Formal concept analysis and semantic file systems

Ben Martin
University of Wollongong

PhD Thesis: Formal Concept Analysis and Semantic File Systems

by

Mr Ben Martin
B.I.T., Queensland University of Technology
M.I.T., Queensland University of Technology

A thesis submitted to the
School of Information Systems and Technology
University of Wollongong in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
School of Information Systems and Technology
2008

Thesis Certification

CERTIFICATION

I, Benjamin M. Martin, declare that this thesis, submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Information Systems and Technology, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

(Signature)
Benjamin M. Martin
19 October 2008

Ben Martin, Mr (Ph.D., Information Science)

PhD Thesis: Formal Concept Analysis and Semantic File Systems

Thesis directed by Prof. Peter Eklund

The thesis is that a branch of discrete mathematics, Formal Concept Analysis, when applied to Semantic File Systems can lead to an improved personal information space. Semantic File Systems share many properties with their non semantic brethren, bringing more rich metadata and the ability to directly resolve user queries within the filesystem interface itself.

A filesystem might offer upwards of a million files each of which having in the order of hundreds of discerning attributes. Formal Concept Analysis has typically been applied to a much smaller input data set and there are issues with scalability both in the initial finding of the set of Formal Concepts and also ongoing issues such as finding the list of files which are currently applicable (the extent) for a Formal Concept.

The thesis is largely dependent on improving the scalability of Formal Concept Analysis in order for it to be applied to such a large dynamic data store.

Dedication

To the authors of great novels:
Though I have enjoyed many of your works, I have enjoyed too few of your works.

Acknowledgements

Professor Peter Eklund has made this PhD possible. I thank him for his understanding of the value of applied research, the difficulty in performing it and his encouragement and guidance throughout the candidature.

Thanks to associate professor Roger Duke for his guidance during the thesis. His sense of humor brightened the days of many administrative difficulties during the middle of the candidature.

Thanks to Robert Murphy for listening to queries about indexing and relational database design and providing valuable insight into SQL throughout the years both of the PhD and predating it. Apologies for always wanting to talk about libferris over the years.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Formal Concept Analysis is a mathematical method which takes as input a binary relation between two sets and offers as output a partially ordered set of clusters which minimally represents this binary relation. The clusters are ordered from least to most specialized cluster and form a finite lattice. The technique is a form of unsupervised machine learning or concept clustering.

Various challenges arise when one attempts to apply Formal Concept Analysis to a large dynamic data source. These include issues of scalability, formatting the formal input data to the Formal Concept Analysis process in a manner to produce interesting results and the actual algorithm chosen to find the clusters from the input data. The very presentation of the clusters and their relations to each other in a compelling manner is also an interesting research topic as the number of clusters increases.

The dynamic information source chosen for this thesis is a modern **Semantic** File System [39, 64]. The differences between a traditional Filesystem and a Semantic File System are detailed in Section 2.3. For this chapter it is sufficient to consider the Semantic File System as a superset of the traditional file system. The file system [90, 59] has become the de facto standard for storage and management of semi structured data on computers. Various file formats such as XML formally describe how a semi structured document is to be serialized to be stored in a single file on a filesystem.

File systems have evolved from presenting a list of named objects, files, which contain a contiguous range of bytes to the modern file system comprised of a tree structure augmented with soft and hard links, sparse files, extended attributes and transparent support for many on-disk storage formats. Extended Attributes (EA)s [2] allow the creation, update and removal of key-value data that is stored on a per file basis.

With the inclusion of soft and hard links, a modern filesystem is no longer a tree structure and in general is not even required to be a Directed Acyclic Graph (DAG).

File systems perform many roles including storage of a user's data as well as metadata and configuration settings. When most users consider metadata that is stored by a file system they think of a file's size, modification time, access permissions etc. While such metadata has been in common use for a very long time, modern file systems allow much more metadata to be stored and retrieved [2]. It has become common place for applications to store their configuration settings in the user's home directory under UNIX systems, extending the use of file systems to containing metadata about application instances themselves.

These many ways in which modern filesystems handle many data sources, together

with the diverse manners in which filesystems are used today, makes them a good choice for a data source when performing Formal Concept Analysis. The scope of what can be analysed is again broadened when applied to a Semantic File System. Choosing to apply Formal Concept Analysis to a modern Semantic File System allows not only traditional filesystems but also relational databases, XML data and graph structured data to be analysed.

