of different entities. It is also possible for these entities to be related by complex foreign key relationships (for example, an Address record could be related to a Person record via an addressID which is linked via a concernRoleAddressID which is in turn linked via a concernRoleID).

Things are made more complex when one of these entities gets updated via the application. When this happens, the Generic Search Server must be able to work out which entity has just been affected, what Searches it is involved in, and how it is related to every other entity included in each Search Service.

Ultimately, one or more Documents on one or more Search Service Indices will need to be updated, and information in these Documents may be gathered from a range of entities, not just the one that just got modified. However, given that Search Services have one and only one Mapper, each Mapper implementation only needs to worry about assembling information for its own Search Service.

The getObjectList interface method addresses this problem. Given a single updated entity record, getObjectList assembles all other entity Dtls records which will be required to update the corresponding Document in the current Search Service Index. The getObjectList method needs to be coded in such a way that any of the entities involved in a Search Service can be used as the starting point of this process. getObjectList is responsible for:

- Working out what entity has been passed to it
- Working out all related entities for the Search Service in question
- Reading and assembling all related entity records based on the data in the parameter entity

The mapper.getobjectList () method is called in the following processes:

- Database Synchronization insert
- Database Synchronization modify
- Initial Database Extraction

Note that for initial Database Extraction, the getObjectList interface method gets invoked for every item fetched from the ReadmultiOperation; typically this will be the top-level entity in this case (for example, for a Person Search Extract, all Person records would be read in a readmulti; getObjectList will then be called for each to retrieve all of the other information required to build a SearchServiceRow).

If this method is called for an input that isn't relevant to this search service, then the implementation should simply return an empty list.

Mapper.getExtKey interface

```
/**
 * Gets the Row external value for the specified object list.
 * @param searchServiceId. the search service identifier
 * @param objList the list of Search Service related entity
 *   objects.
 * @return the externalKey.
 */
String getExtKey(final SearchServiceKey serviceId, List objList) ;
```

The `getExtKey` interface method returns a unique identifier for the specified Search Service. This key is used as the key for each row in the `SearchServiceRow` table in the staging database. Note that the objList parameter is the output of the getObjectList interface method described above. For Example, `calling getExtKey` for the PersonSearch Search Service should return the concernRoleID of the record in question.

If this method is called for data that the search service doesn't care about then it should return null.

Mapper.remove interface

```
/**
 * Deletes the row identified by the specified key from the
 *   staging
 * database.
 * @param serviceId identifier of the service.
 * @param objKey the Key.
 * @throws AppException
 * @throws InformationalException
*/
void remove(SearchServiceKey serviceId, Object objKey)
  throws AppException, InformationalException;
```

Deletes the specified row object from the staging database.

Mapper.getFieldValue Interface

```
/**
 * If a specialized field value can't be covered by the
 * <code>SearchServiceMapper.getValue()
 * <code> functionality this method
 * should be overridden in the mapper for the specific search
 *   service.
 * @param objList list of entity objects for this specific
 *   mappers service id.
 * @param field the field whose value is required.
```

```
 */
Object getFieldValue(final SearchServiceKey serviceId,
   final List objList, final SearchServiceFieldDtls fieldDtls);
```

The Generic Search Server infrastructure will try to retrieve an entity attribute value from an object list by using Field metadata retrieved from the Search Service Field table. Typically, objectLists will contain entity dtls structs, and in such cases it is trivial for the Generic Search Server to use reflection to identify the correct attribute and get its value - this is exactly what is done behind the scenes.

However, if the objectList contains something other than an entity dtls struct (as in the case of Person Search, where an AddressElementDtlsList is present, itself containing a single AddressElement struct) then the Mapper.getFieldValue interface method should be implemented by search developers.

The Mapper.getFieldValue interface method should be implemented if a Mapper cannot automatically map a specific attribute value. The relevant entity and field name is passed in via the fieldDtls struct parameter, and the attribute value can be retrieved from the objList using reflection. It is up to the search developer to implement this method interface for the type or types to be catered for.

Empty strings should not be returned from this method - null should always be returned.

Mapper newInstance()

If the mapper is modelled then the factory class should be specified for the SearchService mapperName property. If the mapper is NOT modelled then the mapper implementation must implement a

```
public static Mapper newInstance();
```

interface returning an new instance of this search service's mapper. In this case the SearchService mapperName property will be the class name of this implementation class.

Search Router and Implementation

As mentioned previously, searching currently uses SQL. In future versions, it is likely that Platform and Solution searches will begin to use the Generic Search Server as the searching method of choice. However, it is likely that SQL searching will also continue to be supported as-is currently, both from an upgrade protection perspective, and from a fallback/failover option perspective in case of network or other deployment problems.

To facilitate this, a Search Router factory class should be implemented which should returns a reference to either the database search implementation or the Generic Search Server based implementation based on a property setting.

