

LINKING DOCUMENTS IN
REPOSITORIES TO STRUCTURED
DATA IN DATABASE

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2007

By

Lu Mao

School of Computer Science

Contents

Abstract	9
Declaration	11
Copyright	12
Acknowledgements	13
1 Introduction	14
1.1 Types of Information Storage Management	15
1.2 Challenges in Distributed Data Management	16
1.3 Project Objectives	19
1.4 Structure of this Dissertation	20
1.5 Summary	21
2 Background	23
2.1 Content Management System	24
2.1.1 The Content to Be Managed	24
2.1.2 Components in a CMS	25
2.2 Content Repository	27
2.3 Java Content Repository API	28
2.3.1 Apache Jackrabbit	31

2.4	IBM Unstructured Information Management Architecture	33
2.4.1	Text Analysis Technology	34
2.4.2	Key Components in UIMA	35
2.4.3	Text Analysis Levels	37
2.5	Apache Derby	38
2.6	Summary	39
3	Linking Data Architectures	40
3.1	User Interfaces	43
3.2	Linking Data System Model	46
3.3	Data Flow Model	47
3.4	Linking Data Architecture	48
3.5	Summary	50
4	Linking Data Extraction	51
4.1	Document Annotation	52
4.1.1	Annotation Function Design	55
4.1.2	Annotator Generation	57
4.2	Annotator Descriptor	59
4.3	Annotation Output	59
4.3.1	Text Span	60
4.3.2	Annotation Result Representation	60
4.4	Summary	61
5	Data Correlation Approach	63
5.1	Correlation Query	64
5.1.1	Correlation Overhead	65
5.1.2	Correlation Query Evaluation	68

5.2	Populating Database with Unstructured Data	70
5.3	Sentiment Analysis	72
5.3.1	Approaches in Sentiment Analysis	72
5.3.2	A Sentiment Analysis Example	73
5.4	Summary	74
6	Experimental Study	76
6.1	Prototype System	76
6.1.1	Technological Solutions	77
6.1.2	Correlation Correctness Test	78
6.2	Performance Experiment Design	79
6.2.1	CQT vs. Data Volume	81
6.2.2	CQT vs. Hit Rate	81
6.2.3	Correlation Overhead vs. Data Volume	82
6.2.4	Correlation Overhead vs. Hit Rate	82
6.2.5	Automatic Document Generator	83
6.3	Result Analysis	83
6.3.1	CQT vs. Data Volume of Unstructured Document	84
6.3.2	CQT vs. Hit Rate	87
6.3.3	Correlation Overhead vs. Data Volume of Documents	87
6.3.4	Correlation Overhead vs. Hit Rate	90
6.4	Summary	90
7	Related Work	91
8	Conclusion and Future Work	94
	Bibliography	97

A Using Apache Jackrabbit and JCR API	100
B Build an Annotator Using the UIMA SDK	105
C Populating Database with Unstructured Data	118
D Implementation of the Correlation Query Evaluation	125
E Lindas GUI Demo	128

List of Figures

2.1	Repository Components	28
2.2	Different ways to use the JSR-170 for accessing content repository	30
2.3	Apache Jackrabbit Architectural Layers (source: [Ove])	32
2.4	Document-Level Text Analysis Engine	38
3.1	Email documents include annotation	42
3.2	Correlation Query GUI in Lindas	44
3.3	Administrator Interface in Lindas	45
3.4	Linking Data System Model	47
3.5	Data-flow Model	48
3.6	Linking Data System Architecture	49
4.1	Customer support requests in web-form	54
4.2	Customer support requests in web-form	55
4.3	Aggregated Text Annotation Engine (Annotator Pipeline)	56
4.4	An annotation algorithm	58
4.5	Document annotation result in XML	62
5.1	Task Decomposition in a Correlation Query	64
5.2	Correlation Overhead	67
5.3	Correlation Query Evaluation	69

5.4	Populating a Database with Analytical Outcomes	71
5.5	Sentiment Relational Table	74
6.1	CQT vs. Data Volume of Documents	85
6.2	CQT vs. Hit Rate	86
6.3	Correlation Overhead vs. Data Volume of Documents	88
6.4	Correlation Overhead vs. Hit Rate	89
A.1	A Jackrabbit repository for emails	101
A.2	Initialising a Jackrabbit repository	102
A.3	Add a file to Jackrabbit repository	103
A.4	Retrieve a files from Jackrabbit repository	104
B.1	Type System Descriptor for an annotator	107
B.2	UIMA SDK Component Descriptor Editor	108
B.3	A Java implementation of the annotation algorithm for email ad- dresses	110
B.4	UIMA SDK Component Descriptor Editor	112
B.5	The Capability page of the UIMA SDK Component Descriptor Editor	113
B.6	the UIMA Document Analyzer	115
B.7	the UIMA 'Analysis Results' window	116
B.8	Viewing the result of annotation	117
C.1	Step 1: prepare input	119
C.2	Step 2a: Analysis unstructured documents	120
C.3	Step 2b: Analysis unstructured documents	121
C.4	Step 3: Output annotated documents	122
C.5	Step 4: Process annotated unstructured documents	123

C.6	Step 5: Store discovered unstructured data into relational database	124
D.1	the Correlation Query Evaluation Implementation	127
E.1	The Correlation Query Control GUI (a)	129
E.2	The Correlation Query Control GUI (b)	130
E.3	The Correlation Query Control GUI (c)	131

Abstract

As business becomes more and more complicate, data is now emerging as a major market battleground. All business information are expressed in three kinds of forms which are unstructured and semi-structured documents in content repositories as well as structured data residing in databases. Faced with growing knowledge management needs, enterprises are increasingly realising the importance of interlinking critical business information distributed across various kinds of data sources. Different forms of data are managed under different data management systems. One of the major challenges that companies face is that of extracting value from the combination of data assets they amass. The question to be addressed here is that how to query in such a way as to obtain added value from the combination of un- and semi-structured documents with structured data.

This dissertation presents the project that aims to specify a software architecture for correlating unstructured and semi-structured documents managed by content management systems and structured data managed by database management systems. Moreover, a prototype system called the Linking Data System (Lindas in short) will be designed and implemented as a tool for realising the architecture above. This system is built upon existing pieces software infrastructures and tools, and capable of eliciting data from unstructured and semi-structured documents; processing and evaluating queries that link unstructured

and semi-structured documents with structured data. Experiments have been conducted for the purpose of evaluating some aspects of the performance of Lindas.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the School of Computer Science.

Acknowledgements

I would like to thank my MSc. project supervisor Dr Alvaro A.A.Fernandes for his ongoing support and guidance throughout the course of this project.

Chapter 1

Introduction

As business becomes more and more complicated, data is now emerging as a major market battleground. In every minute of our life, while business activities conducted by companies around the globe are continuously progressing, increasingly large and varied amounts of data, such as financial reports, customer records, memos, HTML documents, are being generated. At the same time, companies also continuously receive new customer emails, product request forms, field service notes and customer comments. Nowadays, many more database researchers than prior to the acceptance of the Internet being used in business communication are focusing on how to unify data management systems of various for different types of data (customer reports, customer complaint letters and key customer information) so as to provide the ability to jointly query and manipulate data from different sources. This project will make a small contribution to this emerging issue.

1.1 Types of Information Storage Management

Business information is expressed in three major kinds of data, i.e., unstructured and semi-structured documents, and structured data in table form. Different kinds of data are, for the most part, managed by different data management systems.

Unstructured and semi-structured data are normally stored as documents and managed inside document repositories by enterprise content management systems, which take the responsibilities of organising, retrieving and analysing document populations. Examples of unstructured documents include customer comments, financial reports, emails, etc. XML and HTML documents are examples of semi-structured documents.

Unstructured documents usually hold only the data itself with no accompanying metadata. Semi-structured documents, on the other hand, hold data together with metadata. An information retrieval system, such as a content search engine, can be used to retrieve either unstructured documents or semi-structured documents. Content search engines allow one to ask for content that meets specific criteria, typically that of containing a given word, and retrieves a list of items that matches those criteria. A web search engine is a kind of content search engine that targets information in public web sites. Semi-structured HTML documents and unstructured text files are the typical format for holding static web content. Although HTML documents are semi-structured, the metadata in HTML documents consists mostly of instructions on how to present the data content in the documents, and do not describe the structure of the data. Therefore, in most cases, content in HTML documents are searched by using web based content search engines, for instance Google has been widely accepted by web users to find

HTML web pages. By way of contrast, XML documents are also semi-structured but unlike HTML documents they hold the data together with structural information in the form of metadata. The computer may be able to understand the meaning of the data content by the meta-data. These type of documents can be queried collectively by using XQuery [(XQ)] for retrieving a fragment of data. However, it can also be searched as textual documents by content search engines.

Structured data residing in databases are normally managed by Database Management Systems (DBMS). The relational model is one data model for structured data. It is widely implemented in Relational Database Management System (RDBMS). Traditionally, data that originates in business transactions or is required for business transactions to be carried out, e.g., customer data or sale data, is stored as relational data (as collected transactional data, e.g. by means of forms).

1.2 Challenges in Distributed Data Management

Faced with growing knowledge management needs, enterprises are increasingly realising the importance of interlinking critical business information maintained distributively over various kinds of data sources. One of the major challenges that companies face is that of extracting value from the combination of data assets they amass. As mentioned previously, customer emails, customer reports and HTML documents are expressed in the form of unstructured and semi-structured documents managed by document repositories, while customer and sale data expressed in the form of structured data are stored and managed by database management system. To manage data in this isolated way can, however, lead to a

situation when a database management system and a document repositories become silos that isolate transactional data from emails and reports, and prevent its combined use to enhance customer experience and improve business performance. The question to be addressed here is how to query in such a way as to obtain added value from the combination of un- and semi-structured documents with structured data. The results would be in the form of a pairwise combination of structured and un-/semi-structured data.

As an example, in the context of a broadband service provider, there is business value in linking emails from customers with demographics, or with the package and options that the customer subscribes to. While an email might contain the customer reference number as a piece of text, no use can be made of that unless the text is analysed and the corresponding string identified and parsed as of such type (e.g. a key) as to allow the application to retrieve, by querying, the corresponding structured information stored in the company database. If this process could be carried out programmatically, the company would be able to mine the emails from customers, e.g, to identify sentiment. Extracting and analysing data about customers, partners and vendors gives a business the upper hand over their competitors. Anecdotal evidence [Sha07] suggests that customer service personnel in call centres would benefit from having knowledge of customer sentiment to address customer needs in the course of a call, for example.

As another example, consider a banking company, in which the structured data comprises of information about customer accounts, customers and banking services problem. The bank provides an Internet Banking Service for its customers to manage their account via the Internet and receives a steady inflow of comments on the Internet Banking Service which are hold in a centralised complaint repository. These complaints are accepted using a range of means, such

as a web-form, email, fax and voice-mail (which is then transcribed). Each such complaint is typically a free-flow narrative text about one or more service problems raised by the Internet Banking Service, and it might contain the problem reference number or the customer ID. Using this limited information, we can discover matches customer records, together with account information, in the customer and customer account databases, such matches can be used to link the given complaint with the customers and their accounts. Occasionally, we may not be able to find the identifiers of customers; instead, it might disclose, by way of context, limited information such as the customer date of birth, the account sort-code, a simple description of the problem encountered, etc. Using some of this information, we may be able to find the customer's identifier in the customer database, and jointly retrieve customer account information from the database if the customer's identifier appears as the foreign key in the customer account relation.

One other challenge that needs to be overcome is how to integrate structured and unstructured data stored across various data sources. Usually, a set of unstructured documents can implicitly contain some data (John Smith) of the same data type (the name of a person) as some structured data stored in databases. By knowing the purpose of the unstructured documents, we can define a common data schema, either semi-structured or relational, then extract data from the unstructured documents to populate a relational or a semi-structured database. For instance, a set of problem-reporting emails sent by customers of the Internet Banking Service provided by a bank may include customer information which can be collected into a relational database. From those emails, we can find a set of newly raised problems that haven't been seen before or data with a new data type, for example the occupation of customers, which would be useful to

be added into existing relational tables that hold customer information as a new occupation attribute. Likewise, we can extract data in unstructured documents into an XML repository and do analytical work using an XML query language, like XQuery. For example, we can collect customer's opinions about a particular service into a sentiment database table which can be used as the key resource in analysing the performance of a business service (more discussion on this example in Chapter 5).

The respective linkage of unstructured and semi-structured data with structured data residing in databases is a knowledge-intensive, time-consuming process, which makes it both prohibitively expensive to delegate to humans and quite challenging to fully automate.

1.3 Project Objectives

The aim of this project is, firstly, to specify a software architecture for correlating unstructured and semi-structured documents managed by content management systems and structured data managed by database management systems. Secondly, a prototype system called the Linking Data System (Lindas, for short) will be designed and implemented as a tool for realising the architecture above. This system has the following major functional requirements:

1. Eliciting linking data from unstructured and semi-structured documents;
2. Populating data in unstructured and semi-structured documents into relational database;

3. Processing and evaluating queries that link unstructured and semi-structured documents with structured data.

Several pieces of software infrastructure and tools can be coordinated to address some aspects of this data correlation, in a systematic way. These are:

1. **Apache Jackrabbit** – A Java implementation of the Java Content Repository API for content management systems.
2. **UIMA SDK** – The Unstructured Information Management Architecture developed by IBM as a software framework for developing text analytical capabilities.
3. **Apache Derby Database** – An in-memory relational database management system.

1.4 Structure of this Dissertation

The remainder of this dissertation is organised according to the following structure:

1. **Chapter 2:** This chapter addresses various background concepts and technologies that the project builds on. The background consists of Content Management Systems, Text Analysis Engines, the IBM Unstructured Information Management Architecture (UIMA) and it also briefly touches on the Derby relational database system.

2. **Chapter 3:** The model, architecture and data flow description of the Lindas system is discussed in this chapter, which ends with a proposed graphical user interface to Lindas.
3. **Chapter 4:** This chapter discusses how text analysis and mining techniques can be applied for the task of extracting linking data from documents.
4. **Chapter 5:** A set of cases, where data correlation can be applied will be addressed in this chapter.
5. **Chapter 6:** This chapter will firstly discuss how the prototype system, Lindas, is implemented, and then moving onto experimental studies of the implemented prototype system.
6. **Conclusion:** This dissertation will be concluded by discussing related work and by proposing some ideas for future work.

1.5 Summary

This chapter has introduced and motivate the project titled 'Linking documents to repositories with structured data in databases'. Enterprise data are expressed in two ways: structured data, stored in database; as well as unstructured and semi-structured documents, stored in document repositories. The two types of data tend to be managed by separate software artefacts. The challenge that companies face in the management of critical business information that are distributed across various data sources is the inter-linkage of unstructured and semi-structured documents with structured data. The goal of this project is to design an architectural model for eliciting linking data and deliver a prototype software

system, called Lindas, which demonstrates and realises the idea of linking documents to structured data. This software system is built upon a set of existing software infrastructures.

Chapter 2

Background

In the previous chapter, the project objectives have been defined, i.e., to specifying a software architecture for linking unstructured and semi-structured documents (such as Email, financial report, complaint, etc.) in the document repositories to structured data (such as customer and sale information) in databases. Traditionally, the former two types are stored in document repositories and managed by a Content Management System (CMSs); while the last type is stored in database and managed by a Database Management System (DBMSs). The linkage of documents to structured data is created by using text analysis technologies which add structured information about documents into a schema for structured data.

This chapter focuses on the background concepts and technologies that will support the design of the linking data software architectures and the prototype system, Lindas. This chapter organises the discussion as follows: Firstly, the concept of a Content Management System, its functionalities and limitations is addressed. This is followed by a discussion of Content Repositories which are the

back-end storage systems underpinning a CMS. Moving forward, a software architecture and framework developed by IBM, namely the UIMA SDK, is introduced, which has been widely used by developers of text analytical capabilities to build Unstructured Information Management (UIM) applications. Finally, Relational Database, namely the Apache Derby Database, used is briefly described.

2.1 Content Management System

A Content Management System (CMS) is a software system that is used for content management. Its main purpose is to manage the full life cycle of data content and metacontent (the content that is used to describe the main data content) by way of a work flow in a repository, with the goal of allowing dynamic access to the content in an effective and efficient fashion. The most widely used application of a CMS is the Web Content Management System (WCMS) a company's website. A less frequently encountered use is as a document content management system in which a company stores, e.g., scanned copies of physical documents.

2.1.1 The Content to Be Managed

Content is informally described as the 'stuff' that is found on a website. There are two main categories: Information, such as text and images, that can be seen on a website when visiting it; and applications that run on the website for serving requests. Developing a CMS that will works no matter the type of content requires the ability to maintain and follow two distinct work-flows at the same time. It is true that the work-flows of both applications and information have many similarities [Fra02] – both create, change, approve, test and deploy – but

the difference are significant. Very different skill sets are required in creating information as opposed to creating an application, and the difference only widens as are continues through to the stage of deployment.

As mentioned in [Fra02], in practice, most CMSs are not usually devoted to manage only application content or even a combination of information and application content. In most cases, CMS software developers focus on information management only and let other software developers build tools, such as source code management systems, to handle the application content.

2.1.2 Components in a CMS

Usually, a CMS consist of a minimum of three elements [Fra02]: the content management application (CMA), the metacontent management application (MMA), and the content delivery application (CDA). Some CMSs have more elements, but all will have these three in some form. The CMA manages the content of the CMS. The MMA, on the other hand, manages the information about the content. Finally, the CDA provides a way of displaying content.

A CMA [Fra02] manages the full life cycle of content components from inception through removal. A CMA will create, maintain, and remove content to and from a content repository. The Content Repository acts as the back-end data store. The content repository can be a database, a set of files, or a combination of both.

In an ideal CMS, metacontent and content should be kept separate, hence the separation in the CMS between CMA and the Metacontent Management Application (MMA) [Fra02]. The MMA is a software application that manages the full life cycle of metacontent. Metacontent can be thought as information

about the content. The process of managing the progress of metacontent closely resembles that of the CMA but with a different focus. Thus, the main reason to keep the content and metacontent separate is that the CMA and the MMA have different work-flows.

The job of content delivery application (CDA) [Fra02] is to retrieve the content out of the CMS content repository and display them, using metacontent, to the website user. The CDA is driven by with the CMA and the MMA. They determine what to display and how it is displayed.

The following list [Fra02] shows the most commonly seen features in commercial and open source CMS products.

- Standard interface for creating, editing, approving, and deploying content
- Content repository
- Version control, tracking, and rollback
- Content search
- Monitoring, analysing, and reporting content

In this project, no content management system is used. The focus is rather on accessing the content repository than underpins the CMS. The next subsection therefore focuses on this component of the CMS architecture.

2.2 Content Repository

Relational and object databases lack many data management features [Som05] required by modern enterprise applications, such as versioning, rich data references, inheritance, or fine-grained security. Java database programmers need to complete those features, or some of those, by themselves. Content repositories extend database with such additional capabilities. The JSR-170, short for Content Repository for Java™ technology API, promises to greatly simplify Java database programming.

Commercial repositories are often implemented on top of a database on a filesystem. Therefore, repositories can be seen as a layer located between a data store, such as an RDBMS, and an application requiring access to persistent data. A repository typically consists of the following components [Ber98]:

- A repository engine that manages the repository's content.
- A repository information model that describes the repository contents.
- A repository API that allows applications to interact with the repository engine and provides for querying or altering the repository information model.
- Optionally, a repository includes an interface to a persistent store, or a persistence manager.

The relationships between these components are illustrated in Figure 2.1.

According to Microsoft Research's Phil Bernstein, a repository engine offers six features above a traditional relational or object database management system

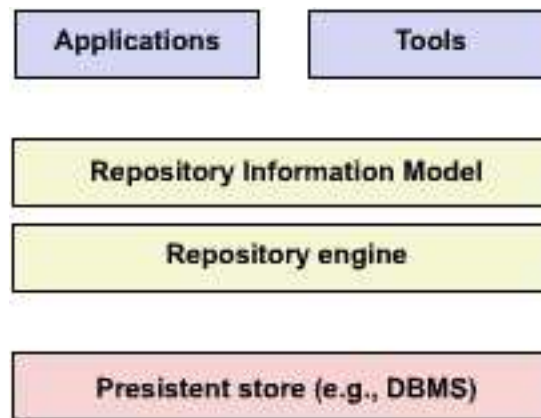


Figure 2.1: Repository Components

[Ber98]: object management, dynamic extensibility, relationship management, notification, version management, and configuration management. Repositories can be more desirable to data intensive application developers than traditional DBMS, if the application can use any of the respective features. Database vendors often ship a repository component as part of their high-end DBMS product (the Microsoft Repository ships with SQL Server).

2.3 Java Content Repository API

The Java Content Repository API (JSR-170) defines a standard to access content repositories from Java code, and promises to greatly simplify Java database programming. This section will review the Java Content Repository API, and in the next section its open-source implementation, Apache Jackrabbit will be discussed.

The reason that JSR-170 [jcp] joined the Java Community Process [Pro] is because experts in Data Management have seen drawbacks in the complexity in Java database programming (discussed in the previous section), and in existing

CMS products. Content management systems from different vendors have been available for quite a long time [Pat06]. All of these vendors ship their own version of a content repository with their CMSs, usually providing a proprietary API for accessing their content repository with their CMS systems. JSR-170 tries to solve this problem by standardising the Java API that should be used for connecting to any content repository. Figure 2.2 reveals an application builds upon the JSR-170 API.

JSR-170 features are divided into three parts [Pie05]. First are the Level 1 features, which provide a read-only view of the repository. Level 2 defines the basic features required for read and write operations. In addition to these two levels, JSR-170 defines five optional features – versioning, locking, SQL search, and transactions – that can be implemented by a content repository. That means a vendor who has a Level 1-compliant repository can decide to implement the SQL search feature but none of the other advanced features. A second vendor might implement a different feature set – for example, a Level 2-compliant repository with versioning.

As shown by Figure 2.2, an application may interact with different the content repositories, namely 1, 2 or 3. We can see that two of these three content repositories, 1 and 3, need JSR-170 drivers for interacting with a JSR-170 application, whereas content repositories 2 is natively JSR-170 compliant. One more thing to be noticed is that an application need not to worry about how actual content is stored. The underlying content repository can be a RDBMS, a filesystem, or a native XML management system.

The JSR-170 Specification [Pie05, jcp, Pro] defined the content repository model as comprising a number of workspaces, which should normally contain

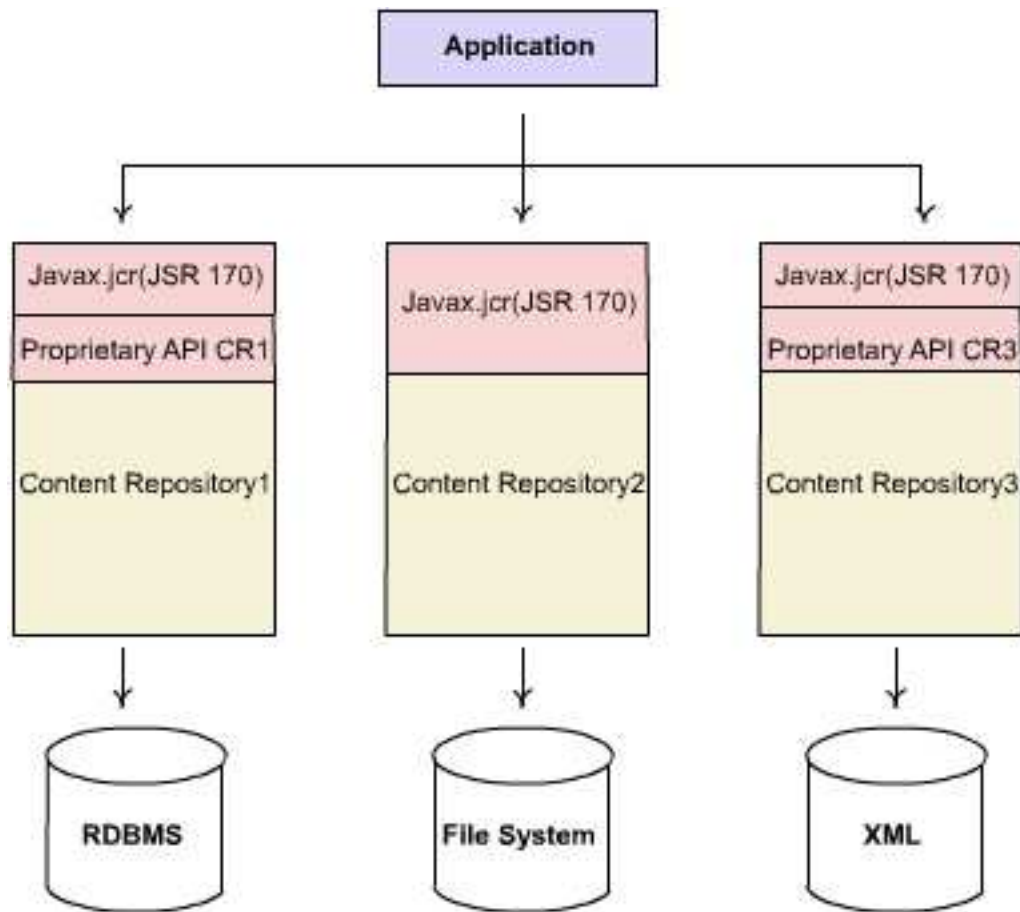


Figure 2.2: Different ways to use the JSR-170 for accessing content repository

similar content. A repository can have one or more workspaces. Each workspace contain a single rooted tree of items. An item is either a node or a property. Properties have one node as a parent and cannot have children; they are leaves of the trees. The actual content in the repository is stored as values of properties.