There are many limitations of a hierarchical file system model which are addressed by using Formal Concept Analysis. The most striking being that a tree structure forces logical containment of files in a directory and the encoding of metadata about each file into its path in the tree [25, 30]. The logical containment issue is that a file normally only exists in a single directory. This can be eased by use of soft and hard links but in so doing the semantics of file access become more complicated (dangling links, cycles during link resolution). Encoding metadata into a file's path hinders the location of conceptually similar documents because a small change in one piece of metadata may require one to scan from the root of the tree to find a document. Consider the example where paths are created by first encoding the year the document was authored and then the general type of document such as audio, video or text. If one is browsing `/2003/text/whitepaper/libferris/fcasurvey.tex` and wishes to find other libferris white papers that were not necessarily authored in 2003 then they must try other branches from the root of the tree and scan down a similar path from each of those.

To alleviate the single access path issue many file systems offer the ability to find conceptually similar documents by showing the results of a query as a file system [39, 47, 41]. Such views have the drawback of being read only or allowing inconsistency and usually being somewhat separate from the standard navigation paths. In moving to the finite lattice structure of Formal Concept Analysis both querying and navigation are presented via the same interface and a user can seamlessly switch between both styles of interaction [30].

Parts of the Introduction Chapter assume that the reader has some familiarity with Formal Concept Analysis. Formal Concept Analysis will be presented in detail in Section 2.2. Very briefly, the "clusters" mentioned above are called Formal Concepts, the ordering of clusters is a Concept Lattice and the input data that the clusters are found from is a Formal Context. The term "Formal" shall be left out where no ambiguity arises, referring to clusters simply as concepts.

1.1 Hypothesis

The thesis is that a branch of discrete mathematics, Formal Concept Analysis, when applied to Semantic File Systems can lead to an improved personal information space. Semantic File Systems share many properties with their non semantic brethren, bringing more rich metadata and the ability to directly resolve user queries within the filesystem interface itself.

The issues mentioned in the previous section about the encoding of file metadata in file system paths and the logical containment of files in directories can be remedied with the integration of Formal Concept Analysis into the system. Formal Concept Analysis has a solid mathematical background with formally defined semantics [38]. The use of Formal Concept Analysis thus provides a solution to the above problems complete with a well defined mathematical treatment as opposed to attempts to simply

hide these issues with ad-hoc solutions.

In order to provide this improved personal information space, issues relating to the scalability of Formal Concept Analysis must be addressed. The application of Formal Concept Analysis must be able to support large dynamic data sets. This will typically be in the order of millions of files each having potentially hundreds or thousands of interesting attributes. This is the key issue of the thesis – if the system requires days or weeks to perform Formal Concept Analysis on a user's filesystem then it will be of no use.

Scalability needs to be addressed in many places: indexing a filesystem such that queries can be performed to obtain a formal context, indexing on the formal context such that the set of concepts can be obtained and updated contingent and extents can be found as the formal context varies with the underlying data set and the ability to turn a set of concepts into a concept lattice in a timely manner.

1.2 Prior work on Formal Concept Analysis and File Systems

Applying FCA to file systems can be seen from many perspectives in the literature. Much research has been done on applying FCA to text document collections [19, 55]. FCA has also been applied to more structured data such as email [25].

Ferré and Ridoux present an alternate representation of FCA as Logical Concept Analysis (LCA) in [31] where the lattice (O, A, I) has the attributes A replaced by an (possibly infinite) lattice of formulas and I attaches formulas to the objects in O. It has been shown that LCA and Formal Concept Analysis are equivalent. LCA is used to apply Formal Concept Analysis to file systems as a whole [30].

Interacting with a filesystem through a concept lattice, directories become concepts, symbolic links are no longer required and files form the object set O in the formal context. There is no requirement for symbolic links because FCA allows a file to exist in many concepts at the same time. Because a lattice structure allows a concept to have multiple parents it allows objects that are conceptually close to each other to be close in the lattice as well [13].

Consider the previous example of looking for other libferris white papers. The example encoded metadata in directory names to form a tree such as `/2003/text/whitepaper/libferris/fcasurvey.tex`. Using Formal Concept Analysis one would only need to move up the lattice to loosen the restriction in the time dimension to see other related libferris papers. To gain access to informal documentation on libferris one could then navigate upward to remove the white paper attribute.

