

DATA ANNOTATION MODELS AND
ANNOTATION QUERY LANGUAGE

A Thesis
Presented
to the Faculty of
California State University, Chico

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
© Neerja Bhatnagar 2006
Spring 2006

DATA ANNOTATION MODELS AND
ANNOTATION QUERY LANGUAGE

A Thesis

by

Neerja Bhatnagar

Spring 2006

APPROVED BY THE INTERIM DEAN OF THE SCHOOL OF
GRADUATE, INTERNATIONAL, AND INTERDISCIPLINARY PROGRAMS:

Susan E. Place, Ph.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

Benjoe A. Juliano, Ph.D., Chair

Renee S. Renner, Ph.D.

PUBLICATION RIGHTS

No portion of this thesis may be reprinted or reproduced in any manner unacceptable to the usual copyright restrictions without the written permission of the author.

For Mummy-Papa

ACKNOWLEDGMENTS

I would like to extend my heartfelt thanks and gratitude to Dr. Benjoe A. Juliano for his moral support, time, energy, and faith. I would not have been able to accomplish this important milestone in my life without his efforts and support. Dr. Juliano motivated me and guided me throughout this research, as well as when I signed up for his classes in Theory of Computing and Advanced Computer Architecture. He was always patient with me, and constantly inspired me to work harder. Although he was on sabbatical during the crucial months of this project, he continued with his extraordinary efforts and support.

I would especially like to thank Dr. Renee S. Renner for her help, support, and guidance. This endeavor would not have been successful without her dedication and invaluable inputs. Despite being on sabbatical Dr. Renner continued to support me and guide me through this thesis.

I would like to express my special heartfelt thanks to Anshul Dawra for his continuous support, constant encouragement, and infinite patience during all my endeavors. Anshul has always put unconditional faith in me, and my abilities. I would especially like to thank him for answering infinite questions, for reviewing the chapters in this thesis, for discussing ideas and concepts, for holding endless discussions in order to help me form ideas. I would also like to express my gratitude to him for providing valuable and timely technical support and help.

I would like to express gratitude to Dr. Phokion G. Kolaitis for teaching me Introduction to Database Systems. I would like to thank Marie Taylor-Harper for introducing me to XML and XML-related technologies. I am really thankful to Dr. Suresh Lodha and Dr. Kevin Karplus for their insights on technical writing.

Last, but not the least, my heartfelt thanks to my Mummy and Papa for their continued support, encouragement, and faith in me. My Mummy and Papa have always believed in me, and have always been there for me both through thick and thin.

TABLE OF CONTENTS

	PAGE
Acknowledgments	v
List of Figures	ix
Abstract	xii
CHAPTER	
I. Introduction.....	1
Proposed Solution	3
Data Annotations Use Case Scenarios.....	4
Organization	8
II. Literature Review.....	9
SLIMPad.....	9
Annotating Scientific Images.....	12
Annotating Audio-Visual Units	16
Analysis	20
III. Data Annotation Models	24
Assumptions	24
Why Extensible Markup Language (XML)	27
Data Annotation Models	28
IV. Annotation Query Language.....	48
The Example <code>REAL_ESTATE</code> Database.....	48
Naïve Storage Scheme.....	51
AnQL Query Engine.....	54
AnQL Query Operations	62
Processing Overhead	84
Limitations of AnQL	84
Related Work.....	85

CHAPTER	PAGE
V. Preliminary Design of a Data Annotation Management System	87
Components of Data Annotation Management System	87
States of Data Annotation Management System	89
VI. Conclusions and Future Work	94
Future Work	98
References	100

LIST OF FIGURES

FIGURE	PAGE
1. Data Annotation Model at the Database Level.....	29
2. Validation Module of Data Annotation Model	30
3. Identification Module of Data Annotation Model.....	30
4. Level Module of Data Annotation Model	31
5. Annotation Module of Data Annotation Model.....	31
6. Annotation Metadata Module of Data Annotation.....	31
7. Cross-Reference Module of Data Annotation Model.....	32
8. Example Database-Level Data Annotation Document	34
9. Data Annotation Model at the Relation Level	35
10. Level Module at the Relation Level	36
11. Example Relation-Level Data Annotation Document	37
12. Data Annotation Model at the Column Level.....	38
13. Level Module at the Column Level.....	38
14. Relations <code>RECIPE_LIST</code> and <code>CONDIMENT_LIST</code> (<code>COOKING Database</code>).....	39
15. Relation <code>CONDIMENT_LIST</code> with column <code>ANNOTATION_QUANTITY</code>	39
16. Example Column-Level Data Annotation Document	41
17. Data Annotation Model at the Tuple Level	42

FIGURE	PAGE
18. Level Module at the Tuple Level	43
19. Relations NEW_STUDENT_INFO with ADMIT_INFO (NEW_STUDENT Database)	44
20. Example Tuple-Level Data Annotation Document.....	45
21. Data Annotation Model at the Cell-Level.....	46
22. Level Module at the Cell Level	47
23. Example Cell-Level Data Annotation Document	47
24. Example REAL_ESTATE Database	49
25. Example Data Annotation Document (REFeaturesLotSizePI2)	51
26. Example Data Annotation Document (REFeaturesPropertyIdPI2).....	52
27. Storage Scheme	54
28. Database-Level Data Annotation Graph	57
29. Relation-Level Data Annotation Graph.....	58
30. Column-Level Data Annotation Graph	59
31. Tuple-Level Data Annotation Graph.....	60
32. Cell-Level Data Annotation Graph.....	60
33. Result Set of a <i>Data Annotation Graph Traversal</i> Function	61
34. Result Set of a <i>Transitive Closure of Data Annotation Graph Traversal</i> Function	62
35. Result Set of a <i>Select</i> Operation	64
36. Result Set of a <i>Project</i> Operation.....	66

FIGURE	PAGE
37. Result Set of a <i>Natural Join</i> Operation.....	69
38. Result Set after the Processing of <i>Natural Join</i>	74
39. Result Set after the Processing of <i>Project Clause</i>	75
40. Result Set of a <i>Select-Project-Natural Join</i> Query	75
41. Result Set of a <i>Union</i> Operation.....	77
42. Result Set of a <i>Select-with-Predicate-Project-with-Constraint</i> <i>SPJ</i> Query	79
43. Result Set of a <i>Select-with-Predicate-Project-without-Constraint</i> <i>SPJ</i> Query	81
44. Result Set of a <i>Select-without-Predicate-</i> <i>Project-without-Constraint SPJ</i> Query	82
45. Combination Queries	83
46. Creating a Data Annotation Document.....	90
47. Modifying a Data Annotation Document.....	91
48. Querying a Data Annotation Document.....	92
49. Querying Base Data.....	92
50. Data Annotation Management System (Screen Shot).....	93

ABSTRACT

DATA ANNOTATION MODELS AND ANNOTATION QUERY LANGUAGE

by

© Neerja Bhatnagar 2006

Master of Science in Computer Science

California State University, Chico

Spring 2006

This thesis presents five different models to express data annotations at the database, relation, column, tuple, and cell levels of base data that resides in a relational database management system. This thesis also presents the Annotation Query Language (AnQL), a query language used to query data annotations. AnQL's fundamental query operations include *select*, *project*, *natural join*, and *union*. AnQL's query engine is an integral part of AnQL. The main operations of the AnQL query engine include *data annotation graph generation*, and *data annotation graph traversal*.

This thesis also presents a preliminary design of a data annotation management system. A data annotation management system allows data annotation users to create, update, query data annotation documents to query base data. The data annotation management system can be visualized as

running on top of the underlying relational database that stores base data. The data annotation management system, although independent of the underlying relational database, runs in conjunction with the underlying relational database system.

CHAPTER I

INTRODUCTION

The metadata schema provided by most relational database management systems is not suitable for expressing *data annotations*. Data annotations are defined as semantically rich metadata, applicable to a particular application domain, that help further clarify *features of interest* [3][13]. A feature of interest is a data item that a user wants to annotate. Types of data annotations include comments, descriptions, definitions, notes, and error messages, among others. In general, most database management systems either do not allow or do not recommend their users to update the data contained in the metadata schema [1]. A majority of the database management systems utilize the metadata schema to store statistical information for constraint checking and query optimization.

As an example, consider the metadata schema provided by IBM DB2 Universal Database (UDB). DB2 UDB automatically maintains and updates the system catalog tables that contain metadata [1]. The `RUNSTATS` utility in DB2 UDB automatically generates statistics for query optimization. The `SYSCAT` schema defines two catalog views – `SYSCAT` and `SYSSTAT` [1]. Most DB2 UDB users are not allowed to update the `SYSCAT` catalog view. However, certain privileged users who are allowed to update the `SYSSTAT` catalog view, manually,

are only allowed to influence the system optimizer [1]. `SYSCAT` catalog views include views like `COLDIST` (each row describes the n^{th} most frequent value of a column or n^{th} quantile value for a column), `KEYCOLUSE` (lists all columns that participate in a key defined by a unique key, primary key, or foreign key constraint) [1]. `SYSSTAT` catalog views include views like `COLDIST` (contains statistics gathered about distribution of data values in columns of base tables to be used by the query optimizer), and `COLUMNS` (contains information about values stored in the columns) [1].

Microsoft SQL Server behaves in a manner similar to IBM DB2 UDB with respect to metadata schema. SQL Server's system tables include tables named `sysforeignkeys`, `sysindexes`, `sysobjects`, `syscolumns`, and `sysconstraints` [2]. These system tables store information about tables, columns, and constraints. These system tables also store access permissions for objects in a database or table [2]. SQL Server provides information schema views in order to retrieve metadata about constraints, columns, tables, and views from system tables [2]. SQL Server also provides system stored procedures to query system tables [2]. However, SQL Server does not ensure that these stored procedures will return the information requested by the user [2].

Hence, most database management systems are not suitable for expressing or querying semantically rich metadata. Moreover, the metadata schema provided by a majority of database management systems overwrites, and hence, loses vital information as soon as there is an update to the database.

Similarly, if the foreign key constraint of a table changes, the pertinent information in the `KEYCOLUSE` view of `SYSCAT` catalog view in IBM DB2 UDB will be updated automatically. However, in this process, the previous record of information is overwritten, and thus, lost.

Proposed Solution

This thesis presents a solution to the problem of the unsuitability of the metadata schema in a relational database management system to express semantically rich metadata. The solution presented in this thesis has three parts. The first part of the solution is to define data annotation models that allow the expression of data annotations at five levels. The five levels at which data annotation models allow data annotation users to express data annotations are database, relation, column, tuple, and cell. The main features of the data annotation models presented in this thesis are that these models are platform-independent and database-neutral, yet flexible and extensible. The data annotation models presented in this thesis do not require any structural and schematic changes to the underlying relational database, and therefore, do not affect query processing, or inflict any additional processing overhead on the underlying relational database.

The second part of the three-part solution presented in this thesis is the Annotation Query Language (AnQL). AnQL is a query language that allows data annotation users to query data annotation documents. AnQL's main functions include *select*, *project*, *natural join*, and *union*. AnQL's query engine

processes AnQL queries issued by data annotation users. The main functions of AnQL query engine include *data annotation graph generation* and *data annotation graph traversal*.

The third part of the solution presented in this thesis is the preliminary design of a data annotation management system. This data annotation management system allows data annotation users to create, query, and update data annotation documents. This data annotation management system also allows its users to view the data that they annotate. The data annotation management system can be visualized as a virtual system or a view running on top of the underlying relational database that stores the data being annotated by data annotation users. The proposed data annotation management system is a collaboration between data annotations and the underlying relational database.

Data Annotations Use Case Scenarios

Almost all database users, including, scientists, researchers, customers, customer service representatives, banks, and credit card companies etc. can benefit from data annotations. Data annotations can significantly reduce communication hassles (for example, coordinating time to call customer service centers). Thus, the use of data annotations presents several potential opportunities to save time and effort. This section presents potential use case scenarios of data annotations.

One of the potential utilization of data annotations is within the scientific community. Suppose a scientist makes a discovery while investigating

and interpreting an image. The scientist wants to share his or her finding(s) with the research community. In the absence of a data annotation management system, the scientist would send the image along with his or her findings via fax or email. However, using the data annotation models proposed in this thesis, the scientist would simply annotate the image with his or her finding(s), and share it seamlessly with his or her research community. When his or her colleagues in the research community make additional discoveries on that image, they can also add their findings via data annotations on the image. The data annotation management system would automatically record and save data annotations on the image, and hence, eliminate the need for separate record keeping. The data annotation management system would provide a medium for data interchange as well as automatic bookkeeping, and thus, save time, money, and effort for scientists and researchers.

Another potential use of data annotations would be to inform the database administrator regarding errors. Assume that an employee notices that the spelling of his or her last name and his or her birth year in his or her personal record is incorrect. The employee could simply annotate the cells that contain his or her last name and his or her birth date with an error message that also lists the corrections to be made. The database administrator can query data annotations and rectify the errors in the employee's record.

Similarly, a credit card customer, who notices a few illegal charges on his or her credit card statement, could simply annotate the relevant tuples with a

comment stating that the charges were illegal. A customer service representative at the customer service center could query the data annotations, launch an inquiry, and update the customer with the status via data annotations.

Scientific database users can also potentially benefit from the use of data annotations. In many cases, data stored in scientific databases is comprised of numeric values. An example of such data stored in a scientific database is the temperature recorded by a buoy on a particular day. In most cases, the data type of such a column would be numerical, such as, `INTEGER` or `DECIMAL`. However, simply by looking at the data, it is difficult to interpret whether the observed temperature is expressed using English units or metric units. One way to overcome this problem is to store the unit of the recorded temperature along with the temperature. This method would require changing the data type of the temperature column to `VARCHAR`. The `VARCHAR` data type is unsuitable for mathematical calculations. The loss of the ability to mathematically manipulate recorded temperature might not be desirable for scientific database users. Another alternative to this problem of interpreting the units of recorded temperature is to declare two separate columns in the relation – one that records observed temperature in centigrade, and another one that records the temperature in Fahrenheit. This method does not make the data stored in these columns unsuitable for mathematical manipulation. However, this solution is also not desirable because the method introduces redundancy, and causes space wastage due to the occurrence of several null values in both columns. In such a

case, the use of data annotations can prove very useful – users can simply annotate the numerical values in the temperature column with their respective units by using the data annotation models proposed in Chapter III.

The examples presented above indicate that potentially almost all database users can benefit from the use of data annotations. Data annotations can help database users reduce communication hassles, and in some cases, also reduce the costs related with communication. A potential cost reduction opportunity for a credit card company that allows its users to communicate via data annotations would be to reduce the number of telephone lines that the company would have to monitor and maintain in order to provide customer service. A credit card company that utilizes data annotations would not have to provide any additional access privileges to allow their customers to annotate data. The reason for this is that the customers do not need to make any changes to their statements in order to annotate them. The customers simply need a mechanism to annotate the data that they want the credit card company to focus on. Moreover, the credit card company would not need to make any schematic or structural changes to the underlying database. It is common knowledge that organizations are highly resistant to making schematic or structural changes to deployed databases. This is because most organizations customize and optimize their database deployments based on the organization's query needs. Making schematic or structural changes to the database might adversely affect query processing and query optimizations made by the organization. Thus, data

annotations not only facilitate smoother on-demand communication, but also provide several potential opportunities for cost reductions, without making any structural or schematic changes to the underlying database.

Organization

The rest of this thesis is organized as follows – Chapter 2 presents literature review; Chapter 3 presents the proposed data annotation models that can be used to express data annotations on the database, relation, column, tuple, and cell level; Chapter 4 presents the proposed query language, Annotation Query Language, and its operations; Chapter 5 presents preliminary design of a data annotation management system; and Chapter 6 presents conclusion and future work.

CHAPTER II

LITERATURE REVIEW

This chapter presents the review of existing related concepts and technologies. The techniques discussed in this chapter include a technique to annotate scientific images, a technique to annotate audio-visual units, and SLIMPad, which is an application that automates sticky notes.