Appendix 1 presents an example to demonstrate how to store and access documents by using proposed methods in JSR-170 specification.

2.3.1 Apache Jackrabbit

Apache Jackrabbit is Level 2 JSR-170 compliant, and implements all optional feature blocks. Beyond the JSR-170 API, Jackrabbit features numerous extensions and administrative features that are needed to run a repository but are not specified by JSR-170. I have decided to use Apache Jackrabbit as the content repository in the Lindas prototype system.

Jackrabbit consists of an general architecture comprising three layers: a Content Application Layer, an API Layer and a Content Repository Implementation Layer. The composition of those layers is shown in Figure 2.3 .

The purpose of each layer is summarised below [Ove]:

- **Content Applications:** Content Applications interact through the JSR-170 API with the Content Repository Implementation. There are numerous applications that are available for JSR-170 repositories.
- **Content Repository API:** This contains two major sections: The Content Repository API defined by JSR-170; and a number of features of a

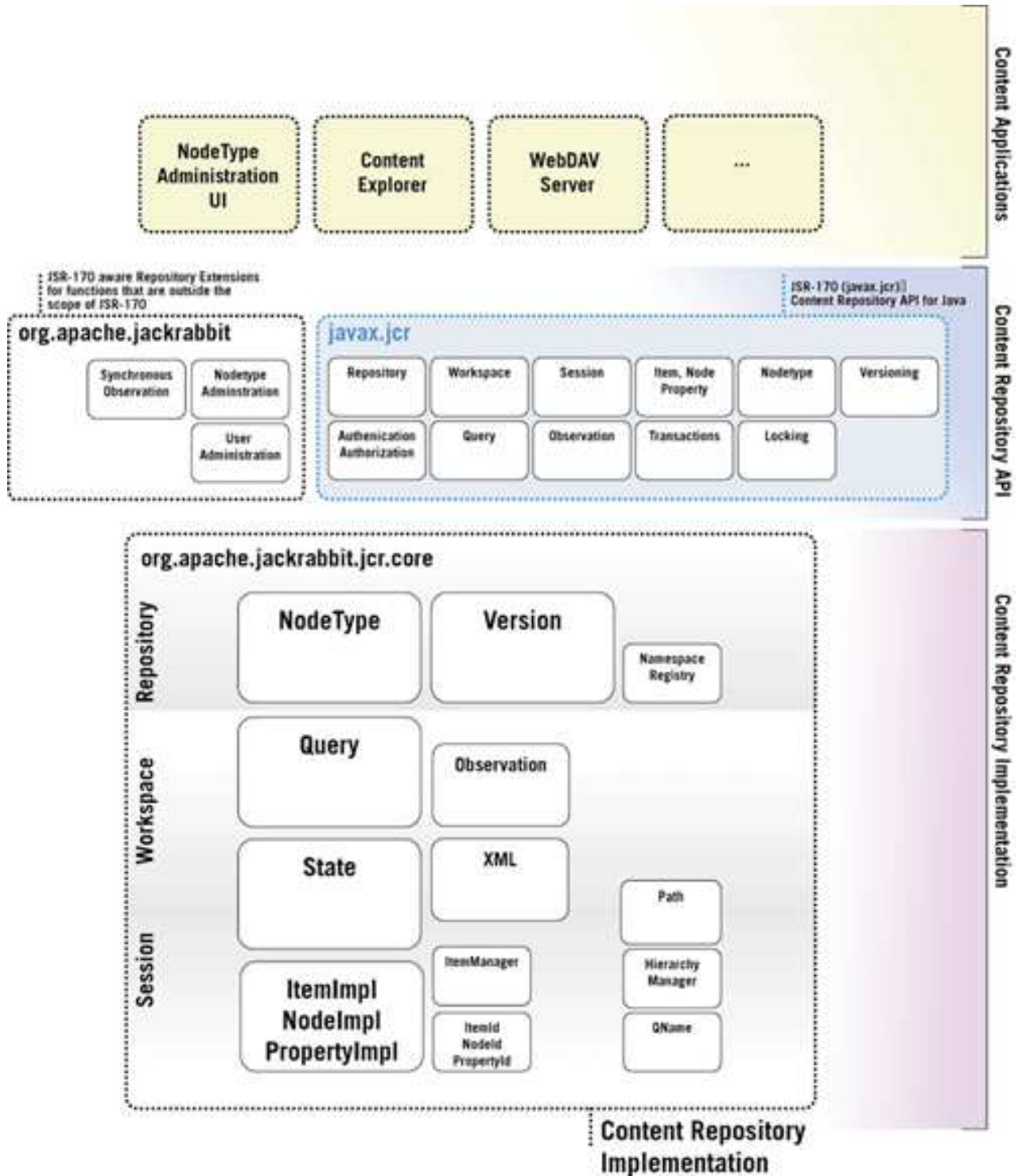


Figure 2.3: Apache Jackrabbit Architectural Layers (source: [Ove])

content repository, that have been removed from the JSR-170 specification since they are difficult to implement on existing non-Java-based content repositories.

- **Content Repository Implementation:** This portion of the architecture chart reflects the major building blocks of the Jackrabbit content repository implementation.

2.4 IBM Unstructured Information Management Architecture

The IBM Unstructured Information Management Architecture (UIMA) [uim06] specifies component interfaces, data representation, design patterns and development roles for [FL04] developing and deploying multi-model analysis capabilities for unstructured information management (UIM) applications. The UIMA framework [uim06] provides a run-time environment in which developers can plug in their UIMA component implementations and with which they can build and deploy UIM applications. The framework is not specific to any IDE or platform. The UIMA also provides a Software Development Kit including an all-Java implementation of the UIMA framework for the implementation, description, composition and deployment of UIMA components and applications. The development of LINDAS was done under the Eclipse Java project development environment which can include a set of tools and utilities for using UIMA.

2.4.1 Text Analysis Technology

As described in [FL04], structured information can be characterised as information whose intended meaning is unambiguous and explicitly represented in the structure or format of the data. The most representative example of a tool for managing structured information is a relational database management system. On the other hand, unstructured information can be defined as information whose intended meaning is only loosely implied by its form and therefore requires interpretation in order to approximate its intended meaning. Examples are email messages, speech, audio, images and video.

One reason for deriving implied meaning from unstructured information is that it is estimated that 80% of all corporate information is unstructured [Moo02]. An even more compelling reason is the rapid growth of the Web and the perceived value of its unstructured information to applications that range from e-commerce and life science applications to business and national intelligence. The high-value content in these increasing amounts of unstructured information is normally hidden by in lots of noise. Searching for fragments of desired content or doing sophisticated data mining over unstructured information sources present new challenges.

An unstructured information management (UIM) application may be generally defined as software that can systematically analyse large volumes of unstructured information (text, audio, video, images, etc.) to discover, organise and deliver relevant knowledge to the client or application end-user. An example [uim06, FL04] is an application that processes millions of medical abstracts to discover critical drug interactions. Another example is an application that processes tens of millions of documents to discover key evidence indicating probable

competitive threats.

The simplest way to describe text analysis on unstructured documents is the process of adding structural information into the documents so that the hidden meaning in unstructured content can be explicitly interpreted. In order to achieve this goal, the unstructured data must be analysed to interpret, detect and locate concepts of interest that are explicitly tagged or annotated in the original artefact, for example, named entities like person, organisations, locations, facilities, products etc. More challenging analytic may detect things like opinions, sentiments, complaints, threats or facts. The list of concepts that are important for applications to detect and find in unstructured resource is large and often domain specific. Therefore, specialised component analytic must be combined and integrated to do the job.

In analysing unstructured content, UIM applications make use of a variety of analysis technologies including: Statistical and rule-based Natural Language Processing (NLP), Information Retrieval (IR), Machine learning, Ontologies, Automated reasoning and, Knowledge Sources. These technologies are developed independently by highly specialised scientists and engineers using different techniques, interfaces and platforms.

2.4.2 Key Components in UIMA

The building blocks of the UIMA architecture are called **Analysis Engines** (AEs). AEs are composed to analyse a **document** and infer and record descriptive attributes about the document as a whole, and/or about regions therein.

This descriptive information produced by AEs is referred to generally as **analysis results**. Analysis results typically represent meta-data about the document

content. Therefore, AEs can, alternatively, be described as software agents that automatically discover and record meta-data about original content.

UIMA is capable of analysing different modalities including text, audio and video. The Lindas system has an analysis engine that only focuses on text documents.

UIMA provides a basic component type intended to house the core analysis algorithms running inside an AE. Instances of this component are called **Annotators**. The primary concern of the developer of an analysis algorithm therefore is the development of annotators. The UIMA framework provides the necessary methods for taking annotators and creating analysis engines. (Appendix 2 discusses the path in developing an annotator and analysis engines, through a detailed example of searching customer's identifiers in unstructured documents, using the UIMA framework). The simplest AE contains exactly one annotator at its core. Complex AEs may contain a collection of AEs, each potentially containing within them other AEs.

How annotators represent and share their results is an important part of the UIMA architecture. UIMA defines a **Common Analysis Structure (CAS)** precisely for these purposes. The CAS is an object-based data structure that allows the representation of objects, properties and values. The CAS logically contains the document being analysed. Analysis developers share and record their analysis results in terms of an object that complies with the CAS.

Annotation types are used to annotate or label regions of an artifacts. Common artifacts are text documents, but they can be other things, such as audio streams. The annotation type for text includes two features, namely begin and end. Values of these features represent offsets in the artifact and delimit a span.

Any particular annotation object identifies the span it annotates with the begin and end features.

For every component specified in UIMA there are two parts required for its implementation: The **declarative part** and the **code part**. The declarative part contains metadata describing the component, its identity, structure and behavior and is called the **Component Descriptor**. Component descriptors are represented in XML. The code part implements the algorithm which may be a program in Java.

2.4.3 Text Analysis Levels

There are two levels in which text analysis can be applied to documents [Moo02]: one document (**document-level**); or a collection of documents (**collection-level**). When document-level analysis is applied, the AE takes the document as an input and produce its output in the form of meta-data describing fragments of the original document. The meta-data represents structural information that refers to the document as a whole or to one of its regions. Examples of document-level analyses including language detection (over semi-structured HTML document or pure by textual document), detagger (remove all the tags that appear in an semi-structured document thereby deriving from it an unstructured document), name-entity detection (identify recognisable entities or relationships that appear in an unstructured document), and tokenization. In each cases, the analysis component examines the document and produces associated meta-data as a result of its analysis.

An AE may be implemented by composing a number of more primitive components, as illustrated in Figure 2.4. In this figure, a number of components

are assembled in series in order to implement a “government official detector” function. The output of each stage consists of a document with the result of the analysis. For example, the output of the detagger component consists of the document with HTML tags identified and context extracted, and so on.

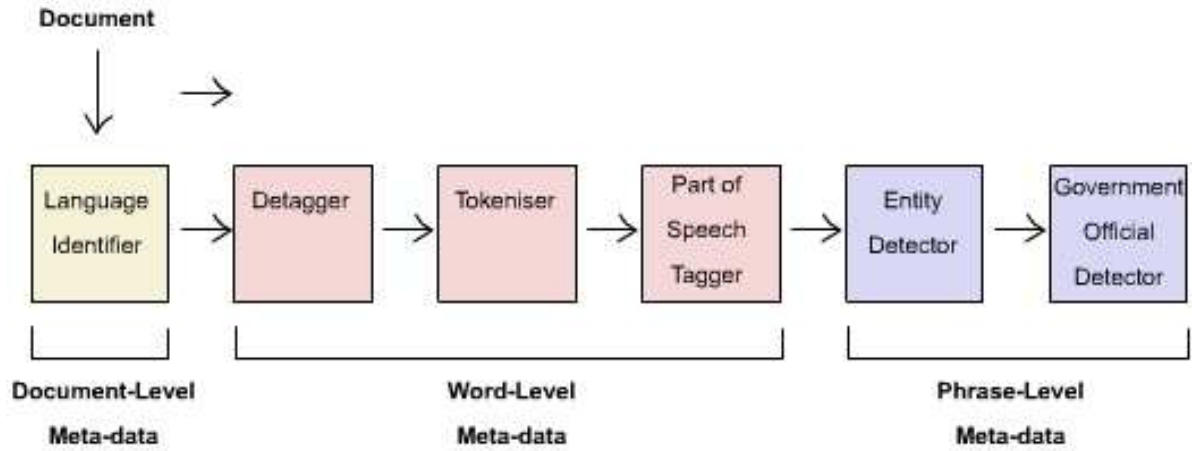


Figure 2.4: Document-Level Text Analysis Engine

In collection-level analysis, the text analysis engine processes an entire collection of documents. Whereas each document may undergo document-level analysis as part of collection processing, the result of collection-level analysis represent inferences made over the entire collection. Example of collection-level results are: glossaries, dictionaries, databases and search indexes. The collection-level text analysis is the approach used in the Lindas system.

2.5 Apache Derby

A brief discussion on the Apache Derby Database is provided, since it has been selected as the structured data management system to be used in Lindas.

Apache Derby is an in-memory relational database developed as an Apache DB subproject implemented entirely in Java. It only needs a small amount of simple code to be easily embedded into any Java application. It makes use of JDBC technology to build connection and access to databases, therefore applications use the Apache Derby Database can conveniently switch to other relational database management systems (RDBMSs), such as MySQL and Oracle, without the need to spend a great amount of work on code changing. Furthermore, Derby uses conventional SQL to achieve the tasks of querying and manipulating a database as other RDBMSs do.

2.6 Summary

This chapter has touched on a set of background concepts and technologies on which the project of Linking Documents in Repositories to Structured Data in Database builds. Three main fields have been introduced: Content Management Systems (CMS); Text Analysis on Unstructured Information; and the Apache Derby Database Management System was briefly introduced as well.

The following three chapters will mainly focus on how to put those background concepts and technologies into practice for realising the idea of linking unstructured, semi-structured and structured information.

Chapter 3

Linking Data Architectures

Having introduced various background concepts and technologies, in this chapter, we discuss the system architecture of Lindas. From an architectural point of view, most of the bottom level components of Lindas, are mapped to existing software introduced in the previous chapter, for example: the Apache Jackrabbit Content Repository; the IBM UIMA Unstructured Information Management (UIM) application development SDK and the RDBMS Apache Derby. The most important point to consider when designing the Lindas architecture is to coordinate the contribution of existing software components to the goal of this project. One approach to produce a comprehensive architecture is to take a 'top-down' approach, which starts by eliciting the business and functional objectives, and then defining a general system model and data-flow model (because Lindas is a data- and query-intensive system). Finally, the system architecture will be designed based on the system and data model.

An Example Use Case

Let us firstly take a use case as an example to understand what functional requirements the Lindas system should have. Consider a banking company that provides its customers with an Internet Banking (IB) service for managing their bank account and money through a web interface. The company has a steady flow of complaints sent through the IB service by means of which customers report any problem they meet while using the service. These complaints take the form of emails which are stored in a centralised content repository. An example of an email used for this purpose is shown in Figure 3.1a. When the user produces an email like this, they may provide information (for example, an account number, a username, etc.) in addition to the substantive content of the email. Albeit limited, such information, can be used to discover potential matches with structured customer data, e.g., account information, held in databases. If so, then the unstructured data that conveys the complaint can be linked with the matching structured data about the customer in a relational database. The customer service manager may subsequently reply to the users (an example message is shown in Figure 3.1b) and ask them to call the IB customer service in person to solve the problem encountered. This reply email message, as well as the email originally sent by the IB user to report the problem, can be stored together in the content repository, and indexed by the username of the customer. Furthermore, the problem-reporting email can be text-analysed for later use. When the user communicates on the phone with the call centre staff to negotiate ways to solve the encountered problem, the staff can use the customer's IB service username to retrieve the text-analysed emails shown in Figure 3.1a and 3.1b. From the annotated meta-content produced as the result of the text analysis function, staff can find collated information about the users, their account and the problem they

have encountered.

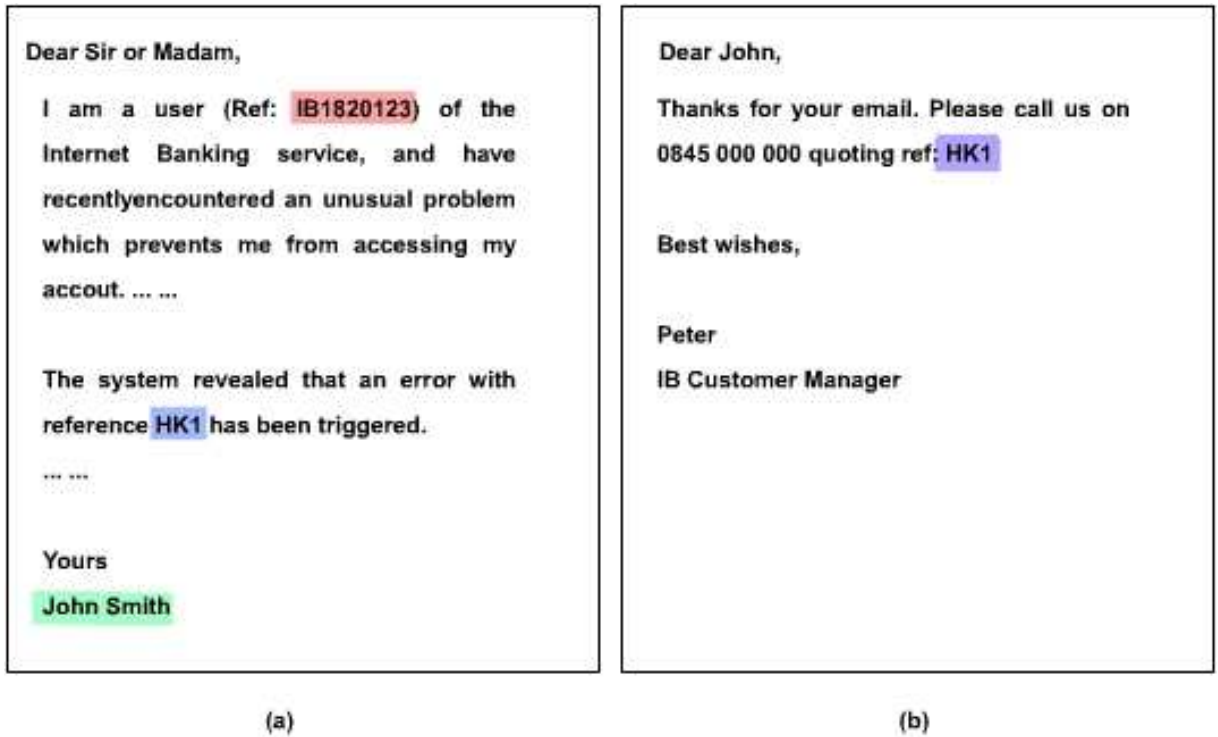


Figure 3.1: Email documents include annotation

As we learnt from this scenario, the Lindas system should be capable of:

1. Eliciting linking data from unstructured and semi-structured documents by using text analysing techniques;
2. Correlating documents with structured data on the basis of elicited linking data;
3. Populating data obtained by analysing unstructured and semi-structured documents in the form of structured data appropriately into a database;
4. Allowing annotators to be defined for use by the text analysis engine when mining desired entities in unstructured documents;

5. Provide users with graphical user interface to drive the system functionality.

3.1 User Interfaces

After eliciting the functional requirements of Lindas, we describe the Graphical User Interface (GUI) proposed for users to drive the functionality provided by system components. This will later help with the task of defining the system and data-flow models, as well as the eventual system architecture of Lindas, Figure 3.2 illustrates the GUI of Lindas.

As it is shown in Figure 3.2, the GUI contains two major sections: the Correlation Query section and the Annotator Control section. Note that an entity is a unit of annotation (see Chapter 4: Eliciting Linking Data for more details). The correlation query section allows the users to input the value of a specific entity type (e.g., an Email, and ID) which Lindas uses to retrieve the set of documents that contain the linking data in question, based on a content search, as well as structured data, in this case relational database entities, based on a SQL query.

In the Annotation Control section of the interface, the user can perform the following tasks:

1. Adding one or more annotators by providing the name of the target annotation entity, the type of the annotation entity and the regular expression to be used by the text search engine;
2. Viewing existing annotator informations;