Add Synchronization to each Search Entity

As noted earlier, the Generic Search Server staging database must be updated in a timely manner when modifications are made to Search Service related entities. A single entity may well be being used in more than one Search Service, and each of these Search Services must reflect changes to that entity.

The *SearchController* class is responsible for insuring that all staging database information is up to date. The *SearchController insert*, *modify* and *remove* methods must be called from the application when the corresponding Search

Service entity operation is executed. The *insert* and *modify* SearchController operations modify the SearchServiceRow table information with the specified entity details struct data. The remove interface requires a key identifying the entity object being removed and the name of the entity.

```
/**
 * Generic insertion of entity updates to the database.
 *
 * @param details the object details.
 * @param entityName the name of the entity
 * @throws AppException application exception retrieving the
 *   registrar
 * or during Mapper insert.
 * @throws InformationalException information exception.
 */

public final void insert(final Object details,
  final String entityName)
    throws AppException, InformationalException
/**
 * Generic Modify of entity updates to the database.
 *
 * @param details the object details.
 * @param entityName the name of the entity
 * @throws AppException application exception retrieving the
 *   registrar
 * or during Mapper modify.
 * @throws InformationalException information exception.
 */
public final void modify(final Object details,
  final String entityName)
    throws AppException, InformationalException
/**
 * Generic remove of entity from the database.
 *
 * @param key the object key.
 * @param entityName the name of the entity
 * @throws AppException application exception.
 * @throws InformationalException information exception.
 */
public final void remove(final Object key,
  final String entityName) throws AppException,
    InformationalException
```

Pull Mapper

Introduction

In the previous chapter we described the events mechanism and how you can use it to keep your data synchronized with your search service. The Generic Search Server now provides another way to keep your search service up to date, called the Pull Mapper. This chapter describes how the Pull Mapper works and how you can use this with new searches you are developing.

Pull Mapper Overview

The event mechanism is by far the most efficient method of keeping your search services up to date. However, if your searches are complex, developing and fully testing your search service may be cumbersome. This is the problem the Pull Mapper sets out to solve.

The pull mapper uses timestamps on application records to find records that have been created or updated since the pull mapper or the extractor last ran. When it

finds such records it hands them off to the Search Controller to update the search services, and from here the process is exactly the same as the standard event mechanism. This process requires that all database tables involved in a search service are scanned, which does obviously require database resources. In essence the Pull Mapper sacrifices some runtime performance to provide a quicker and easier way to develop searches.

Developing with the Pull Mapper

This section will walk you through the process of developing a search service using the Pull Mapper.

Enable Last Updated Field on your searchable entities

Timestamps are required on all your database entities that are involved in search services and that use a Pull Mapper. These timestamp columns are automatically added and kept up to date by infrastructure when you enable the Last Updated Field feature for the entity in the model. The process for enabling this feature is documented in the `Server Modelling Guide`.

Modelling the table scan

Another modelling requirement imposed by the Pull Mapper to model an operation called `searchByLastwritten` (you must use this exact spelling/case.

This operation should be a nsmulti. The value for no generated SQL should be no. The operation should take a struct called *key*. You should model your own struct as a parameter, but it must have an attribute called *datetime*, which must be a `DateTime`. Later you will specify the classname of this struct in the GSSEntity table, as described below.

You need to provide SQL for the operation. Here is a simple example for a simple entity called Customer:

```
Select Customer.customer_id, Customer.name,
      recordStatus from Customer
 WHERE Customer.lastwritten >= :datetime
 INTO :customer_id :name :recordStatus
```

You must ensure you are selecting all the columns used by the search service.

In addition to the table scan method, you must have a standard read method on all your searchable entities.

Defining your search service

Your search service should be defined in the usual way (see “Implementing a Search with the Generic Search Server” on page 18

In addition to the SearchService and SearchServiceField tables you must add definitions to the GSSMapperType and GSSEntity tables.

GSSMapperType: This table simply maps the Search Service name to a string defining the mapper type. The default is the standard event mapper, which does not need to be specified. To use the pull mapper with a particular search service, a row should be added to this table mapping the Search Service name to the mapper type “PULL”.

searchServiceId

The Search Service Identifier; a string used to uniquely identify a Search Service. This is a foreign key of the SearchService table.

mapperType

Set this to 'PULL' (must be uppercase) to enable the Pull Mapper for the search service.

GSSEntity: When the pull mapper is in use GSS requires more information about the entities being used in the search services. For each unique entity listed in the child searchServiceField records belonging to each SearchService using the Pull Mapper, a GSSEntity record must be added (however if multiple fields belong to the same entity, you don't need to repeat the information).

searchServiceId

The Search Service Identifier; a string used to uniquely identify a Search Service. This is a foreign key of the SearchService table.

tblScanKeyStruct

This is the full classname of the struct that is the parameter to your modelled `searchByLastwritten` method described here: “Modelling the table scan” on page 24.

entityKeyStruct

This is the full classname of the parameter struct to your entity's read method.