SLIMPad

Delcambre *et al.* [10] propose an architecture for the management of superimposed information. Some examples of superimposed information include citation indices, and commentaries etc. The development of this architecture was inspired by the authors' observation of clinicians working at a hospital's intensive care units. While developing this architecture, Delcambre *et al.* follow three basic design principles. These design principles include keeping the architecture of superimposed information lightweight and flexible, and making minimum assumptions about the base layer. In order to achieve the goal of keeping superimposed information lightweight, the superimposed information is designed to exist as a thin layer over the more extensive base layer. In order to ensure flexibility of the architecture, Delcambre *et al.* did not wire the superimposed information layer to a single data model. Moreover, Delcambre *et al.* make only

two minimal assumptions about the base layer in order to achieve their third goal. The assumptions made by the authors are that the base layer is capable of providing the address of a particular information element, and that the base layer is also capable of returning an information element if its address is provided.

In addition to proposing an architecture for the management of superimposed information, Delcambre *et al.* also propose a prototype application, named Superimposed Layer Information Manager scratchPad (SLIMPad), that implements this architecture. SLIMPad is an application that represents and organizes *bundles* digitally. A bundle is defined as a grouping of information selected, collected, elaborated, and structured by a user during problem-solving. The main characteristics of bundles are that they do not have a pre-defined structure or content. Typical examples of bundles include notes jotted down on the back of an envelope or a piece of paper, or entries printed on a printed card. The notes jotted down may or may not be organized into headings and groups. SLIMPad utilizes the RDF superimposed metamodel. RDF triples represent superimposed model, schema, and instance data. An RDF triple consists of a property, a resource, and a value.

SLIMPad's model is composed of four major entities. These entities are – SLIMPad, *bundle*, *scrap*, and *MarkHandle*. SLIMPad is the top-level object, and designates a root bundle. Each bundle is represented by a label and a position. A bundle may contain zero or more scraps or bundles. A scrap, also referred to as an information element, is characterized by a label and a

MarkHandle object. A MarkHandle object is characterized by a *mark identifier*.

A mark identifier refers to a mark object inside the Mark Manager. A mark contains the address of the marked information element, and the Mark Manager is responsible for communicating with the base layer.

SLIMPad's Data Manipulation Interface (DMI) allows users to create, update, remove, store, and load bundles and scraps. In order to create a new scrap, a SLIMPad user chooses an information element using a base-layer application. The base-layer applications are modified in order to support the creation of marks. SLIMPad supports marks for Microsoft Excel spreadsheets, Microsoft Word documents, Microsoft PowerPoint presentations, XML documents, Adobe PDF documents, and HTML pages. A mark is attached to a scrap. The attaching of a mark to a scrap signifies the creation of a computerized "sticky-note".

SLIMPad can interact with base-layer applications in three ways – simultaneous viewing, enhanced base-layer viewing, and independent viewing. During simultaneous viewing, SLIMPad users are able to access both the superimposed information layer and the base-layer applications at the same time. SLIMPad achieves this by showing two active windows on the computer screen simultaneously. One window represents the superimposed layer and another window represents the base layer. While in the enhanced base-layer viewing mode, the functionality of a base-layer application is enhanced in order to manage the superimposed information. In the independent viewing mode, the

base layer applications are hidden, and the user can only see the superimposed information layer. Currently, SLIMPad supports simultaneous viewing only.

SLIMPad, with its scratch pad-like look and feel, does not impose any structure on scraps and bundles. SLIMPad allows superimposed information to exist in a freeform over elements that reside in multiple and heterogeneous base layers. SLIMPad also allows the linking of superimposed information back to the original contexts of elements. SLIMPad allows its users to construct bundles manually, and does not impose any rules or views on its users in order to construct bundles or information elements. SLIMPad also does not require that information elements pre-exist in a dictionary.

Annotating Scientific Images

Gertz *et al.* [3] define an extensible model, and its components, that defines semantic rich metadata schemes in order to describe instances of features discovered in neuroanatomical images at various levels of granularity. This extensible model is specifically geared towards neuroscientists, and employs domain-specific concepts as metadata schemes for the description of features of interest in neuroanatomical images. Besides enabling the annotation of images at a fine granularity, instead of just whole images, the model also allows various text-based data retrieval scenarios.

The proposed model keeps data annotations separate from the images that the annotations annotate. Images are stored in an image repository, and are assumed to be web accessible. A URI locates and identifies an image. The

separate storage of images and annotations enables multiple annotations of the same image using different concepts.

The main steps involved in the annotation of an image, or parts thereof, are the identification of features of interest in an image; the selection of a concept that provides a metadata template for annotating the feature of interest; and the instantiation of text-based metadata for the feature of interest through the process of data annotation.

The model described by Gertz *et al.* [3] utilizes an annotation graph model. An annotation graph model allows the expression of annotation concepts, annotations, and images. Annotation concepts, annotations, and images are represented as different types of nodes in an annotation graph. Annotation concepts, also referred to as concepts, provide templates for annotations that are associated with regions of interest in images. Concepts define a set of properties that must be instantiated during the definition of an annotation. Concepts are represented by concept nodes. A concept node is composed of four components – a concept identifier, a definition that gives the concept's meaning in natural language, synonyms for the concept, and a set of property definitions. Default concepts indicate specific meaning of different types of nodes and relationships among them. Default concepts include `annotates` (represents edges from annotations to images), `annotatedBy` (inverse of `annotates`, and represents edges from images to annotations), `ofConcept` (represents that an annotation is based on a certain concept), and `hasAnnotation` (inverse of `ofConcept`).

Annotation nodes provide the basis for specifying links between concepts and regions of interest in an image. An annotation node is characterized by an identifier, and a set of property instantiations. Although creational properties, such as author and date of creation, are associated with each type of node, these are not specified explicitly. Edges between the nodes describe relationships between the nodes. Typical relationships include how concepts are related, and how images are annotated using concepts. In order to avoid inconsistency, redundancy, and incompatibility, the model checks for compatibility between annotations and their underlying concepts. These checks are based on concept classification hierarchies. Concept classification hierarchies are of two types – spatial containment, and type-based classification. Whenever a new concept is created, the check mechanism determines the extent of similarity among the existing concepts and the new concept.

The annotation graph is queried using a query language modeled in the spirit of XPath. The query language proposed by Gertz *et al.* [3] consists of two operations – selection and path traversal. The selection operation takes as input annotations, concepts, and images, (but not their union), and a boolean predicate of the form `prop <op> value`, where `<op>` is a logical connective. The selection operation returns, as output, the subset of the input that satisfies the boolean predicate. The path traversal function enables the traversal of edges between nodes in an annotation graph. The path traversal operation takes as input a start node and a relationship type concept, and returns the set of target

nodes, based on respective edges. The transitive closure of the path traversal operation traverses the path indicated by the relationship as long as edges can be found that have not been previously visited. The model defined by Gertz *et al.* [3] also allows view mechanisms on annotation graphs. A view specifies a virtual sub-graph of an annotation graph, and is specified by formulating queries in the query language described above.

Gertz *et al.* [3] also describe a prototype application that implements the framework described above. This prototype application, implemented using client-server paradigm, provides services to define concepts, to assign annotations, and to formulate queries. The application, also referred to as the annotation system, is implemented using Java programming language, and utilizes Java Database Connectivity (JDBC) to access the underlying relational database. The interface to the annotation system is implemented using Single Object Access Protocol (SOAP). The annotation system displays annotations and concepts associated with an image via links that refer to details. Related concepts and images are also shown via links. The annotation system described by Gertz *et al.* [3] is composed of four major components – the graph mapping module, the query component, the consistency checker, and the search engine. The graph mapping module represents an annotation graph, and maps the nodes and edges of an annotation graph to relations that reside in a relational database. The query component of the annotation system is further comprised of three components – the query parser, the query translator, and the SQL builder.

The query parser parses the query language described above. The query translator, using a set of transformation rules, transforms the parsed query into an equivalent relational algebra expression. The SQL builder derives SQL queries from the relational algebra expressions generated by the query translator. The consistency checker checks for consistency, redundancy and compatibility whenever a new concept is defined. It notifies the user when a conflict occurs. The search engine, using the query engine, evaluates queries and displays corresponding results.

The annotation system consists of a concept browser and an annotation tool. The concept browser is used to browse and query concepts. The concept browser allows users to view concepts in tabular or tree formats. The annotation tool allows users to utilize the concept browser to select a concept for creating a new annotation. The annotation tool also allows users to mark features or regions of interest in an image, and to assign annotations to these chosen regions of interest based on a previously-selected concept.

Annotating Audio-Visual Units

Zsigmond *et al.* [12] developed a graph-based annotation and browsing system for audio-visual (AV) documents. The system presented by Zsigmond *et al.* is based on the theoretical AI-strata model. The AI-strata model is a general graph-based annotation model designed specifically for AV documents. This system is equipped with a user-friendly interface that provides a dialog-based interface as well as a Graphical User Interface (GUI) in order to

navigate annotation graphs. This interface is written in Visual C++ and allows the manipulation of annotations based on the AI-strata model. This system's internal data structures are based on the LEDA graph structure, and XML Document Object Model (DOM).

This annotation and browsing system enables users to annotate and exploit AV documents. Annotating an AV document is described as the action of associating a descriptor to an AV document, or a fragment thereof, or the result of its action. A descriptor is defined as a text describing something in the AV document. A descriptor should be locatable. Annotations are represented in Extensible Markup Language (XML). Exploitation of AV documents implies the ability of this annotation and browsing system to receive and process complex research requests. The main objective of the system is to obtain answers to these complex research requests. A typical example of an exploitation would be a research request to obtain all film sequences, covered by a particular television station, where Jacques Chirac is speaking with somebody.

The annotation and browsing system described by Zsigmond *et al.* [12] is based on the premise that linear search of the space that holds AV documents is not possible. This inability to conduct linear search of this space stems from two factors. The first factor is the large size of the AV documents, and the second factor is the large number of AV documents that reside in this space. Therefore, the annotation and browsing system described by Zsigmond *et al.* [12] searches the space via structured descriptors. In essence, the annotation and browsing

system can be thought of as a mechanism for creating, structuring, and exploiting These structured descriptors. The annotation and browsing system discussed by Zsigmond *et al.* [12] is built upon a graph-based annotation model. This system allows manual as well as automatic annotation of AV streams. Automatic annotations involve the extraction of low-level abstractions from AV streams. Examples of low-level abstractions include color histograms, movement detection, and camera effects. Manual annotation involves the visualization of a particular AV stream, and the manual attachment of suitable descriptor(s) at higher-levels of abstraction.

The basic graph that defines annotations primarily consists of three types of nodes – link elements, annotation elements, and abstraction annotation elements. The first type of nodes in the graph contains link elements. Link elements associate annotations with binary AV data. The audio-visual units (AVUs) nodes materialize AV segments in the graph. These nodes are characterized by the Uniform Resource Identifier (URI) of the AV stream, and the beginning and end of the strata on the AV stream's time scale. The second type of nodes in the graph is referred to as the Annotation Elements (AEs). AEs form the actual descriptors, and are represented by their names, and a set of attribute-value pairs. AEs derive from a knowledge base or a thesaurus. The knowledge base or thesaurus, also referred to as the Analysis Dimensions (AD) consists of Abstract Annotation Elements (AAEs), and is structured in form of a specialization/abstraction tree. AAEs form the third type of nodes in the

annotation graph. AAEs define the attributes of AEs. AAEs can be selected manually or automatically via designation methods. Designation methods are materialized by potential graphs that search AAEs dynamically. The main purpose of designation methods is to constrain the growth of AD. The nodes in the graph form a connected graph, and the labeled edges of the graph signify a set of relations that connect various elements. Typical relations used in a graph are time-related, and include, "after", "before", "during".

A research request, similar to the one described above, is expressed through a potential graph. A potential graph is a graph that consists of several partially-filled nodes. A research request received by the annotation and browsing system described by Zsigmond *et al.* [12] is processed by searching for a matching potential graph in the global graph. Searching algorithm utilized in this annotation and browsing system is described to be "any-time". In other words, the searching algorithm delivers results as soon as they are found, rather than when the searching algorithm has completed processing.

In order to annotate an AV document, the user of this annotation and browsing system begins with the creation of the knowledge base or the thesaurus. Standard tree manipulating functions are used to create and manage the knowledge base. After the creation of the knowledge base, the user proceeds to annotate the AV document. Annotating an AV document involves the creation of AEs, and the specification of an AV stream. AEs are created by selecting terms from AAEs. The AEs are then manually or automatically designated to

AVUs. The annotation and browsing system described by Zsigmond *et al.* [12] allows users to annotate a particular AV stream, or sub-segments thereof. A particular AV stream or its sub-segments may be annotated multiple times. If the user chooses an AV stream that has already been annotated, then the system's GUI displays the associated annotation.

Analysis

The superimposed information management application, SLIMPad, presented by Delcambre *et al.* [10], is not wired to a single data model. SLIMPad presents annotations in the context of physicians providing health care in a hospital's intensive care unit. The base layer of SLIMPad may consist of several applications, such as Microsoft Excel spreadsheets, Microsoft Word documents, Microsoft PowerPoint presentations, XML documents, and Adobe PDF documents. On the other hand, the annotation system presented by Zsigmond *et al.* [12] and by Gertz *et al.* [3] are wired to a single data model. The annotation model described by Zsigmond *et al.* [12] is based on the AI-Strata model, and allows the annotation of AV documents only. Annotations themselves are expressed using XML. The annotation model presented in Gertz *et al.* [3], on the other hand, uses simple text to express and store annotations. However, this model is also wired to a single data model, allows annotation of neuroanatomical images only.

Similar to these two models, the data annotation models presented in this thesis are also wired to a single data model, and allow the annotation of base

data expressed using the relational data model only. The data annotations themselves are expressed using XML.

Zsigmond *et al.* [12], restrict the definition of annotations with the existence of a pre-defined knowledge base or thesaurus. Similarly, Gertz *et al.* [3] restrict the definition of annotations with the existence of pre-defined concepts. The system described by Delcambre *et al.* [10], on the other hand, allows freeform annotations. Additionally, this system allows annotations to cross-reference each other. In contrast, annotations presented by Zsigmond *et al.* [12] and Gertz *et al.* [3] are not capable of cross-referencing each other.

The content or the context of data annotation documents presented in this thesis are not restricted by concepts or a thesaurus. However, the data annotation documents presented in this thesis must adhere to the structure provided by the data annotation models. The data annotation models presented in this thesis allow data annotation documents to cross-reference each other.

The annotation systems presented by Zsigmond *et al.* [12] and Gertz *et al.* [3] allow their users to express annotations at various granularities. All of the three systems described in the previous sections allow their users to express multiple annotations on the same piece of data.

Similar to these systems, the data annotation models proposed in this thesis allow data annotation users to express data annotations at various granularities. The data annotation models presented in this thesis allow users to annotate at five levels – database, relation, column, tuple, and cell. These data

annotation models also allow data annotation users to express multiple annotations on the same piece of data.

The models presented by Zsigmond *et al.* [12] and Gertz *et al.* [3] utilize annotation graphs to browse through annotations. The annotation system presented by Zsigmond *et al.* [12] utilizes a graph based on LEDA. The annotation graph utilized by Gertz *et al.* [3] is composed of three types of nodes – concepts, annotations, and images. The edges in this annotation graph depict specific relationships, such as how concepts are related, and how images are annotated using concepts.

Similar to these models, the query language, named Annotation Query Language (AnQL), presented in this thesis utilizes a data annotation graph to traverse through data annotation documents. The nodes in this data annotation graph depict the structural elements of the data annotation model that the data annotation document follows. The edges of this data annotation graph depict hierarchical relationships between the nodes. The data annotation graph presented in this thesis is specifically designed to facilitate AnQL query processing and data retrieval.

The annotation systems presented by Delcambre *et al.* [10] and Zsigmond *et al.* [12] do not utilize a query language to query annotations. However, the annotation system presented by Gertz *et al.* [3] utilizes a query language to query annotations. This query language consists of two operations – selection and path traversal.

This thesis proposes a new query language, named AnQL, in order to query data annotations. AnQL's query operations include *select*, *project*, *natural join*, and *union*. AnQL's query processing operations include *data annotation graph generation*, and *data annotation graph traversal*.