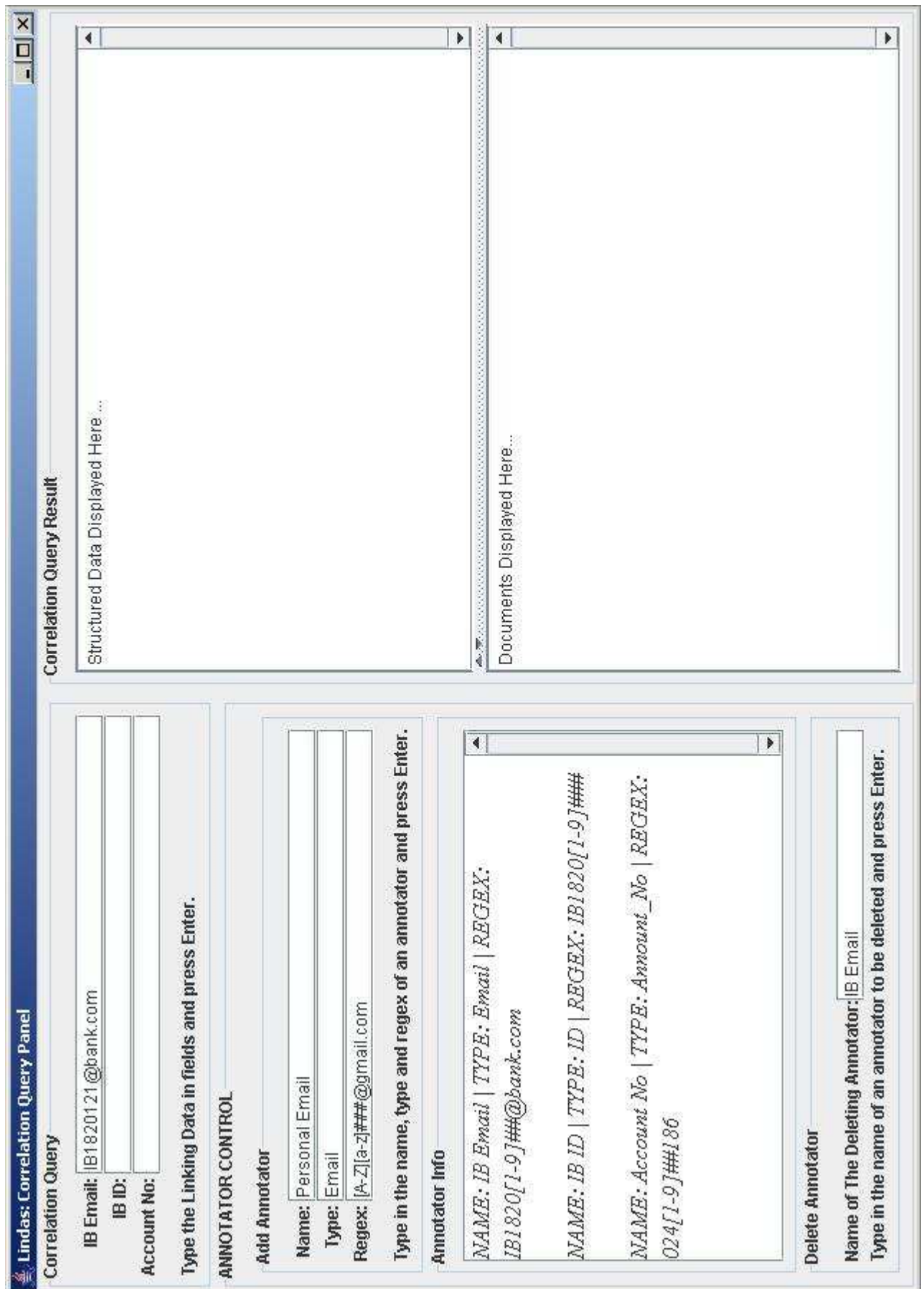


Figure 3.2: Correlation Query GUI in Lindas

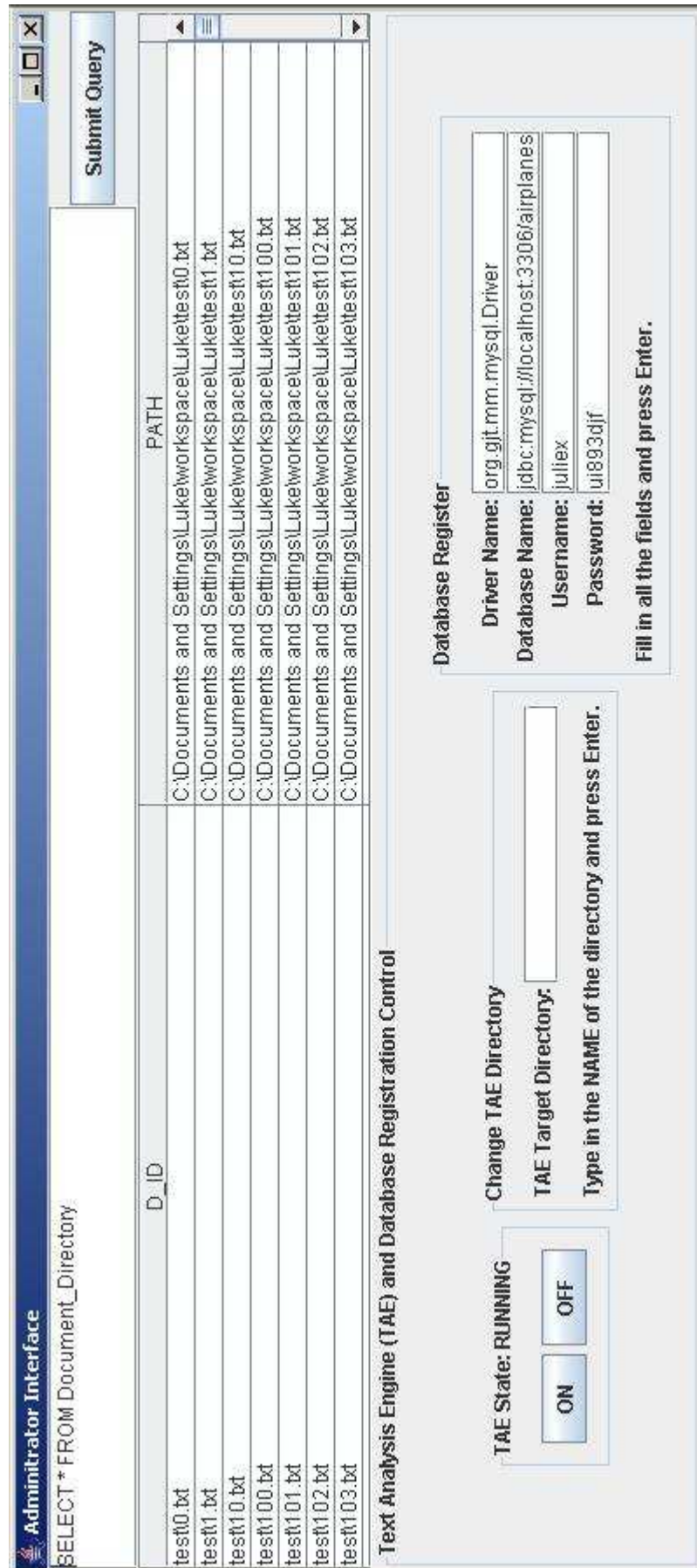


Figure 3.3: Administrator Interface in Lindas

3. Deleting an annotator using its name.

Lindas also provides a correlation overhead monitor interface (discussed in Chapter 5) that allows a Lindas administrator to view and query information (e.g., elicited linking data and annotated documents) generated via document analysis which can be used to assist the correlation query evaluation process. Furthermore, there is also an annotator control panel for administrators to pause or start an annotation task targeting a directory of documents using a specified path. The reason for providing this interface function is that annotation is a background process. In Lindas, a batch of documents are continuously processed with a 5-second break at the end of each round, therefore the administrators must be allowed to terminate this repetitive process or to adjust the annotation target with this interface function. In addition, a database registration interface is provided that allows the user to flexibly switch to other databases which also have the Java DataBase Connection capability installed. All these interfaces are illustrated in Figure 3.3.

3.2 Linking Data System Model

The Lindas system model, illustrated in Figure 3.4, consists of four main components: a Correlator, a document repository, a database and a text analysis engine. The Correlator takes responsibility for populating unstructured data into the database, or solving a Correlation Query by retrieving data from both the document repository and the database. Unstructured and semi-structured documents are stored and managed in the document repository, while the database holds structured data. Finally, the text analysis engine processes documents in

the document repository in order to generate metacontent. The result of the annotation (the meta-data) is stored and managed by the document repository.

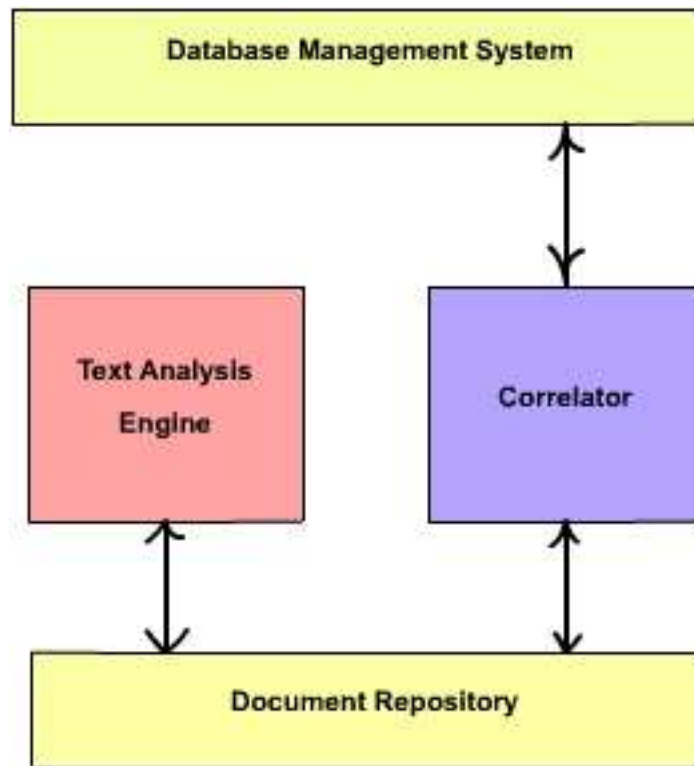


Figure 3.4: Linking Data System Model

3.3 Data Flow Model

Because Lindas is a data-intensive system purposed for data management, it is used to clearly identify the flows of data (un-/semi-structured documents and structured data) that in Lindas. Figure 3.5 illustrates this.

As is shown in Figure 3.5, newly received unstructured and semi-structured documents are continuously added into the document repository. The (original)

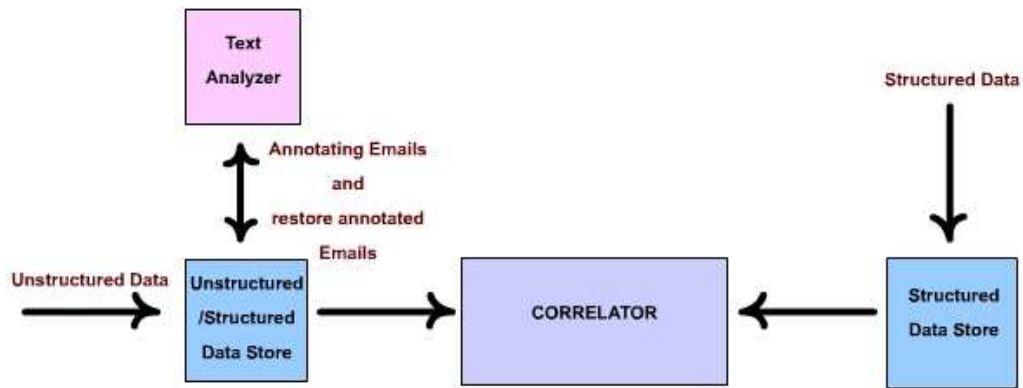


Figure 3.5: Data-flow Model

documents held in the document repository are retrieved by the text analysis engine, and the generated metacontent from the annotation process (represented in document format) is stored separately from the original document in the document repository. On opposite side of the flow, new structured data is also continuously added into the database. With regard to a correlation query issued by a user, the correlator retrieves documents and structured data and assembles them into a unit as an element in the output correlation query result.

3.4 Linking Data Architecture

After defining the GUI, data-flow and system model of Lindas, we can now derive the system architecture (Figure 3.6). As illustrated in the general system model, documents are stored in a document repository. This document repository is managed by Apache Jackrabbit which is the implementation of the JSR-170 API defined for building Java content management systems. All the invocations of the Jackrabbit functions for the purpose of storing, accessing, deleting and manipulating the unstructured content have to be done through the JSR-170 API.

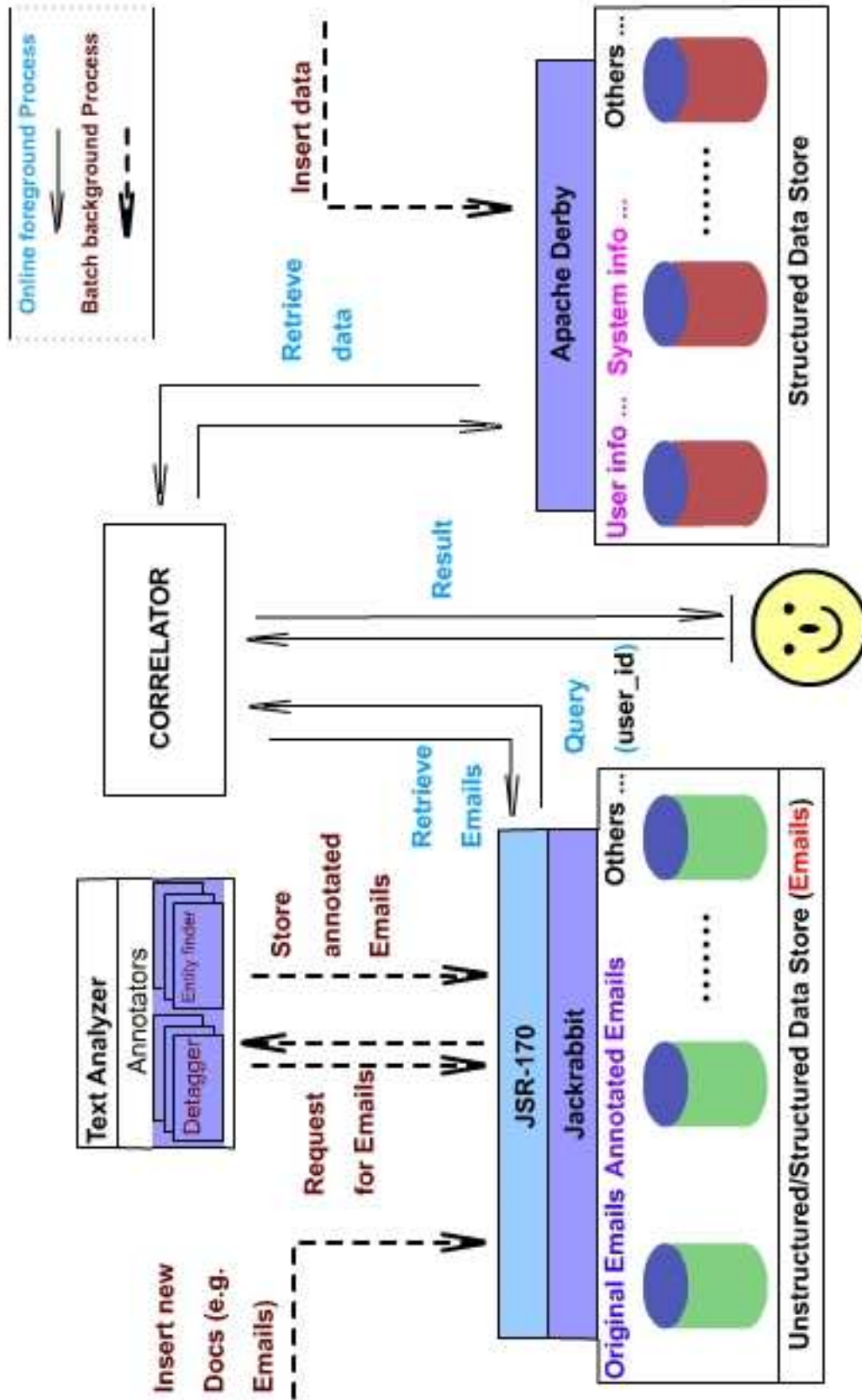


Figure 3.6: Linking Data System Architecture

The text analysis engine, contains a set of user-defined annotators and continuously works on the documents stored in the document repository. The text analysis engine retrieves the documents by invoking Jackrabbit functions, and this is also the mechanism used for storing back the annotated metacontent. The regular expressions used by the annotators for searching linking data in documents are provided by the users of Lindas. The process of creating annotators is done automatically by the analysis engine given the name of the annotator, the type of linking data to search for, and the required regular expressions to drive the search.

The Apache Derby Database is used as the default management and storage system for structured data, unless the user specifies other database management systems, e.g. MySQL, to be used for such purpose.

3.5 Summary

This chapter defined the Lindas GUI, data-flow model, and system model, from which the system architecture derived which are the blueprint for building the Linking Data System. The next two chapters will move onto discussing the Text Analysis Engine and Correlator components of the system architecture.

Chapter 4

Linking Data Extraction

The previous chapter has specified the system architecture of Lindas and various components that realise the idea of linking documents in repositories to structured data in database. This chapter shall focus mainly on the text analysis component of the system architecture to further understand its purpose, i.e., why text analysis can help us build bridges to connect unstructured and semi-structured documents to structured data residing in, e.g., relational databases.

Unstructured and semi-structured data constitute the majority of corporate information generated from everyday business activities. Even today, only a fraction of such information is used by most companies, the rest are buried inside files and, more rarely but increasingly, in document repositories. According to [Sha07], more than 85% of the data within most organisations falls outside of the traditional unstructured data category. Most of this data appears in documents (e.g., in HTML websites, in customer reports and customer comments), still remains unanalysed and can not fit smoothly into typical data creation, query and manipulation processes, e.g. those developed for relational database management

system (RDBMS). A key process to gain corporate insight and to obtain knowledge from data that resides in document repositories is to perform text analysis.

Consider the Internet Banking (IB) service introduced in the previous chapter. The bank that provides this service has a steady inflow of customer support requests into a centralised customer request repository. These customer requests are accepted using means, such as a web-form (e.g., Figure 4.1) or email (e.g., Figure 4.2). Each of these requests may disclose limited information such as the service reference number or information about the problem encountered, such as a reference code. By using this embedded information, we can query a customer database to retrieve related (linkable) customer information.

Section 4.1 considers how to use document annotation to extract linking data in unstructured and semi-structured document, and guidelines for the design of annotation functions inside text analysis engines. Then, Section 4.2 briefly touches on the topic of annotator descriptors. The last section will concentrate on discussing the format in which the result of document annotation can be represented. This formatted annotation result is an important input for the correlator component in the system architecture.

4.1 Document Annotation

Linking data is defined to be an entity or object that can be used as a reference across, on the one hand, either unstructured or semi-structured documents, and, on the other, tuples in a database. Tables in a relational database can be called entity sets, holding tuples of information that belong and describe a certain type of entity or object. For example, for a relation might store customer information

according to the schema: `customer(cid, c_name, acco_no, dob, email)`; another might have a unary schema `email(address)`. The former relation holds information about a set of ‘customer’ entities, each described by value in the columns row of the table carry information about the `cid`(for customer id), `c_name`(for customer name), `acco_no` (for account number), `dob` (for date of birth) and `email`(for email address). The second relation stores a set of email entities with the email address as the single attribute describing each email entity.

As represented by the underline mark in each relational schema, the `cid` and the `address` attributes are the primary keys for the respective tables. As represented by the *italic* font, both `acco_no` and `email` are foreign key attributes that point to other tables in which values in the same domain appear as a primary key.

Data about the customer and email entities mentioned above can also frequently appear in unstructured and semi-structured documents. As shown in Figure 4.2, the web-form asks customers to first fill in some required information (e.g., Internet Banking ID, email address) before writing out their comments. We can apply text analysis techniques to analyse this HTML page and annotate (e.g., extract) those entities (i.e., linking data) in unstructured or semi-structured documents. Moreover, the email message sent from the customer, shown in Figure 4.1a also disclosed customer information, such as the Internet Banking ID (IB1820191), which can be picked up by text analysis. By giving the ID, email and account numbers, we can, furthermore, retrieve structured information about the customer and his account together with the email document, and possibly chain it to the web form as well.

Technical Support

This form is to be used for Internet Banking technical support only. If you have any suggestions or general feedback relating to our Internet Banking service please use our [Feedback form](#). For advice on how to use our Internet Banking service or specific transaction enquiries please telephone our Helpdesk on 0845 600 2290 (overseas +44 1226 261226).

Please provide the following information :

We can then contact you to help solve your particular problem

▶ Name:

▶ Enter a single valid e-mail address:

▶ Preferred Telephone Contact Number (optional):

▶ Internet Banking ID:

Please help us identify your problem :

▶ Operating System Details:

▶ Browser Details :

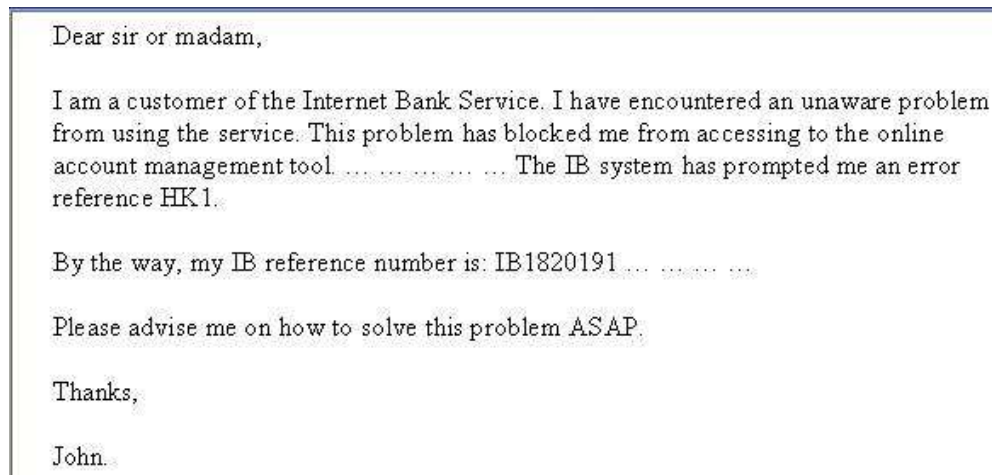
▶ Name of Internet Service Provider:

▶ Modem Speed:

Nature of problem :

Please enter details below:

Figure 4.1: Customer support requests in web-form

A screenshot of a web-form containing a customer support request. The text is as follows:

Dear sir or madam,

I am a customer of the Internet Bank Service. I have encountered an unaware problem from using the service. This problem has blocked me from accessing to the online account management tool The IB system has prompted me an error reference HK1.

By the way, my IB reference number is: IB1820191

Please advise me on how to solve this problem ASAP.

Thanks,

John.

Figure 4.2: Customer support requests in web-form

4.1.1 Annotation Function Design

As discussed in Chapter 2, several Analysis Engines (AEs) can be composed to analyse a document inferring and recording descriptive attributes about it, as a whole and/or about regions therein. A simple (or primitive) UIMA AE contains a single annotator working on a specific annotation task, such as de-tagging. However, a complex AEs may be designed to form a chain of AEs organised into a workflow. These more complex analysis engines are called Aggregate Analysis Engines. Annotators tend to perform fairly granular functions, e.g., language detection, tokenisation or named-entity detection.

Advanced analysis, however, may involve an orchestration of many of these primitive functions. An AE that performs named-entity detection, e.g., to retrieve an email address in a fragment of text, may comprise a pipeline of annotators, as is the case in Lindas, such as document type detector, de-tagger, entity-type-1 detector (targeting email addresses), entity-type-2 detector (targeting IDs), etc. Figure 4.3 depict an example aggregate AE pipeline. Each step in the pipeline

is required by the subsequent analysis.

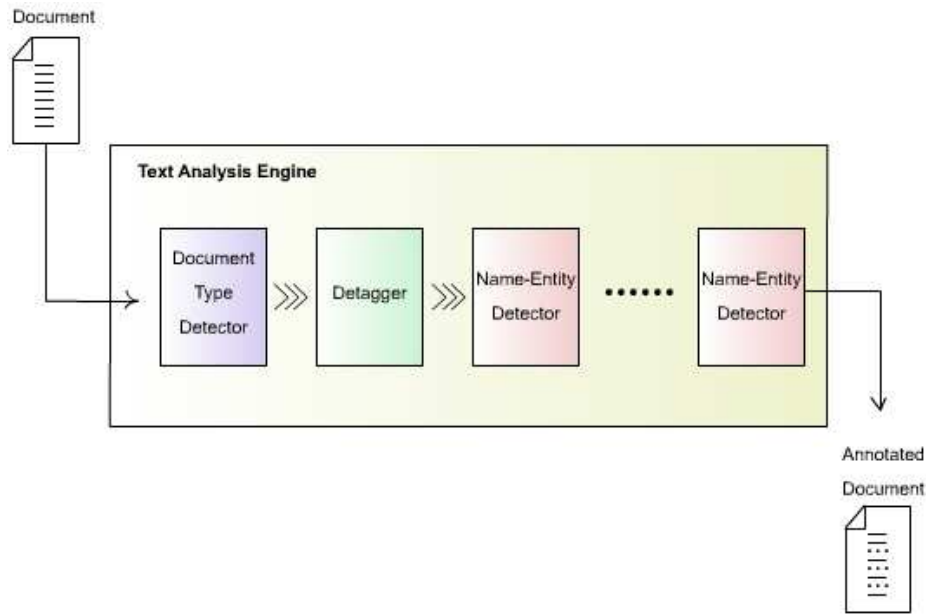


Figure 4.3: Aggregated Text Annotation Engine (Annotator Pipeline)

In Figure 4.3, the document type detector has the responsibility of identifying whether a document is unstructured or semi-structured. If the document is semi-structured, e.g., an XML or HTML document, we firstly remove all the tagging information, leaving only the data content to be treated as an unstructured document, and then use a set of name-entity detectors to identify specific entities that appear in the document. Lindas allows the user of the system to specify the entity that they want the Lindas system to annotate by specifying the regular expression that selects the entity value (e.g., a name, an email address, etc.). Lindas can then automatically create and attach a new AE to the chain.

Users of the AE only need to know its name, its published input requirements and output types (will be discussed in the next section), not how it is constructed internally. To develop aggregate AEs in the IBM UIMA framework, these must be

declared in the aggregate AE descriptor. Aggregate AE's descriptors declare the components they contain and a flow specification. The flow specification defines the order in which the internal component AEs should be run. The internal AEs specified in an aggregate AE are also called the delegate analysis engines.

The aggregate analysis engine descriptor is given to the UIMA framework which uses the information to run all delegate AEs, ensuring that each one gets access to the Common Analysis Structure (CAS) in the precise sequence determined by the flow specification. The UIMA framework is required to handle different deployments where the delegate engines, for example, are tightly-coupled (running in the same process) or loosely-coupled (running in separate processes or even on different machines).

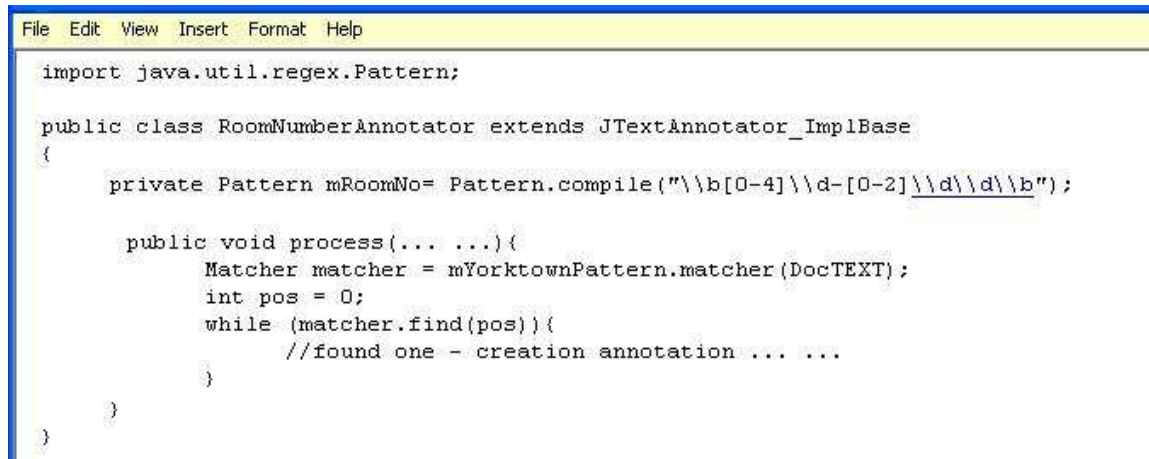
4.1.2 Annotator Generation

For the purpose of extracting linking data from documents in repositories, a single primitive annotator can be built through the following procedural steps:

1. **Identify the target (input) documents:** e.g., a batch of emails or web-forms in HTML format that are to be sent to the annotator as input to be processed.
2. **Identify the targeted entities or objects to search for:** We might be interested in annotating some of the attributes in a relational schema, e.g., `customer(cid, email, ... room_no)`.
3. **Generate the corresponding regular expressions:** For example, the Java package `java.util.regex` provides a language for defining regular expressions so that text fragments can be processed based on them:

- (a) room_no (Pattern: ##-[0-2]##): 20-001, 31-206;
- (b) IB_ID (Pattern: IB1820[0-8]##): IB1820491.