Ferré and Ridoux [30] generalize the current working directory `pwd(1)` into a history stack of working concepts. This is done to allow one to move to the correct parent concept easily. The familiar command `cd ..` becomes a pop operation on the history stack or a move to the root concept if the history is empty. The semantics of `cd ..` become more of a navigation backwards than a navigation to the last direct super concept as detailed in [30]. The change in semantics is because the relative or absolute move in the lattice is not broken into sub parts and pushed individually but each refinement provides a single push onto the stack. If one is doing FCA using this working concept stack then one could break a navigation into its component attributes and push them as individual refinements. For example, given the working concept

"/2003/text" and a command `cd whitepaper/libferris` the stack could have the two attributes pushed onto it in the order presented on the command line.

Due to concepts being multi parented there can be many paths from the root concept to any concept in the lattice. The "parent concept" stack should however still be maintained in the order given by the user so that only the last attribute in the path is removed by a `cd ..` command, ie. the particular absolute path chosen to find a concept is only relevant to future relative path operations. If one can easily strip off the last attribute of a path then one only needs to store the path used to reach the current concept to allow relative path operations. Presumably such a technique was not chosen for [30] due to the use of formulas for attributes in Logical Concept Analysis.

The `ls` command is made modal in [30] by separating out the query of the extent of a concept (`ls -r f`) from query for the refinements available from a working concept (`ls f`). This seems somewhat artificial as the traditional UNIX `ls` command makes no distinction between showing only places to navigate to against showing only the files in the current directory.

Using the working concept one can easily navigate the concept lattice using familiar commands `cd`, `ls` and `pwd` modifying the lattice using `mv`, `cp` and `rm` as has been done in the Conceptual Shell [30]. Although altering the current working directory with `cd` should be easy enough, explicit detail is not given in [30] about how one resolves copy, move and remove operations on objects in the lattice. The most challenging operation would be the `mv` command. Consider the case of moving an image from a subconcept of "true colour" to a subconcept of "monochrome". Such an operation would require a lossy transformation to occur on the actual image data in order to maintain the semantic consistency of the objects in each concept in the lattice.

A commonly noted distinction is between intrinsic attributes of an object which may be mechanically determined from an object's byte content, and extrinsic properties which require user interaction to determine. Extraction of intrinsic properties from documents covers many domain specific algorithms such as by using the Ripple-Down Rule (RDR) knowledge acquisition and maintenance methodology from knowledge based systems [55], email headers, regular expression matches and machine learning algorithms [25, 92], or an arbitrary extraction function [30].

Extrinsic attributes for objects are discussed less than intrinsic. The Conceptual Email Manager (CEM) [25] uses extrinsic attributes to allow the user to override intrinsic attributes to always be `true` or `false` for a particular message and also to allow CEM to update attributes such as "mail read" and "new mail" [25].

The collection of intrinsic and extrinsic attributes is used to form the attribute set A for FCA. In CEM [25] the attributes create a partial order $(A, \leqslant)$ such that the transitivity in the partially ordered attributes is also reflected in the relation I of the formal context (O, A, I), ie. If for an object $o \in O$ and an attribute $a \in A$ if oIa then $\forall \mu \in$ transitive-parent(a), $oI\mu$. This allows one to not only tag files with the most specific attributes but to find them in the formal context using less specific attributes relative to $(A, \leqslant)$. The partial order $(A, \leqslant)$ in CEM is edited using a tree widget in which multi parented attributes appear under each of their parent attributes in the tree.

Although Ferré and Ridoux use formulas as their attributes they too apply an ordering to their attributes [31, 30]. Their formulas are ordered by a possibly infinite lattice and they present methods to enable navigation of the concept lattice built from these formulas using contextualized logic.

CEM [25] creates conceptual scales automatically based on the partial order it maintains over the attributes in the formal context. A default scale $\mathbb{S}_\mu$ is created $\forall \mu \in (A, \leqslant)$ such that $\mathbb{S}_\mu$ is true iff an object o has any of the direct children attributes of μ in $(A, \leqslant)$. Using LCA for file systems [30] has a similar setup using the lattice of formulas it follows that one formula μ deduces all formula below it in the lattice.

1.3 Methodology

This thesis directly concerns **applied** Formal Concept Analysis. As such the complexity of the algorithms, the amount they require to use the systems hard disk and seek disk heads and ultimately the amount of time required to perform certain tasks is critical to the thesis.