CHAPTER III

DATA ANNOTATION MODELS

This chapter presents the data annotation models that allow data annotation users to express data annotations on the database, relation, column, tuple, and cell levels. Along with the data annotation models, this chapter also presents the justification for five separate models.

Assumptions

The data annotation models presented in this chapter are based on three assumptions. The first assumption is that the *base data* resides in a relational database management system. Base data is defined as the data expressed using the relational data model. The second assumption is that data annotations reside outside the relational database that hosts the base data, and the third assumption is that data annotation documents are stored on the file system provided by the operating system of a server or a computer system. The reasons for making these assumptions are explained in later sections.

Base Data in a Relational Database

The first assumption is that the base data is stored in a relational database management system, and hence, is expressed using the relational data model. It is common knowledge that relational database management systems

form the majority of existing database deployments. It is also widely known that most organizations are highly resistant and reluctant to the idea of making structural and schematic changes to their database implementation. Databases are repositories of valuable corporate and business information that is critical to the success of the organization. Moreover, relational databases are highly optimized for fast data retrieval and query processing. Therefore, a data annotation system that "retro-fits" itself to run in conjunction with the currently deployed relational database management systems has a greater chance of acceptance and success, as compared to a system that makes structural and schematic changes to the relational database.

Data Annotations Outside Base Data

The second assumption, on which the data annotation models are based, is that data annotation documents are stored outside the relational database management system that hosts the base data. Given the current data types in a relational database that can support data annotation documents, it would be very inefficient to store data annotation documents inside a relational database management system.

Columnar data types in IBM DB2 UDB that can be used to store documents in a relation include BLOB (Binary Large Object), CLOB (Character Large Object), and DBCLOB (Double Byte Character Large Object) [1]. All three columnar data types BLOB, CLOB, and DBCLOB can have varying lengths similar to VARCHAR [1]. However, the maximum length of each of these data types must

be declared at the time when the columns are created [1]. It is difficult to estimate the length of a data annotation document in advance. Allocating large amounts of space in BLOB, CLOB, or DBCLOB would be very wasteful. This is because every cell, tuple, relation, or column may not need annotating. It would also be extremely difficult to increase the size of a column if a data annotation document outgrew its initial allocation.

It has also been proposed that an XML document can be shredded and parsed into tabular format, and thus stored in a relational database [16]. Shredding an XML document requires considerable processing. Moreover, it defeats the whole concept of semi-structured or structured data format – the data to begin with is not suitable for expression in tabular format. Putting the XML document back together is also very processor intensive because it requires computing several joins.

Hence, given the current columnar data types (BLOB, CLOB, DBCLOB), their maximum allocation sizes, declaration restrictions imposed by the database system, and the unsuitability of shredding an XML document to parse it into a tabular format, it is recommended that data annotation documents reside external to the relational database management system that hosts the base data.

Data Annotations on a File System

The third assumption, on which the data annotation models proposed in this thesis are based on, is that the data annotation documents are stored on the file system of a server. Until the number and size of data annotation

documents grows prohibitively large, the efficiencies incorporated in the file system of a server would be sufficient. However, if the number of data annotation documents grows prohibitively large, a smart indexing scheme would become necessary to search through data annotation documents, and ensure the continued extensibility of the data annotation management system.

It may also be argued that data annotation documents may be stored in an XML native database. However, XML native database are still in their infancy, and will require substantial research to become efficient and optimal.

Why Extensible Markup Language (XML)

The data models used to annotate at the database, relation, column, tuple, and cell levels of a relational database utilize XML in order to structure and express data annotations. XML was chosen because certain characteristics of XML provide several advantages over other data formats. These data formats include simple text documents, or electronic mail memoranda, or electronic forms.

One of the characteristics of XML is that the tags used in XML are meta-tags [4]. These meta-tags allow even naïve and unsophisticated users to understand the structure and semantics of XML documents. Secondly, XML is database neutral [4]. Hence, XML can be used to connect heterogeneous databases i.e. the same data annotation data model can be used to express data annotations on base data residing in heterogeneous databases (e.g. IBM DB2 UDB, Oracle, Microsoft SQL Server, etc.). This allows the data annotation user

community to express, share, and utilize data annotations irrespective of the database environment and platform that hosts their base data.

XML is platform-independent. Hence, data annotation users can express, store, and share data annotation documents regardless of the operating system (e.g. Windows, Unix, Linux, etc.) and the platform they use for data annotation management system, and for their relational database system. XML also supports Unicode [4]. Thus, the support for expressing data annotations in several different languages is readily available. Additionally, XML is extensible, flexible, and can be used to interchange data among heterogeneous applications and platforms seamlessly [4].

Data Annotation Models

This section presents the five data annotation models that allow data annotation users to express data annotations at the database, relation, tuple, column, and cell level. XML is used to structure and express these data annotation models. Data annotation documents expressed using these models are simple, intuitive, easy to understand, platform-independent, database-neutral, and can be interchanged seamlessly. Moreover, the data annotations expressed using these data models are extensible and flexible, and can be easily customized for a particular application domain. The metatag `<applicationDomainSpecificMetaTag>` allows data annotation users to customize data annotations. Some examples of `<applicationDomainSpecificMetatag>` are `<comment>`, `<example>`,

<note>, <error>, <description>, <definition>, among several others.

This section also discusses alternatives to the proposed data annotation models, and their disadvantages.

Data Annotation Model at the Database Level

Fig. 1 presents the data annotation model that allows data annotation users to annotate at the database level.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>uniqueName</documentName>
  <documentId>uniqueId</documentId>
  <annotationAttachedTo>
    <database>databaseName</database>
  </annotationAttachedTo>
  <!--annotation and annotationMetadata may occur
    multiple times in a single document -->
  <annotation>
    <applicationDomainSpecificMetatag>
      dataAnnotation
    </applicationDomainSpecificMetatag>
    <annotationMetadata>
      <author>someone</author>
      <recorded>someDateAndTime</recorded>
    </annotationMetadata>
  </annotation>
  <referencedAnnotations>
    <documentNameList>
      <documentName>uniqueName</documentName>
      <documentName>uniqueName</documentName>
      ...
    </documentNameList>
  </referencedAnnotations>
</annotationDocument>
```

Fig. 1 – Data Annotation Model at the Database Level

The data annotation model presented in Fig. 1 may be divided into several modules. These modules include the validation module, the identification module, the level module, the annotation module, the annotation metadata module, and the cross-reference module. The validation module (see Fig. 2) of the data annotation model presented in Fig. 1 consists of a valid XML schema that validates data annotation documents. The identification module (see Fig. 3) of the data annotation model presented in Fig. 1 uniquely identifies a data annotation document. The level module (see Fig. 4) of the data annotation model presented in Fig. 1 represents the level of the base data that a data annotation document annotates. The level module presented in Fig. 4 indicates the database level.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:noNamespaceSchemaLocation=
"DataAnnotationDataModel.xsd">
```

Fig. 2 – Validation Module of Data Annotation Model

```
<documentName>uniqueName</documentName>
<documentId>uniqueId</documentId>
```

Fig. 3 – Identification Module of Data Annotation Model

```

<annotationAttachedTo>
  <database>databaseName</database>
</annotationAttachedTo>

```

Fig. 4 – Level Module of Data
Annotation Model

The annotation module (see Fig. 5) of the data annotation model presented in Fig. 1 contains the actual data annotation that a data annotation user uses to express semantically rich metadata on the base data.

```

<applicationDomainSpecificMetatag>
  dataAnnotation
</applicationDomainSpecificMetatag>

```

Fig. 5 – Annotation Module of Data
Annotation Model

The annotation module of the data annotation model presented in Fig. 1 is always accompanied by the annotation metadata module of the data annotation model. The annotation metadata module (see Fig. 6) contains the name of the author of the data annotation and the timestamp when the data annotation was created.

```

<annotationMetadata>
  <author>someone</author>
  <recorded>someDateAndTime</recorded>
</annotationMetadata>

```

Fig. 6 – Annotation Metadata Module of
Data Annotation Model

The cross-reference module (see Fig. 7) of the data annotation model presented in Fig. 1 allows data annotation users to list data annotation documents that are related to the current data annotation document.

```
<referencedAnnotations>
  <documentNameList>
    <documentName>
      uniqueName
    </documentName>
    <documentName>
      uniqueName
    </documentName>
    ...
  </documentNameList>
</referencedAnnotations>
```

Fig. 7 – Cross-Reference Module of
Data Annotation Model

The data annotation model presented in Fig. 1 provides data annotation users a consistent, yet flexible and extensible, mechanism to annotate base data at the database level. An alternative to annotating at the database level without using the data annotation model presented in Fig. 1 is to keep notes in a text document. Although keeping notes in text documents is possible, the approach presents a few problems. The first problem is that this alternative does not provide its users a consistent mechanism to annotate at the database level. The primary reason for the lack of consistency is that every database user would use his or her personal format to express data annotations. The second problem with keeping notes in a text document is that sharing text documents across platforms is rather difficult. The third problem with keeping notes in a text

document to express annotations is that this method does not provide means to automatically maintain metadata information or record-keeping about data annotations. In addition to providing various other valuable features, the data annotation model at the database level addresses all the problems listed above.

Assume that a real estate firm has a database that maintains records of properties for sale, real estate agents, and which real estate agent sold which property. The manager of the real estate firm decides to transfer the records for properties listed for over a million dollars to a separate database. The manager can use the data model for the database level to annotate the existing database. Fig. 8 presents this example data annotation. The database administrator can review the data annotations made by the manager, make appropriate changes, and inform the manager regarding the change via a data annotation. In this case, data annotations save time and resources by eliminating the need to set up a meeting with the database administrator in order to explain a simple change to him or her.

Data Annotation Model at the Relation Level

This section presents the data annotation model that data annotation users can use to annotate at the relation level (see Fig. 9). The data annotation model presented in Fig. 9 is similar to the data annotation model presented in Fig. 1. The only difference between the two data annotation models is the level module. The level module (see Fig. 10) of the data annotation model at the relation level (see Fig. 9) contains the name of the relation and the name of the

database to which the relation belongs to. Together, the database name and the relation name uniquely identify the relation that is being annotated by a data annotation user.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>RE1</documentName>
  <documentId>RE1</documentId>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
  </annotationAttachedTo>
  <annotation>
    <actionRequired>
      Please create a separate database for properties
      listed at $1,000,000 or more. Separate out real
      estate agents that deal exclusively in these
      properties. Proposed database name - LUXURY_ESTATES.
      Name the relations using firm's standard naming
      conventions.
    </actionRequired>
    <annotationMetadata>
      <author>theManager</author>
      <recorded>April 2, 2004 10:37:15 PM</recorded>
    </annotationMetadata>
  </annotation>
</annotationDocument>
```

Fig. 8 – Example Database-Level Data Annotation Document

The data annotation model presented in Fig. 9 provides data annotation users with a model that allows expressing data annotations at the relation level in a consistent and standardized manner. An alternative to the data annotation model presented in Fig. 9 in order to annotate at the relation level is to keep notes in a text document. However, this approach suffers from the same

problems as those suffered by keeping database-level data annotations in text documents.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "DataAnnotationDataModel.xsd">
  <documentName>uniqueName</documentName>
  <documentId>uniqueId</documentId>
  <annotationAttachedTo>
    <database>databaseName</database>
    <relation>relationName</relation>
  </annotationAttachedTo>
  <!--annotation and annotationMetadata may occur
    multiple times in a single document -->
  <annotation>
    <applicationDomainSpecificMetatag>
      dataAnnotation
    </applicationDomainSpecificMetatag>
    <annotationMetadata>
      <author>someone</author>
      <recorded>someDateAndTime</recorded>
    </annotationMetadata>
  </annotation>
  <referencedAnnotations>
    <documentNameList>
      <documentName>uniqueName</documentName>
      <documentName>uniqueName</documentName>
      ...
    </documentNameList>
  </referencedAnnotations>
</annotationDocument>
```

Fig. 9 – Data Annotation Model at the Relation Level

Assume a paleontology database that consists of relations that describe the main characteristics of each type of dinosaur that existed during a certain period. As an example, the paleontology database would contain a

relation named `LateTriassic`. The relation `LateTriassic` would contain information about `Coelophysis`, `Cynodont`, and `Placerias`, among other dinosaurs. Other relations in this database would include `EarlyCretaceous`, `LateJurassic`, `LateCretaceous`.

```
<annotationAttachedTo>
  <database>databaseName</database>
  <relation>relationName</relation>
</annotationAttachedTo>
```

Fig. 10 – Level Module at the Relation Level

However, these relations would not contain any information on the state of the earth or the weather or the habitat during any of these time periods. A paleontologist can utilize the relation-level data annotation model to express additional semantic information regarding the habitat during a particular period. Fig. 11 presents such an example data annotation document.

Data Annotations at the Column Level

This section presents the data annotation model that data annotation users can use to annotate at the column level (see Fig. 12). The data annotation model presented in Fig. 12 is similar to the data annotation model presented in Fig. 1. The only difference between the two data annotation models is the level at which they allow users to annotate base data. The level module (see Fig. 13) of the data annotation model presented in Fig. 12 contains the name of the database, the name of the relation, and the name of the column that the data annotation user annotates. Together, the database name, the relation name, and

the column name uniquely identify the column that a data annotation user is annotating.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>plntlogyLateTriassic1</documentName>
  <documentId>Pal1</documentId>
  <annotationAttachedTo>
    <database>PALEONTOLOGY</database>
    <relation>LATE_TRIASSIC</relation>
  </annotationAttachedTo>
  <annotation>
    <description>
      Neither flowering plants nor grass existed. The
      ground was covered with ferns and mosses. Plant
      life
      was very drab - just green and brown in color.
    </description>
    <annotationMetadata>
      <author>paleontologist</author>
      <recorded>Apr 17, 2004 8:02:08 PM</recorded>
    </annotationMetadata>
  </annotation>
</annotationDocument>
```

Fig. 11 – Example Relation-Level Data Annotation Document

The alternative to using the data annotation model presented in Fig. 12 is to declare an annotation column for each column in the relation [17]. Consider a COOKING database (see Fig. 14) with the following schema –

```
RECIPE_LIST (RECIPE_ID, RECIPE_NAME);
CONDIMENT_LIST (CONDIMENT_ID, CONDIMENT_NAME, QUANTITY);
```

(Underlined column names indicate primary keys.)