4. Implement the annotation algorithm (an example is shown in Appendix 2).



```
File Edit View Insert Format Help
import java.util.regex.Pattern;

public class RoomNumberAnnotator extends JTextAnnotator_ImplBase
{
    private Pattern mRoomNo= Pattern.compile("\\b[0-4]\\d-[0-2]\\d\\d\\b");

    public void process(... ..){
        Matcher matcher = mYorktownPattern.matcher(DocTEXT);
        int pos = 0;
        while (matcher.find(pos)){
            //found one - creation annotation ... ..
        }
    }
}
```

Figure 4.4: An annotation algorithm

The first step in writing the Java code for an annotation algorithm (as shown in Table 4.1) that uses the Java Regular Expression package is to import the package: `java.util.regex.Pattern`. Then, within the class definition, we firstly give the regular expression as a `Pattern` object. Secondly, we use the `process` method to find words that can be matched to the provided regular expression.

We can design, implement and test each individual primitive annotators first, then compose them together to form an annotator chain. A detailed walkthrough on how to develop a text analysis application by using UIMA is discussed in Appendix 2.

4.2 Annotator Descriptor

As mentioned in Chapter 2, Annotators and AEs are two of the basic building blocks in the UIMA framework. Developers implement them to build and compose analysis capabilities and ultimately applications. For every annotator component, there are two parts required for its implementation: one is the declarative part and the second one is the code part. The UIMA architecture requires that descriptive information about an annotator be represented in an XML file and provided along with the compiled annotator class file(s) to the UIMA framework at run time. The code part implements the algorithm, typically a program in Java.

4.3 Annotation Output

The purpose of document annotation in Lindas is to extract linking data from unstructured and semi-structured documents in document repositories that are valuable in correlating documents to structured data residing in databases. In order to realise this goal, apart from designing the text annotation function (clearly, the most crucial task) we also need to think about how the annotation results are shared and represented in order to facilitate future use. Although that we can visualize the annotation in a graphical interface as color-highlighted regions in a text span, this is only human understandable. We need to consider alternative, machine-readable format, and UIMA uses XML (discussed in section 4.3.1.), to represent annotation results.

4.3.1 Text Span

Text analysis results may include statements about the content of a document. For example, the following is an assertion about the topic of a document:

(1) The Topic of Email in Figure 4.1(b) is ‘‘Error HK1 and User IB1820191’’

Analysis results may include statements describing smaller regions than an entire document. We use the term **span** to refer to a sequence of characters in a text document. Consider that the email document in Figure 4.1 (b) contains a span, ‘‘IB1820191’’ starting at character position 101. An AE that can detect Internet Banking (IB) ID in text documents may represent the following statement as an analysis result:

(2) The span from position 101 to 112 in the email document denote an IB ID

In both statements (1) and (2) above there is a special pre-defined term or what is called in UIMA a Type. They are ID, error reference, etc.

4.3.2 Annotation Result Representation

UIMA defines a Common Analysis Structure (CAS) [uim06] precisely for the purpose of sharing and representing annotation results.

The CAS is an object-based data structure that allows the representation of objects properties and values. Object types may be related to each other in a single-inheritance hierarchy. The CAS logically contains the document being

analysed. The UIMA framework includes an implementation and interfaces to the CAS.

A CAS that logically contains statement (2) above would include objects of the (IB) ID type. For each ID found in the body of a document, the AE would create an ID object in the CAS and link it to the span of text where the ID was mentioned in the document.

The UIMA can convert the annotation result represented in CAS to an XML document which contains the original text as the text content of an XML element together with the annotated span in terms of the character position in the analysed text fragment. Figure 4.5 shows an example of the annotation results being represented in an XML document.

4.4 Summary

This chapter has discussed the text analysis component in the Lindas system architecture. Text analysis is a key process to gain corporate insight and to obtain knowledge from data that resides in document. In this project, it is applied to extract linking data from unstructured and detagged semi-structured documents so as to facilitate the future use of that data in correlating documents to structured information held in databases. The design of the annotation function involves identifying how to assign responsibilities to each annotator in the annotation chain. The overall process is guided by database schemas, insofar as linking data is that which appears both in documents and in database tuples.

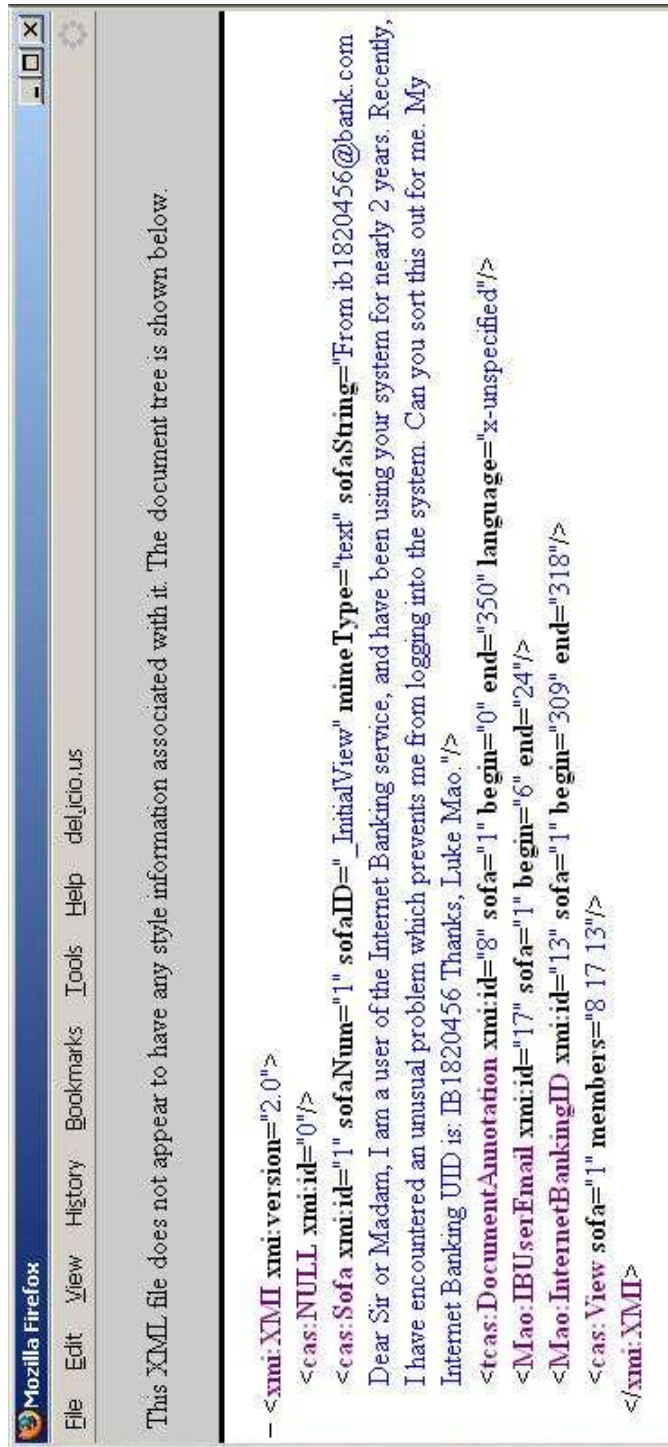


Figure 4.5: Document annotation result in XML

Chapter 5

Data Correlation Approach

Having discussed the Text Analysis component of the Linking Data System Architecture in the previous chapter, this chapter moves on to discuss the Correlator component, which does the job of accessing document repositories and databases to solve correlation queries. A correlation query asks for a result that pairs unstructured or semi-structured documents with structured data. This can be done after the outcome of analysis of unstructured and semi-structured data is made as part of the database in which structured data resides. Unstructured and semi-structured documents are continuously added into document repositories and continuously analysed by the Analysis Engine described in the previous chapter. Similarly, on the structured side, information in the form of relational data is continuously populated into relational databases.

Section 5.1 introduces the notion of correlation query and the evaluation strategy for solving correlation queries implemented in Lindas. Section 5.2 describes how analyses unstructured data are used to populate a database. Finally, Section 5.3 briefly discuss how to perform sentiment analysis by using advanced text analysis techniques and the correlation query function.

5.1 Correlation Query

Supporting correlation queries requires supporting content search over a set of unstructured and semi-structured documents, as well as database query evaluation for retrieving structured data residing in databases. This split into two tasks can be visualised in Figure 5.1. One or more items of linking data are given as the input condition for the correlation query. Based on such linking data, we can narrow down the size of query result. For the content search part of the correlation query, only the documents that contain the linking data in question need to be retrieved. For the database query, a table of tuples, which have one or more attributes that match the linking data, are returned. The final result of the correlation query is a pair combining documents with corresponding structured data that share the linking data items given.

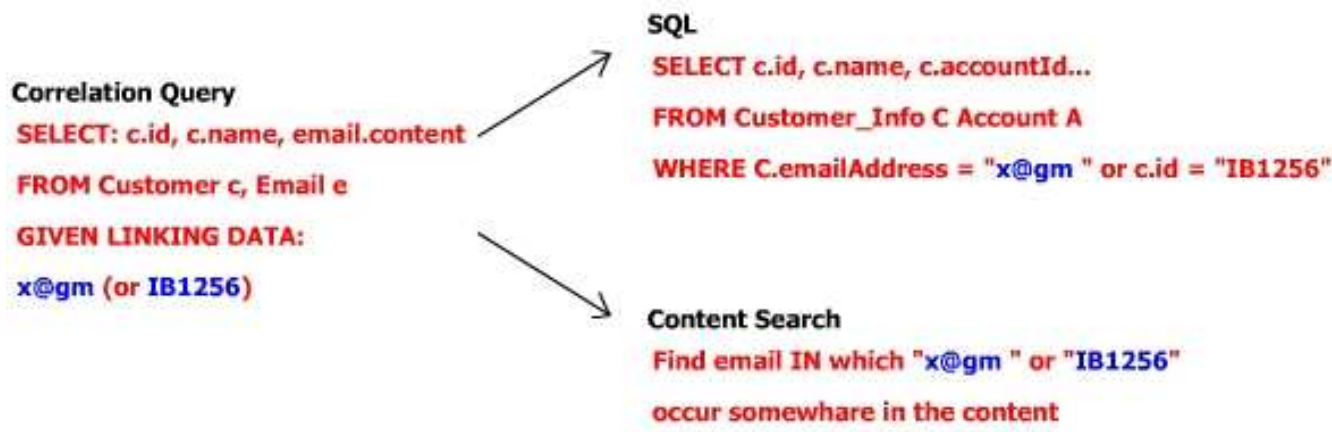


Figure 5.1: Task Decomposition in a Correlation Query

The query in Figure 5.1 is issued for the purpose of retrieving the (in this case, email) documents that contain the linking data ‘x@gm’ or ‘IB1256’, paired with

some relational data retrieved by a SQL query. The next subsection describes how this decomposition is carried out.

The formalisation of the Correlation Query will be as one of the future work. For the scope of this project, all the correlation queries are issued by from the Correlation Query Control Panel of the Graphical Interface described in Chapter 3.

5.1.1 Correlation Overhead

By correlation overhead is meant the additional tables that are required in order to evaluate correlation queries. These are described now.

Unstructured and semi-structured documents are passed through one or more application-specific text analyser for extracting linking data. An item of linking data acts as a bridge (in a similar way to a foreign key) for connecting an unstructured document with relational data. The outcomes of document analysis are stored as - XML documents, each of which includes the original document content together with annotation information (the start index and end index of a character span in a text fragment). In order to evaluate the correlation query, we need to provide some subsidiary information.

Firstly, a relational table is created, called Document Directory, so that each tuple in the table holds an assigned document identifier as well as the path of the XML document in which the result of document annotation is recorded. The Document Directory is assumed by Lindas (i.e., it has to have been created before Lindas can do any useful work). Potentially, a new entry is added to the table whenever a document has been processed by the Text Analyser.

Furthermore, for each detector target, i.e., each user-defined named-entity, such as an email address, there is potentially a set of documents in which it could occur. A Correlation Table is created to hold the value of that entity and the identifier of each document in which the value appears. The identifier, therefore, acts as a foreign key which points to a document identifier in the Document Directory table. A correlation table is created whenever a new annotator (named-entity detector) is started. Henceforth, an entry is added to the table whenever the corresponding target is found in a new document (or in a new place in an existing document).

Finally, a Correlation Table Registry is created as an index for all the Correlation Tables. An entry is added to the registry when a new annotator is started. It contains the name of the annotating entity and its type. It is used for finding which type a linking data entity belongs. Two entities may have different names (e.g., “Internet Banking User Email” and “Complaint Sender Email”), but they belong to the same type (e.g., “Internet Banking User Email” and “Complaint Sender Email” all belong to the type “Email”). Figure 5.2 shows, with an example, an overview of the additional tables required by correlation process, i.e., the Document Directory and the Correlation Table Registry together with one Correlation Table per target, so that, in the running example, there are two Correlation Tables: one holds the results of Email annotation; the other holds the results of IB_ID annotation.

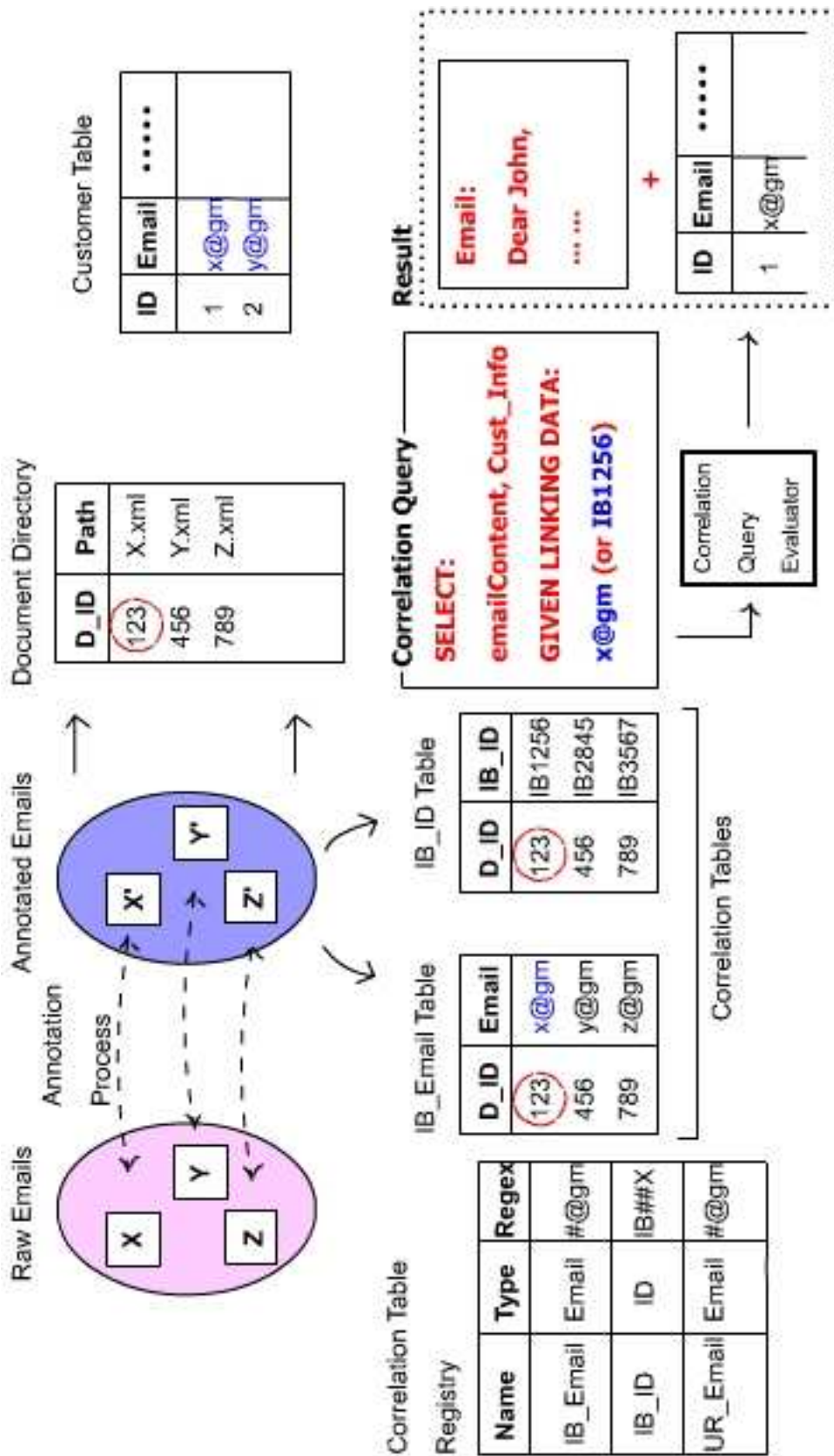


Figure 5.2: Correlation Overhead

5.1.2 Correlation Query Evaluation

The processes involved in the evaluation of correlation queries are assisted by the correlation overhead. In the document search task, all the documents that contain the linking data in question are retrieved according to the following steps (which are depicted in, and based on, the example in Figure 5.3):

1. The issuer of the correlation query asks for all the documents that contain an `IB_Email` entity with type `Email` and has value: `"x@gm"`.
2. We firstly scan down the Correlation Table Registry (CTR) to find, as mentioned in the last section, which type the string `"x@gm"` belongs. The CT search can be narrowed to the attribute name `"IB_Email"` or more generally to any attribute with domain `"Email"` (e.g., maybe another CT has the email value `"x@gm"`, but as a value in a domain named, e.g., `"Internet_Banking_Email"`). This is also why we need to remember the entity type in CTR.
3. We then take each CT (found in the previous step) to search for document identifiers such that the documents pointed to by them contain the linking data.
4. By using all the returned document identifiers, we can subsequently reach the original document using the content repository API.

In principle, it should be possible to link up a series of CTs, even reach documents from external sources. The above procedure has been implemented into

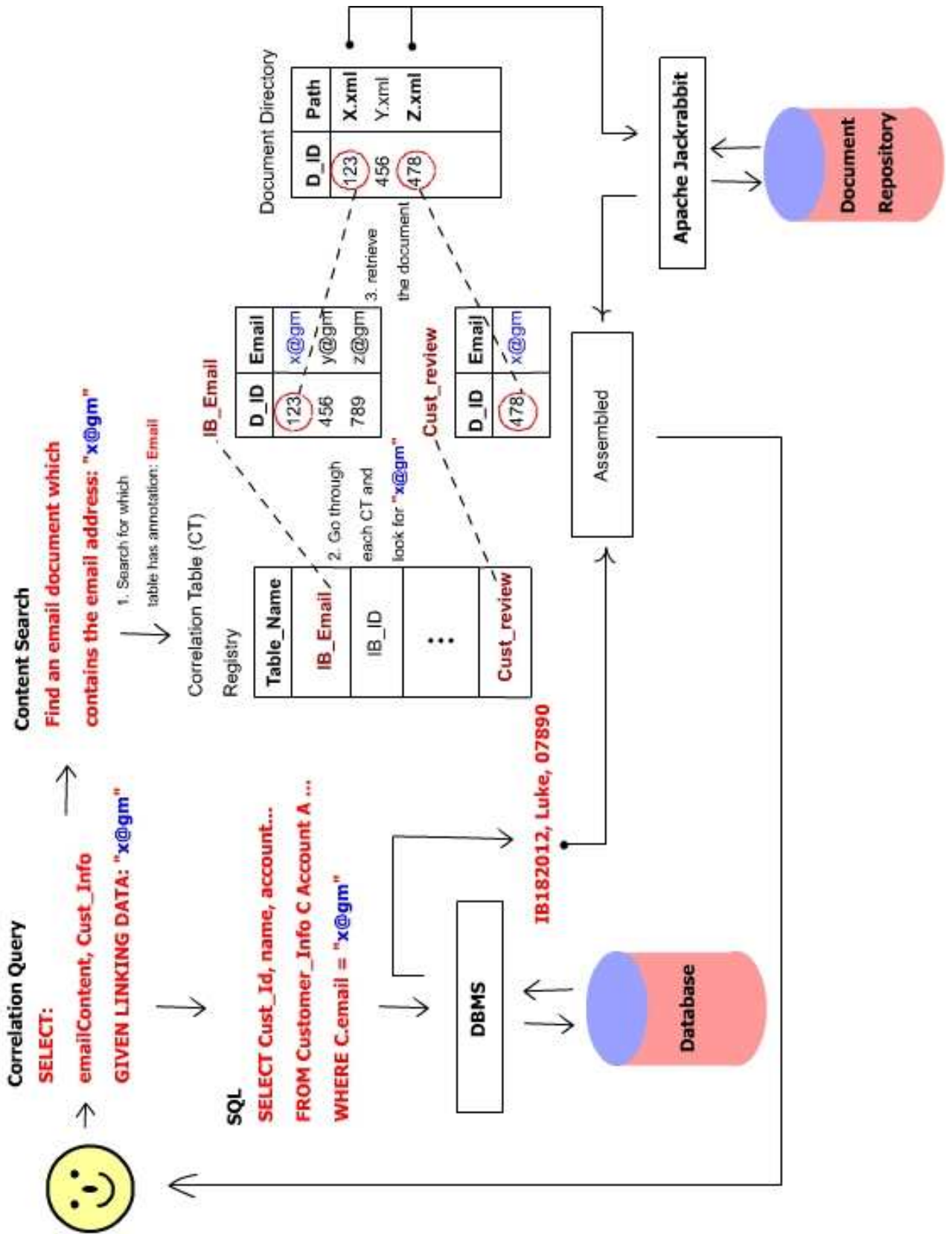


Figure 5.3: Correlation Query Evaluation

the Lindas system, the correctness has been verified by conducted experiments (which will be discussed in Chapter 6).

In the structured data query, we retrieve information about the customer (for example: ID, name, age, occupation, account number. etc) who owns the IB_Email given as the input linking data to the query evaluator. At the same time, we can, at the same time, find the account information relating to this customer using other linkable information, i.e., by providing keys, e.g. his or her identifier included in the retrieved customer information, and foreign keys, e.g. his or her account number.

The implementation of the Evaluation Methodology is shown in Appendix 4.

5.2 Populating Database with Unstructured Data

As mentioned in previous chapters, unstructured documents hold the majority of information generated by modern business activities. It is a time-consuming task for humans to interpret and retrieve information that has business value but lies hidden inside massive amounts of unstructured documents such as service reports, customer reviews, etc.. A portion of those documents can be processed to supplement the information held in the database. In some situations, when there data is missing in the database, it may be possible to obtain it from annotated unstructured documents. Consider the example shown in Figure 5.4, the Customer_info table in the database is missing the email address of customer with ID: IB1820789 but it can be found (with some likelihood which would need to be considered) in the message at the bottom left of the diagram. Likewise, we could use annotated unstructured documents to correct database information.

Also shown in Figure 5.4 is the fact that the customer (with ID IB1820123) sent an email to request an update to the information held about him. If he disclosed his ID, a Correlator would be able to support a process by means of which the annotated version of the email underpins the replacement of the old email address with the new one.