The complexity of the application of Formal Concept Analysis is directly dependent on the input data set that forms the formal context. The analysis of many of the algorithms relies on the output of the process: the number of concepts, how many connections there are between concepts, or how wide the concept lattice is. Note that the output and input are directly related but not in a manner that allows algorithmic complexity expressed in terms of the number of concepts to be easily expressed in terms relating to the formal context.

This makes traditional algorithm performance analysis such as the "Big-O" average and worst case complexity more complex to derive. Moreover the worst case complexity may be largely irrelevant because the cases which can trigger such worst case are not encountered in reality.

Attempts to present more accurate algebraic performance of algorithms would require intimate knowledge of the data distribution of the formal context. In the cases in this thesis the performance relies heavily on the use of sophisticated custom indexing tailored for performance on secondary storage. Attempting to derive algebraic complexity for these systems would thus require analysis of both the data distribution and recursive page splitting and subsequent distributions of the pages in the index structure. Such analysis would likely lead to very complex formulae and would be difficult to apply to new data sets where the exact data distribution has not been mathematically abstracted.

As such, empirical testing has been performed to validate the improvements claimed by the research. Many sections of the thesis thus contain empirical test results to explicitly quantify the improvements offered. As these results drawn are from many publications [75, 76, 74, 73] and were performed using different software and hardware configurations they are spread over the thesis rather than combined into a single chapter.

1.4 Major Results

Major contributions are presented in the area of indexing support for Formal Concept Analysis on large dynamic data sets.

It has been found that the use of traditional B-Tree indexing by relational databases is ineffective for applied Formal Concept Analysis [75]. The thesis starts by presenting spatial indexing methods which offer query resolution 80 times faster when Formal Concept Analysis is applied to a large data source. This performance is obtained with

RD-Tree spatial indexing. This is more impressive again when one considers that hundreds of these queries might be lodged at the database in order to visualize a single concept lattice. Such results make the application of Formal Concept Analysis to a large data set in the order of hundreds of thousands to a million objects tractable.

The RD-Tree index structure mentioned above is further improved by the use of customized asymmetric page splitting algorithms and custom compression. The compression must have knowledge of the Formal Concept Analysis scales that are used to generate the data that is being indexed. When employing such page splitting and customized compression tailored to Formal Concept Analysis only 16% the number of internal keys are touched relative to a RD-Tree for a 16 attribute index. For the IBM synthetic Data Mining input the Formal Concept Analysis based page split with Formal Concept Analysis tailored index compression could actually reduce the index tree height from 4 to 3 which will directly reduce the number of disk seeks required for each query. This impact will be more again when one considers that a relational database will cache the root index node and quite possibly the direct children of the root node. This leads to the Formal Concept Analysis based index requiring one disk seek compared to the standard RD-Tree requiring two. Disk head seeks are the single slowest action performed when accessing an index stored on a hard disk drive.

Investigations into applying Formal Concept Analysis to Semantic File Systems is presented in Chapter 6. This looks at the structure of the concept lattice that is created in various circumstances and ways to create a lattice which is more interesting for the user. This chapter diverges from typical Formal Concept Analysis application in the removal of the assumption that a "domain expert" will be present to generate scales and assist the end user in the creation of concept lattices.

The link between Data Mining and the finding of Closed Frequent Itemsets and the process of finding the set of all concepts for a formal context is built on in Chapter 5. Although mathematically the ordering relation among concepts is implicitly available once the set of all concepts is found, in a computer system this ordering must be explicitly recorded. A new algorithm for quickly making the covering relation explicitly available is presented and empirically tested. This new "border" algorithm is found to be very effective for the target that it was designed for – making the covers explicit for up to a few thousand concepts. The major advantage of the border algorithm is that it does not require the formal context at all. This makes it efficient to apply where the data source is large and dynamic and simply reading the formal context sequentially implies a large performance cost.

Chapter 7 investigates the application of Formal Concept Analysis to system security. It was found that the two security policies offered in Fedora Linux 7, the targeted and strict security policies, both offer the same level of information flow protection when considered in a transitive manner. Considering transitive information flow with Formal Concept Analysis in this manner allows one to see where information can move to in a computer system when a number of intermediate security states may be required. The use of Formal Concept Analysis in this manner allows one to see which security states in the system share a "transitive cluster" in that information can more readily flow between these security states than others.