EntityFactClass

This is the full classname of the generated factory class for your entity.

Writing your mapper class

A SearchServiceMapper implementation with the PullMapper is very much like a standard SearchServiceMapper implementation as described in the Implementing a Search with GSS chapter of this guide. However, there are some additional considerations.

When using the Pull Mapper with a complex search service that is composed of several related entities, ensure that your SearchServiceMapper implementation will behave appropriately when it has to deal with incomplete sets of entities, i.e. if entities A, B and C together comprise a search service your mapper may get called when only A and C exist. Depending on your search service the correct behaviour may be to add the incomplete set of data to the search service, or to do nothing until the set is complete.

Delete operations

The Pull Mapper cannot deal with standard delete operations. If you have a searchable entity that can be deleted then you must use another mechanism to deal with this operation (e.g the event based mechanism described in this guide).

However, the Pull Mapper can deal with standard logical delete operations, i.e. where a recordStatus column is set using the `RecordStatus` codetable values.

Searches and Queries in Depth

Introduction

Like any other piece of software, your GSS enabled searches must conform to certain design constraints if they are to perform acceptably and work as users expect. This chapter described in depth the process of designing a GSS search and proper use of GSS queries.

The Search Service - general guidelines

Your first design task is to decide what data you want to be able to search. Which fields do you want to be able to search on? What data do you want your search to return? There are several tradeoffs here so it's worth thinking about these things carefully.

Firstly, your index should contain as few fields as possible. Less fields mean a smaller index at runtime, and less use of system resources. Don't put it in your search service unless you need it.

Each field in your index can be indexed (i.e. searchable), stored (i.e. you can retrieve its value), or both. The reasons you would want to index a field are obvious - you want to be able to search based on it. However, some fields you might not want to search on - such as non-human-readable IDs. You might wish to add these to your search service as stored but non-indexed fields, so that you can perform database lookups based on the results of your searches. If you don't need to index a field, then don't - your extract processes will run faster and your index will consume less system resources.

Likewise, you may choose to store field values or not. In general, the index does not store the original value of a field, but keeps a searchable representation only. In general, to be useful, a search must store at least one field (the corresponding primary key of the database record).

After that, whether or not to store fields is a tradeoff. You could store all the fields you need in order to display your search results, or you could store only the database IDs and use these to retrieve the data from the database to display. The first option will result in a much larger index, but a faster display of search results because the database is not required.

Mapping your database structure to an Index - Denormalization

You may wish to include data from several different entities in your search. Unlike database searching, searching with indexes is not conducted using joins. Remember, the main benefit of using an index is to allow the work of searching to essentially happen up-front, when the index is created rather than when the search is invoked. Accordingly, all database tables should be denormalized for indexing. The alternative, which is to create separate indexes, search them separately, then attempt to merge results is much more complex and inefficient.

Example say you have the following entities: Entity Person with attributes name, date of birth, and a foreign key pointing to an Address entity Entity Address with attributes street address, city, and country. You wish to create a search that allows you to search for persons by name, DOB, street address, city and country. You would create a searchable index that contains all the data from both tables.

When you have multiple entities contributing to a single search index, bear in mind that updates to any of the tables concerned can lead to the search index requiring an update.

Tokenized and Untokenized Fields

We have already briefly touched on the issue of tokenization of search fields. What tokenization entails is essentially breaking up the indexed data into units called tokens. This is done by use of an analyzer. Different analyzers behave differently, some may break tokens at whitespace, some at punctuation, etc. The resulting tokens are also usually transformed to lowercase. For tokenized fields query strings are tokenized in the same way, so searches are case insensitive, among other benefits.

For some fields it doesn't make sense to tokenize. Good examples of this are computer generated values, such as codetable codes. In general, however, most of your fields should be tokenized. In particular, the behaviour of multi-word untokenized fields and searches is counterintuitive. If you find your searches are not returning the data you expect consider whether this may be the case.

Example: Take an address field, with a document containing "Joyce Way Parkwest Dublin". If this were a tokenized field using the standard analyzer, then the index will contain four terms: joyce, way, parkwest and dublin. Any query string that contains terms matching these terms (exactly or via a wildcard) will find this document. For instance: "Dublin", "Joyce Way", "park*", etc.

However, if this field is untokenized and the same document is added, the index will contain a single term: "Joyce Way Parkwest Dublin". Much fewer query strings will match this, essentially only the string itself or the first part of the string as a prefix search. The search will also be case sensitive.

Wildcards

GSS supports single character and multi-character wildcards. The question mark symbol, “?” matches any single character. The asterisk symbol, “*” matches any sequence of characters. Neither of these may be used as the first character in a search term because this results in poor performance. When implementing a search developers should consider whether users should be allowed enter these characters in searches, and if so provide useful online help. Otherwise they can be escaped with an escape character: “\”. It may also be useful to check that these characters do not occur at the start of search terms and return a more specific error message to the user than the GSS infrastructure is capable of doing (a generic exception to indicate that the query is invalid will be returned, but the developer implementing the search will be able to add more information regarding which field is invalid).