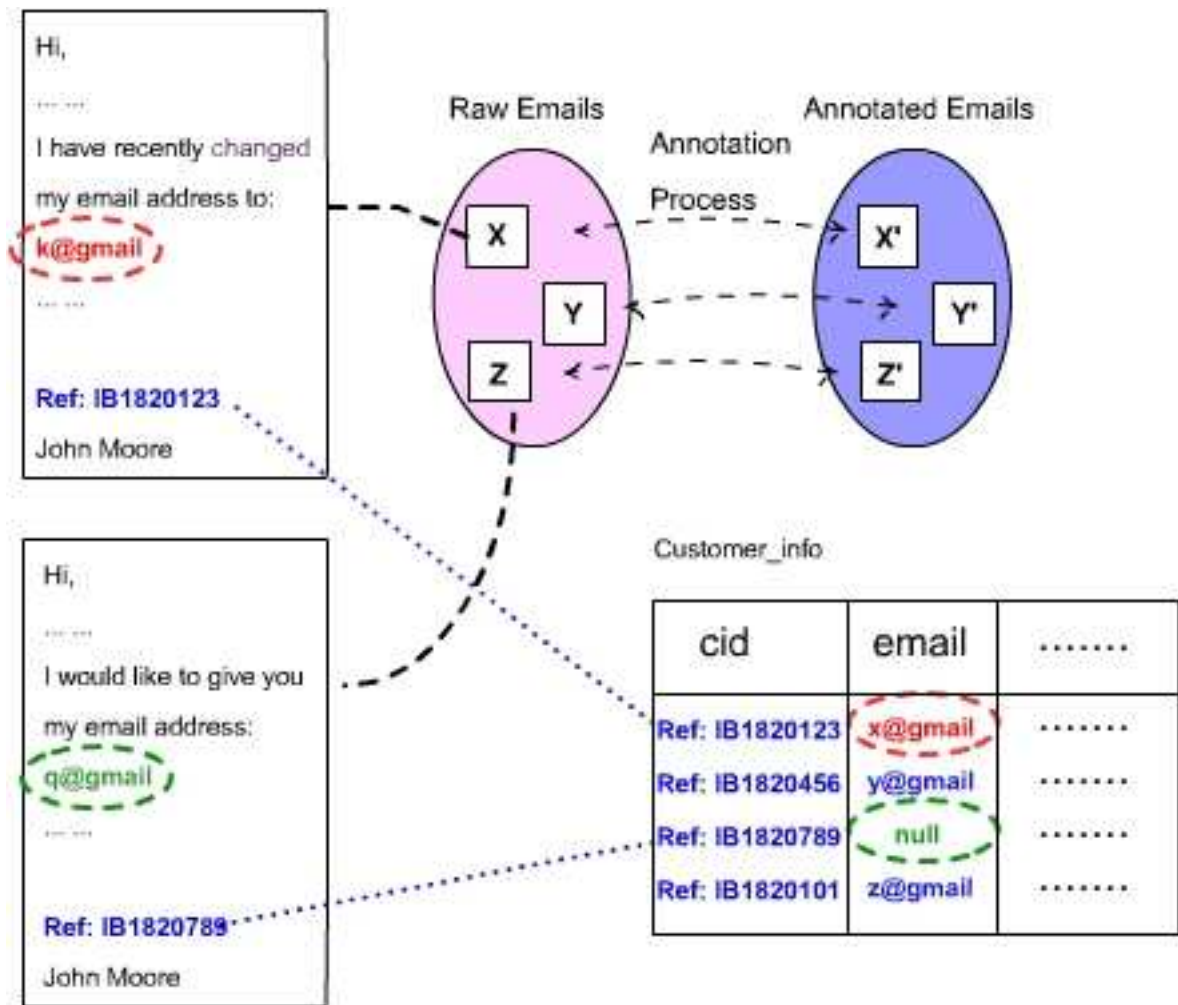


Figure 5.4: Populating a Database with Analytical Outcomes

5.3 Sentiment Analysis

Sentiment Analysis is a technique to detect favourable and unfavourable opinions toward specific subjects within large number of documents. It offers enormous opportunities for various applications. It can provide powerful functionality for competitive analysis, marketing analysis, and detection of unfavourable rumours for use in risk management.

5.3.1 Approaches in Sentiment Analysis

Various methodologies have been developed by researchers in the areas of artificial intelligence and computational linguistics. Applications have been developed from such ideas that can extract sentiment expression from massive amount of text fragments. One example was proposed in [TJ03]. It starts from a critique of the traditional approach in sentiment analysis of classifying the entire document into positive polarity and negative polarity and proposes, instead, that we should focus on extracting sentiments associated with polarities of positive or negative for specific subjects from a text fragment. Therefore, two essential issues need to be dealt with when developing sentiment analysis applications are:

1. To identify how sentiments are expressed in texts and whether the expression indicates a positive or negative opinion towards the subjects.
2. To improve the accuracy of sentiment analysis, it is important to properly identify the semantic relationship between the sentiment expression and the subject.

5.3.2 A Sentiment Analysis Example

Natural language processing (NLP) techniques have an important role in identifying sentiment expressions and analysing their semantic relationships with the subject term. By integrating NLP algorithms into Lindas, in principle, we would be able to perform sentiment analysis through the Correlator component of Lindas. This would add even more value (e.g., in scenarios that are typical of customer service call centres) by offering not just paired unstructured and structured information but also in providing members of the customer service team with an indication as to what a caller has a positive or negative opinion of the services, thereby allowing the staff member to predict better what the caller's behaviour and expectations might be.

Consider a situation when customers of a service provided by a company frequently send their criticism of the service via emails. Assume that the received emails are analysed by the company side with a view to finding out the sentiments expressed on the subject 'service S'. We would be able to perform sentiment analysis and adding sentiment annotations about email documents to a database table, called Sentiment Table (ST). ST holds (the name of) the email sender, the email address, and sentiment words (such as: 'good', 'so-so', or 'bad'), the name of the service the sentiment expressed refers to, and a timestamp. Each tuple in ST is called a Sentiment Tuple. An example of this table is shown in Figure 5.5. Based on the information stored in ST, we can assess the customer view and the performance of the service. For example, we can query the Sentiment Table to retrieve all the sentiment tuples for the customer with email address: x@gmail. Thus, we can check whether the service has improved or degraded by interpreting the change in sentiment words over a period of time. For instance, if last year

this time, the customer felt the service was ‘bad’, but one year later, during a customer survey, he or she now votes ‘good’ this means that, in the eyes of the customer, the service has improved.

SENDER	EMAIL	SENTIMENT	DATE
John	Jhn@gm	Bad	15/08/06
Peter	ptr@gm	So-So	15/08/06
Kate	kt@gm	So-So	15/04/06
Ben	bn@gm	So-So	15/06/06
Peter	ptr@gm	Good	15/07/07
John	jhn@gm	Good	15/08/07
Nick	nck@gm	Bad	15/08/04



Figure 5.5: Sentiment Relational Table

5.4 Summary

This chapter has described another major component of the Linking Data system and the functionalities it provides. After annotating unstructured and semi-structured documents through the Text Analyser, and based on supporting information provided by Correlation Tables, the Document Repository and the Correlation Table Registry, we can issue correlation query to retrieve documents in repositories that are paired to data residing in databases. This solution allows structured, semi-structured and unstructured data to be linked thereby making it available to be analysed further using commercial business intelligence and reporting tool. The solution is based on retrieving data across various data sources

with a single correlation query. We can, alternatively, supplement and update database by processing annotated unstructured documents. Finally, the Linking Data Architecture can be applied in more sophisticated analysis, e.g., for performing sentiment analysis on communications received from customers.

Chapter 6

Experimental Study

Based on the Linking Data System Architecture defined in Chapter 3, and by deploying a set of document annotators into the Text Analysis Engine for the purpose of extracting linking data from unstructured documents, we can move toward the implementation of a prototype system that is capable of performing some of the data correlation functions identified in Chapter 5. The prototype system has been named Lindas (for **L**INking **D**Ata **S**ystem). It is built upon various currently available software products and with specific software development tools. This chapter concentrates on describing how Lindas is built and proceeds to discuss the experiments conducted on Lindas to examine its performance.

6.1 Prototype System

Lindas was developed for the purpose of demonstrating the idea of linking unstructured documents with relevant structured data. Lindas views structured data as comprising a predefined set of ‘entities’ and annotates documents with

the ‘entities’ found inside them, thereby effectively creating links between documents and structured data. Lindas allows users to issue correlation queries through a graphical interface. Such a query is evaluated according to the evaluation methodology defined in Chapter 5. The implementation of the Correlation Query Evaluation is shown in Appendix 4. Moreover, Lindas is capable of populating the database with data from annotated documents. Appendix 3 shows the implementation of this function.

Lindas has been implemented in Java according to the functional requirements addressed in Chapter 1. It was developed using Eclipse, the open-source integrated software development environment (ISDE). In its default form, Eclipse is a Java ISDE. Users can extend its capabilities by installing plug-ins written for the Eclipse, seen as a software framework.

6.1.1 Technological Solutions

From the viewpoint of the system architecture, the Text Analysis Engine and the Correlator have to interact with a content repository and a database, deployed to store and retrieve documents and structured data, respectively. Therefore, Lindas can be seen as having some aspect of an integrated data management system, to the extent required by the proposed data correlation capability. In this section, we briefly explore what current data management technologies are available to be used in implementing the idealised linking data system.

The Text Analysis Engine component of Lindas was developed and deployed using the IBM UIMA SDK. There are two benefits gained from using the UIMA SDK: firstly, UIMA is implemented in Java and thus could be easily integrated

into the development of Lindas; and secondly, some useful UIMA SDK tools run as plug-ins to the Eclipse platform.

The content repository was implemented using Apache Jackrabbit, an open-source Java content repository that implements the Java Content Repository (JSR-170) API. JSR-170 defines a standard to access content repositories from Java. Appendix 1 shows the Java implementation of using Jackrabbit to initialise and access content repository.

On the structured data management side, the Apache Derby in-memory database was selected. As a full-featured, open source, embedded relational database management system, Derby is written and implemented completely in Java. Derby comprises a SQL-based database engine that was tightly couples into Lindas.

6.1.2 Correlation Correctness Test

Applications built upon the Linking Data System expect to be able to retrieve semantically correct correlation results. Lindas has passed two types of functional tests that focus on verifying the correctness requirements.

Firstly, for a result to be correct it has to return the correct number of documents. Given a linking data value (such as an Email[ohn@gmail.com]) and given test datasets of which we have full knowledge, we can anticipate that Lindas should retrieve N documents from the repository, with each of the N documents containing an annotated entity, namely “Email”, with the given value, say, “john@gmail.com”. If the actual number of documents retrieved is M , then M should equal N .

Secondly, for a result to be correct it has to return, together with each retrieved document, the related structured data retrieved from database according to the linking data obtained by annotation. So, if we assume John's email to be the linking data in a correlation query that requests customer details and account information, then John's details and account information should be returned as stored in the databases, but not other information about John and not any information about any other person. Such correctness requirements are strictly imposed on the correlation query evaluation methodology, which make use of the correlation overhead comprising the Document Directory, Correlation Tables and the Correlation Query Table Registry.

In order to focus on testing the correctness of the correlation query function, we assume that the regular expression used for the annotation process correctly captures all occurrences of the targeted 'entity'. In other words, the correctness of the annotator is not in the scope of the functional test conducted here. It is the responsibility of users of Lindas to define a fully deterministic regular expression for each annotator.

6.2 Performance Experiment Design

A set of experiments has been conducted with the purpose of evaluating some aspects of the performance of Lindas. The performance indicator we use is the response time for correlation queries, which we refer to as CQT, i.e., the time to retrieve all pairs of documents and related structured data, given the linking data in the correlation query.

The expectation is that CQT should not increase too sharply as the data volume or the hit rate goes up. By *data volume* we mean the total size of unstructured documents residing in the document repository. Note that this accepts the widespread assumption that, in comparison with that taken by structured data, the space consumed by unstructured documents is dominant by far. By *hit rate* we mean the ratio of the total number of documents that need to be returned by a correlation query to the total number of documents that the query ranges over.

The efficiency of Lindas is also influenced by the size of the correlation overhead, which is generated by text analysis and used for assisting correlation query evaluation. The higher the ratio of the correlation overhead to the data volume, the lower the expected efficiency is.

The rest of this section introduces how each experiment is set. The design of each experiment addresses the following points: purpose, expected outcome, and the independent and dependent variables to be used.

We used synthetic data for two main reasons: firstly, Lindas cannot be expected to be robust enough to cope with public data sets, which tend to be overwhelmingly large; secondly, the same data set used in performance experiments is also used to test correctness. Thus, we prepared a batch of documents to use in the experiments. Rather than relying on very large (e.g. up to 500 Megabyte) document collections available from the Internet, I have developed an Automatic Document Generator (ADG) to address this need. The functions of ADG are discussed at the end of this section. Apart from generated documents, the experiments are also supplied with an example of customer and account relational information stored in database.

6.2.1 CQT vs. Data Volume

Purpose:

This experiment has been designed to examine the incremental pattern of CQT with regard to data volumes.

Expected Outcome:

This experiment is expected to show that CQT doesn't increase too sharply as the data volume increases.

Independent Variable: Data Volume (Unit: Megabyte).

Dependent Variable: CQT (Unit: Second).

Fixed Variables: Number of Entity Detection Annotators = 1; Hit Rate = 50%; Size of Structured Data = 50 KB.

6.2.2 CQT vs. Hit Rate

Purpose:

This experiment has been designed to examine the incremental pattern of CQT with regard to the hit rate.

Expected Outcome:

The CQT doesn't increase too sharply as the number of documents to be retrieved by a correlation query increases.

Independent Variable: Hit Rate (as a percentage).

Dependent Variable: CQT (Unit: Second).

Fixed Variables: Number of Entity Detection Annotators = 1; Data Volume = 50 Megabyte; Size of Structured Data = 50 KB.

6.2.3 Correlation Overhead vs. Data Volume

Purpose:

This experiment has been designed to examine how the correlation overhead grows as the data volume increases.

Expected Outcome:

The correlation overhead is expected to grow at a rate that is not overly faster than that by which the data volume increases when the hit rate is fixed.

Independent Variable: Data Volume (Unit: Megabyte)

Depend Variable: Correlation Overhead (Unit: Megabyte)

Fixed Variables: Number of Entity Detection Annotator = 1; Hit Rate = 50%; Size of Structured Data = 50 KB.

6.2.4 Correlation Overhead vs. Hit Rate

Purpose:

This experiment has been designed to examine how the correlation overhead grows as the hit rate grows.

Expected Outcome:

The correlation overhead is expected not to grow too sharply as the hit rate increases.

Independent Variable: Hit Rate (as a percentage)

Depend Variable: correlation overhead (Unit: Megabyte)

Fixed Variables: Number of Entity Detection Annotator = 1; Data Volume = 50 Megabytes; Size of Structured Data = 50 KB.

6.2.5 Automatic Document Generator

The experiments presuppose appropriately prepared datasets. For the reasons already given, an Automatic Document Generator (ADG) has been developed. ADG is capable of dynamically generating either unstructured or semi-structured documents up to a specified data volume. The input to ADG is two documents made up of random text: Document A has linking data randomly scattered into different locations in the text content. Document B contains random text only. When ADG is given the total number of documents of type A and type B that is to be generated, it creates a data set up to a desired size.

6.3 Result Analysis

The correlation query that used for all the experiments described in the previous section is:

```
SELECT c.id, c.name, c.email, a.account, Email
FROM Customer c, Account A, Email e
WHERE c.id = a.id
```

```
GIVEN LINKING DATA: Email('IB1820481@bank.com')
```

During each experiment, this query is issued from the Correlation Query panel of the Graphical Interface (described in Chapter 3).

6.3.1 CQT vs. Data Volume of Unstructured Document

Figure 6.1 shows that as the data volume doubled from 50 MB to 100 MB, almost the same growth rate was observed on the CQT. The same happened when the data volume goes from 250 MB to 500 MB. Moreover, the rate of increase on CQT remains as it is when the data volume grows from 50 MB to 250 MB (i.e. five times). This is because, as the hit rate is fixed, the size of the Document Directory and Correlation Tables increases as fast as the data volume. Therefore the time taken for correlation queries to retrieve all the desired documents won't grow faster than the growth of the data volume.

The size of Document Directory depends on the number of documents, since we need to allocate one tuple of path and document ID per document. Besides, the size of each Correlation Table grows as the linking data appears more and more frequently in documents.

Furthermore, changes in CQT are predominantly contributed by the increment of time spent to access and search the correlation overhead rather than the time spent for retrieving out files from document directory.

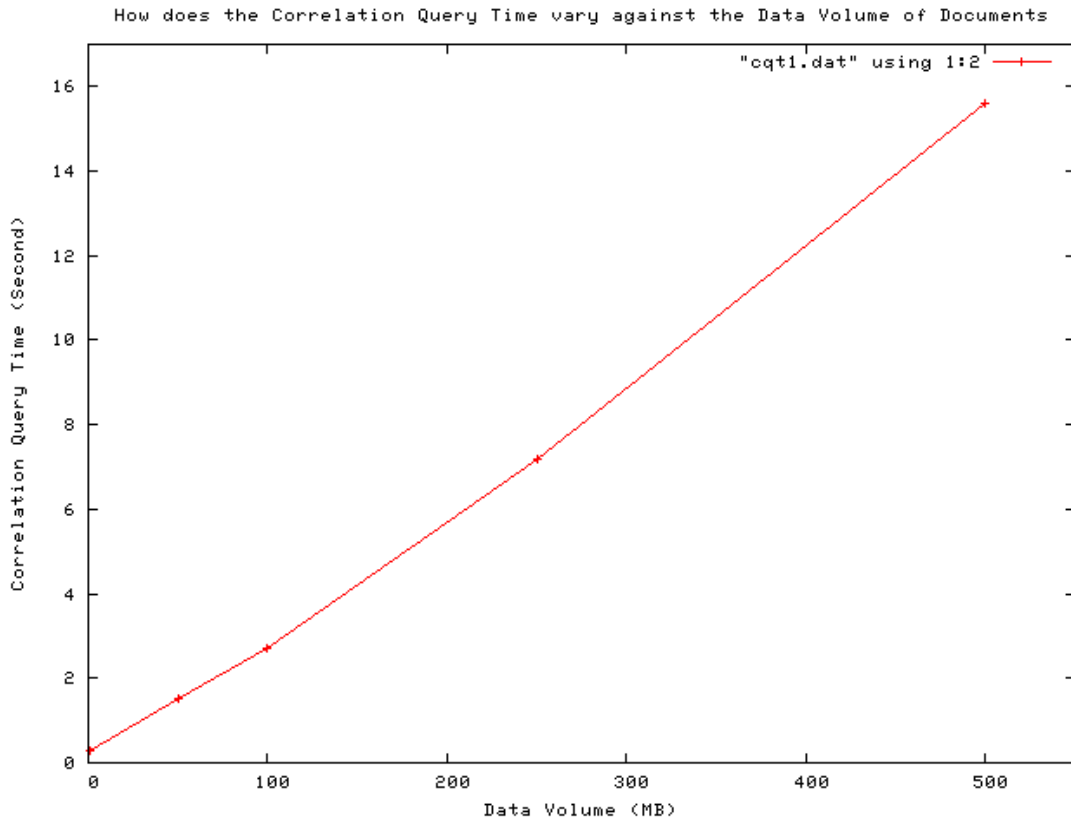


Figure 6.1: CQT vs. Data Volume of Documents

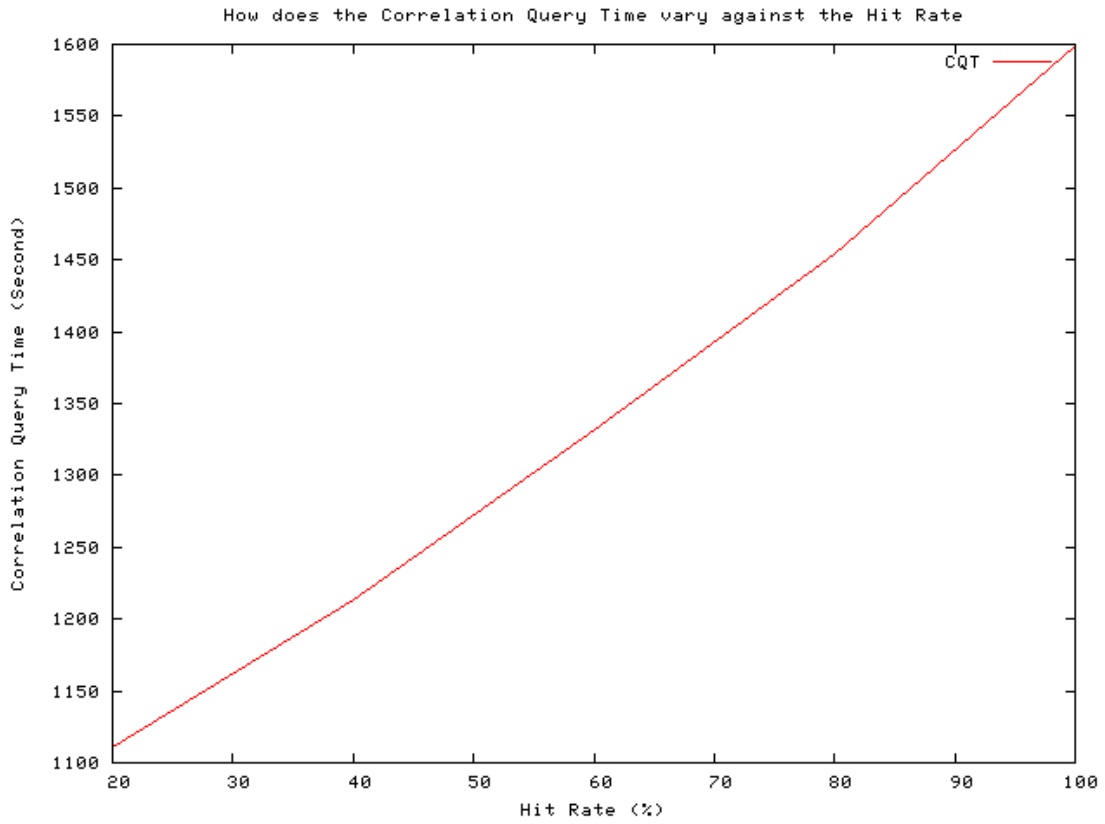


Figure 6.2: CQT vs. Hit Rate

6.3.2 CQT vs. Hit Rate

Figure 6.2 shows that the CQT increases roughly linearly as the hit rate increases from 20% up to 100%. This can be explained by the fact that for a given data volume if the hit rate is increased, which means that more and more documents contain the targeted linking data, then the size of corresponding correlation tables would grow too, linearly, as is shown, with the hit rate. The overall effect is that, as expected, more time is needed for evaluating the correlation query.

6.3.3 Correlation Overhead vs. Data Volume of Documents

Figure 6.3 shows that the correlation overhead does increase in the same rate as the growth of data volumes under a fixed hit rate. In other words, the correlation overhead doubled whenever the total data volume of documents was doubled too. This is because the correlation overhead consists of three kinds of relations: the Document Directory, a set of Correlation Tables and the Correlation Table Registry. The Document Directory increases at the same rate as the total number of documents increases, so does the size of Correlation Table (since the hit rate is fixed in the experiment). The size of the Correlation Table Registry remains the same, and is maybe negligible (if the number of entity annotations is no more than, for example, 50), provided that we do not add new entity detection annotators into the text analysis engine.