Chapter 8 presents investigations into Semantic File Systems themselves. The major contribution being how a Semantic File System can in many ways be seen as XML and vice versa. This allows things like arbitrary bidirectional mappings to be

setup between Semantic File Systems to enable schema mutation. Schema mutation inside the Semantic File System allows possibilities such as the schema of a relational database to be mutated to the schema of an Office document. With such a bidirectional schema mutation in place, an end user application such as OpenOffice can be used to directly open and edit a table in a relational database.

1.5 Impact and Importance

Although the thesis is primarily concerned with the application of Formal Concept Analysis to a Semantic File System data set many of the methods and results can be translated to other large data sources. In particular the indexing improvements presented have been empirically tested against standard large datasets such as the IBM synthetic data generator [88] and the mushroom and covtype databases from the UCI dataset [17].

The scalability offered by the methods in this thesis enable Formal Concept Analysis to be applied to datasets which would previously have been avoided. A single query once took over 55 minutes and can now be resolved in just over 40 seconds on the same computer. When one considers that the new resolution does not require any sequential scans over the data set and only relies on the index structure for resolution and the previous best practice needed 90 sequential scans to resolve, the scalability of the index structure presented becomes even more impressive. As the size of the data set is expanded an order of magnitude the old method would gain more than a linear number of sequential scans. This only serves to increase the performance advantage of the new indexing structures.

The Border algorithm in Chapter 5 also directly supports application to larger data sets because it operates only on the set of concepts and does not require any knowledge of the formal context.

Chapter 6 works towards a system where an average user who is ignorant of Formal Concept Analysis can still take advantage of a system employing Formal Concept Analysis where a "domain expert" is not present. Various methods are presented which modify the concept lattice to be more user friendly. Although the techniques presented in Chapter 6 assume that the chapter reader has knowledge of Formal Concept Analysis the techniques themselves can be offered by a computer system for the use of system users ignorant of Formal Concept Analysis.

The link of Semantic File Systems to XML shown in Chapter 8 has already been noticed by private industry with the publication in the Linux Journal [70] and a presentation of the technique at the Ottawa Linux Symposium 2007 [69].

1.6 Overall Structure

Background information on both Formal Concept Analysis and Semantic File Systems is presented in Chapter 2. This is followed by a description of building filesystem indicies and the various trade-offs to this new field of "desktop search" in Chapter 3. Improvements to indexing for Formal Concept Analysis is presented in Chapter 4. Without the work in Chapter 4 Formal Concept Analysis could not be applied in a timely manner to data sources as large as a filesystem. A new algorithm for finding the covering relationship among concepts is then presented in Chapter 5 which does not require

access to the formal context at all in order to find the covering relation. With the ability to resolve Formal Concept Analysis queries in a timely manner and the ability to find the covering relation on the concepts, the concept lattice, Chapter 6 investigates the application of Formal Concept Analysis to Semantic File Systems and various ways to modify the application to achieve more pleasing results. Attention is then turned to applying Formal Concept Analysis to the security of filesystems in Chapter 7. Security is considered in the context of a modern Mandatory Access Control system. Various advances to Semantic File Systems themselves that were researched during the PhD candidature are then presented in Chapter 8 followed by the Conclusion of the thesis.

The work in Section 8.2 appeared in ADCS 2003 [61]. Section 1.2 appeared in ICFCA 2004 [62]. Section 2.3 contains information from ICFCA 2004 [62]. Section 6 contains information from ADCS 2005 [73]. Section 4.2 and Section 4.3 contain information from ICFCA 2006 [75]. Section 4.4 contains information from ISMIS 2006 [74] and ICFCA 2007 [76].

Chapter 2

Background

2.1 Introduction

This section presents information on both Formal Concept Analysis and Semantic File Systems.

2.2 Formal Concept Analysis Preliminaries

This section will first present Formal Concept Analysis in an informal manner using a running example. This will be proceeded by a more formal treatment. The interested reader will find Formal Concept Analysis presented with full mathematical rigour in [38]. The example is the Hungarian educational film taken from [38].

Formal Concept Analysis is concerned with the relation of a set of Objects to a set of Attributes which those objects may or may not posses. It is convenient to present this relation in a cross table as shown in Figure 2.1. The objects in Figure 2.1 are various living things and the attributes describe their desired living space, ability to move around, having limbs, and other features.

In its purest form, Formal Concept Analysis is only concerned with binary cross tables where objects either have an attribute or do not. Input data which is not in the form of a binary relation must first be converted into this format. For example, if there was a "mass" for each object that would likely be expressed as grams or some other numeric value. To use such information with Formal Concept Analysis one must first translate the values into binary attributes. For example, one could assign light, average and heavy classes with fixed mass values which delineate each class. These three binary attributes could then be used with Formal Concept Analysis.