```

<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "DataAnnotationDataModel.xsd">
  <documentName>uniqueName</documentName>
  <documentId>uniqueId</documentId>
  <annotationAttachedTo>
    <database>databaseName</database>
    <relation>relationName</relation>
    <column>columnName</column>
  </annotationAttachedTo>
  <!--annotation and annotationMetadata may occur
    multiple times in a single document -->
  <annotation>
    <applicationDomainSpecificMetatag>
      dataAnnotation
    </applicationDomainSpecificMetatag>
    <annotationMetadata>
      <author>someone</author>
      <recorded>someDateAndTime</recorded>
    </annotationMetadata>
  </annotation>
</referencedAnnotations>

```

Fig. 12 – Data Annotation Model at the Column Level

```

<annotationAttachedTo>
  <database>databaseName</database>
  <relation>relationName</relation>
  <column>columnName</column>
</annotationAttachedTo>

```

Fig. 13 – Level Module at the Column Level

Fig. 15 presents how Chef Alice can annotate the `QUANTITY` column by adding a column named `ANNOTATION_QUANTITY`.

RECIPE_LIST

RECIPE_ID	RECIPE_NAME
1	PAD THAI
4	YELLOW CURRY

CONDIMENT_LIST

CONDIMENT_ID	CONDIMENT_NAME	QUANTITY
1	SALT	3
8	TURMERIC	2

Fig. 14 – Relations RECIPE_LIST and CONDIMENT_LIST (COOKING database)

CONDIMENT_LIST

CONDIMENT_ID	CONDIMENT_NAME	QUANTITY	ANNOTATION_QUANTITY
1	SALT	3	TABLESPOON
8	TURMERIC	2	TABLESPOON

Fig. 15 – Relation CONDMIENT_LIST with column ANNOTATION_QUANTITY

Although this technique allows the chef to express data annotations, it presents a few problems. The first problem with this technique is that it does not allow the chef to express and store the relationship between a tablespoon and a milligram (for example, 1 tablespoon = 5 milligrams). The second problem with this technique of adding an annotation column for each column in a relation is that it is redundant, and wastes a huge amount of space. It is not necessary that each and every column in the relation may need to be annotated. The third problem with this technique is that it might affect query processing, and hurt database performance due to the unnecessary increase in the number of columns in the relation. This technique also violates the principles of sound

database design. The fourth problem is that due to the size limitations and declaration constraints of data types, such as BLOB, CLOB, and DBCLOB, the annotation columns created within a relation might not be extensible. The fifth problem with the creation of annotation columns within a relation is the resistance of corporations and database administrators against making any structural and schematic changes to deployed databases. It is common knowledge that database administrators and corporations usually spend a large amount of time and effort to customize and optimize their databases for faster query processing. Structural or schematic changes made to a database might adversely affect query processing, and hence, such a change meets with a high degree of resistance. Another problem with this technique is that it is not possible to query specific portions of data annotation documents stored in these annotation columns using Structured Query Language (SQL).

Fig. 16 presents an example data annotation document that Chef Alice can utilize in order to annotate the `QUANTITY` column. This example data annotation document utilizes the column-level data annotation model presented in Fig. 12.

Data Annotations at the Tuple Level

This section presents the data annotation model that data annotation users can use to annotate at the column level (see Fig. 17). The data annotation model presented in Fig. 17 is similar to the data annotation model presented in Fig. 1. The only difference between the two data annotation models is the level at

which the data annotation models allow data annotation users to annotate. The level module (see Fig. 18) of the data annotation model presented in Fig. 17 contains the name of the database, the name of the relation, and the primary key of the tuple that the data annotation user annotates. Together, the database name, the relation name and the primary key of the tuple uniquely identify the tuple that is being annotated. A composite primary key may be represented by a comma-separated list.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>cookingRecipeQuantity1</documentName>
  <documentId>cooking1</documentId>
  <annotationAttachedTo>
    <database>COOKING</database>
    <relation>RECIPE</relation>
    <column>QUANTITY</column>
  </annotationAttachedTo>
  <annotation>
    <note> All quantities in tablespoons.
      Conversion: 1 tablespoon = 5 milligrams </note>
    <annotationMetadata>
      <author>Chef Alice</author>
      <recorded>May 27, 2004 9:12:24 AM</recorded>
    </annotationMetadata>
  </annotation>
</annotationDocument>
<referencedAnnotations>
  <documentNameList>
    <documentName>uniqueName</documentName>
    <documentName>uniqueName</documentName>
    ...
  </documentNameList>
</referencedAnnotations>
</annotationDocument>
```

Fig. 16 – Example Column-Level Data Annotation Document

```

<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "DataAnnotationDataModel.xsd">
  <documentName>uniqueName</documentName>
  <documentId>uniqueId</documentId>
  <annotationAttachedTo>
    <database>databaseName</database>
    <relation>relationName</relation>
    <!--for composite primary keys, list values separated
      by commas-->
    <tuple>primary key</tuple>
  </annotationAttachedTo>
  <!--annotation and annotationMetadata may occur multiple
    times in a single document -->
  <annotation>
    <applicationDomainSpecificMetatag>
      dataAnnotation
    </applicationDomainSpecificMetatag>
    <annotationMetadata>
      <author>someone</author>
      <recorded>someDateAndTime</recorded>
    </annotationMetadata>
  </annotation>
  <referencedAnnotations>
    <documentNameList>
      <documentName>uniqueName</documentName>
      <documentName>uniqueName</documentName>
      ...
    </documentNameList>
  </referencedAnnotations>
</annotationDocument>

```

Fig. 17 – Data Annotation Model at the Tuple Level

In the absence of the tuple-level data annotation model presented in Fig. 17, a tuple can be annotated by adding a column (named ANNOTATION) to each relation. This column can be of data type VARCHAR, BLOB, CLOB, or DBCLOB.

```

<annotationAttachedTo>
  <database>databaseName</database>
  <relation>relationName</relation>
  <!--for composite primary keys, list
       values separated by commas-->
  <tuple>primary key</tuple>
</annotationAttachedTo>

```

Fig. 18 – Level Module at Tuple Level

However, this method presents a few problems. The first problem presented by this method is that it wastes space. The reason for space wastage is that although the storage space for every tuple is allocated, every tuple might not require annotating. Another problem with the method of adding an annotation column to a relation to annotate at the tuple level is the inability of the column to grow when need arises. The third problem with this method is the resistance of corporations and database administrators to make structural and schematic changes to an already deployed database. The fourth problem with this method is that it is not possible to query and retrieve specific portions of data annotation documents stored in the annotation column using SQL.

Consider the `NEW_STUDENTS` database (see Fig. 19), with the following schema –

```

NEW_STUDENT_INFO ( STUDENT_ID, NAME, DOB );
ADMIT_INFO ( STUDENT_ID, YEAR, DEGREE, PROGRAM );

```

When John Doe accepts an offer of admission, the department secretary informs the graduate coordinator to change John Doe's status from

"offered" to "accepted". This change of status also encompasses adding John Doe's record to the record of currently enrolled student database (the example assumes that the secretary does not have authority to change student status). The data annotation document presented in Fig. 20 illustrates how easy and convenient it is to express data annotation at the tuple level using the data model proposed in Fig. 17.

NEW_STUDENT_INFO		
STUDENT_ID	NAME	DOB
999888777	John Doe	1/1/1984
123456789	Jane Smith	1/2/1986

ADMIT_INFO			
STUDENT_ID	YEAR	DEGREE	PROGRAM
999888777	2004	B.S.	ELEC. ENGG.
123456789	2004	M.S.	COMP. SCI.

Fig. 19 – Relations NEW_STUDENT_INFO with ADMIT INFO (NEW STUDENT database)

Data Annotations at the Cell Level

This section presents the data annotation model that data annotation users can use to annotate at the column level (see Fig. 21). The data annotation model presented in Fig. 21 is similar to the data annotation model presented in Fig. 1. The only difference between the two data annotation models is the level at which the data annotation model allows data annotation users to annotate. A cell in a database is defined as an intersection of a column and a tuple. Therefore, the cell-level data annotation model uniquely identifies a cell by the combination of the database name, relation name, column name, and the primary key of the tuple. A composite primary key may be represented by a comma-separated list.

Fig. 22 presents the level module of the data annotation model that allows data annotation users to annotate base data at the cell level (presented in Fig. 21).

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>admissionsNewStudentInfo</documentName>
  <documentId>admnl</documentId>
  <annotationAttachedTo>
    <database>ADMISSIONS</database>
    <relation>NEW_STUDENT_INFO</relation>
    <tuple>999888777</tuple>
  </annotationAttachedTo>
  <annotation>
    <comment>
      John Doe (Id 999888777) accepted offer. Please change
      status.
    </comment>
    <annotationMetadata>
      <author>departmentSecretary</author>
      <recorded>May 27, 2004 4:03:08 PM</recorded>
    </annotationMetadata>
  </annotation>
</annotationDocument>
```

Fig. 20 – Example Tuple-Level Data Annotation Document

It is not possible to express data annotations at the cell level without using the cell-level data annotation. Fig. 23 presents an example data annotation document that Chef Alice can use to annotate the cell that contains the ingredient turmeric (see Fig. 23) in relation CONDIMENT_LIST (see Fig. 14). This example data annotation document utilizes the using the cell-level data annotation model presented in Fig. 21.

```

<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>uniqueName</documentName>
  <documentId>uniqueId</documentId>
  <annotationAttachedTo>
    <database>databaseName</database>
    <relation>relationName</relation>
    <column>columnName</column>
    <!--for composite primary keys, list values separated
      by commas-->
    <tuple>primary key</tuple>
  </annotationAttachedTo>
  <!--annotation and annotationMetadata may occur multiple
    times in a single document -->
  <annotation>
    <applicationDomainSpecificMetatag>
      dataAnnotation
    </applicationDomainSpecificMetatag>
    <annotationMetadata>
      <author>someone</author>
      <recorded>someDateAndTime</recorded>
    </annotationMetadata>
  </annotation>
  <referencedAnnotations>
    <documentNameList>
      <documentName>uniqueName</documentName>
      <documentName>uniqueName</documentName>
      ...
    </documentNameList>
  </referencedAnnotations>
</annotationDocument>

```

Fig. 21 – Data Annotation Model at the Cell Level

```

<annotationAttachedTo>
  <database>databaseName</database>
  <relation>relationName</relation>
  <column>columnName</column>
  <!-- for composite primary keys, list
        values separated by commas-->
  <tuple>primary key</tuple>
</annotationAttachedTo>

```

Fig. 22 – Level Module at the Cell Level

```

<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>cookingCondimentList</documentName>
  <documentId>cooking2</documentId>
  <annotationAttachedTo>
    <database>COOKING</database>
    <relation>CONDIMENT_LIST</relation>
    <column>CONDIMENT_NAME</column>
    <tuple>8</tuple>
  </annotationAttachedTo>
  <annotation>
    <caution>
      Be very careful with turmeric. Turmeric leaves
      yellow stains on everything incl. counter tops, and
      clothes. These stains are extremely difficult to
      remove.
    </caution>
    <annotationMetadata>
      <author>alice</author>
      <recorded>Apr 27, 2004 4:18:16 AM</recorded>
    </annotationMetadata>
  </annotation>
</annotationDocument>

```

Fig. 23 – Example Cell-Level Data Annotation Document

CHAPTER IV

ANNOTATION QUERY LANGUAGE

This chapter presents the Annotation Query Language (AnQL), a query language to query data annotations, and the AnQL query engine. In order to better understand the AnQL query operations, this chapter first introduces the AnQL query engine, and its query processing operations. AnQL's query processing operations include *data annotation graph generation*, and *data annotation graph traversal*. AnQL's query operations include *select*, *project*, *natural join*, and *union*.

The Example REAL_ESTATE Database

An example database, REAL_ESTATE, is used throughout this chapter to illustrate AnQL's fundamental as well as query-processing operations. The example REAL_ESTATE database maintains records of property listings, real estate agents, and which real estate agent sold which property. The schema for the example REAL_ESTATE database is as follows –

```
Listings (PropertyId, MLSNumber, StreetAddress, City, Zip, County);
```

```
Features (PropertyId, Area, Beds, Baths, Type, LotSize, Age, Style);
```

Details (DetailId, Detail);

Contains (PropertyId, DetailId);

Agents (AgentId, AgentName, AgentEmail, AgentPhone);

Sold (AgentId, PropertyId);

Fig. 24 presents relations (along with data) in the example REAL_ESTATE database.

LISTINGS

PROPERTY_ID	MLS_NUMBER	STREET_ADDRESS	CITY	ZIP	COUNTY
PI1	387811	2928 CHILTERN WY	SAN JOSE	95127	SANTA CLARA
PI2	401891	OUT OF AREA	OUT OF AREA	95148	OTHER

FEATURES

PROPERTY_ID	AREA	BEDS	BATHS	TYPE	LOT_SIZE	AGE	STYLE
PI1	1384	4	2	SINGLE FAMILY		47	DETACHED
PI2	27007			LAND PROPERTY	0.62		LAND

DETAILS

FEATURE_ID	FEATURE_NAME
F11	FIREPLACE
F10	L-SHAPED LIVING-DINING COMBO

AGENTS

AGENT_ID	AGENT_NAME	AGENT_EMAIL	AGENT_PHONE
AG1	RUDY CAMPOS	rudy@ere.com	(805) 485-2616
AG2	CARLA GALLEGOS	carla@ere.com	(408) 979-2899

CONTAINS

PROPERTY_ID	FEATURE_ID
PI1	F11
P21	F12

SOLD

AGENT_ID	PROPERTY_ID
AG4	PI1

Fig. 24 – Example REAL_ESTATE Database

Fig. 25 and Fig. 26 present example data annotation documents used in this chapter to illustrate AnQL's functions. The example data annotation

document presented in Fig. 25 annotates the cell defined by the column `LOT_SIZE` and the tuple with primary key `PI2`. This data annotation helps save storage space in the underlying database. This is because in order to store the unit that corresponds with each and every lot size within the relational database, the database administrator would need to create an additional column in the relation `FEATURES`. The value stored in this column would repeat for most of the tuples that describe a lot. However, this cell's value for all those tuples that describe other types of properties would be null. Another reason that the creation of an additional column to store the unit for a lot size would be wasteful is that typically the number of lots offered for sale by a real estate company is much less than the number of other types of properties offered for sale.

The example data annotation document presented in Fig. 26 helps save storage space in a manner similar to the one presented in Fig. 25. Typically, an empty lot would have very different features than a constructed home or property. However, in a typical real estate database, the number of empty lots listed for sale is much less than the number of homes listed for sale. Hence, to declare a few columns specifically to describe an empty lot might prove to be wasteful. Most of the values in these columns would be null. This is because the number of listed homes would be much greater than the number of listed lots. Hence, the second data annotation document would serve the purpose of better describing the listed lot, as well as, help save storage space in the underlying database.

```

<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
  "DataAnnotationDataModel.xsd">
  <documentName>REFeaturesLotSizePI2</documentName>
  <documentId>RE9</documentId>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>LOT_SIZE</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <description>
      Lot size is in acres.
    </description>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>May 28, 2004 12:15:14 PM</recorded>
    </annotationMetadata>
  </annotation>
</annotationDocument>

```

Fig. 25 – Example Data Annotation Document
(REFeaturesLotSizePI2)

Naïve Storage Scheme

In order to facilitate the processing of AnQL queries, a clever, yet, simple and naïve storage scheme is employed. According to this storage scheme, all data annotation documents that annotate at one level are kept in a separate directory from the data annotation documents that annotate at another level. In other words, data annotation documents that annotate at the cell level are kept in a separate directory from the data annotation documents that annotate at the relation level, and so on. Similarly, all data annotation

documents, irrespective of their levels, for one database are kept separate from all data annotation documents for another database. In other words, data annotation documents that pertain to the REAL_ESTATE database would be kept in a separate directory from the data annotation documents that pertain to the ACTORS database.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotationDocument
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "DataAnnotationDataModel.xsd">
  <documentName>REFeaturesPropertyIdPI2</documentName>
  <documentId>RE11</documentId>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>PROPERTY_ID</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <comment>
      Build your dream home. Sunny and private. Water
      and electricity at site. Plans and permits
      approved and ready for a 2683+ sq. ft home.
      Septic and geo approved.
    </comment>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>Apr 1, 2004 10:15:25 AM</recorded>
    </annotationMetadata>
  </annotation>
</annotationDocument>
```

Fig. 26 – Example Data Annotation Document
(REFeaturesPropertyIdPI2)

An alternative to the storage scheme described above is a tagged file format. In the tagged file format, all annotations are kept in a single file with tags

that uniquely identify the start and end of a data annotation document. However, this file format presents a few problems. The first problem presented by the tagged file format is that an additional index file, that contains information on the tagged data annotation documents, would have to be maintained. This index file would contain information about the start and end of a particular data annotation document that is stored in the tagged file format. This index file would be an equivalent to the data structure that an operating system maintains in order to store information on files stored on a disk. In order to process each query, the query engine would have to first look up in the index file to find the correct position of the data annotation document in the tagged file. This index file would have to be updated every time a new data annotation document is created, and whenever a data annotation document is deleted. The second problem with the tagged file format is that the query engine would require two file accesses in order to locate and retrieve data annotation documents for query processing. The third problem presented by the tagged file format is that the usage of the tagged file format makes the data annotation management system based on the data annotation models proposed in Chapter III and AnQL system-dependent. Therefore, the storage scheme discussed above is preferred over the tagged file format because it is better suited for AnQL query processing. Fig. 27 illustrates the naïve storage scheme utilized in this thesis.