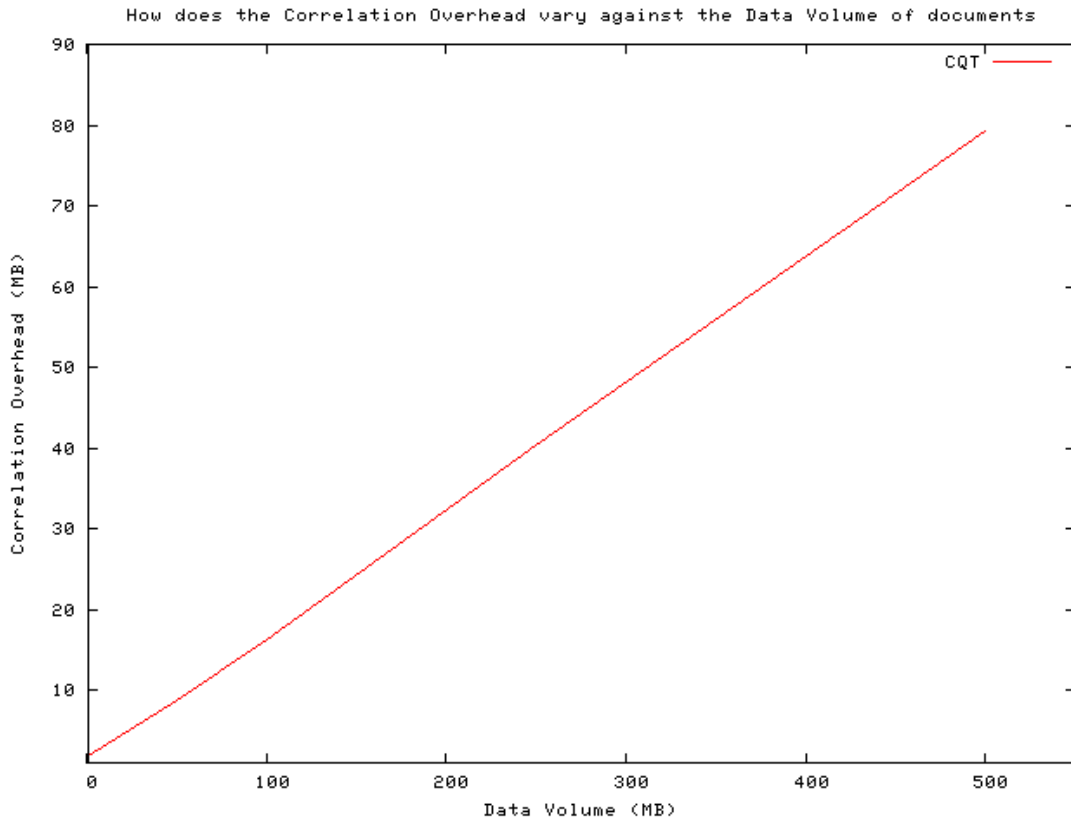


Figure 6.3: Correlation Overhead vs. Data Volume of Documents

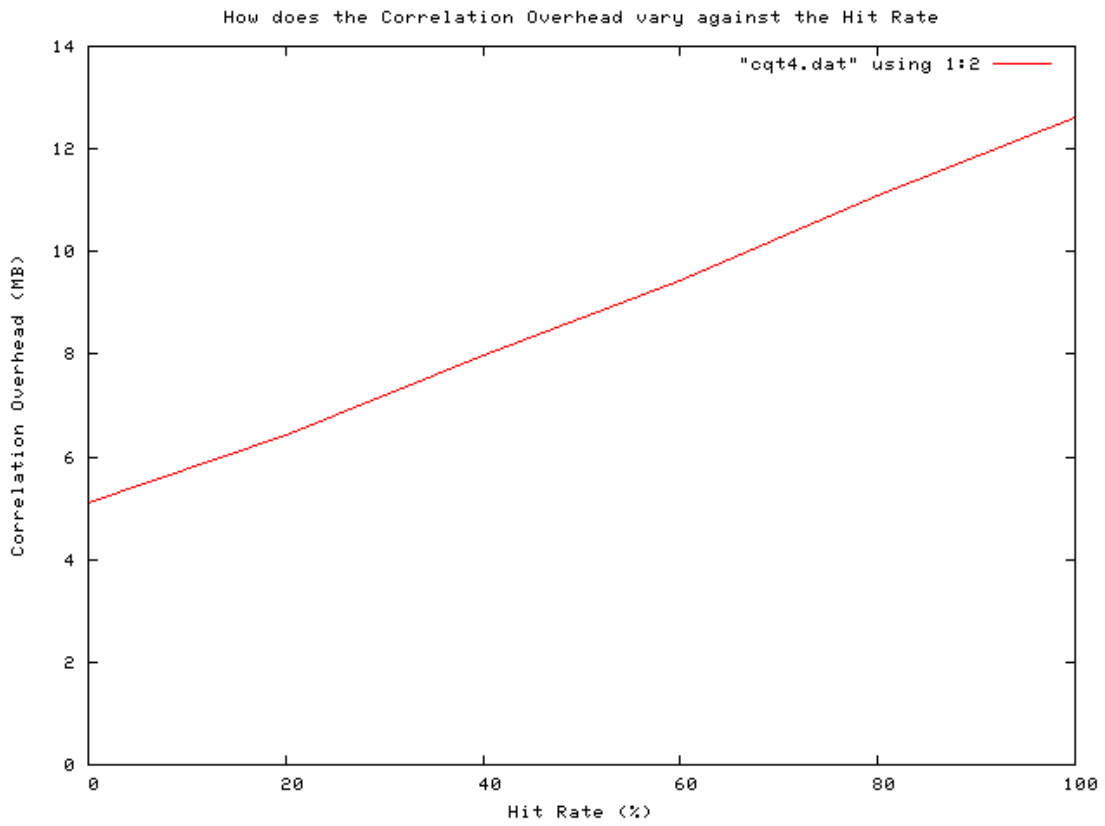


Figure 6.4: Correlation Overhead vs. Hit Rate

6.3.4 Correlation Overhead vs. Hit Rate

Figure 6.4 shows how the correlation overhead is affected when the hit rate varies and both the data volume and the number of annotators remain unchanged.

As expected, as the hit rate increases, the total size of correlation tables increases even if the data volume remains unchanged.

6.4 Summary

To summarise, the overall outcome of the performance experiments shows that correlation query evaluation performs reasonably when several aspects of the environment vary. For example, increases in the data volume cause roughly linear changes on the CQT. As for space efficiency, the average ratio of correlation overhead to the data volume remains at 6.5:1. For example, when the data volume is 500 MB, each 6.5 MB of data volume needs 1 MB of overhead information.

Chapter 7

Related Work

This chapter aims to survey related works in the area of linking text retrieval systems and structured databases. Traditionally, the mainstay of this area has been integration of data across heterogeneous structured databases [DH05]. Recently, there has been significant work on keyword-based search in relational databases. A number of research prototypes have been developed to address the problem of supporting keyword queries over structured data. In DBXplorer [ASG02] and BANKS [BGS02], given a set of terms as input, the task is to retrieve the candidate join-trees (groups of rows across multiple tables in the database interlinked through foreign-key relationships) that collectively contain these terms; these candidate join-trees are also ranked against each other based on relevance. The entity identification task of Lindas is quite different; it interprets the input document as a sequence of terms, and identify the best matching entities as well as their embeddings in the document. In [VL03], algorithms for returning the top-k results in this context are presented.

Recent work has also addressed the query languages that integrate information retrieval related features such as ranking and search into XML queries have been

proposed in [AKSD02, AYSJ03]. together with proposed techniques to evaluate these ranked queries. In [ea98], the problem of ranking SGML documents using term occurrences is considered.

In [AY03], an approach is presented for adding structure to the (unstructured) HTML web pages present on the world-wide-web. The paper argues that creating structured data typically requires technical expertise and substantial upfront effort; hence people usually prefer to create unstructured data instead. The main contribution of the system is to make structured content creation, sharing, and maintenance easy and rewarding. They also introduced a set of mechanisms to achieve this goal. Notice how this approach is different from the approach, where here, in Lindas, we use text analytics to add structure to unstructured data.

There is a long history of previous work that directly address linking across structured data and documents. The EROCS [VTM06] prototype system can identify and entity link even when the relevant entity is not explicitly mentioned in the document. An efficient algorithm has been proposed in [VTM06] that, while keeping the amount of information retrieved from the database at a minimum, find the best matching entities as well as their optimal embedding in the document in the document. In [HZ05], the IBM AVATAR system was developed to have the use of text analytics as a mechanism to bridge this structured-unstructured divide, and utilizes an existing commercial relational DBMS as the underlying storage engine. Also mentioned in [HZ05] that recent advances in text analytics for free-form text. However, for these techniques to be deployed in large scale enterprise applications, there is a need for contribution from data management research. By way of contrast to EROCS and AVATAR, Lindas has further step into more practical work on the area of linking structured data to unstructured text documents. Based on the previous research contributions on

identifying possible text analytical techniques and algorithms, in this dissertation, we have proposed a more refined architecture that is composed of content repository systems, database, and a correlation query methodology, for the purpose of integrating data linkage with management.

Chapter 8

Conclusion and Future Work

This dissertation presents the project work on developing a Linking Data System (Lindas) aims for linking documents in repositories to structured data residing in database. The system has been developed based on a proposed architecture that laid out text analytic techniques, content repository and relational database system and defined how to compose them together to achieve this goal. A Correlation Query methodology is also proposed in this project that performs the tasks of retrieving text documents with relevant structured data. This chapter will close the main content of this dissertation by addressing several further improvements that can be done to Lindas.

Further Improvements

The architecture of linking data system is aimed to connect the previous research works on defining text analytical techniques, with the addition of document repository and relational database, to build an integrated data management system with data linkage capabilities. However, the text analytic component of Lindas

can only annotate out entity terms under the assumption that the terms are explicitly mentioned in the document. Various text analytic algorithms, developed by the UIMA SDK, have already been mentioned in EROCS [VTM06] and IBM AVATAR [HZ05] can handle the situations when terms are not explicitly recognisable. A further improvement can be made to Lindas by integrating some of those algorithms to reach a more reliable and intelligent level on extracting linking data.

In principle, it is also feasible to add more data correlation capabilities to the Correlator of the linking data architecture. One of such example is mentioned in Chapter 5, namely sentiment analysis. Sentiment Analysis can provide powerful functionality for competitive analysis, making analysis and the detection of unfavorable rumors of risk management. [TJ03] has proposed a sentiment analysis algorithm that can be added into the text analyser. This algorithm has the goal of:

1. Identifying the polarity of the sentiments
2. Identifying sentiment expression that are applied, and
3. Identifying the phrases that contain the sentiment expressions.

One more major improvement can be made is to extend the linking data architecture toward web services. The data correlation functions can be invoked remotely by using message passing mechanisms, e.g. SOAP. The Text Analysis and the Correlator components can be grouped into a middleware layer so that the correlator can interact with multiple requests concurrently and the text analysis engine can be more conveniently switched to process different document repositories.

Finally, a minor improvement can be made is to the Graphical User Interface proposed in this project in which there still are gaps to be filled. For example, the control panels could be reorganised; the Correlation Query and Annotator Information result display can also be modified. We can refine the interface components based on Human-Computer Interaction design principles and usability guidelines.

Bibliography

- [AKSD02] Yu C. Al-Khalif S. and Srivastava D. Querying structured text in an xml database. 2002.
- [ASG02] Chaudhuri S. Agrawal S. and Das G. Dbxplorer: A system for keyword-based search over relational database. 2002.
- [AY03] An Hai Doan et al. Alon Y., Halevy O. Crossing the structure chasm. 2003.
- [AYSJ03] Cho S. Amer-Yahia S. and H.V. Jagadish. Tree pattern relaxation. 2003.
- [Ber98] Philip A. Bernstein. *Repositories and Object Oriented Database, Research Survey*. SIGMOD Record, 1998.
- [BGS02] Nakhe C. Bhalotia G., Hulgeri A. and Sudarshan S. Keyword searching and browsing in database using banks. 2002.
- [DH05] A. Doan and A. Halevy. Semantic integration research integration in the database community: A brief survey. 2005.
- [ea98] S. Myaeng et al. A flexible model for retrieval of sgml documents. 1998.

- [FL04] D. Ferrucci and A. Lally. Building an example application with the unstructured information management architecture. *IBM System Journal*, Vol 43, (3), 2004.
- [Fra02] Stephen R.G. Fraser. Building a content management system. *Apress*, 2002.
- [HZ05] S.Vaihyathan H. Zhu, S. Raghavan. Avatar: Using text analytics to bridge the structured-unstructured divide. 2005.
- [jcp] Jcp home page for jsr 170, the java content repository api. <http://www.jcp.org/en/jsr/>.
- [Moo02] C. Moore. Diving into data. *InforWorld*, October 2002.
- [Ove] Apache Jackrabbit Jackrabbit Architecture Overview. <http://jackrabbit.apache.org>.
- [Pat06] Sunil Patil. Advanced java content repository api. *ONJava.com*, 2006.
- [Pie05] Peeter Piegaze. *Content Repository API for Java Technology Specification 1.0*. Day Management AG, 2005.
- [Pro] The Java Community Process. <http://www.jcp.org/>.
- [Sha07] Mark Shainman. Uncover buried treasures. *Teradata Magazine*, 2007.
- [Som05] Frank Sommers. Catch jackrabbit and the java content repository api. *Artima Developer*, 2005.
- [TJ03] Nasukawa T. and Yi J. Sentiment analysis: Capturing favorability using natural language processing. October 2003.

- [uim06] *IBM Unstructured Information Management Architecture (UIMA) SDK User's Guide and Reference*. July 2006.
- [VL03] Hristids V. and Gravano L. Ir-style keyword search over relational databases. 2003.
- [VTM06] Prasan M. Venkatesan T., Gupta H.R. and Mohania M. Efficiently linking text documents with relevant structured information. 2006.
- [(XQ)] W3C XML Query (XQuery). <http://www.w3.org/xml/query>.

Appendix A

Using Apache Jackrabbit and JCR API

This Appendix shows the implementation of storing and retrieving text documents via the Apache Jackrabbit. The content repository model to be used in this example (as shown in Figure A.1) has a repository that holds email documents.

The Java code shown below has three parts:

1. Initialising a repository (Figure A.2)
2. add a file to the initialised repository (Figure A.3)
3. and retrieve the added file (Figure A.4).

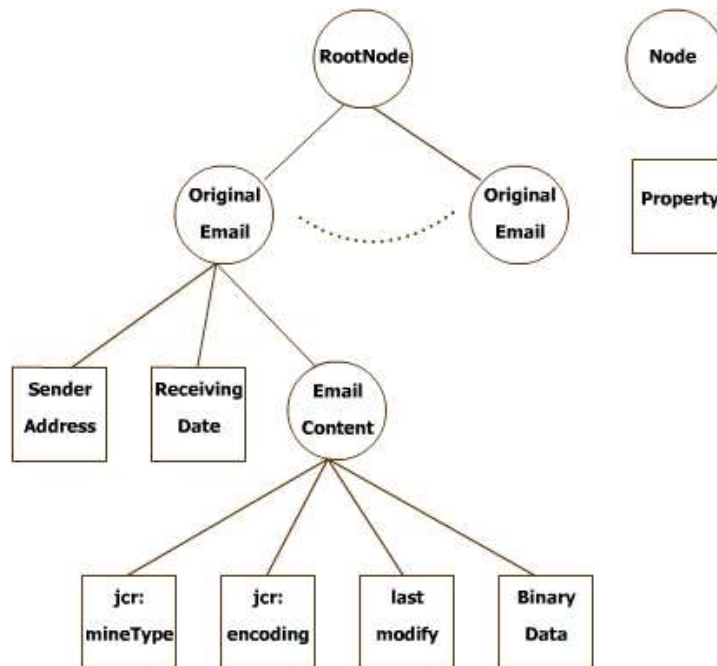


Figure A.1: A Jackrabbit repository for emails

```
File Edit View Insert Format Help
Session s;

void init( String username, String password, String homeDir,
          String workspace) throws Exception{

    String configFile = "repository.xml";
    String repHomeDir = homeDir;

    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.apache.jackrabbit.core.jndi" +
            ".provider.DummyInitialContextFactory");

    env.put(Context.PROVIDER_URL, "localhost");

    InitialContext ctx = new InitialContext(env);

    RegistryHelper.registerRepository(
        ctx,
        "repo",
        configFile,
        repHomeDir,
        true);

    Repository r = (Repository) ctx.lookup("repo");
    SimpleCredentials cred = new SimpleCredentials(
        username, password.toCharArray());
    s = r.login(cred, workspace);
    Workspace ws = s.getWorkspace();
    setNamespace(ws, "emails",
                "http://www.barik.net/wiki/1.0");
    Node rn = s.getRootNode();
    System.out.println(rn.hasNode("emails:orgnemails"));
    if(!rn.hasNode("emails:rawemails")){
        Node rawemails = rn.addNode("emails:orgnemails");
    }
}
```

Figure A.2: Initialising a Jackrabbit repository

```
File Edit View Insert Format Help

void addEmail(Session s, String sender, String date, String
filename) throws Exception{

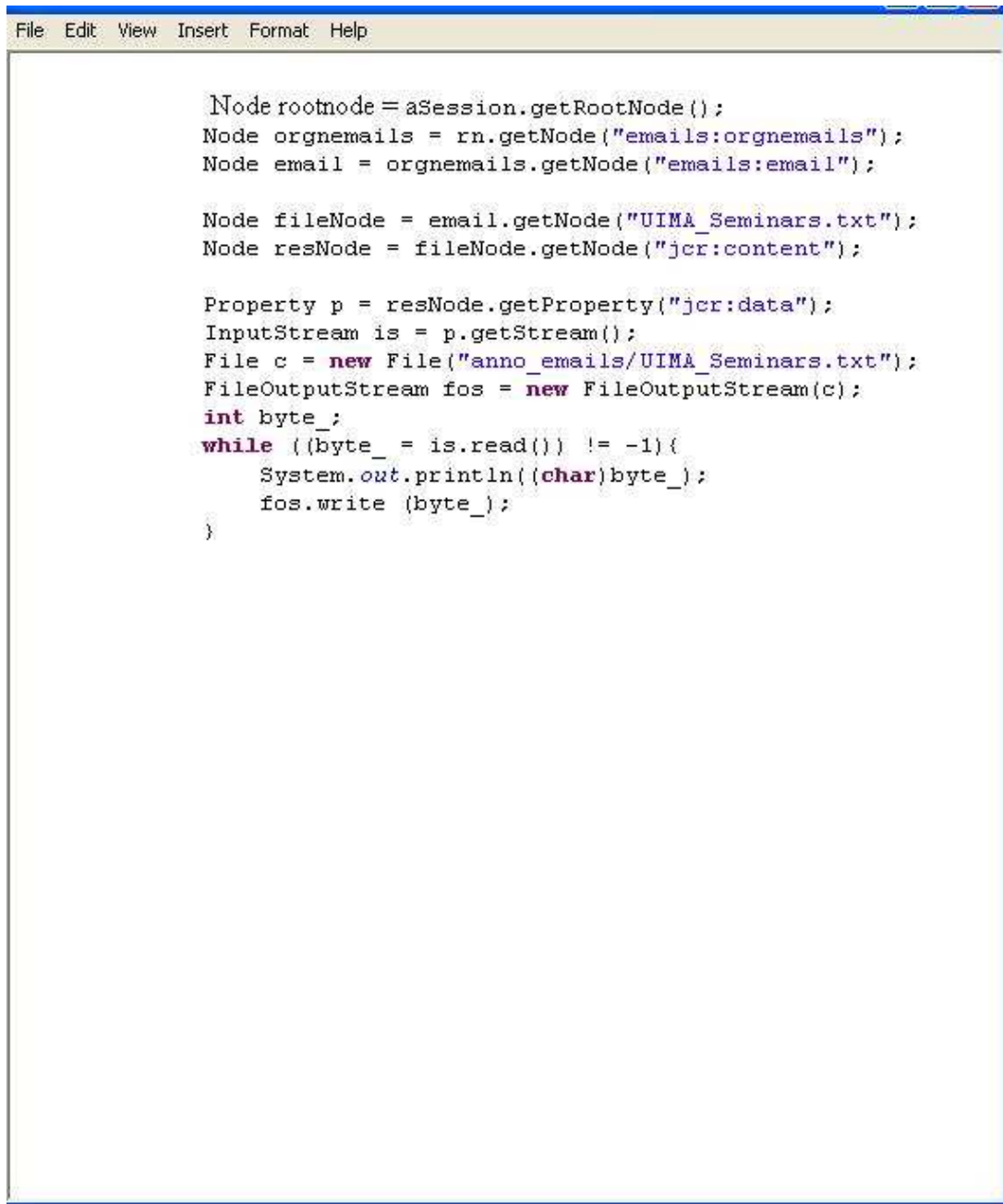
    Node rn = s.getRootNode();
    Node rawemails = rn.getNode("emails:orgnemails");

    Node email = rawemails.addNode("emails:email");
    email.setProperty("emails:sender", new
        StringValue(sender));
    email.setProperty("emails:date", new
        StringValue(date));
    s.save();

    //add a binary file
    File file = new File(filename);
    MimeTable mt = MimeTable.getDefaultTable();
    String mimeType = mt.getContentTypeFor(file.getName());
    if (mimeType == null)
        mimeType = "application/octet-stream";

    System.out.println(file.getName());
    Node fileNode = email.addNode(file.getName(),
        "nt:file");
    Node resNode = fileNode.addNode("jcr:content",
        "nt:resource");
    resNode.setProperty("jcr:mimeType", mimeType);
    resNode.setProperty("jcr:encoding", "");
    resNode.setProperty("jcr:data", new
        FileInputStream(file));
    //mandatory property
    Calendar lastModified = Calendar.getInstance();
    lastModified.setTimeInMillis(file.lastModified());
    resNode.setProperty("jcr:lastModified", lastModified);
    s.save();
}
```

Figure A.3: Add a file to Jackrabbit repository



```
File Edit View Insert Format Help

Node rootnode = aSession.getRootNode();
Node orgnemails = rn.getNode("emails:orgnemails");
Node email = orgnemails.getNode("emails:email");

Node fileNode = email.getNode("UIMA_Seminars.txt");
Node resNode = fileNode.getNode("jcr:content");

Property p = resNode.getProperty("jcr:data");
InputStream is = p.getStream();
File c = new File("anno_emails/UIMA_Seminars.txt");
FileOutputStream fos = new FileOutputStream(c);
int byte_;
while ((byte_ = is.read()) != -1){
    System.out.println((char)byte_);
    fos.write (byte_);
}
```

Figure A.4: Retrieve a files from Jackrabbit repository

Appendix B

Build an Annotator Using the UIMA SDK

This appendix describes how to develop Annotators and Analysis Engines using the UIMA SDK.[uim06] An example annotator to be built will detect Email Address with the regular expression: `IB1820[1-9][4-9]#@bank.com`.

There are several steps to develop a simple UIMA annotator:

1. Define the CAS type that the annotator will use.
2. Generate the Java classes for these types.
3. Write the actual annotator Java code.
4. Create the Analysis Engine Descriptor.
5. Testing the Annotator.

These steps are discussed in the next sections.

Defining Types

The first step in developing an annotator is to define the CAS Feature Structure types that it creates. This is done in an XML file called a Type System Descriptor. UIMA includes an Eclipse plug-in for editing Type System Descriptors (See Chapter 3 of the UIMA SDK Users Guide and Reference [uim06] for instructions on setting up Eclipse and installing the plugin).

The Type System Descriptor for the Email annotator is shown in Figure B.1.

After creating the Type System Descriptor, we can view and modify the descriptor in Eclipse. In the Eclipse navigator panel, right click on the file and select Open With -> Component Descriptor Editor. Once the editor opens, click on the “Type System” tab at the bottom of the editor window. A screenshot of the editor is shown in Figure B.2

The email annotator needs only one type – `Mao.IBUserEmail`. Just as in Java, types have supertypes. The supertype is listed in the second column of the left table. In this case the `IBUserEmail` annotation extends from the built-in type `uima.tcas.Annotation`.

Generating Java Source File for CAS Types

After a modified descriptor has been saved, the Component Descriptor Editor will automatically generate Java classes corresponding to the types that are defined in that descriptor, using a utility called `JCasGen`. These Java classes will have the same name as the CAS types, and will have get and set methods for each of the features that have been defined.



```
File Edit View Insert Format Help
<?xml version="1.0" encoding="UTF-8"?>
<typeSystemDescription
xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <name>TutorialTypeSystem</name>
  <description>Type System Definition for the
Internet Banking User Email Address examples
</description>
  <version>1.0</version>
  <vendor>IBM</vendor>
  <types>
    <typeDescription>
      <name>Mao.IBUserEmail</name>
      <description/>
      <supertypeName>uima.tcas.Annotation
</supertypeName>
    </typeDescription>
  </types>
</typeSystemDescription>
|
```

Figure B.1: Type System Descriptor for an annotator

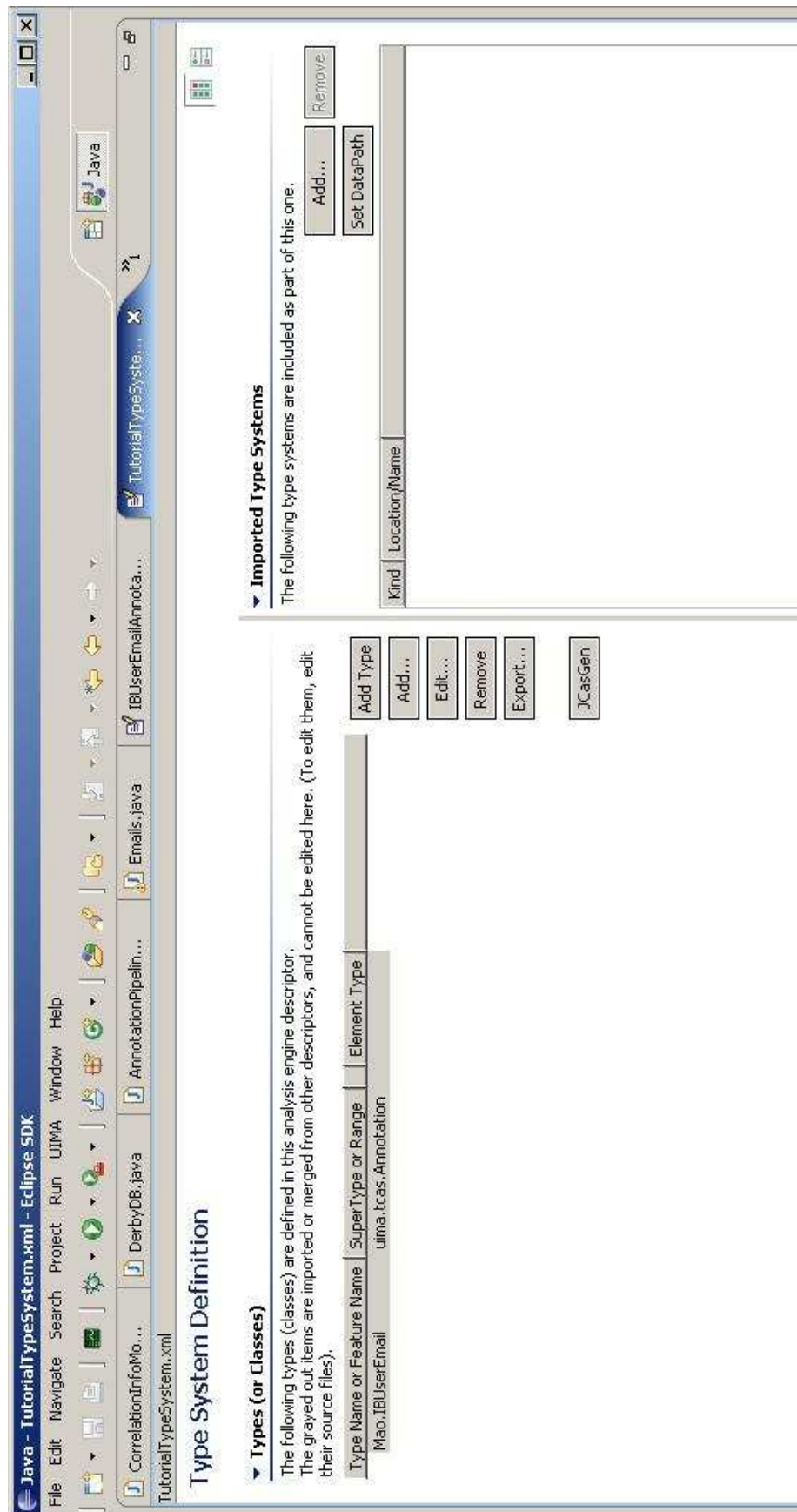


Figure B.2: UIMA SDK Component Descriptor Editor

Developing Annotator Code

All the annotator implementations have to implement a standard interface that has several methods, the most important of which are:

1. `initialize`,
2. `process`, and
3. `destroy`.