Analyzers in Depth

As previously introduced, Analyzers prepare your searchable text for indexing and searching.

Your choice of analyzers is very important. Analyzers are concrete classes that extend the class org.apache.lucene.analysis.Analyzer. The GSS comes complete with several analyzers, and you can create and use your own. Sometimes when you are tempted to define a field as untokenized you may want to consider your choice of analyzer more carefully instead.

Each Search Service has a default analyzer, and any Search Service Field can override that analyzer to define a specific analyzer for use with that field (see “analyzerName” on page 12) GSS will use the same analyzer both for indexing and for searching.

The Generic Search Server provides the following predefined analyzers.

LUCENESTANDARD
Splits text at punctuation characters, removing punctuation. However, a dot that's not followed by whitespace is considered part of a token. Splits words at hyphens, unless there's a number in the token, in which case the whole token is interpreted as a product number and is not split. Recognizes email addresses and internet hostnames as one token. Normalizes token text to lower case and removes common English stop words.

STANDARD
Similar to LUCENESTANDARD analyzer but common stopwords are removed from the tokenized terms and if the content to be tokenized is a single number it will not be altered (making it suitable for processing generated infrastructure IDs which may be negative numbers).

SIMPLE
Splits text at non-letter characters and normalizes token text to lower case.

STOP Splits text at non-letter characters, normalizes token text to lower case and removes common English stop words.

WHITESPACE
Splits text at whitespace. Adjacent sequences of non-Whitespace characters form tokens.

KEYWORD
"Tokenizes" the entire stream as a single token. This is useful for data like zip codes, ids, and some product names.

Note that if you are using an analyzer other than a predefined GSS analyzer or analyzers shipped with Lucene the class must be available on the Generic Search Server classpath.

Running the Generic Search Server in Eclipse

Introduction

This chapter describes how to configure the development environment to run the Generic Search Server in the Eclipse IDE for development and test purposes.

The Generic Search Server can be run in RMI mode for development purposes, in a similar way to the Cúram application itself. This chapter details how to set this up.

Bootstrap.properties

Before starting development, the relevant settings should be added to your `Bootstrap.properties` file, where necessary. See “Configuration Properties” on page 35 for a description of the configuration properties.

Launching the Cúram Generic Search Server from Eclipse

Like the Cúram application, in development mode the Generic Search Server requires a tnameserv process to be running on your machine.

In your development installation, navigate to the EJBServer/components/core/lib/core.jar file in Eclipse:

- Right-click on the core.jar file and select **Run as Java Application**
- From the list of classes select SearchDataExtractor and click OK. This will build your staging database.
- Right-click again on the core.jar file and select **Run as Java Application**
- From the list of classes select StartUpSearchServer and click OK. This will start the GSS search server.

Run the SearchDataExtractor to build your staging database before StartSearchServer. And run the StartSearchServer process whenever you need to run a Search Server instance to test your search functionality. You should rerun your SearchDataExtractor before you start your SearchServer if you have rebuilt your application database.

Note: If any of your Search Services use third party or custom Analyzers (i.e. Analyzers that do not come as part of the Lucene distribution), ensure that they are added to the classpath of the EJBServer project.

Deploying the Generic Search Server

Introduction

This chapter describes the process of deploying the Cúram Generic Search Server onto your application server. This chapter is aimed at administrators who will be deploying the Search Server alongside Cúram application and who are familiar with the relevant Cúram Deployment Guide.

Deployment Options

GSS is deployed as part of the Cúram ear file, which is useful for testing purposes or small deployments. You can also deploy GSS in its own ear file for a higher performant deployment configuration. There are build targets that will create a SearchServer.ear file, which can then be deployed separately.

Deployment Process

The deployment process consists of the following steps:

- Set up your Bootstrap.properties with your configuration properties and any properties related to your Search Server. See "Configuration Properties" on page 35 for a description of the configuration properties.
- Build your Cúram application ear file as usual (this will also build your GSS ear file).
- Set up your database as usual.
- Run the Cúram Generic Search Server search database extractor.
- Deploy all your application ear files, including SearchServer.ear
- Log into the application as an administrator, and set up the system properties to enable the GSS-supported searches that you wish to use and to enable the synchronization mechanism. See "Generic Search Server enabled searches" on page 8
- Run the generic search server startup process.

The Generic Search Server should then be available to respond to queries.

Clustering

Deploying multiple instance of GSS is supported on a cluster environment. Extended discussion of advanced cluster deployment topologies is beyond the scope of this guide. However it is important to set the following properties in your `Bootstrap.properties` file:

- `curam.searchserver.server.host`
- `curam.searchserver.server.port`
- `curam.searchserver.sync.username`
- `curam.searchserver.sync.password`

See "Configuration Properties" on page 35 for details of these and other configuration properties. Also see "Recommended configuration for Production Environment" on page 35.