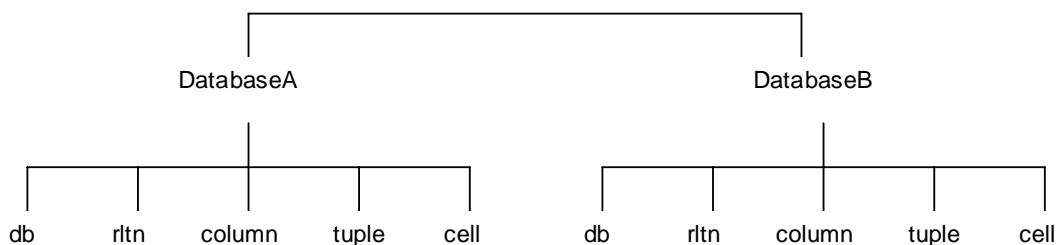


Fig. 27 – Storage Scheme

AnQL Query Engine

AnQL's query engine is at the heart of AnQL. It is equivalent to an operating system kernel. The query engine executes the queries issued by data annotation users. The query engine also interfaces with the underlying storage mechanism that hosts the data annotation documents. The AnQL query engine functions include *data annotation graph generation*, and *data annotation graph traversal*.

In order to process an AnQL query, the AnQL query engine generates *data annotation graph(s)* corresponding to the input data annotation document(s). The query engine uses the *data annotation graph generation* function to generate these data annotation graph(s). The query engine then utilizes the *data annotation graph traversal* function to traverse through these data annotation graph(s).

Data Model for Data Annotation Graph

In order to process AnQL queries, the AnQL query engine generates a *data annotation graph* using the *data annotation graph generation* function. A

data annotation graph is a special graph modeled in spirit of the XQuery data model [6]. The data annotation graph is modeled and structured to especially facilitate AnQL query processing. A data annotation graph contains nodes and edges that correspond to the hierarchical structure of a data annotation document. Data annotation documents are based on the data annotation models presented in Chapter III. The nodes in a data annotation graph correspond to the hierarchical elements in a data annotation document. The edges in a data annotation graph represent the hierarchical structure, i.e. the ancestor-descendent relationship, between the elements of a data annotation document. The nodes in a data annotation graph can be classified as follows –

- *root* – The *root* node of a data annotation graph is *annotationDocument*.

The *root* node represents that the graph contains the hierarchical structure of the data model that expresses data annotation at one of the five levels (database, relation, column, tuple, and cell).

- *element* – An *element* node contains an element defined in a data annotation document. Based on the data models defined in Chapter III, element nodes include *annotationDocument*, *documentName*, *documentId*, *annotationAttachedTo*, *database*, *relation*, *column*, *tuple*, *annotationMetadata*, *author*, *recorded*, and the *domain-specific-tag* used within the start and end tags of *annotation*.

- *elementValue* – A node of type *elementValue* is a child of a node of type *element*. The *elementValue* node contains the actual value of an *element*.

- *textValue* – A node of type *textValue* (named *annotation* in data models defined in Chapter III) is a child of a node of type *element*. The *textValue* node contains the actual data annotation. A node of type *textValue* is enclosed within the start and end tags of element named *annotation*.

Data Annotation Graph Generation Function (ϕ)

The *data annotation graph generation* function generates a data annotation graph corresponding to the data annotation document provided as input. The data annotation graph is modeled and structured to facilitate AnQL query processing. Depth first traversal of a data annotation graph ensures that document order is preserved during AnQL query processing. The *data annotation graph generation* function takes as input a *well-formed* and *validated* data annotation document. A well-formed data annotation document is one that has a correct and complete nesting structure [4]. A validated data annotation document is one that follows the nesting and hierarchical structure stipulated by a Document Type Definition (DTD) or XML Schema [4]. Formally, the *data annotation graph generation* function is defined as –

ϕ (data annotation document) = G(V, E) where,

G is the data annotation graph,

V is the set of nodes in the data annotation graph,

E is the set of edges in the data annotation graph,

$E = \{v_s, v_t\}$, such that,

- v_t is a node of type *elementValue* and is a child of

element v_s , or

- v_t is a node of type *element* and is a child of

element v_s , or

- v_t is a node (named *annotation*) of type *textValue* and is a child of *element* v_s , or

- v_t is a node of type *element* and is a child of root v_s .

Fig. 28 – 32 present generic data annotation graphs that correspond to the five data annotation models presented in Chapter III.

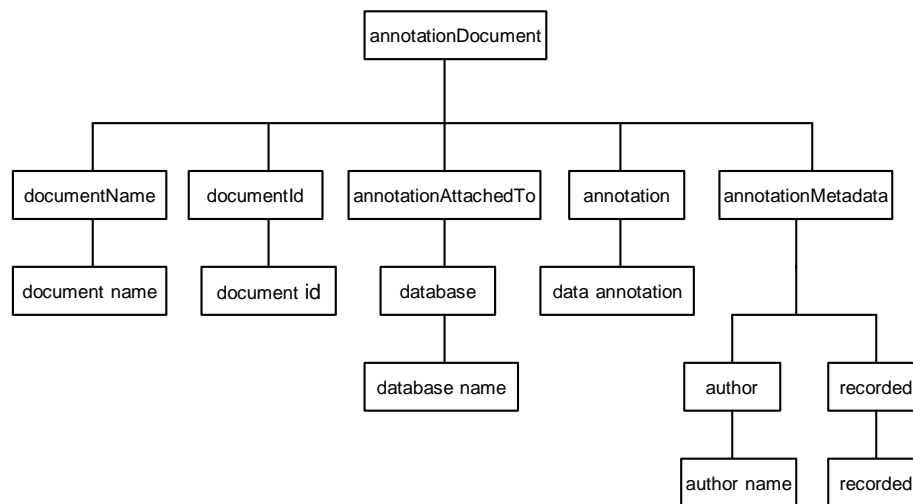


Fig. 28 – Database-Level Data Annotation Graph

Data Annotation Graph Traversal Function (Σ)

The *data annotation graph traversal* [3] function traverses a data annotation graph using depth-first traversal. The *data annotation graph traversal* function accepts as input a start node and a well-formed and validated data

annotation document. The *data annotation graph traversal* function returns as output a set of nodes that are directly connected to the start node (provided as one of the inputs). For the function to return meaningful results, it is important that the data annotation document specified as one of the inputs contain the start node (specified as the second input). If the data annotation document does not contain the start node, the *data annotation graph traversal* function generates an error.

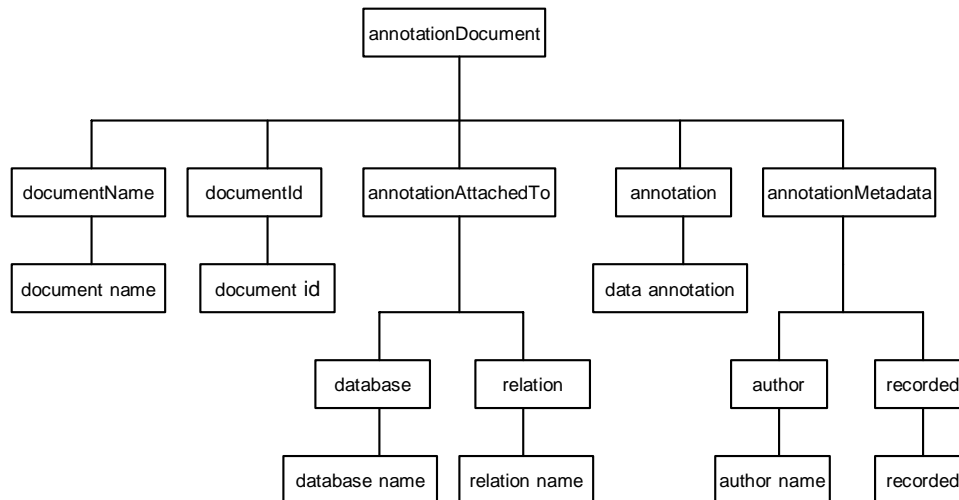


Fig. 29 – Relation-Level Data Annotation Graph

Formally, *data annotation graph traversal* function is defined as –

$$\Sigma_{\text{start node}}(\text{data annotation document}) = \{v_t \text{ such that } v_t \in V$$

and $(v_s, v_t) \in E\}$, where,

V is the set of nodes in the data annotation document,

E is the set of edges in the data annotation document, and

v_s and v_t are related to each other in one of the following

ways -

- v_t is a node of type *elementValue* and is a child of *element* v_s , or
- v_t is a node of type *element* and is a child of *element* v_s , or
- v_t is a node (named *annotation*) of type *textValue* and is a child of *element* v_s .

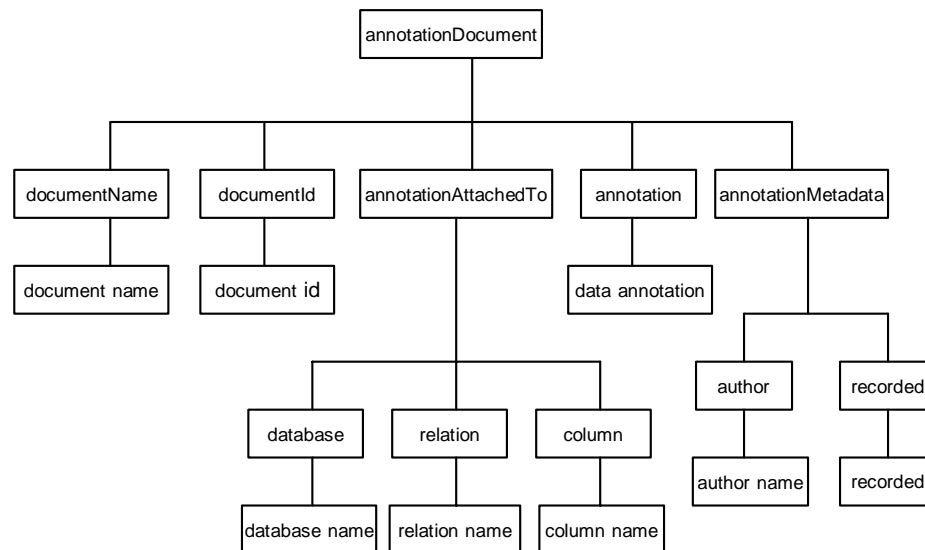


Fig. 30 – Column-Level Data Annotation Graph

Fig. 33 presents the result set generated by the function, $\Sigma_{\text{documentName}}$ (REFeaturesLotSizePI2). In this function, *documentName* is the start node, and REFeaturesLotSizePI2 is the input data annotation document (presented in Fig. 26). The *data annotation graph traversal* function returns the node that is directly connected to the node *documentName*.

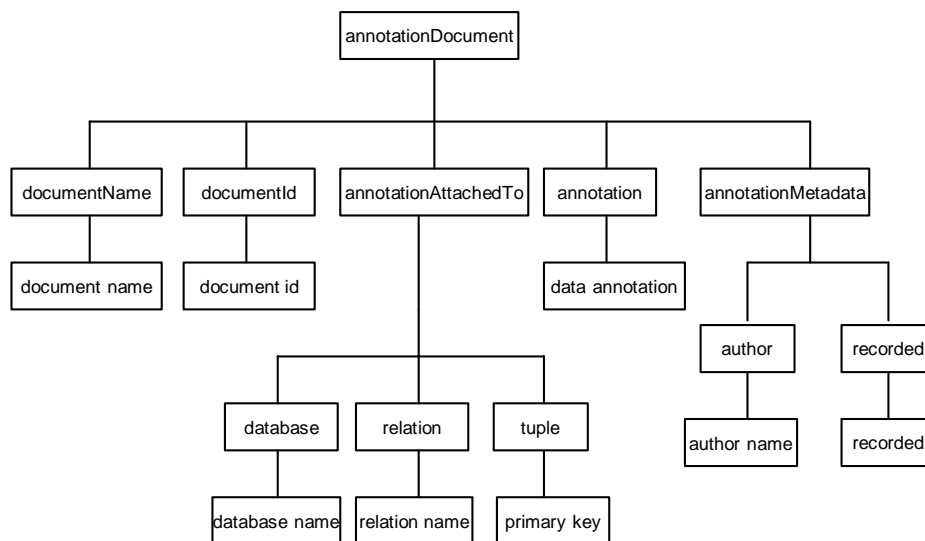


Fig. 31 – Tuple-Level Data Annotation Graph

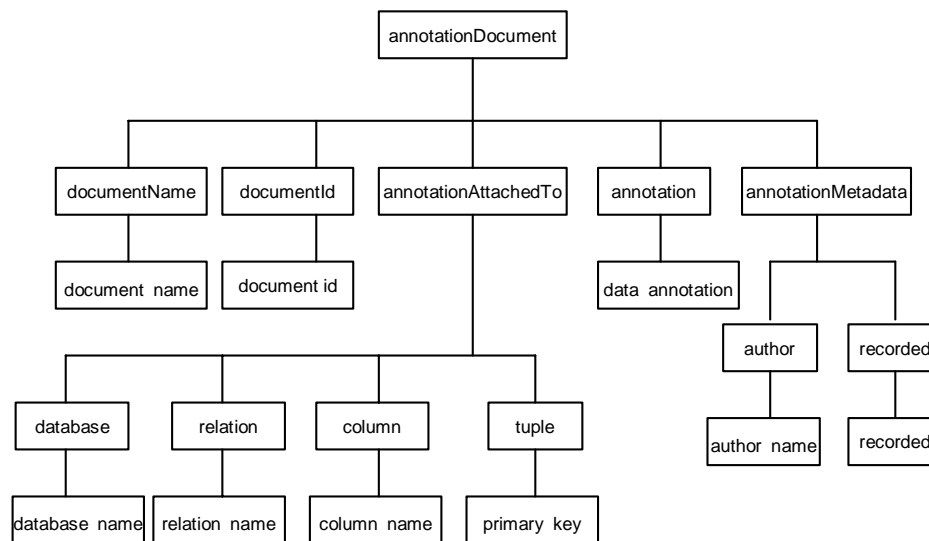


Fig. 32 – Cell-Level Data Annotation Graph

REFeaturesPropertyIdPI2

Fig. 33 – Result Set of a *Data Annotation Graph Traversal* Function

Transitive Closure of Data Annotation Graph Traversal Function (Σ^+).

The *transitive closure of the data annotation graph traversal* function [3] accepts a start node, and a well-formed and validated data annotation document as input. As output, the *transitive closure of the data annotation graph traversal* function generates the set of all nodes that are connected directly or indirectly to the start node.

Fig. 34 presents the result set generated by the *transitive closure of the data annotation graph traversal* function, $\Sigma^+_{\text{documentName}}$ (REFeaturesLotSizePI2). In this example, the *transitive closure of the data annotation graph traversal* function is provided with `documentName` as the start node, and the example data annotation document of Fig. 26. The function returns the nodes that are connected, directly or indirectly, to the start node specified as one of the inputs.

Both the *data annotation graph traversal* function and its *transitive closure* also accept a single input – a well-formed and validated data annotation document. If the start node is not specified, both *data annotation graph traversal* function, and its *transitive closure* begin to traverse the data annotation graph at the *root* node.

```
<documentName>  
  REFeaturesPropertyIdPI2  
</documentName>
```

Fig. 34 – Result Set of *Transitive Closure of Data Annotation Graph Traversal Function*

AnQL Query Operations

Data annotation users can utilize AnQL query operations in order to query data annotation documents. AnQL query operations include *select*, *project*, *natural join*, and *union*. All these operations accept well-formed and validated data annotation documents as inputs.

Select (σ) Operation

The *select* operation selects portion(s) of a data annotation document based on a boolean predicate. The *select* operation takes as input a boolean predicate, and a well-formed and validated data annotation document. The boolean predicate is generally of the form `element = "someValue"` or `elementValue = "someValue"`. If the boolean predicate evaluates to true, the *select* operation retrieves the portion of the input data annotation document that satisfies the boolean predicate. The subset of the data annotation document returned contains the immediate hierarchy of nodes in the input data annotation document that satisfy the boolean predicate. If the specified boolean predicate matches with a node that has one or more descendents, then the *select* operation returns the entire node hierarchy enclosed between the start and end tags of this node. However, if the node that satisfies the boolean predicate has

no descendents, then the *select* operation simply returns this node. By default, the *select* operation returns the entire node hierarchy enclosed within the start and end tags of the node `<annotationAttachedTo>`. This is to give data annotation users a reference to the corresponding base data.