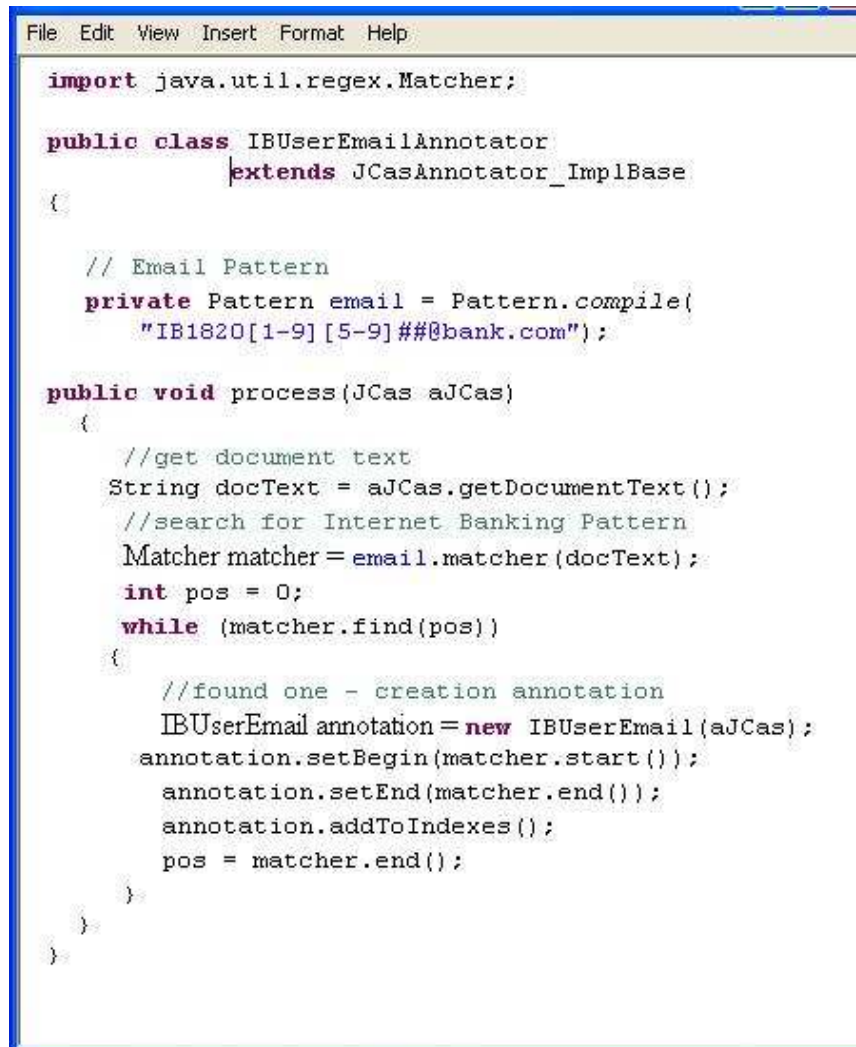
`initialize` is called by the UIMA framework once when it first creates the annotator. `process` is called once per item being processed. `destroy` may be called by the application when it is done.

The class definition for the `EmailAnnotator` implements the `process` method and it is shown in Figure B.3.

Creating the XML Descriptor

The UIMA architecture requires that descriptive information about an annotator be represented in an XML file and provided along with the annotator class file(s) to the UIMA framework at run time. This XML file is called an Analysis Engine Descriptor. The descriptor includes:

1. Name, descriptor, version, and vendor
2. The annotator's inputs and outputs, defined in terms of the types in a Type System Descriptor

A screenshot of a Java IDE window with a menu bar (File, Edit, View, Insert, Format, Help) and a code editor. The code defines a class IBUserEmailAnnotator that extends JCasAnnotator_ImplBase. It includes a private static Pattern for email addresses and a process method that iterates through the document text to find and create annotations for email addresses.

```
File Edit View Insert Format Help

import java.util.regex.Matcher;

public class IBUserEmailAnnotator
    extends JCasAnnotator_ImplBase
{

    // Email Pattern
    private Pattern email = Pattern.compile(
        "IB1820[1-9][5-9]##@bank.com");

    public void process(JCas aJCas)
    {
        //get document text
        String docText = aJCas.getDocumentText();
        //search for Internet Banking Pattern
        Matcher matcher = email.matcher(docText);
        int pos = 0;
        while (matcher.find(pos))
        {
            //found one - creation annotation
            IBUserEmail annotation = new IBUserEmail(aJCas);
            annotation.setBegin(matcher.start());
            annotation.setEnd(matcher.end());
            annotation.addToIndexes();
            pos = matcher.end();
        }
    }
}
```

Figure B.3: A Java implementation of the annotation algorithm for email addresses

3. Declaration of the configuration parameters that the annotator accepts.

The Component Descriptor Editor plugin, which introduced previously for editing the Type System descriptor, can also be used to edit Analysis Engine Descriptors.

The Component Descriptor Editor consists of several tabbed pages; but only a few of them are used to build the email annotator. The initial page of the Component Descriptor Editor is the Overview page, which appears as in Figure B.4

UserEmailAnnotator Analysis Engine (AE). The left side of the page shows that this descriptor is for a primitive AE (meaning it consists of a single annotator), and that the annotator code is developed in Java. Also, it specifies the Java class that implements the logic. Finally, on the right side of the page are listed some descriptive attributes of the annotator.

The other page that needs to be filled out is the Capabilities page. We can switch to these pages using the tabs at the bottom of the Component Descriptor Editor. On the Capabilities page, we define our annotator's inputs and outputs, in terms of the types in the type system. The Capability page is shown in Figure B.5.

The IBUserEmailAnnotator is very simple. It requires no input types, as it operates directly on the document text – which is supplied as a part of the CAS initialisation. It only produces one output type (IBUserEmail), and it sets the value of the building feature on that type. This is all represented on the Capabilities page.

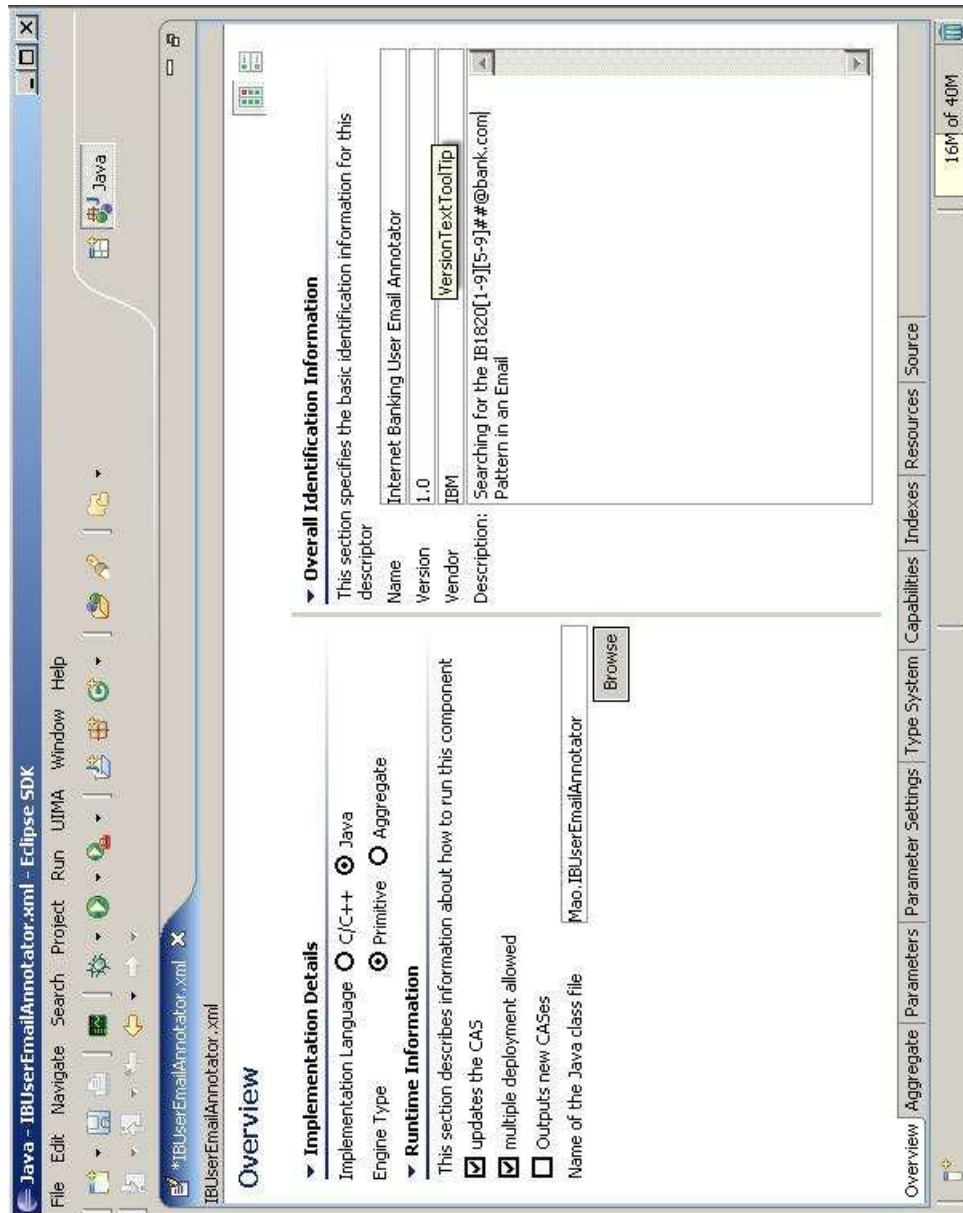


Figure B.4: UIMA SDK Component Descriptor Editor

Testing the Annotator

Having developed an annotator, we can test it on some example documents. The UIMA SDK includes a tool called the Document Analyser that will allow the annotator developers to do this. To run the Document Analyzer in an Eclipse project, execute the “UIMA Document Analyzer” run configuration supplied with that project. A screenshot of the Document Analyzer with its six options is shown in Figure B.6.

The six options are:

1. Directory containing documents to analyse
2. Directory where analysis results will be written
3. The XML descriptor for the TAE to be run
4. An XML tag, within the input documents, that contains the text to be analysed
5. Language of the document
6. Character encoding

After filling in the six option fields with corresponding information, we can click on ‘Run’ to start processing. When the document processing completes, an “Analysis Results” window (shown in Figure B.7) should appear.

We can double-click on a document to view the annotations that were discovered. This view is shown in Figure B.8

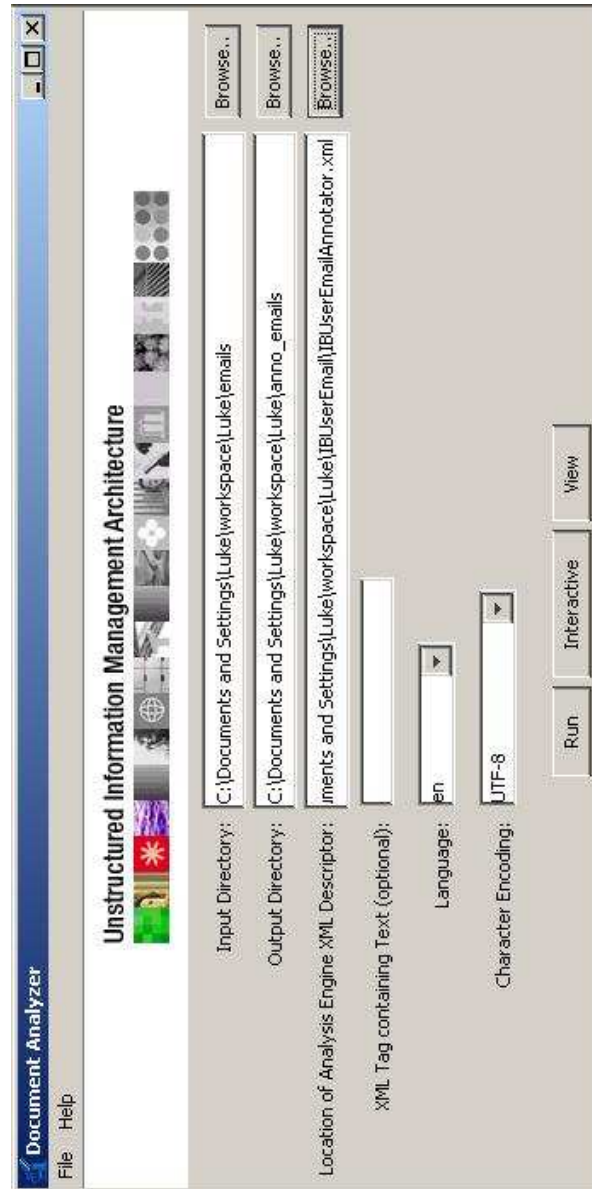


Figure B.6: the UIMA Document Analyzer

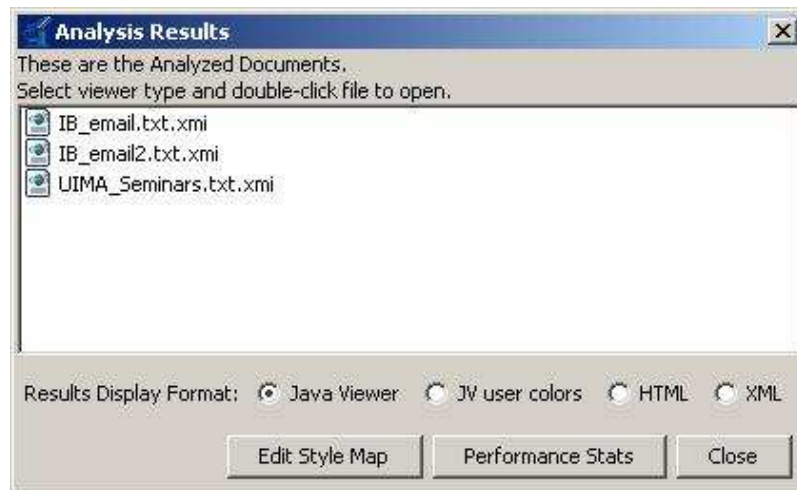


Figure B.7: the UIMA 'Analysis Results' window

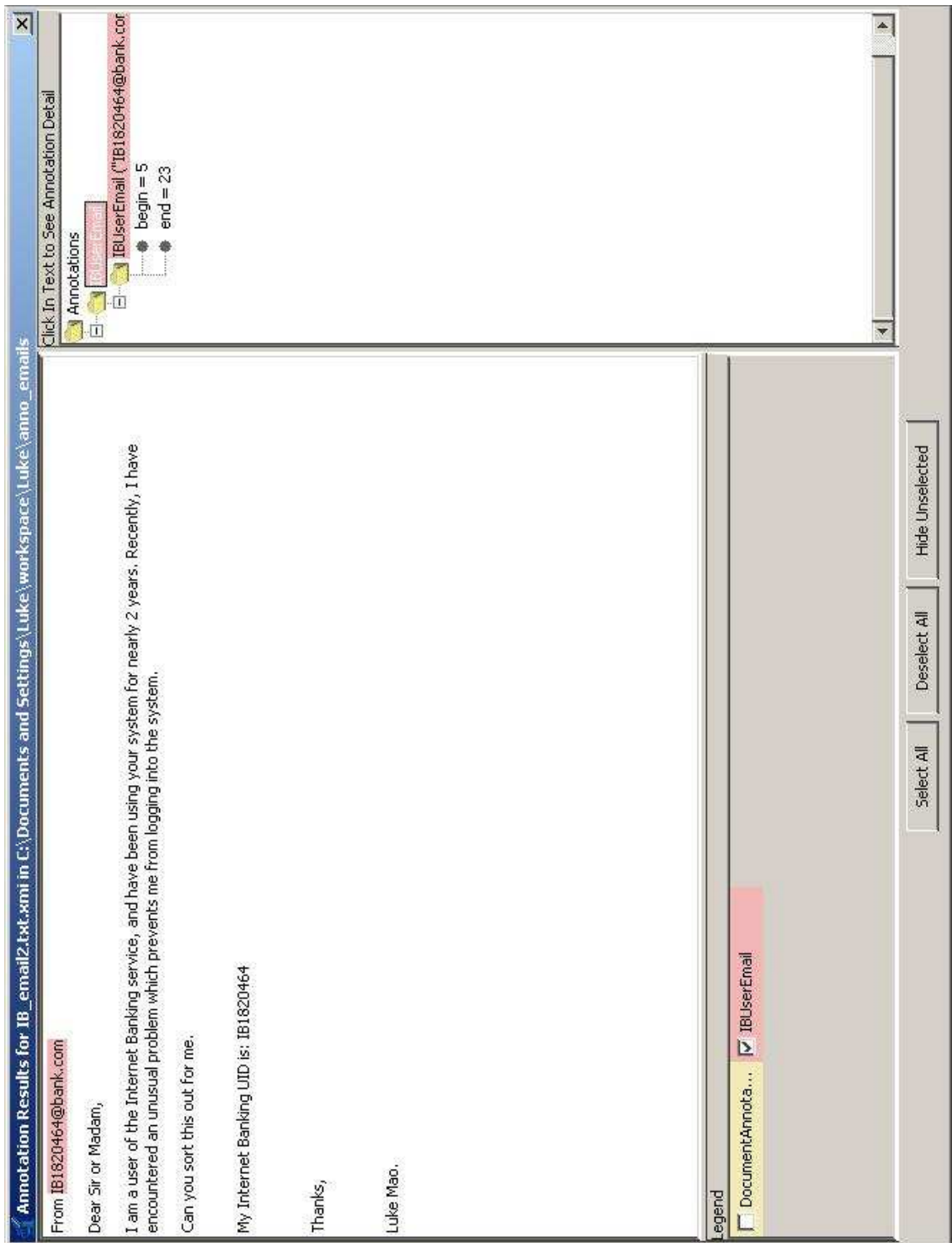


Figure B.8: Viewing the result of annotation

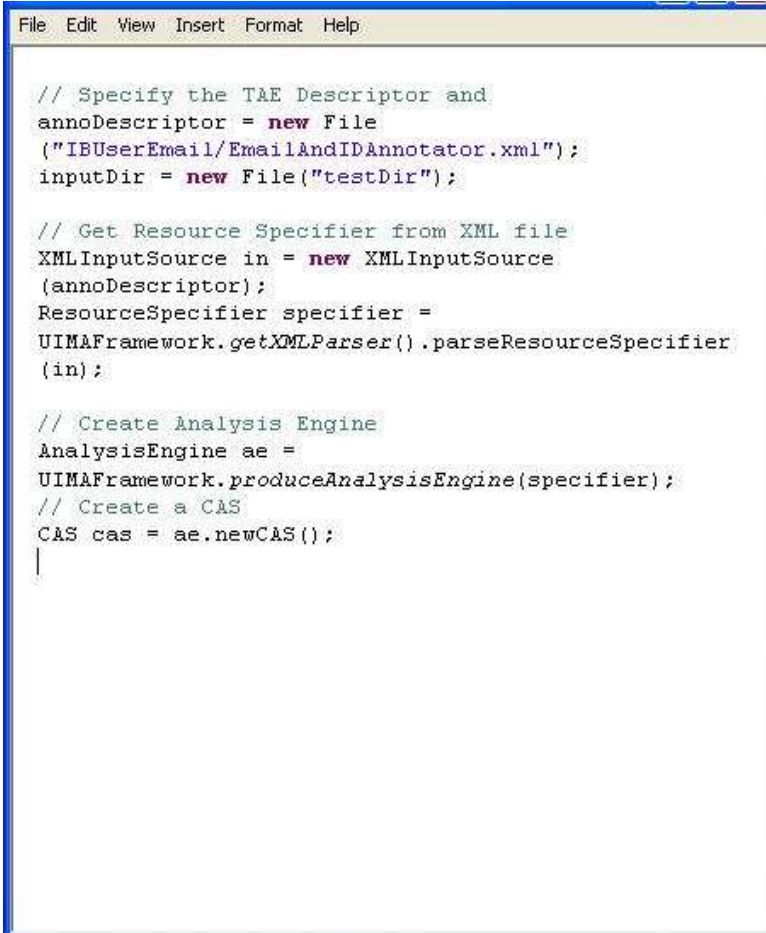
Appendix C

Populating Database with Unstructured Data

The procedure below describes how to populate a table in database with unstructured data. The code accompanied with each individual step is the Java implementation of that step. This procedure take a directory of text documents as input, annotate them by the text analysis engine of Lindas, and populated linking data into database. We assume that all the document contain the ID of a user as well as his email address.

Step 1: Prepare input

The Annotator Descriptor and the path of the directory that hold all the unstructured documents are given to the Text Analysis Engine (TAE) for discovering ID and email address. The Java implementation for this step is shown in Figure C.1.