Note: It is advised to deploy GSS in its own cluster.

Build Targets

The following build targets are specific to the Cúram Generic Search Server.

weblogicEARGSS

This target builds the SearchServer.ear file and copies it to the EJBServer/build/ear/WLS/ directory, alongside your Cúram ear file. It is run automatically as part of the **weblogicEAR** target. The SearchServer ear file must be built after the Cúram ear file. After the SearchServer ear file has been build the application is ready for deployment onto Oracle WebLogic Application Server using the same build targets or manual processes as the Cúram ear file.

websphereEARGSS

This target builds the SearchServer.ear file and copies it to the EJBServer/build/ear/WLS/ directory, alongside your Cúram ear file. It is run automatically as part of the **websphereEAR** target. The SearchServer ear file must be built after the Cúram ear file. After the SearchServer ear file has been build the application is ready for deployment onto IBM®WebSphere® Application Server using the same build targets or manual processes as the Cúram ear file.

runExtractor

This target must be run after your application database has been configured. By default it extracts all data related to the CEF search services and any other search services you have defined out of your application database and transforms it into a format suitable for indexing. The length of time that this process will take will increase with the amount of data to be extracted. This target may be rerun multiple times if required.

This target may executed against a single search service by specifying the "SERVICE" property. E.g: "build runExtractor -DSERVICE=PersonSearch"

runPersist

If you are using a persisted database index (see "Index Persistence" on page 32, this target builds the index from the staging database tables. It should only be run after your application database has been configured and the runExtract target has been run. The runExtract target will build your persisted index if persistence is configured, therefore this target only needs to be run separately if you have changed your configuration since running the runExtractor target.

startupSearchServer

This target is optional. If it is to be run it must be run after your Generic Search Server has been deployed onto your application server. It triggers the Search Server to set up its indexes so that they are available for searching. The length of time that this process will take will increase with the amount of data to be indexed. If you don't run the startup target explicitly, the search server will initialize its indexes on the first search request. This feature is primarily there for ease of testing with small datasets. For large datasets the automatic startup feature should not be used. You can disable the automatic startup by setting the property "curam.searchserver.autostartup.disabled" to true in your `Bootstrap.properties`. when you set up your ear file - this is recommended.

Database Performance

The Cúram application and the Search Server application share a common database, but impose quite different demands on it. The SearchServiceRow table will see the bulk of writes and accesses, and it will grow very large, as it essentially contains a version of all the searchable data. The Cúram application will write to this table as searchable entities are inserted or updated. Periodically, if your Search Server is restarted or when it synchronizes, there will be a lot of reads from this table. It may make sense to place the SearchServiceRow table in a different tablespace to the rest of the application tables, depending on your organizations resources and needs.

Time Considerations

If different machines are used to run instances of the Curam application and the Generic Search Server then all systems must have their clocks in sync and remain in the same time zone. We recommend that a software solution such as NTP (depending on your deployment platform) is employed to ensure this remains the case. If this is not done then there can be no guarantee that all updates to application data will be accurately reflected by the Generic Search Server.

Performance

Introduction

This chapter describes Cúram Search Server performance and how various deployment scenarios and configuration settings may influence it.

Index types

As described in "Indices" on page 1 an index is the data structure that powers GSS searches. It can be a fairly sizable data structure (see "Index Size Calculation" on page 34 and this begs the question: where to store it? GSS provides two options: memory or file. For information on how to configure these properties see "Configuration Properties" on page 35

RAM (in-memory) directories must be reconstructed each time an application server is started (unless persistence is used, see "Index Persistence" on page 32. They are fast to access but their memory requirements may exceed the resources available. RAM directories may be very useful for testing however, as they do not hold state.

File indexes use the local file system to store the index. Even though the Java Platform, Enterprise Edition specification does not cover file system access in practice this works with all supported versions documented in a separate

document, *Curam Supported Prerequisites* document. Naturally the better the performance of the underlying filesystem used the better the performance of GSS will be.

Index Persistence

Each Search Service has an associated index that is queried during each search. This index is generated from the staging database tables when the Search Server initializes. A substantial amount of time may be required to read all the search service data from the staging database tables and subsequently to generate the relevant indices for this data.

The Generic Search Server provides the means to persist the current index on the database so as to improve the startup time. When index persistence is enabled, and before the staging tables are interrogated, the persisted index is loaded if available. If it is not available, all data is read from the staging tables and startup will be slower.

The persisted index has a timestamp associated with it and this is stored in the appropriate Search Service table for that index. This timestamp indicates the time that RAM index was last persisted to disk. Knowing this time enables the Generic Search Server to retrieve any new or modified Search Service data from the staging tables. The persisted index and the new/modified data from the staging tables provide for a complete in-memory index ready for searching. Time is saved by reducing the access to the staging tables and the associated processing during index construction.