Fig. 35 presents the result set generated by issuing the query, $\sigma_{\text{element}} = \text{"annotation"} (\text{REFeatureLotSize})$, on the data annotation document of Fig. 25. The query, $\sigma_{\text{element}} = \text{"annotation"} (\text{REFeatureLotSize})$, signifies the selection and retrieval of the part of the data annotation document (presented in Fig. 25) that occurs within the start and end tags of the `annotation` element in the document.

The result set of the *select* query issued above contains the entire node hierarchy enclosed within the start and end tags of the element `annotation` in the example data annotation document of Fig. 25. The hierarchy within the start and end tags of `annotationAttachedTo` is always returned within an AnQL query result set. This hierarchy is returned in order to relate the data annotation to the base data that it annotates. For better semantics, in addition to returning the hierarchy within the start and end tags of `annotationAttachedTo`, an AnQL result set also returns the `annotationMetadata` hierarchy.

Loosely, a *select* query is similar to an SQL query of the form –

```
"SELECT <LIST OF COLUMN NAMES>
FROM TABLENAME ;"
```

If the boolean predicate is not specified, a *select* query corresponds to an SQL query of the form –

```
"SELECT *
FROM TABLENAME;" .
```

Without the specification of the boolean predicate, the *select* operation returns the entire input data annotation document.

```
<result>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>LOT_SIZE</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <description>
      Lot size is in acres.
    </description>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>May 28, 2004 12:15:14 PM</recorded>
    </annotationMetadata>
  </annotation>
</result>
```

Fig. 35 – Result Set of a *Select* Operation

Processing of *Select* Operation. After generating the data annotation graph corresponding to the input data annotation document, AnQL's query engine utilizes the *data annotation graph traversal* function to traverse through the graph. The strategy for graph traversal depends on the boolean predicate of the *select* operation. If the boolean predicate is of the form `element = "someValue"`, then the *data annotation graph traversal* function starts to look

for a match among nodes of type *element*. If the boolean predicate is of the form `elementValue = "someValue"`, then the *data annotation graph traversal* function tries to find a match among nodes of type *elementValue*, and so on. If the query engine finds a match, it traverses to the descendents of the node, and returns the entire hierarchy of nodes enclosed within the start and end tags of the start node. If the start node does not have any descendents, then the query engine simply returns the start node.

Project (π) Operation

The *project* operation selects portion(s) of data annotation documents based on a non-boolean constraint. The *project* operation takes as input a non-boolean constraint and a *project criterion*. The *project criterion* specifies the level of data annotation documents to be queried. The *project criteria* include `database`, `database/relation`, `database/relation/tuple`, `database/relation/column`, and `database/relation/cell`. The second input of *project* operation, a non-boolean constraint, is a single keyword or a set of keywords. The *project* operation returns the portion(s) of data annotation documents that contain the keywords. The cardinality of the set of keywords is restricted to a maximum of three words. The keywords may be the value of an *element* (i.e. occur in a node of type `elementValue` of a data annotation document), or they might occur in the data annotation itself (i.e. occur within the start and end tags of `element annotation` of a data annotation document). The keywords are joined together using *and* logic. In other words, the *project*

operation conducts an exact phrase search on data annotation documents at the specified level. By default, in the result set, the *project* operation returns the entire node hierarchy enclosed within the start and end tags of the node `<annotationAttachedTo>`. This is to give data annotation users a reference to the corresponding base data.

Fig. 36 presents the result set generated by issuing the query, $\pi_{\text{"sunny"}}(\text{REAL_ESTATE/FEATURES/PROPERTY_ID/PI2})$, on the data annotation documents of Fig. 25 and 26. This query signifies the search for the keyword “sunny” in the example data annotation documents.

```

<result>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>PROPERTY_ID</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <comment>
      Build your dream home. Sunny and
      private. Water and electricity at
      site. Plans and permits approved and
      ready for a 2683+ sq. ft home. Septic
      and geo approved.
    </comment>
  </annotation>
</result>

```

Fig. 36 – Result Set of a *Project* Operation

The result set of the *project* query issued above contains the entire node hierarchy enclosed within the start and end tags of the element that

contains the specified keyword. The hierarchy within the start and end tags of `annotationAttachedTo` is always returned within an AnQL query result set. This hierarchy is returned in order to relate the data annotation to the base data that it annotates. For better semantics, in addition to returning the hierarchy within the start and end tags of `annotationAttachedTo`, an AnQL result set also returns the hierarchy within the element `annotationMetadata`.

Processing of *Project Operation*. In order to process a *project* operation, the AnQL query engine browses through a directory (determined by the specified level), and generates a data annotation graph corresponding to each data annotation document that occurs within that directory. The AnQL query engine, utilizing the *data annotation graph traversal* function, traverses through these graphs. The query engine conducts an exact phrase search among nodes of type *elementValue* and *textValue* of these documents. The query engine returns the node(s), from all data annotation documents at the specified level, that contain the keywords specified as the non-boolean constraint.

Natural Join (nj) Operation

The *natural join* operation joins data annotation documents, at a specified level, based on a *natural join criterion*. The *natural join* operation takes as input the level at which to join data annotation documents, and a *natural join criterion*. The levels of *natural join* include `database`, `database/relation`, `database/relation/tuple`, `database/relation/column`, and `database/relation/cell`. The *natural join* criteria include the author of the

data annotation documents, and the date of the data annotation documents.

If the *natural join criterion* is not specified, the *natural join* operation simply appends all the data annotation documents at the specified level. By default, in the result set, the *natural join* operation returns the entire node hierarchy enclosed within the start and end tags of the node `<annotationAttachedTo>`. This is to give data annotation users a reference to the base data that the data annotation documents annotate. Fig. 37 presents the result set generated by issuing the query, `njauthor(REAL_ESTATE/FEATURES/LOT_SIZE/PI2)`, on the example data annotation documents of Fig. 25 and 26. The query, `njauthor(REAL_ESTATE/FEATURES/LOT_SIZE/PI2)`, indicates to the AnQL query engine to compare the authors within the cell-level example data annotation documents presented in Fig. 25 and 26, and join these data annotation documents if their authors are the same.

The result set of the *natural join* query issued above contains the entire node hierarchy enclosed within the start and the end tags of the element `annotation`. The hierarchy enclosed within the start and the end tags of `annotationAttachedTo` is always returned within an AnQL result set. The hierarchy is returned in order to relate the data annotation to the base data that it annotates. For better semantics, in addition to returning the hierarchy within the start and end tags of `annotationAttachedTo`, an AnQL result set also returns the hierarchy within the element `annotationMetadata`.

```

<result>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>PROPERTY_ID</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <comment>
      Build your dream home. Sunny and private. Water
      and electricity at site. Plans and permits
      approved and ready for a 2683+ sq. ft home. Septic
      and geo approved.
    </comment>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>Apr 1, 2004 10:15:25 AM</recorded>
    </annotationMetadata>
  </annotation>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>LOT_SIZE</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <description>
      Lot size is in acres.
    </description>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>May 28, 2004 12:15:14PM</recorded>
    </annotationMetadata>
  </annotation>

```

Fig. 37 – Result Set of a *Natural Join* Operation

Intersection (&) Operation. The definition of the *natural join* operation in AnQL also includes the definition of an *intersection* operation [5]. This is because the *intersection* operation can be defined in terms of the *natural join* operation. In

fact, the *intersection* operation in relational algebra is an additional operation defined in order to simplify common queries [5]. Consider two relations, R and S , such that, $R(A, B, C)$ and $S(A, B, C)$. In relational algebra, R intersection $S = \pi_{R.A, R.B, R.C}(\sigma_{(R.A = S.A) \wedge (R.B = S.B) \wedge (R.C = S.C)}(R \times S))$ [5].

For the above operation to be meaningful, it is necessary that both relations R and S have the same arity and data types. The corresponding condition in AnQL is that the data annotation documents are defined at the same level (i.e. all data annotation documents to be *natural joined* should be defined at one of the following levels – database, database/relation, database/relation/column, database/relation/tuple, and database/relation/cell), and are well-formed and validated.

The query, $nj_{author}(REAL_ESTATE/FEATURES/LOT_SIZE/PI2)$, (issued above) also illustrates the *intersection* operation. This query joins data annotation documents based on the *natural join criterion* $author$. The query searches through the data annotation documents at the cell level, and joins those data annotation documents that have the same author as the one specified in the query. Thus, this query “intersects” data annotation documents with the same author.

Processing of Natural Join Operation. The strategy to process *natural join* operation depends on the specified level, and the *natural join criterion*. If the *natural join criterion* is not specified, then the query engine simply appends all

the data annotation documents at the specified level. However, if the *natural join criterion* is `author`, then, the query engine compares the value of `author` node in all the data annotation graphs that were generated corresponding to the data annotation documents at the specified level. If the values of the *natural join criterion* match, the query engine returns the hierarchy of nodes enclosed within the start and end tags of element `annotation`. If the values do not match, then the query engine stops processing the query and generates an exception. The query engine proceeds in a similar fashion if the *natural join criterion* is `date`.

A Select-Project-Natural Join Example

This section presents an example *select-project-natural join* query, the result set generated by this query, and the steps that the AnQL query engine takes in order to process this query.

The query, $\sigma_{\text{element=annotation}} \pi_{\text{"water"}} \eta_{\text{Rudy Campos}} (\text{REAL_ESTATE/FEATURES/PROPERTY_ID/PI2})$, is issued on the example data annotation documents presented in Fig. 25 and 26. This query signifies the joining of all the cell-level data annotation documents whose author is `Rudy Campos`. The cell is defined by the intersection of the column `PROPERTY_ID` and the tuple whose primary key is `PI2`. The query also specifies that the result set should contain the node hierarchy within the start and the end tags of the element `annotation` (via its *select* clause), and should retrieve only those data annotations that contain the word "sunny" (via its *project* clause).

In order to process the query issued above, the AnQL query engine first processes the *natural join* clause of the query. In order to compute the *natural join* of the cell-level data annotation documents, the AnQL query engine drills down to the directory that contains data annotation documents at the cell level. Using the *data annotation graph generation* function, the query engine generates data annotation graphs corresponding to all the data annotation documents that pertain to the particular cell specified in the query. Next, using the *data annotation graph traversal* function, the query engine traverses to the author node of data annotation graphs, and compares them. If the values of the author node match, the query engine returns the entire hierarchy of nodes enclosed within the start and the end tags of element annotation. Fig. 38 presents the intermediate result set generated by the AnQL query engine. The intermediate result set presented in Fig. 38 indicates that the query engine finds a match between the authors of the data annotation documents for the cell defined by the intersection of the column `PROPERTY_ID` and the tuple whose primary key is `PI2`. Thus, in the intermediate result set, the query engine appends the two data annotation documents.

The next step that the AnQL query engine takes in order to process the query, $\sigma_{\text{element=annotation}} \pi_{\text{"water"}} \eta_{\text{Rudy}} \text{Campos}(\text{REAL_ESTATE/FEATURES/PROPERTY_ID/PI2})$, is to process the *project* clause of the *select-project-natural join* query. In order to process the *project* clause, the query engine traverses through the nodes of

type `elementValue` and `textValue` of the data annotation graphs corresponding to the two data annotation documents.

Since the intermediate result set after the processing of *natural join* operation has already been generated, the nodes of type `elementValue` and `textValue` that the query engine searches through to look for the keyword “sunny” include the values contained within the start and end tags of `database`, `relation`, `column`, `tuple`, `comment`, `description`, `author`, and `recorded`. The intermediate result set generated by the AnQL query engine after processing the *project* clause of the above query is presented in Fig. 39.

The next step that the AnQL query engine takes in order to process the *select-project-natural join* query, $\sigma_{\text{element=annotation}} \pi_{\text{“water”}} \eta_{\text{Rudy Campos}}$

(`REAL_ESTATE/FEATURES/PROPERTY_ID/PI2`), is to compute the *select* clause of this query. In order to process the *select* clause, the query engine first processes the boolean predicate, `element=annotation`. Since the boolean predicate is of the form `element=“someValue”`, the query engine uses the *data annotation graph traversal* function to traverse through the nodes of type `element`, and tries to find a match. Upon finding the annotation node in the intermediate result set, the query engine returns the entire hierarchy of nodes enclosed within the start and end tags of the start node. Fig. 40 presents the result set generated by the AnQL query engine upon the completion of the processing of the query, $\sigma_{\text{element=annotation}} \pi_{\text{“water”}} \eta_{\text{Rudy Campos}}$

(`REAL_ESTATE/FEATURES/PROPERTY_ID/PI2`). This result set is an XML

document, and contains the entire hierarchy within the start and end tags of the element `annotationAttachedTo`, so that data annotation users can refer to the base data that the data annotation documents annotate.

```

...
<annotationAttachedTo>
  <database>REAL_ESTATE</database>
  <relation>FEATURES</relation>
  <column>PROPERTY_ID</column>
  <tuple>PI2</tuple>
</annotationAttachedTo>
<annotation>
  <comment>
    Build your dream home. Sunny and private.
    Water and electricity at site. Plans and
    permits approved and ready for a 2683+ sq.
    ft home. Septic and geo approved.
  </comment>
  <annotationMetadata>
    <author>Rudy Campos</author>
    <recorded>Apr 1, 2004 10:15:25 AM </recorded>
  </annotationMetadata>
</annotationAttachedTo>
<annotationAttachedTo>
  <database>REAL_ESTATE</database>
  <relation>FEATURES</relation>
  <column>LOT_SIZE</column>
  <tuple>PI2</tuple>
</annotationAttachedTo>
<annotation>
  <description>
    Lot size is in acres.
  </description>
  <annotationMetadata>
    <author>Rudy Campos</author>
    <recorded>May 28, 2004 12:15:14PM </recorded>
  </annotationMetadata>
...

```

Fig. 38 – Result Set after the Processing of
Natural Join Operation

```

...
  <annotation>
    <comment>
      Build your dream home. Sunny and
      private. Water and electricity at
      site. Plans and permits approved and
      ready for a 2683+ sq. ft home.
      Septic and geo approved.
    </comment>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>Apr 1, 2004 10:15:25 AM
      </recorded>
    </annotationMetadata>
  </annotation>
...

```

Fig. 39 – Result Set after the Processing of
Project Clause

```

<result>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>PROPERTY_ID</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <comment>
      Build your dream home. Sunny and private.
      Water and electricity at site. Plans and
      permits approved and ready for a 2683+
      sq. ft home. Septic and geo approved.
    </comment>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>May 28, 2004 12:15:14 PM</recorded>
    </annotationMetadata>
  </annotation>
</result>

```

Fig. 40 – Result Set of *Select-Project-Natural Join Query*

Union (!) Operation

The *union* operation allows data annotation users to retrieve a “big picture” view or a consolidated view of data annotations defined at the database, relation, column, tuple, and cell levels. The *union* operation takes as input a *union criterion*. The *union criterion* specifies the level for applying the *union* operation. *Union criteria* for AnQL consist of database, database/relation, database/relation/column, database/relation/tuple, and database/relation/cell. In other words, the *union criterion* specifies the level of commonality for the consolidated view of data annotations.

Fig. 41 presents the result set generated by the query,

| database/relation/cell, on the example data annotation documents of Fig. 25 and 26. The query, | database/relation/cell, indicates to the AnQL query engine to *union* all cell-level data annotation documents. The result set generated is in the form of a report that groups data annotation documents by cells. In other words, all data annotation documents that pertain to a particular cell appear within a single group. Similar result sets are generated in case of other *union criteria*. Loosely speaking, AnQL’s *union* operation is similar to SQL’s GROUP BY clause.