The translation of one or more non binary attribute(s) into one or more binary attribute(s) using a predicate like this is called Logical Scaling [86, 87] in the Formal Concept Analysis community. The input binary relation is referred to as a Formal Context.

Given a cross table like Figure 2.1 as input, Formal Concept Analysis will find all the strongest groupings (cluster) of objects and attributes. A strongest grouping can be found by starting at any particular cross in the table and trying to expand a rectangle out in all directions. The catch is that arbitrary reordering of the rows and columns in the cross table is allowed if it will allow the rectangle to expand. For example, consider the "One seed leaf" attribute for the Maize object. If the Maize row is exchanged with the row above it then the "One seed leaf" rectangle can be expanded to include both

	Needs water to live	Lives in water	Lives on land	Needs chlorophyll	two seed leaves	one seed leaf	can move	has limbs	suckles offspring
Leech	×	×					×		
Bream	×	×					×	×	
Frog	×	×	×				×	×	
Dog	×		×				×	×	×
Spike - weed	×	×		×		×			
Reed	×	×	×	×		×			
Bean	×		×	×	×				
Maize	×		×	×		×			

Figure 2.1: Context of an educational film “Living Beings and Water”.

the "Spike - weed" and "Reed". Both of these objects have "Needs chlorophyll" so the rectangle can be expanded to include that attribute. Note that both "Spike - weed" and "Reed" have "Lives in Water" but this attribute cannot be included because the third object (which we originally started with) Maize does not have "Lives in Water". We now have a 2 row by 2 column sized rectangle. This rearrangement process must be repeated until it is no longer possible to expand the rectangle. At that stage a single "strongest grouping" has been found. Then a different starting cross in the table is to be selected and the process repeated until all crosses have been used as a starting point.

As can be seen from the above informal description the location of the strong groupings can be a very computationally expensive task. Let us assume that the number of crosses in the cross table is $|I|$, the number of attributes is $|A|$, the number of objects is $|O|$ and $|C|$ is the number of strong groupings (formal concepts). The upper bound complexity for computing the number of strong groupings is $2/3 \times 2^{\sqrt{|I|+1}}$ but much better algorithms have been developed with $O(|A||C||O|)$. In practice, computing the concept lattice of a formal context is cubic in the size of the number of objects or attributes, whichever is larger. As discussed in Section 5 there are techniques to obtain a well defined subset of the strong groupings which offer even better performance. A comprehensive survey of algorithms and a characterization of their space and time complexity can be found in [20].

When all the strong groupings have been found they can be arranged in a Hasse diagram by considering groupings which have more attributes to be extensions or children of groupings with a subset of their attributes. For example, a grouping with the attributes "Can_move, Has_limbs" might have a grouping with attributes "Can_move, Has_limbs, Suckles_offspring" as an extension or child grouping.

The cross table shown in Figure 2.1 has its strong groupings shown in a Hasse diagram in Figure 2.2. The attributes for a grouping are shown above the node for that group. The objects in a grouping are listed below its node. Note that when an attribute appears on a node x it automatically applies to all the other nodes which are connected (transitively) below x. An attribute label appears as high as it possibly can on the diagram. Conversely an object label is added as low in the diagram as possible. The object label attached to node y applies to all other nodes connected (transitively) above y. This specialized form of Hasse diagram is known as a labelled line diagram and is a diagrammatic representation of a concept lattice.

Notice that since all objects "need water to live" that the strong grouping shown at the top of the diagram contains that attribute. Looking at the strong grouping for Leech we can see a nearby grouping containing Bream. The only strong grouping that is not a parent of Leech is "Has limbs" and so we can see that this is the discerning fact between the two objects. Considering the two attributes "Lives on land" and "Needs chlorophyll" if we want to find objects which have the latter but not the former we need to trace downwards from "Needs chlorophyll" being careful not to move to a strong grouping which has "Lives on land" as a (transitive) parent. It turns out that the only such object is "Spike - weed".

A strong grouping is referred to as a Formal Concept in the Formal Concept Analysis community. The arrangement of strong groupings (cf. Formal Concepts) in a Hasse diagram based on their attributes (cf. Formal Attributes) is called a Concept Lattice. The discussion now focuses on giving a more formal treatment to the above prose.