Persisted index data is stored in BLOB format, therefore performance of reading and writing a large index from and to the database is optimal.

Persistence Operation Invocation

The Batch operation DataBaseIndexPersist.persistIndex() is executed to perform the backup for all indices. The process for persisting each index is to:-

1. Read current persisted index
2. Read new or modified data from staging table data
3. Generate an in-memory index with 1) + 2) above.
4. Save newly generated in-memory index to the database.
5. Repeat 1) to 4) for each search service.

Testing and operational considerations

Persisted indexes, FILE indexes are designed to retain built indexes between server resets.

The data also persists between database rebuild operations, and this may cause issues for testers if index data becomes inconsistent with the current database.

Similarly, in an operational setting, if database updates occur without search index updates being enabled in the application (via the "curam.lucene.luceneOnlineSynchronizationEnabled" property) the data in the index will become out of date and problems may occur.

In the event of either of the above scenarios, persisted data can be removed manually from the database by dropping all database tables that begin with "GSS_" (there will be one table for each Search Service). The persisted indexes will be rebuilt as normal when an extract or persist operation is run.

In the case of a FILE index the file may be deleted, and in the event of a standard RAM search service encountering such issues, rerunning the extract process will fix the problems.

Performance Tuning

This section describes parameters that influence the performance of reading and writing the search index. They determine how the index is constructed and how new entries are to be written to it.

Max Merge Documents

`curam.searchserver.luceneadaptor.searcher.index.maxmergedocs`

This property improves search times for higher values and for lower values gives better results when an index encounters frequent updating. Small values (e.g., less than 10,000) are best if the index is frequently updated, however, search times performance will be impacted. The default is 10000000. If the search performance is most important this value should be large, for example the default value, or else if the search data updating performance is more important then the value should to a small value, for example 10,000.

Merge Factor

`curam.searchserver.luceneadaptor.searcher.pool.mergefactor`

This property has an impact on RAM used while updating an index. The index requires updating as a result of search affecting application data updates. For small values(less than 10), searches will be faster, however, search index updates will be slower. With larger values(greater than 10), more RAM is used during index updating, and while searches are slower, index updating is faster. The default value is 10; If the search performance is most important this value should be less than 10 or else if the search data updating performance is more important then the value should be greater than 10.

Enable Persistence

`curam.searchserver.server.index.persistence.enable`

add curam.searchserver.server.index.persistence.enable=true to Bootstrap.properties to enable index persistence.

Note:- If this property is enabled, during the Database extraction execution, the new persisted indices will also be generated.

References

For more information of parameters discussed in this section refer to the javadoc for Apache Lucene 2.2.0.

Searcher Pooling

This section describes the how to configure Search Pools and the influence this has on search performance.

Overview

Lucene has an internal caching mechanism which makes searches using long-lived IndexSearcher objects faster than searches with newly created IndexSearcher instances. One shared IndexSearcher instance would be enough to get fast searches in single-user environment, but a standard use case in a server environment is that multiple clients search the index simultaneously. To avoid sequencing the search

requests in this setting, which would degrade individual search performance, the GSS uses an IndexSearcher pool that keeps a defined number of IndexSearcher instances for reuse by simultaneous search requests.

An IndexSearcher will only see the index as of the "point in time" that it was opened. Any updates to the index after the IndexSearcher was opened are not visible until the IndexSearcher is re-opened. Each IndexSearcher instance can use a very significant amount of memory depending on index size and whether the index has been updated in the meantime or not. The IndexSearcher pool takes care of closing and reopening IndexSearcher instances when an index update occurs.

Pool configuration properties

IndexSearcher pool has two basic options - initial size and maximum size. The following parameter

```
curam.searchserver.luceneadaptor.searcher.pool.initialsize
```

specifies how many IndexSearcher instances will be open at startup and kept open at all times for use by search clients. This is a required option and takes positive integer values including 0. If not specified the default value is "0". Typically this property should be set to the anticipated maximum number of simultaneous client searches.

```
curam.searchserver.luceneadaptor.searcher.pool.maxsize
```

specifies what is the maximum number of IndexSearcher instances allowed to be open at any given time. If more than this number of searches happens at any time an exception will be thrown and logged for diagnostic purposes. This option takes positive integer numbers, and if not specified the default value is "100". There is also the associated

```
curam.searchserver.luceneadaptor.searcher.pool.maxsizeunbounded
```

option which means the maximum pool size is unlimited. The option accepts values of "true" or "false". If not specified default is "true". If this option is set to "true" the curam.searchserver.luceneadaptor.searcher.pool.maxsize option value will be ignored. One of those two associated options is required.