Processing of Union Operation. In order to process a *union* query, the AnQL query engine browses through a directory and groups data annotations at one level into one document to generate a comprehensive view of these data annotation documents. The processing of *union* operation is tricky and tedious. Since no indexing mechanism has been employed, it is very inefficient to

compute the *union* of data annotation documents at the request of a user. Hence, the processing of *union* operation is done routinely when the data annotation management system is not very busy.

```

<result>
  <annotationAttachedTo>
    <database>databaseName</database>
    <relation>relationName</relation>
    <column>columnName</column>
    <tuple>primaryKey</tuple>
  </annotationAttachedTo>
  <annotation>
    <domainSpecificTag> ... </domainSpecificTag>
    <annotationMetadata> ... </annotationMetadata>
  </annotation>
  <annotation>
    <domainSpecificTag> </domainSpecificTag>
    <annotationMetadata> ... </annotationMetadata>
  </annotation>
  <annotationAttachedTo>
    <database>databaseName</database>
    <relation>relationName</relation>
    <column>columnName</column>
    <tuple>primaryKey</tuple>
  </annotationAttachedTo>
  <annotation>
    <domainSpecificTag> </domainSpecificTag>
    <annotationMetadata> ... </annotationMetadata>
  </annotation>
  ...
  <annotation>
    <domainSpecificTag> </domainSpecificTag>
  </annotation>
  ...
</result>

```

Fig. 41 – Result Set of a *Union* Operation

The problem with such offline processing is that the result set presented to the user upon the issuance of a *union* query might not include the

most recent dataset. Therefore, the processing of *union* operation needs to be done on a regular basis, so that the result set provided to data annotation users is not outdated by more than a few hours or days.

Combination Queries

Select, *project*, and *natural join* operations may be combined together to form *Select-Project-Join (SPJ)* queries. *SPJ* queries may be further classified into four categories – *select-with-predicate-project-with-constraint SPJ* query, *select-without-predicate-project-with-constraint SPJ* query, *select-with-predicate-project-without-constraint SPJ* query, and *select-without-predicate-project-without-constraint SPJ* query. In the case of all *SPJ* queries, AnQL query engine performs the *natural join* operation before it performs the *select* and *project* operations.

Select-with-Predicate-Project-with-Constraint SPJ Query. In a *select-with-predicate-project-with-constraint SPJ* query, a predicate for the *select* operation as well as a constraint for the *project* operation is specified. A *select-with-predicate-project-with-constraint SPJ* query corresponds to an SQL query of the form –

```

"SELECT <LIST OF COLUMNNAME>
FROM TABLENAME1, TABLENAME2
WHERE TABLENAME1.COLUMNNAME = TABLENAME2.COLUMNNAME;".

```

Fig. 42 presents the result set generated by the query, $\sigma_{\text{annotation}}$
 $\pi_{\text{water}} \text{ nj}(\text{REAL_ESTATE/FEATURES/PROPERTY_ID/PI2})$. This *select-with-*

predicate-project-with-constraint SPJ query utilizes the data annotation documents presented in Fig. 25 and 26.

```
<result>
  <annotationAttachedTo> ... </annotationAttachedTo>
  <annotation>
    <comment>
      Build your dream home. Sunny and private. Water and
      electricity at site. Plans and permits approved and
      ready for a 2683+ sq. ft home. Septic and geo
      approved.
    </comment>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>Apr 1, 2004 10:15:25 AM</recorded>
    </annotationMetadata>
  </annotation>
</result>
```

Fig. 42 – Result Set of a *Select-with-Predicate-Project-with-Constraint SPJ* Query

Select-without-Predicate-Project-with-Constraint SPJ Query. In a *select-without-predicate-project-with-constraint SPJ* query, a predicate for the *select* operation is not specified. However, a constraint for the *project* operation is specified. A *select-without-predicate-project-with-constraint SPJ* query corresponds to an SQL query of the form –

```
“SELECT *
FROM TABLENAME1, TABLENAME2, . . . , TABLENAMEK
WHERE TABLENAME[1..K].COLUMNNAME =
TABLENAME[1..K].COLUMNNAME AND
TABLENAME[1..K].COLUMNNAME = ‘KEYWORD’ ;”.
```

Issuing the query,

$\sigma_{\pi_{\text{water}} \text{ nj}_{\text{author}}}(\text{REAL_ESTATE}/\text{FEATURES}/\text{PROPERTY_ID}/\text{PI}2)$, on the example data annotation documents of Fig. 25 and 26 yields an empty result set.

Select-with-Predicate-Project-without-Constraint SPJ Query. In a *select-with-predicate-project-without-constraint SPJ* query, a predicate for the *select* operation is specified. However, a constraint for the *project* operation is not specified. A *select-with-predicate-project-without-constraint SPJ* query corresponds to an SQL query of the form –

```

“SELECT <LIST OF COLUMNNAME>
FROM TABLENAME1, TABLENAME2, . . . , TABLENAMEK
WHERE TABLENAME[1..K].COLUMNNAME =
TABLENAME[1..K].COLUMNNAME;”

```

Fig. 43 presents the result set generated by issuing the query,

$\sigma_{\text{annotationMetadata}} \pi_{\text{nj}}(\text{REAL_ESTATE}/\text{FEATURES}/\text{PROPERTY_ID}/\text{PI}2)$, on the example data annotation documents of Fig. 26 and 27.

Select-without-Predicate-Project-without-Constraint SPJ Query. In a *select-without-predicate-project-without-constraint SPJ* query, neither a predicate for *select* operation nor a constraint for *project* operation is specified. A *select-without-predicate-project-without-constraint SPJ* query corresponds to an SQL of the form –

```

“SELECT *
FROM TABLENAME1, TABLENAME2, . . . , TABLENAMEK

```

```
WHERE TABLENAME[1..K].COLUMNNAME =
TABLENAME[1..K].COLUMNNAME;”.
```

```
<result>
  <annotationMetadata>
    <author>Rudy Campos</author>
    <recorded>May 28, 2004 12:15:14 PM</recorded>
  </annotationMetadata>
  <annotationMetadata>
    <author>Rudy Campos</author>
    <recorded>Apr 1, 2004 10:15:25 AM</recorded>
  </annotationMetadata>
</result>
```

Fig. 43 – Result Set of a *Select-with-Predicate-Project-without-Constraint SPJ* Query

Fig. 44 presents the result set generated by issuing the query, $\sigma \pi_{nj}(\text{REAL_ESTATE/FEATURES/PROPERTY_ID/PI2})$, on the example data annotation documents of Fig. 25 and 26.

Select, *project*, and *union* operations may be combined together to form *Select-Project-Union (SPU)* queries. *SPU* queries may be further classified into four categories – *select-with-predicate-project-with-constraint SPU* query, *select-without-predicate-project-with-constraint SPU* query, *select-with-predicate-project-without-constraint SPU* query, and *select-without-predicate-project-without-constraint SPU* query. These queries can be utilized to retrieve a subset of a large data annotation document generated as a result of a *union* operation performed according to one of the five *union criteria*. In order to process *SPU* queries, AnQL query engine performs *union* before the *select* and *project*.

```

<result>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>LOT_SIZE</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <description>Lot size is in acres.</description>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>May 28, 2004 12:15:14 PM</recorded>
    </annotationMetadata>
  </annotation>
  <annotationAttachedTo>
    <database>REAL_ESTATE</database>
    <relation>FEATURES</relation>
    <column>PROPERTY_ID</column>
    <tuple>PI2</tuple>
  </annotationAttachedTo>
  <annotation>
    <comment>Build your dream home. Sunny and private. Water
      and electricity at site. Plans and permits approved
      and ready for a 2683+ sq. ft home. Septic and geo
      approved. </comment>
    <annotationMetadata>
      <author>Rudy Campos</author>
      <recorded>Apr 1, 2004 10:15:25 AM</recorded>
    </annotationMetadata>
  </annotation>
</result>

```

Fig. 44 – Result Set of a *Select-without-Predicate-Project-without-Constraint SPJ* Query

Select-with-Predicate-Project-with-Constraint SPU Query. In a *select-with-predicate-project-with-constraint SPU* query, both a predicate for the *select* operation, and a constraint for the *project* operation are specified.

Select-without-Predicate-Project-with-Constraint SPU Query. In a *select-without-predicate-project-with-constraint SPU* query, a predicate for the *select* operation is not specified. However, a constraint for the *project* operation is specified.

Select-with-Predicate-Project-without-Constraint SPU Query. In a *select-with-predicate-project-without-constraint SPU* query, a predicate for the *select* operation is specified. However, a constraint for the *project* operation is not specified.

Select-without-Predicate-Project-without-Constraint SPU Query. In a *select-without-predicate-project-without-constraint SPU* query, neither a predicate for the *select* operation nor a constraint for the *project* operation is specified.

The presence or absence of a predicate for the *select* operation, and a constraint for the *project* operation in *SPJ* and *SPU* combination queries is summarized in Fig. 45.

SPJ/SPU Queries	Predicate for <i>Select</i> Specified	Constraint for <i>Project</i> Specified
<i>Select-with-Predicate-Project-with-Constraint</i>	Yes	Yes
<i>Select-without-Predicate-Project-with-Constraint</i>	No	Yes
<i>Select-with-Predicate-Project-without-Constraint</i>	Yes	No
<i>Select-without-Predicate-Project-without-Constraint</i>	No	No

Fig. 45 – Combination Queries

Processing Overhead

Processing AnQL queries does not incur any additional overhead on the relational database management system that hosts the base data. The reason for this is that data annotation documents reside outside the relational database that hosts the base data. The data annotation management system allows AnQL query engine to run in conjunction with the relational database query engine. However, the two query engines are completely separate, and can reside in different systems. These two query engines can also function completely independent of each other. The relational database is primarily utilized to query and view the base data that the data annotation users want to annotate.

Limitations of AnQL

One of the limitations of AnQL is that its *select* operation is restricted to a single data annotation document. Another limitation of AnQL is that the keyword search defined within the *project* operation is an exact phrase search, and is restricted to a maximum of three keywords. In other words, the *project* operation conducts an exact phrase search using *and* logic. AnQL's *project* operation is unable to conduct searches using *or* and *not* logic. The third limitation of AnQL is that the *natural join criteria* are restricted to author and date. The fourth limitation of AnQL is that the *difference* operation has not been defined in AnQL. Moreover, the *union* operation in AnQL is highly processor-intensive.

Related Work

This section compares and contrasts the data annotation models, AnQL, and the data annotation management system presented in this thesis with the data annotation management systems discussed in Chapter II.

The data annotation models presented in Chapter III of this thesis are specifically designed to annotate base data expressed using the relational model, and residing within a relational database management system. Similarly, the annotation management system presented by Gertz *et al.* [3] is restricted to annotating scientific images, and the one presented by Zsigmond *et al.* [12] is restricted to the annotating audio-visual units. In contrast, SLIMPad, presented by Delcambre *et al.* [10], exists as a thin layer over an extensive base layer, which may be a spreadsheet, or a word processor, or a database system.

This thesis assumes that data annotation documents are stored outside the relational database system that hosts the base data. This behavior is similar to that of the annotation management system discussed by Gertz *et al.* [3]. In the system presented by Gertz *et al.*, the data annotations reside separately from the data that these data annotations annotate.

Similar to AnQL, the query language presented by Gertz *et al.* [3] also utilizes data annotation graphs to query data annotation documents. The nodes in annotation graphs utilized by the query language presented by Gertz *et al.* in [3] represent *concepts*, *annotations*, and *images*. The nodes in the data annotation graphs generated by AnQL's query engine represent the structural

elements of the data annotation document. The edges in the annotation graph utilized by Gertz *et al.* in [3] describe relationships between nodes. However, the edges of the data annotation graph presented in this thesis depict only the ancestor-descendant relationship between the nodes they connect. Similar to the data annotation models presented in this thesis, the annotations presented by Zsigmond *et al.* [12] are structured in a graph, and are represented in XML. However, the structure of annotations presented by Zsigmond *et al.* [12] is restricted to three types of nodes – set I contains link elements between annotations and binary AV data; set II consists of annotation elements (the actual descriptors); and set III contains abstract annotation elements.

The data annotation management system presented in this thesis generates data annotation graphs only during AnQL query processing. This is in contrast to the data annotation management system presented by Gertz *et al.* [3]. The system presented by Gertz *et al.* in [3] maps the nodes and edges of an annotation graph to relations stored in a relational database. Moreover, the annotation management system presented by Gertz *et al.* [3] parses and translates queries over annotations into equivalent SQL queries. AnQL does not behave in this manner. AnQL query directly processes AnQL queries.

CHAPTER V

PRELIMINARY DESIGN OF A DATA ANNOTATION MANAGEMENT SYSTEM

This chapter presents the preliminary design of a data annotation management system which combines the functionalities of the relational database management system that hosts the base data, and that of the annotation system that implements data annotations. The proposed data annotation management system allows data annotation users to create, query, and view data annotation documents, as well as to view the base data that they annotate.

Components of Data Annotation Management System

The proposed data annotation management system is composed of two major components – the base data component, and the data annotation component. The base data component of the proposed data annotation system consists of the relational database management system that hosts the base data. The proposed data annotation management system does not make any structural or schematic changes to the relational database that stores the base data. The base data component allows data annotation users to query, and hence, view the base data that they annotate. The base data component utilizes

SQL to query the data stored in the relational database. Data annotation users do not have the privilege or permission to create, modify, update, or delete base data.

The data annotation component of the proposed data annotation management system deals with data annotation documents, and has the following features –

- It allows data annotation users to create data annotation documents.
- It allows data annotation users to modify data annotation documents.
- It allows data annotation users to update data annotation documents.
- It allows data annotation users to delete data annotation documents.
- It allows data annotation users to query data annotation documents.

The data annotation component of the proposed data annotation management system utilizes AnQL to query data annotation documents.

The base component and the data annotation component of the proposed data annotation management system work in conjunction with, but, independent of each other. The proposed data annotation management system does not incur any processing overhead related to the processing and querying of data annotation documents on the relational database management system that stores the base data.

The main functions of the proposed data annotation management system can be summed up as follows –

- Allow data annotation users to create new data annotation documents.

- Allow data annotation users to modify or update existing data annotation documents.
- Allow data annotation users to query existing data annotation documents.
- Allow data annotation users to view the base data that they annotate.
- Display corresponding base data when the result set of a data annotation query is displayed, or when a data annotation document is created, modified, updated, or deleted.

The proposed data annotation management system enables data annotation users to view the base data that they annotate. This is a huge advantage for data annotation users because it gives them the ability to view the base data alongside their data annotations. It saves data annotation users from the labor and inconvenience of looking up the corresponding base data, printing it, and then trying to annotate it, or from looking at the base data and data annotations in separate windows.

States of Data Annotation Management System

This section presents the state diagrams that correspond to each of the functions listed above. The state diagrams show the steps that the proposed data annotation management system takes in order to complete the function that the data annotation user desires to undertake.

Fig. 46 shows the states that the proposed data annotation management system undergoes when a data annotation user creates a new data annotation document. The user issues an SQL query to retrieve the base data that he or she wants to annotate. The base data component of the system processes the SQL query, retrieves the base data, and displays it. The data annotation component of the proposed data annotation management system displays a blank document. The data annotation user utilizes this document to type the data annotation. The typed data annotation document is validated against a schema. If the data annotation document is successfully validated, the proposed system saves the data annotation document. If the validation of the data annotation document is unsuccessful, the proposed system alerts the user.

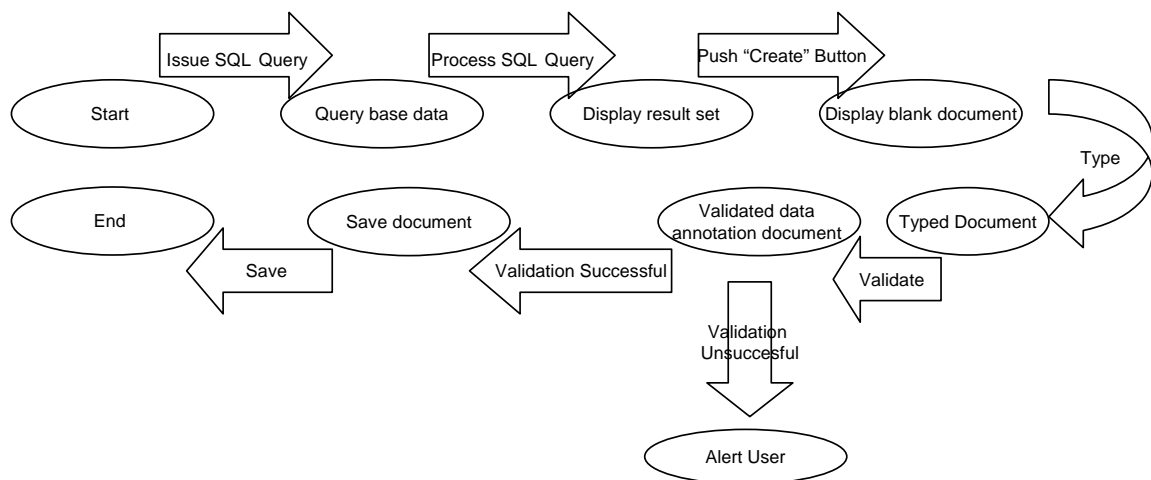


Fig. 46 – Creating a Data Annotation Document

Fig. 47 presents the states that the proposed system goes through when a data annotation user modifies an existing data annotation document. The

user begins by retrieving the data annotation document. After the user modifies the retrieved data annotation document, the data annotation document is validated against a schema. If the data annotation document is successfully validated, then the proposed system saves the data annotation document. If the validation fails, the proposed system alerts the user.