A screenshot of a code editor window with a blue border. The window has a menu bar at the top with the items "File", "Edit", "View", "Insert", "Format", and "Help". The main area contains Java code with syntax highlighting. The code is as follows:

```
// Specify the T&E Descriptor and
annoDescriptor = new File
("IBUserEmail/EmailAndIDAnnotator.xml");
inputDir = new File("testDir");

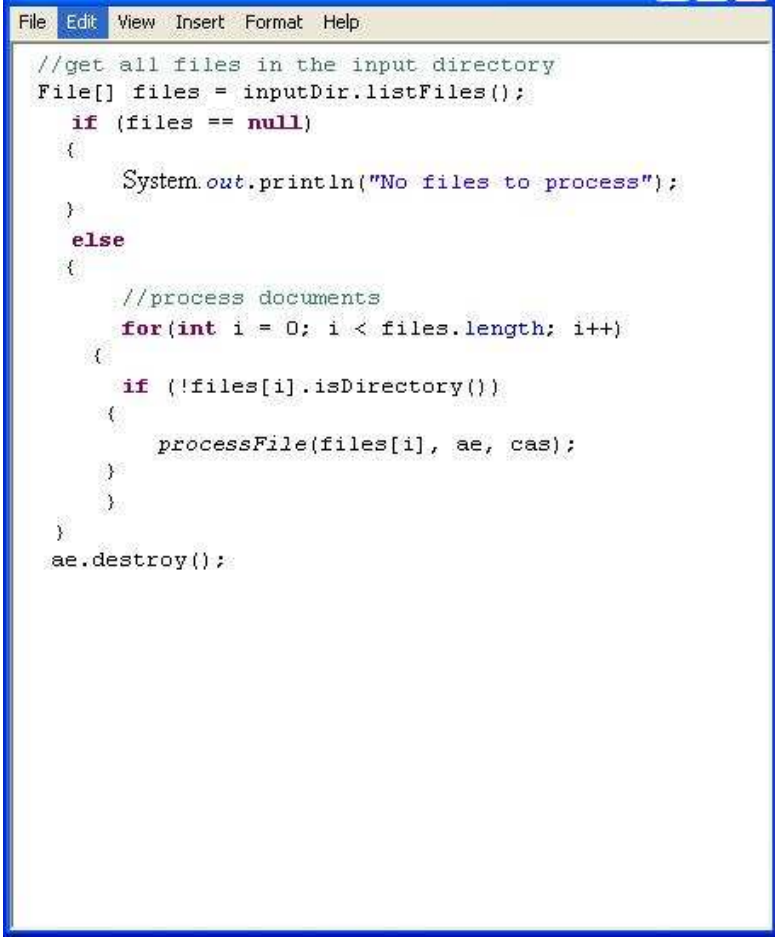
// Get Resource Specifier from XML file
XMLInputSource in = new XMLInputSource
(annoDescriptor);
ResourceSpecifier specifier =
UIMAFramework.getXMLParser().parseResourceSpecifier
(in);

// Create Analysis Engine
AnalysisEngine ae =
UIMAFramework.produceAnalysisEngine(specifier);
// Create a CAS
CAS cas = ae.newCAS();
|
```

Figure C.1: Step 1: prepare input

Step 2: Analysis unstructured documents

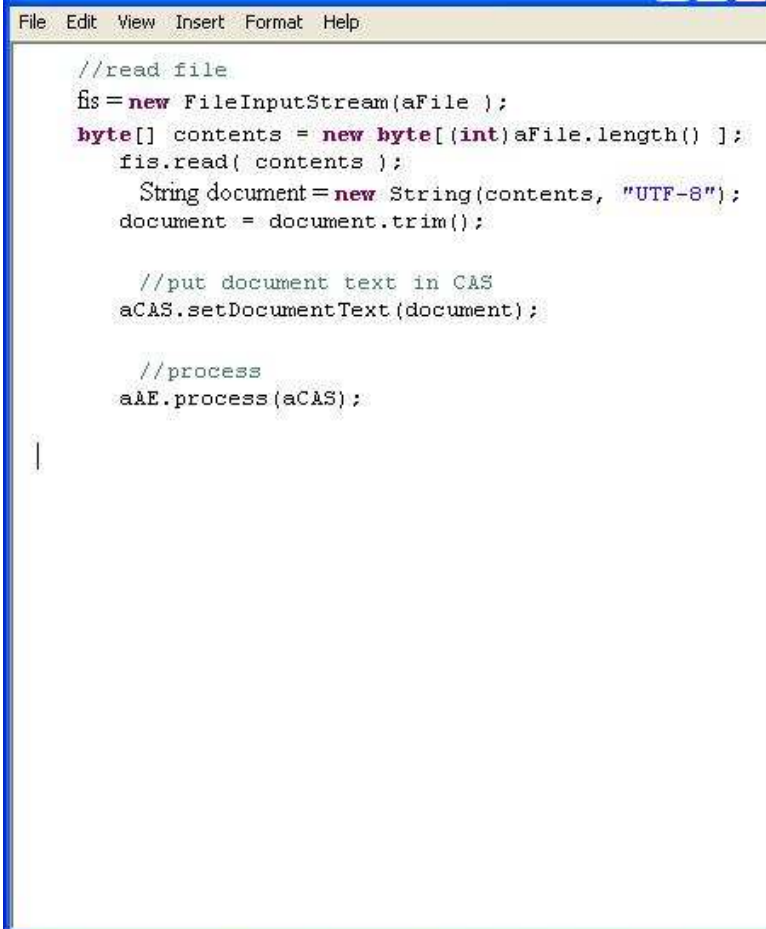
In this step, The TAE loop through each file in the directory, and then process (text-analyse) the file. The Java implementation for this step is shown in Figure C.2.

A screenshot of a Java IDE window with a menu bar (File, Edit, View, Insert, Format, Help) and a text area containing Java code. The code lists files in a directory and processes each file if it is not a directory. The code is as follows:

```
//get all files in the input directory
File[] files = inputDir.listFiles();
if (files == null)
{
    System.out.println("No files to process");
}
else
{
    //process documents
    for(int i = 0; i < files.length; i++)
    {
        if (!files[i].isDirectory())
        {
            processFile(files[i], ae, cas);
        }
    }
}
ae.destroy();
```

Figure C.2: Step 2a: Analysis unstructured documents

The `processFile` method start to process each file, output the annotation as an XML file (Step 3), and then process the annotated file (Step 4). The code shown below is the first part of the `processFile` method. The Java implementation for this step is shown in Figure C.3.



```
File Edit View Insert Format Help

//read file
fis = new FileInputStream(aFile );
byte[] contents = new byte[(int)aFile.length() ];
fis.read( contents );
String document = new String(contents, "UTF-8");
document = document.trim();

//put document text in CAS
aCAS.setDocumentText (document);

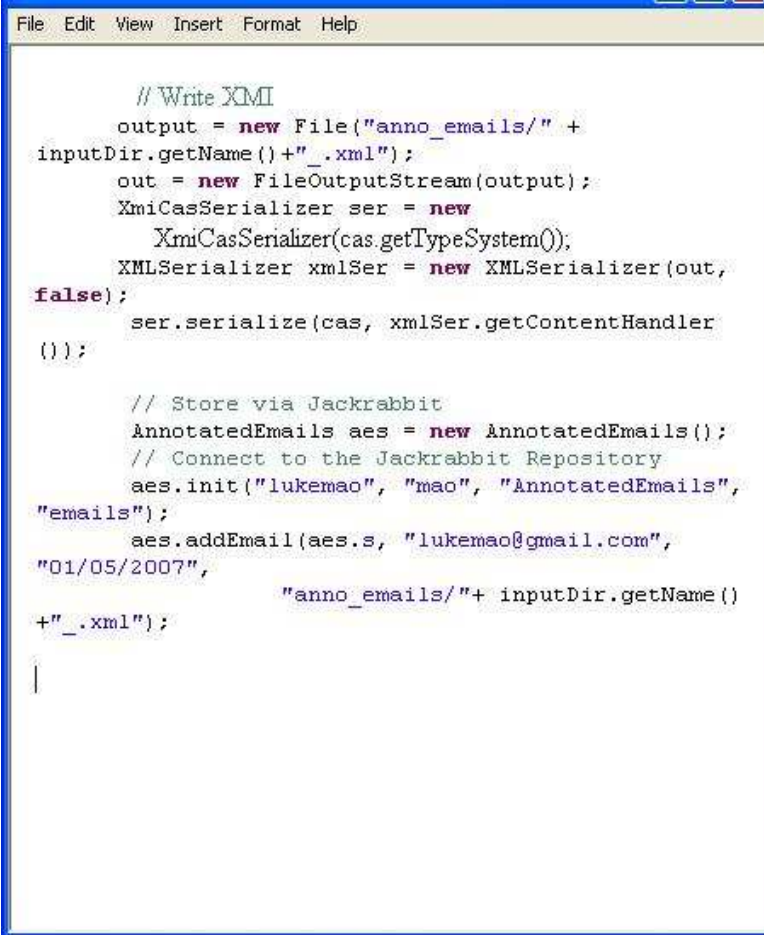
//process
aAE.process (aCAS);

|
```

Figure C.3: Step 2b: Analysis unstructured documents

Step 3: Output annotated documents

We store annotation output (the original text, all the found entities and their region appears in the original text) into an XML documents. The Java implementation for this step is shown in Figure C.4.



```

File Edit View Insert Format Help

    // Write XML
    output = new File("anno_emails/" +
inputDir.getName()+".xml");
    out = new FileOutputStream(output);
    XmiCasSerializer ser = new
        XmiCasSerializer(cas.getTypeSystem());
    XMLSerializer xmlSer = new XMLSerializer(out,
false);
    ser.serialize(cas, xmlSer.getContentHandler
());

    // Store via Jackrabbit
    AnnotatedEmails aes = new AnnotatedEmails();
    // Connect to the Jackrabbit Repository
    aes.init("lukemao", "mao", "AnnotatedEmails",
"emails");
    aes.addEmail(aes.s, "lukemao@gmail.com",
"01/05/2007",
        "anno_emails/" + inputDir.getName()
+".xml");
    |

```

Figure C.4: Step 3: Output annotated documents

Step 4: Process annotated unstructured documents

The output XML document can be processed by a document parser (e.g., the Simple API for XML). The Java implementation for this step is shown in Figure

C.5.



```

File Edit View Insert Format Help
SAXParserFactory factory =
SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse(output, new Correlator() );

... ..

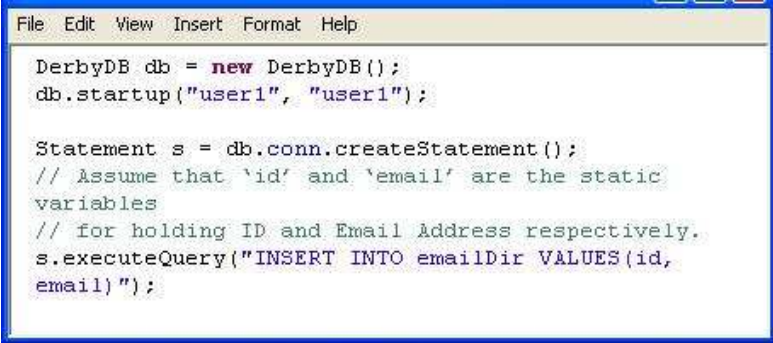
public void startElement (String name, AttributeList
attrs) throws SAXException{
if(name.equals("cas:Sofa")){ ... }
else if(name.equals("Mao:IBUserEmail")){ ... }
else if(name.equals("Mao:InternetBankingID")){ ... }

```

Figure C.5: Step 4: Process annotated unstructured documents

Step 5: Store discovered unstructured data into relational database

An SQL INSERT query is created to store discovered ID and Email into relational database. The Java implementation for this step is shown in Figure C.6.

A screenshot of a Java IDE window with a menu bar (File, Edit, View, Insert, Format, Help) and a code editor. The code in the editor is as follows:

```
DerbyDB db = new DerbyDB();
db.startup("user1", "user1");

Statement s = db.conn.createStatement();
// Assume that 'id' and 'email' are the static
variables
// for holding ID and Email Address respectively.
s.executeQuery("INSERT INTO emailDir VALUES(id,
email)");
```

Figure C.6: Step 5: Store discovered unstructured data into relational database

Appendix D

Implementation of the Correlation Query Evaluation

The Correlation Query Evaluation methodology proposed in this project comprised of a content search for retrieving all the documents from content repository that contain the Linking Data, as well as related structured data from databases. In the content search task, all the documents that contain the linking data in question are retrieved according to the following steps:

1. The issuer of the correlation query asks for all the documents that contain an IB_Email entity with type Email and has value: “x @gm”.
2. We firstly scan down the Correlation Table Registry (CTR) to find which Correlation Table (CT) that holds the annotation of IB_Email. The CT search can be narrowed to the the attribute name “IB_Email” or more generally to any attribute with domain “Email” (e.g., maybe another CT has the email value “x@gm”, but as a value in a domain named, e.g., “Internet_Banking_Email”). This is also why we need to remember the entity type in CTR.

3. We then take each CT (found in the previous step) to search for document identifiers such that the documents pointed to by them contain the linking data.
4. By using all the returned document identifiers, we can subsequently reach the original document using the content repository API.

This process involved in the evaluation of correlation queries are assisted by the Correlation Overhead. The code shown in Figure D.1 is the implementation of the content search part of the Correlation Query.

```

File Edit View Insert Format Help

//The Correlation Overhead is stored in the Database(db)
Statement s = db.conn.createStatement();

//CORRELATION TABLES SEARCH
//STEP 1: Search CT_Registry
ResultSet rs = s.executeQuery(
    "SELECT t_name, annotation FROM CT_Registry");
String CTName = "";
String annotation = "";
while( rs.next()){

    //index of the result set start from 1 ---
    CTName = (String)rs.getObject(1);
    annotation = (String)rs.getObject(2);

    //found a Correlation Table that has annotation (Entity Type)

    if(ld.getMetadata().toUpperCase().equals(annotation)){
        //STEP 2: Search Correlation Table 'CTName'
        Statement sCT = db.conn.createStatement();
        System.out.println("Start Timer: :: ::STEP 2: Search CTs");
        ResultSet rsCT = sCT.executeQuery(
            "SELECT path, "+annotation+" FROM "+CTName+
            " C, Document_Directory D WHERE "+annotation+" =
            '"+ld.getData()+"' AND C.d_id = D.d_id");

        while( rsCT.next() ){
            String path = (String)rsCT.getObject(1);
            //STEP 3: RETRIEVE THE DOCUMENT VIA JACKRABBIT
            ... ..
        }

        rsCT.close();
    }
}
//STEP 1 END
s.close();

```

Figure D.1: the Correlation Query Evaluation Implementation

Appendix E

Lindas GUI Demo

In this appendix, a simple demonstration on the Correlation Query GUI in Lindas is shown. When a user wants to issue a Correlation Query, he firstly needs to run the Correlation Query Control Panel. The initial view of the GUI is displayed in Figure E.1.

As shown in the screenshot E.1, user has to input the Linking Data in the Correlation Query panel. After filling the Linking Data, press Enter to execute the query, and then the output document contents and structured data will be shown in the right displays (as shown in E.2).

Apart from the function of issuing correlation queries, user can also use the Correlation Query Panel to create annotators. After given the name, type and regular expression of the annotator (enter into the corresponding text fields), Lindas will automatically create the annotator and print out its information in the Annotator Info display:

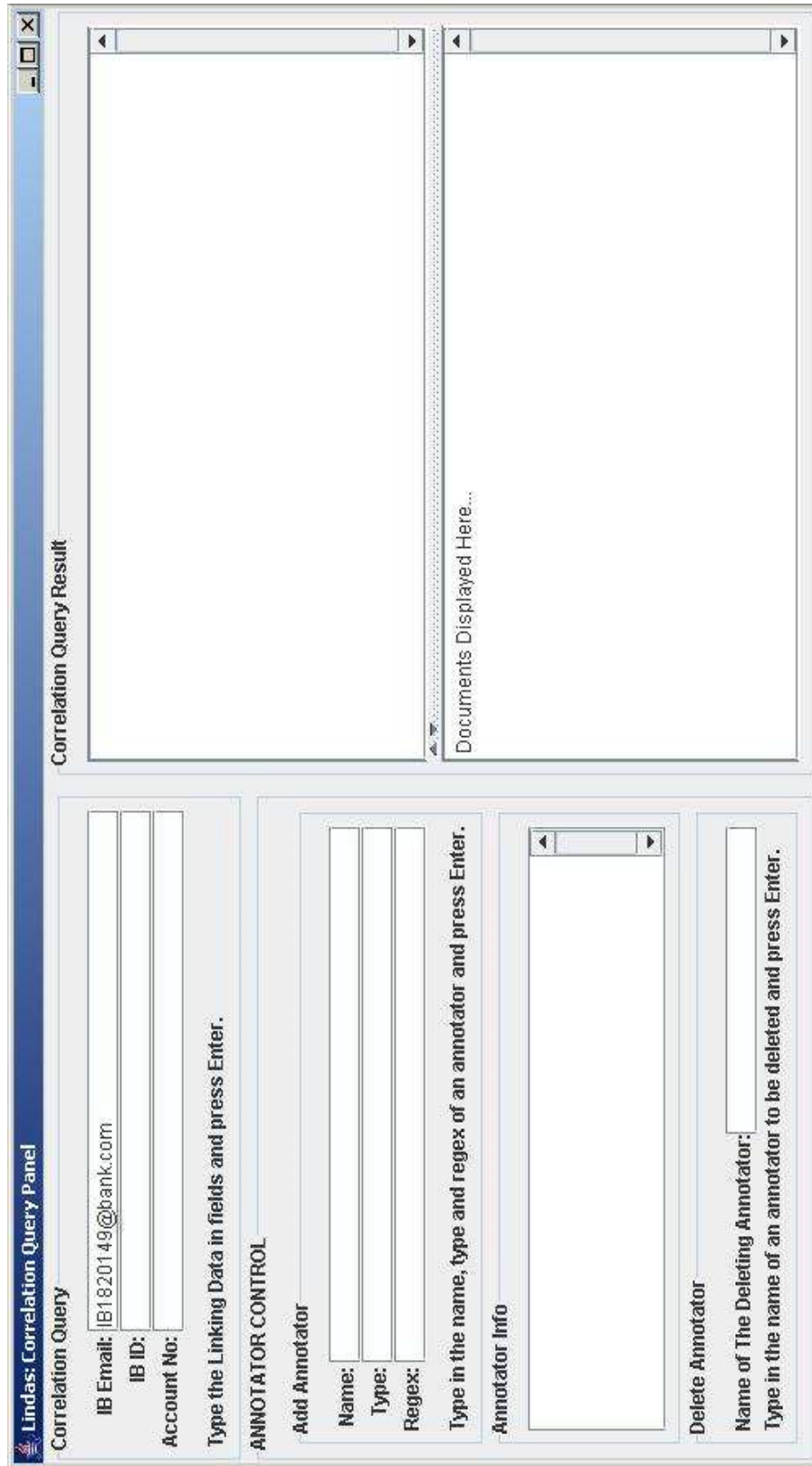


Figure E.1: The Correlation Query Control GUI (a)

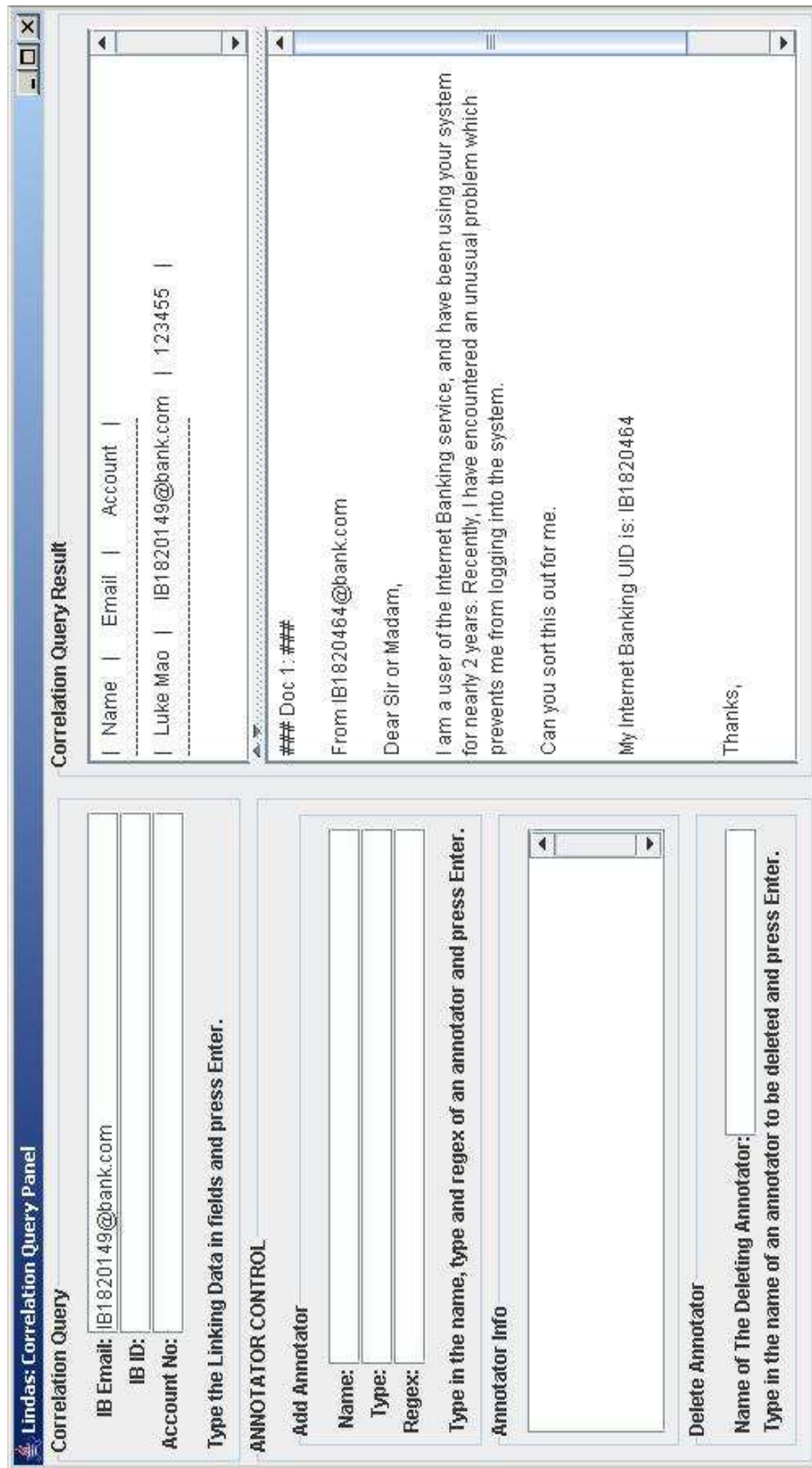


Figure E.2: The Correlation Query Control GUI (b)

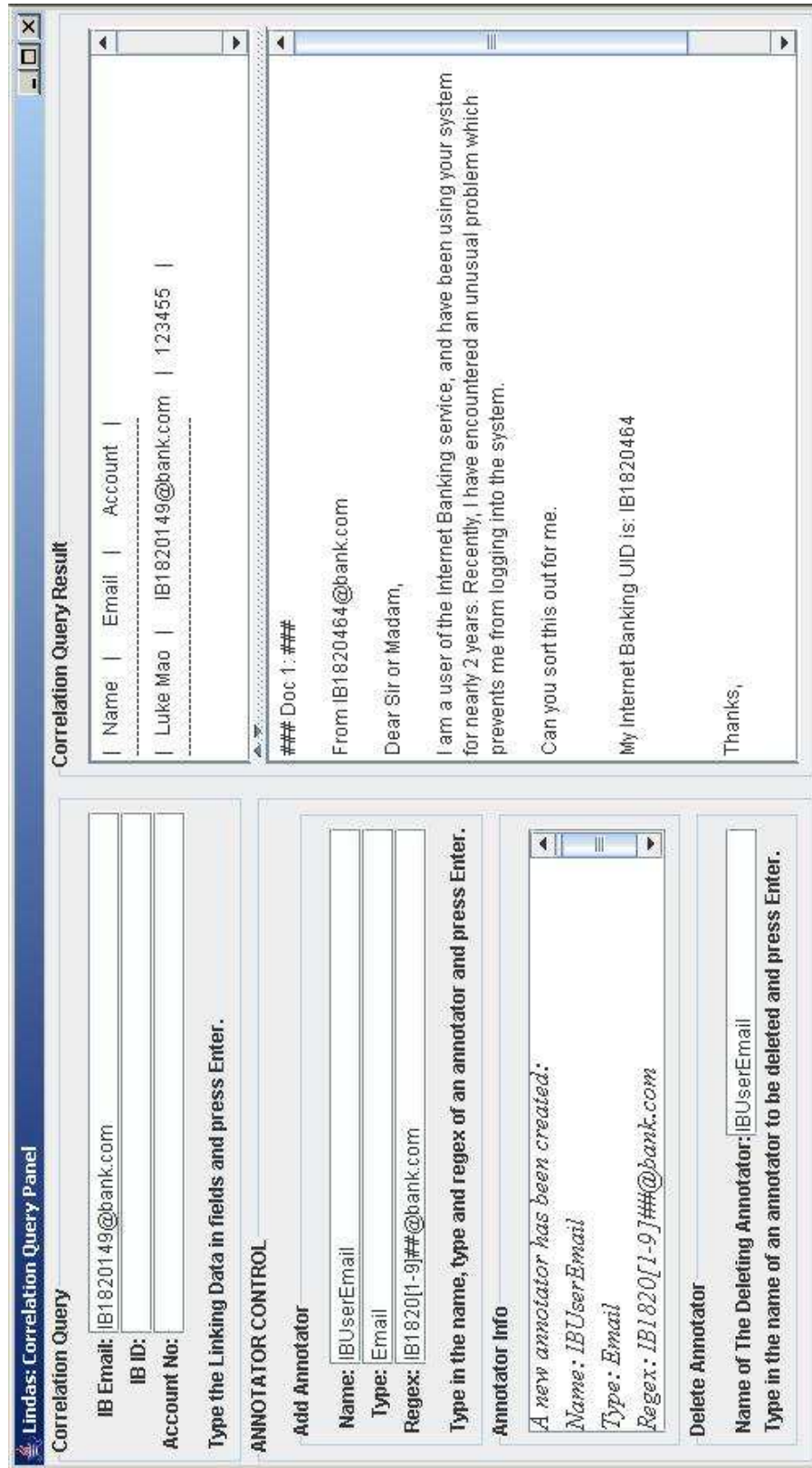


Figure E.3: The Correlation Query Control GUI (c)