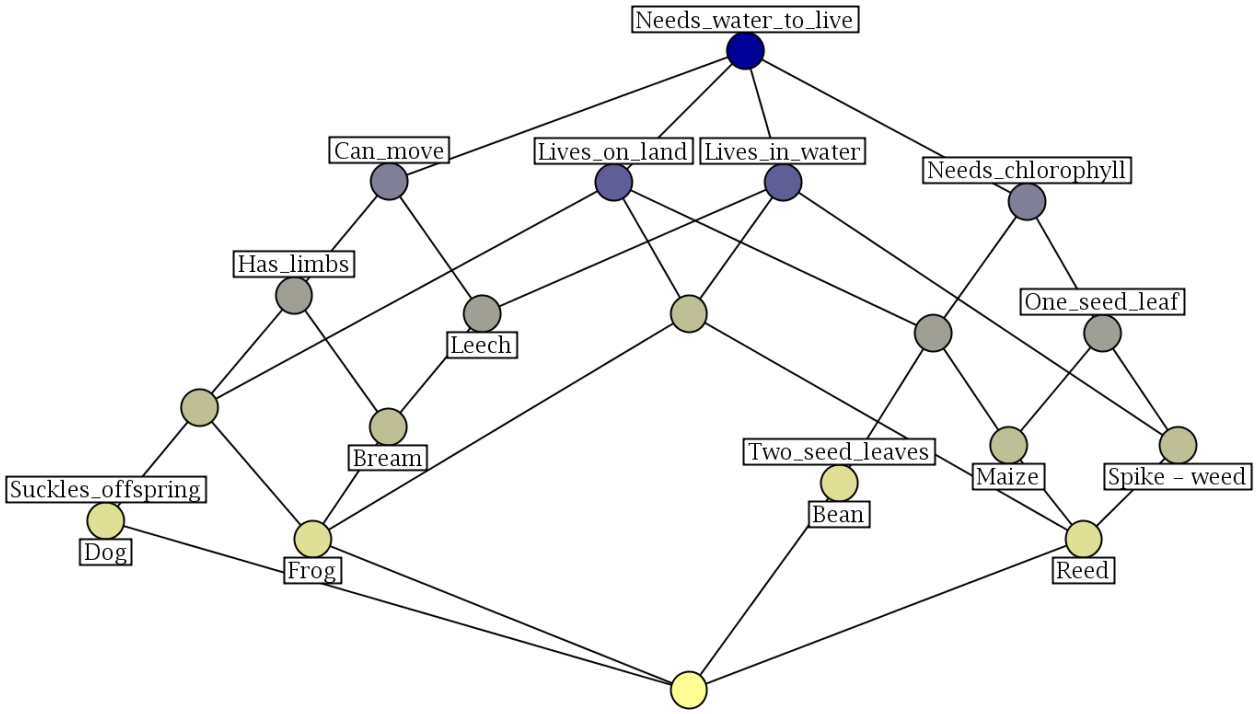

Figure 2.2: Hasse diagram for the strong groupings for the cross table in Figure 2.1. In Formal Concept Analysis terminology this is the Concept Lattice for the Formal Context shown in Figure 2.1.

Formal Concept Analysis [38] is a branch of discrete mathematics concerned with the study and application of the Galois connection between two sets [27] (for this reason a **Concept lattice** is often called a **Galois lattice** in the French speaking world). The input to Formal Concept Analysis is a binary relation between two sets. The binary relation is referred to as a Formal Context. A formal context is a triple (O, A, I) where O is a set of objects, A is a set of attributes, and I is a binary relation between the objects and the attributes, i.e. $I \subseteq O \times A$. There is no requirement that O and A be different or disjoint sets.

The Formal Context can be displayed to the user as a Concept Lattice. A Formal Concept of a Formal Context (O, A, I) is a pair (X, Y) where $X \subseteq O$, $Y \subseteq A$, $X = \{o \in O \mid \forall m \in Y : (o, a) \in I\}$ and $Y = \{a \in A \mid \forall o \in X : (o, a) \in I\}$. For a concept (X, Y), X is called the extent and is the set of all objects that have all of the attributes in Y, similarly Y is called the intent and is the set of all attributes possessed in common by all the objects in X. As the number of attributes in Y increases, the concept becomes more specific, i.e. a specialization ordering is defined over the concepts of a formal context by:

$$(X_1, Y_1) \leq (X_2, Y_2) :\Leftrightarrow Y_2 \subseteq Y_1$$

Formal Concept Analysis is always performed on a binary relation. To extend the possibilities for input data of Formal Concept Analysis, a non binary relation can be converted into a binary relation (Formal Context) by a process called scaling.