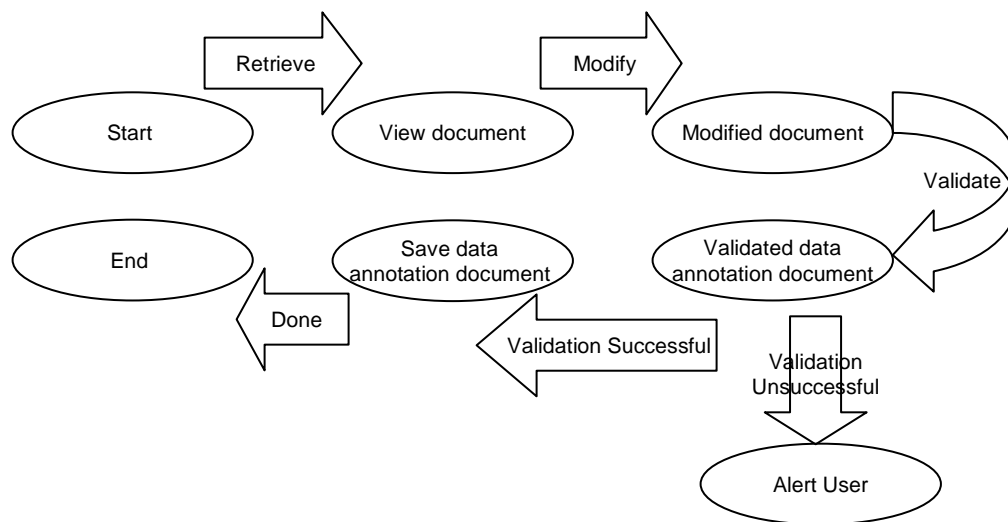


Fig. 47 – Modifying a Data Annotation Document

Fig. 48 presents the states that the proposed data annotation management system goes through when a data annotation user queries a data annotation document. The data annotation user issues an AnQL query. The data annotation component of the proposed system processes the AnQL query, and displays the result set.

Fig. 49 displays the states that the base data component of the proposed data annotation management system goes through in order to process

an SQL query. This allows data annotation users to view the base data that they annotate.

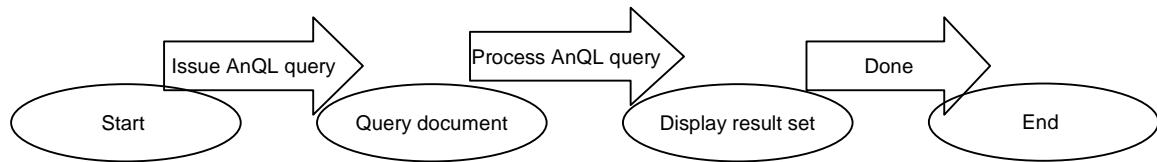


Fig. 48 – Querying a Data Annotation Document

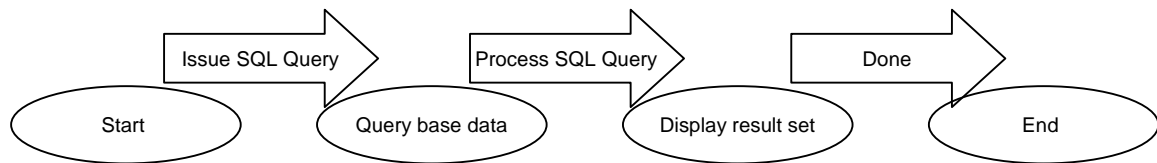


Fig. 49 – Querying Base Data

Fig. 50 presents a screen shot of the proposed data annotation management system. The data annotation component provides data annotation users with the options to create, save, modify, delete, query, and validate data annotation documents. The base data component of the proposed system provides the users with the option to query base data. Separate space is provided to view result sets of SQL queries, and to type and view data annotation documents.

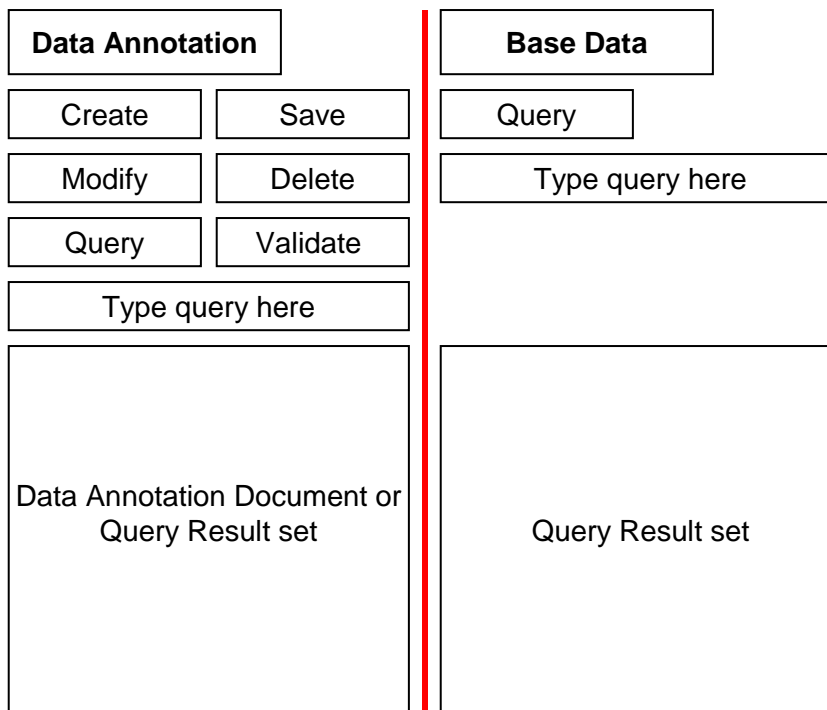


Fig. 50 – Data Annotation Management System (Screen Shot)

CHAPTER VI

CONCLUSION AND FUTURE WORK

This thesis presents a three-part solution to the problem that the metadata schema provided by most relational database management systems is not suitable for expressing semantically rich data annotations. A majority of relational database management systems utilize metadata schema to store statistical information for constraint checking and query optimization. Moreover, a majority of relational database management systems either do not allow or do not recommend that normal users update the data contained in the metadata schema. Data annotations allow users to express semantically rich metadata to help further clarify features of interest. Almost all database users, including, scientists, researchers, customers, customer service representatives, banks, and credit card companies, among others, can benefit from the use of data annotations. Data annotations help reduce communication hassles, and offer several potential opportunities to save time and effort. A few potential use case scenarios for data annotations are presented in Chapter I.

The first part of the solution presented in this thesis is the definition of data annotation models that allow users to express data annotations at five levels of granularity. These levels include the database level, the relation level, the column level, the tuple level, and the cell level. These data annotation models

are flexible, extensible, database-neutral, and platform-independent. The data annotation models are expressed using XML. These data annotation models do not require any structural and schematic changes to be made to the underlying relational database. Additionally, the data annotation models do not inflict any additional processing overhead on the underlying relational database, and therefore, do not adversely affect query processing. Another important feature of the data annotation models presented in this thesis is that the data annotation documents can cross-reference each other.

The data annotation models presented in this thesis are based on three assumptions. The first assumption is that the base data resides in a relational database management system. The second assumption is that data annotations reside outside the relational database that hosts the base data. The third assumption is that data annotation documents are stored on the file system provided by the operating system of a server or computer system.

A clever, yet naive, storage scheme is employed in order to facilitate AnQL query processing. According to this storage scheme, all data annotation documents that annotate at one level are kept in a separate directory from data annotation documents that annotate at another level. All data annotation documents for one database are kept separate from all data annotation documents for another database.

The second part of the three-part solution presented in this thesis is the Annotation Query Language (AnQL). AnQL is a query language that allows

data annotation users to query data annotation documents. AnQL's main functions include *select*, *project*, *natural join*, and *union*. AnQL's functions accept as input well-formed and validated data annotation documents. A DTD or an XML schema is employed in order to ensure that data annotation documents are well-formed and valid.

The *select* operation selects portion(s) of data annotation documents based on a boolean predicate. The *project* operation selects portion(s) of data annotation documents based on a non-boolean constraint and a *project criterion*. The *natural join* operation joins data annotation documents at a specified level based on a *natural join criterion*. The definition of *natural join* operation in AnQL includes the definition of an *intersection* operation. The *union* operation allows data annotation users to retrieve a consolidated view of data annotation documents at a specified level.

Select, *project*, and *natural join* operations are combined together to form *Select-Project-Natural Join (SPJ)* queries. *SPJ* queries are further categorized into *select-with-predicate-project-with-constraint SPJ* queries, *select-without-predicate-project-with-constraint SPJ* queries, *select-with-predicate-project-without-constraint SPJ* queries, and *select-without-predicate-project-without-constraint SPJ* queries. Similarly, *select*, *project*, and *union* operations may be combined to form *Select-Project-Union (SPU)* queries. *SPU* queries are further categorized into *select-with-predicate-project-with-constraint SPU* queries, *select-without-predicate-project-with-constraint SPU* queries, *select-with-*

predicate-project-without-constraint SPU queries, and *select-without-predicate-project-without-constraint SPU queries*.

AnQL query engine processes AnQL queries issued by data annotation users. The AnQL query engine is at the heart of AnQL. The AnQL query engine interfaces with the underlying storage mechanism that hosts the data annotation documents. *Data annotation graph generation* and *data annotation graph traversal* are the main functions of the AnQL query engine.

In order to process an AnQL query, the AnQL query engine generates data annotation graph(s) corresponding to the input data annotation document(s). The query engine uses the *data annotation graph generation* function to generate data annotation graph(s) corresponding to the input data annotation document(s). A data annotation graph is a special graph modeled in spirit of the XQuery data model, and is specially structured to facilitate AnQL query processing. The nodes in a data annotation graph correspond to the hierarchical elements in the data annotation document, and are classified as *root*, *element*, *elementValue*, or *textValue*. The edges in a data annotation graph depict the hierarchical (ancestor-descendant) relationship among nodes. The query engine utilizes the *data annotation graph traversal* function and its *transitive closure* in order to traverse through these data annotation graph(s). Depth first traversal of the data annotation graph ensures that document order is preserved during AnQL query processing.

Processing AnQL queries does not incur any additional overhead on the relational database management system that hosts the base data. The reason is that data annotation documents reside outside the relational database management system that hosts the base data.

The third part of the three-part solution presented in this thesis is the preliminary design of a data annotation management system. The proposed data annotation management system combines the functionalities of the underlying relational database management system, and the annotation management system that manages data annotation documents. The data annotation management system allows users to create, query, and view data annotation documents, and to view the base data that they annotate.

Future Work

The area of semi-structured data is relatively new, and a lot of research is being conducted in the area. As the number of data annotation documents grows, a smart indexing scheme will become necessary to search and locate data annotations efficiently. This indexing scheme might also be helpful in speeding up the processing of *union* operation. Most of the operations defined in AnQL need to be further optimized. Keyword search operation in *project* operation can be extended to search phrases of size greater than three words. The *project* operation can also be extended to include joining of words using *or* and *not* logic. Additionally, the cross-referencing section in the data

models can be extended or modified to use XML Linking Language (XLink) [22] or XML Inclusions (XInclude) [23]. Data models as well as AnQL operations can be extended to annotate and query base data expressed using the object-oriented and hierarchical data models.

REFERENCES

REFERENCES

1. Don Chamberlin, *A Complete Guide to DB2 Universal Database*, p. 713. San Francisco, CA: Morgan Kaufman, 1998.
2. Dianne Siebold, "Access SQL Server Metadata Learn how SQL Server stores metadata and what you can do with it.," *Visual Studio Magazine*, Jun. 2000
3. Michael Gertz, Kai-Uwe Sattler, Fredric Gorin, Michael Hogarth, Jim Stone, "Annotating Scientific Images: A Concept-based Approach," *Proc. Int'l Conf. Scientific and Statistical Database Management*, pp. 59-68, 2002.
4. Kenneth B.Sall, *XML Family of Specifications A Practical Guide*. Boston, MA: Addison Wesley, 2002.
5. Silberschatz, Korth, Sudarshab, *Database System Concepts*, New York, N.Y.: McGraw Hill, 2002.
6. Don Chamberlin et al., *XQuery from the Experts A Guide to the W3C XML Query Language*. Boston, MA: Addison-Wesley, 2004
7. José Kahan, Marja-Riitta Koivunen, Eric Prud'Hommeaux, Ralph R. Swick, "Annotea: An Open RDF Infrastructure for Shared Web Annotations," *Computer Networks: the International Journal of Distributed Informatique*, vol. 39, pp. 589-608, Aug. 2002.
8. Recommendation, Extensible Markup Language 1.0, W3C, 2004.

9. Kevin Williams, "XML for Data: Native XML databases: a bad idea for data?," <http://www-106.ibm.com/developerworks/xml/library/x-xdnat.html>, 12 Apr. 2004.
10. Lois Delcambre, David Maier, Shawn Bowers, Mathew Weaver, Longxing Deng, Paul Gorman, Joan Ash, Mary Lavelle, Jason A. Lyman, "Bundles in Captivity: An Application of Superimposed Information," *Proc. Int'l Conf. Data Eng.*, pp. 111-120, 2001.
11. Wang-Chiew Tan, "Containment of Relational Queries with Annotation Propagation," *Proc. Int'l Conf. DBPL*, pp. 37-53, 2003.
12. Elöd Egyed-Szigmond, Yannick Prié, Alain Mille, and Jean-Marie Pinon, "A Graph-based Audiovisual Document Annotation and Browsing System," *RIAO 2000*, vol. 2, pp. 1381-1389, Apr. 2000.
13. Michael Gertz, Kai-Uwe Sattler, "A Model and Architecture for Conceptualized Data Annotations," *Proc. 14th Int'l Conf. Scientific and Statistical Database Management*, IEEE CS Press, pp. 59-68.
14. Anni Coden, Norman Haas, Robert Mack, "Multisearch of Video Segments Indexed by Time-Aligned Annotations of Video Content," *IBM Thomas J. Watson Research Center Exploratory Computer Vision Group*, <http://www.research.ibm.com/ecvg/pubs/haas-vista.pdf>, 22 Jan. 2004.
15. Ronald Bourret, "Annotated XML Specification, " <http://www.xml.com/axml/axml.html>, 30 Jun. 2004.

16. Abdul H. Al-Azzawe, "IBM Video Online for e-business - DB2 Inbound XML Data Fragments",
<http://www-106.ibm.com/developerworks/db2/library/techarticle/0204alazzawe/0204alazzawe.html>, June 2004.
17. Wang-Chiew Tan, D. Bhagwat, L. Chiticariu, G. Vijayvargiya, "An Annotation Management System for Relational Databases,"
<http://www.cs.ucsc.edu/~wctan/papers/2004/ams-vldb04.pdf>, 2004.
18. XSL Transformations (XSLT), W3C, 2003.
19. Cascading Style Sheets (CSS), W3C, 2003.
20. "Chronology Prehistoric Life – The Inside Story, "
www.bbc.co.uk/dinosaurs/chronology
21. Architecture Domain, XML Schema, W3C, 2000.
22. Recommendation, XML Linking Language 1.0, W3C, 2001.
23. Working Draft, XML Inclusions (XInclude) 1.0, 2003.