RAM Limitations

The Global Search Server indices are stored in-memory if configured to do so. If using a 32-bit JVM A memory limitation of ~3GB is encountered. However, this figure is not only the memory available to GSS but also to all other system processes. It is important to note that very large Search Service indices could exceed the maximum RAM available to the GSS and other deployed processes.

Index Size Calculation

The index size is approximately 30% of the text indexed. The Search Service's indexed and stored properties (these can be obtained from the SearchServiceField attributes where indexed=true and stored=true) are used to estimate the index size.

- 1 million Person records. where 1 record = 1 index document.
- 1 document may contain the following indexed and stored properties determined from the SearchServiceField table for a PersonSearch service:- refnumber(10) forename(20), surname(20), AddressLine1(30), AddressLine2(30), city(20), country(15), gender(10). where (*) = max value size in character for that field.
- 1 document = (155 characters for stored value) + (66 characters for each field/term name.) = 221.

- Memory 1M Person documents and Java using 16-bit unicode per character. Total indexed and returned text 442MB * 30% = 132MB.

Recommended configuration

The recommended configuration for Cúram Generic Search server is the use of a FILE index type with index persistence turned off as standard. This should provide good performance without sizing worries. The search server should be deployed as a separate application and not co-located with Cúram application (see "Deploying the Generic Search Server" on page 29.

Recommended configuration for Production Environment

FILE index type is the only supported configuration in production environment.

Cúram Generic Search Server Configuration Properties

Configuration Properties

Before starting development, or deploying your Cúram Generic Search Server the following settings should be added to your `Bootstrap.properties` file, where necessary.

Table 3. Cúram Generic Search Server Basic Configuration Settings

Property name	Description
curam.searchserver.sync.interval	The interval in milliseconds between Generic Search Server synchronization invocations. This is effectively the maximum time between data being updated and it being available for search. If this property is not set, the default is to synchronize every 3 seconds.
curam.searchserver.sync.username	The username used for logging into the application to perform synchronization. The user must be authorized to run the DoGSSSync.sync function identifier. Required when running under WebSphere application server only. Omitting to specify this property and the associated password will not prevent the sync operation from running but it will result in security warnings being written to the logfiles on each synchronization.
curam.searchserver.sync.password	Password associated with the curam.searchserver.sync.username described in the entry above. This password should be encrypted with the standard Cúram encrypt build target.
curam.searchserver.environment.vendor	This property should be set to "ITD", "IBM", or "BEA" depending on whether you are using the Search Server in development mode or deploying to WebSphere or WebLogic. If this property is not set the Search Server will default to using curam.environment.as.vendor property.
curam.searchserver.server.host	The domain name or IP address of the server on which your Search Server is running. This must be set in order for you to be able to run the server startup process from the command line. If this property is not set the default is localhost.
curam.searchserver.server.port	The port on which your application server's RMI service is available. This must be set in order for you to be able to run the server startup process from the command line.
curam.searchserver.autostartup.disabled	For testing and development purposes, the Search Server will initialize its indexes on the first search request, unless it has already been started up. In a deployment scenario, you may want to disable this behaviour and ensure that the startup process is run from the command line, to give you more control over the process. Setting this property to true disables the automatic startup behaviour. Note that the search server will throw an exception in response to any search attempts that occur before the startup is complete.

Table 3. Cúram Generic Search Server Basic Configuration Settings (continued)

Property name	Description
curam.searchserver. luceneadaptor.searcher.index.maxmergedocs	This property is used to tweak the performance of index reading and writing. Larger values "1,000,000" are best for batched index writing and speedier searches. Smaller values "10,000" are best for interactive indexing where numerous individual index updates occur.
curam.searchserver.luceneadaptor.document.flush.count	Indicates the count of documents to update before flushing to the index, when dealing with a large batch of documents. If not specified, this defaults to 1000 documents. Tuning this property can reduce the time required to build your index initially on index persistence or server startup.
curam.searchserver.term.min.length	Minimum allowable length of a search term. Defaults to two characters. Using very short search terms will result in poor search performance, and usually in poor quality of search results.
curam.searchserver.directory.type	This specifies the type of storage to use for search services - may be RAM, FILE. RAM is the default index type and suitable for smaller indexes that require very fast performance. FILE setting provides storage for large indices on the File System.
curam.searchserver.file.index.location	This property indicates where to store the file index on the File System if curam.searchserver.directory.type=FILE with more data. If deploying to multiple machines the file location should exist on each targeted machine.

Table 4. Cúram Generic Search Server Searcher Pool Settings

Property name	Description
curam.searchserver.luceneadaptor.searcher.pool.initialsize	This property initializes the number of searchers within the searcher pool on startup. The default is 0.
curam.searchserver.luceneadaptor.searcher.pool.maxsize	This property indicates the maximum number of IndexSearchers within the searcher pool. The default is 100.
curam.searchserver.luceneadaptor .searcher.pool.maxsizeunbounded	This property set to "true" overrides curam.searchserver.luceneadaptor.searcher.pool.maxsize and indicates there is no maximum number of IndexSearchers allowed within the searcher pool. The default is "true".
curam.searchserver.luceneadaptor.searcher.pool. mergefactor	This property is used to tweak the performance of index reading and writing. The default value is "10". Minimum value is "2". Higher values result in more RAM usage, slower searching, but quicker index writing.

Table 5. Cúram Generic Search Server Persistence Settings

Property name	Description
curam.searchserver.server.index.persistence.enable	This property should be set to "true" to enable index persistence. If this property is not set the default is "false".
curam.searchserver.custom.db.init	This property should be set to "true" when customizing index persistence database tables. It indicates that the default index persistence tables are not to be used and the `CustomDBSearchServices.properties` file should be used to set up these tables.

Sample DMX Listings: PersonSearch

Search Service Record

```
     <?xml version="1.0" encoding="UTF-8"?>
<table name="SEARCHSERVICE">

  <column name="
      searchServiceId
```

```
      " type="text" />
  <column name="
      name
      " type="text" />
  <column name="
      extKeyName
      " type="text" />
  <column name="
      analyzer
      " type="text" />
  <column name="
      locked
      " type="bool" />
  <column name="
      forcedReindexTimeStamp
      " type="timestamp" />
  <column name="
      mapperName
      " type="text" />
  <column name="
      prstBlobSize
      " type="text" />
  <row>
    <attribute name="searchServiceId">
      <value>
      PersonSearch
      </value>
    </attribute>
    <attribute name="name">
      <value>
      PersonSearch
      </value> </attribute>
    <attribute name="extKeyName">
      <value>
      ConcernRoleID
      </value> </attribute>
    <attribute name="analyzer">
      <value>
      STANDARD
      </value>
    </attribute>
    <attribute name="locked">
      <value>
      0
      </value>
    </attribute>
    <attribute name="forcedReindexTimeStamp">
      <value>
      SYSTIME
      </value>
    </attribute>
    <attribute name="mapperName">
      <value>
      curam.core.impl.PersonSearchMapper
      </value>
    </attribute>
    <attribute name="prstBlobSize">
      <value>
      50M
      </value>
    </attribute>
  </row>
</table>
```

Search Service Field Record

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="SEARCHSERVICEFIELD">

  <column name="
      searchServiceFieldId
      " type="text" />
  <column name="
      searchServiceId
      " type="text" />
  <column name="
      name
      " type="text" />
  <column name="
      indexed
      " type="bool" />
  <column name="
      type
      " type="text" />
  <column name="
      stored
      " type="bool" />
  <column name="
      entityName
      " type="text" />
  <column name="
      analyzerName
      " type="text" />
  <column name="
      untokenized
      " type="bool" />

  <row>
    <attribute name="searchServiceFieldId">
      <value>
      field0
      </value>
    </attribute>
    <attribute name="searchServiceId">
      <value>
      PersonSearch
      </value>
    </attribute><attribute name="name">
      <value>
      primaryAlternateID
      </value>
    </attribute><attribute name="indexed">
      <value>
      1
      </value>
    </attribute><attribute name="type">
      <value>
      String
      </value>
    </attribute><attribute name="stored">
      <value>
      1
      </value>
    </attribute>
    <attribute name="entityName">
      <value>
      Person
      </value>
    </attribute>
    <attribute name="analyzerName">
      <value></value>
```

```
    </attribute>
    <attribute name="untokenized">
      <value>
      1
      </value>
    </attribute>
  </row>

  <row>
    <attribute name="searchServiceFieldId">
      <value>
      field1
      </value>
    </attribute>
    <attribute name="searchServiceId">
      <value>
      PersonSearch
      </value>
    </attribute><attribute name="name">
      <value>
      firstForename
      </value>
    </attribute><attribute name="indexed">
      <value>
      1
      </value>
    </attribute><attribute name="type">
      <value>
      String
      </value>
    </attribute>
    <attribute name="stored">
      <value>
      1
      </value>
    </attribute>
    <attribute name="entityName">
      <value>
      AlternateName
      </value>
    </attribute>
    <attribute name="analyzerName">
      <value>
      STANDARD
      </value>
    </attribute>
    <attribute name="untokenized">
      <value>
      0
      </value>
    </attribute>
  </row>

.....

</table>
```

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing

Legal and Intellectual Property Law.

IBM Japan Ltd.

19-21, Nihonbashi-Hakozakicho, Chuo-ku

Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Dept F6, Bldg 1

294 Route 100

Somers NY 10589-3216

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources.

IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Privacy Policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user’s name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and

IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/us/en/copytrade.shtml.

Apache is a trademark of Apache Software Foundation.

Oracle, WebLogic Server, Java and all Java-based trademarks and logos are registered trademarks of Oracle and/or its affiliates.

Other names may be trademarks of their respective owners. Other company, product, and service names may be trademarks or service marks of others.

Printed in USA