Many-valued contexts and scaling are now presented more formally.

For non binary relations, data is organized as a table and is modeled mathematically as a many-valued context, (O, A, W, I_w) where O is a set of objects, A is a set of attributes, W is a set of attribute values and I_w is a relation between O, A, and W such that if $(o, a, w_1) \in I_w$ and $(o, a, w_2) \in I_w$ then $w_1 = w_2$. In the table there is one row for each object, one column for each attribute, and each cell is either empty or asserts an attribute value.

A many-valued context holds many similarities to a single table in the relational model [52, 78].

A refined organization over the data is achieved via **conceptual scales**. A conceptual scale maps attribute values to new **binary** attributes and is represented by a mathematical entity called a formal context. A conceptual scale is defined for a particular attribute of the many-valued context: if $\mathbb{S}_a = (O_m, A_m, I_m)$ is a conceptual scale of $a \in A$ then we define $W_a = \{w \in W \mid \exists (o, a, w) \in I_w\}$ and require that $W_a \subseteq O_a$. The conceptual scale can be used to produce a summary of data in the many-valued context as a derived context. The context derived by $\mathbb{S}_a = (O_a, A_a, I_a)$ w.r.t. to plain scaling from data stored in the many-valued context (O, A, W, I_w) is the context (O, A_a, J_a) where for $o \in O$ and $n \in A_a$

$$\begin{aligned} o J_a n \;\Leftrightarrow\colon\quad & \exists w \in W : (o, a, w) \in I_w \\ & \text{and } (w, n) \in I_a \end{aligned}$$

Scales for two or more attributes can be combined in a derived context. Consider a set of scales, S_a, where each $a \in A$ gives rise to a different scale. The new attributes supplied by each scale can be combined:

$$N := \bigcup_{a \in A} A_a \times \{a\}$$

Then the formal context derived from combining these scales is:

$$gJ(a,n) \Leftrightarrow: \quad \exists w \in W : (o,a,w) \in I_w \text{ and } (w,n) \in I_a$$

Creating a binary relation that can be used as $I \subseteq O \times A$ can be done by creating conceptual scales as outlined above [24, 25, 23, 86] or using logical scaling [86, 87]. Both conceptual and logical scaling can be seen as a method to take one or more columns in a many valued context and generate one or more new binary columns as the result.

Some standard scaling techniques include: nominal, ordinal and inter-ordinal.

A nominal scale for an attribute A_y generates a new attribute in the output for each value of W_y which A_y takes in the input. If an object $o \in O_y$ has value $w \in W_y$ for attribute $a \in A_y$ then it will have attribute A_{ay} in the output.

An ordinal scale takes an attribute A_y which has a naturally ordered set of values W_y and divides the input value range into many linear intervals to form output attributes. An inter-ordinal scale combines two ordinal scales, one using $\leq$ the other $\geq$ on its ordinal range.

An ordinal scale can be used on a many-valued attribute for which there is a natural ordering, for example, size<=4096,size<=1Mb and so on.

In this representation more specific concepts have larger intents and are considered "less than" ($<$) concepts with smaller intents. The analog is achieved by considering extents, in which case, more specific concepts have smaller extents. The partial ordering over concepts is always a complete lattice [38].

For a given concept $C = (X, Y)$ and its set of lower covers $(X_1, Y_1)...(X_n, Y_n)$ with respect to the above $<$ ordering the object contingent of C is defined as $X - \bigcup_{i=1}^{n} X_i$. The object contingent shall subsequently be referred to as the **contingent** where ambiguity does not arise.

2.3 Semantic File Systems

The notion of a semantic filesystem was originally published by David K. Gifford et al. in 1991 [39].

A semantic file system differs from a traditional file system in two major ways:

- Interesting meta data from files is made available as key-value pairs.
- Allowing the results of a query to be presented as a virtual file system.

Interesting meta data is extracted from a file's byte content using what are referred to as transducers in [39]. An example of a transducer would be a little bit of code that can extract the width of a specific image file format. A transducer to extract some image related metadata is shown in Figure 2.3. Image dimensions are normally available at specific octet offsets in the file's data depending on the image format. A transducer which understands the PNG image encoding will know how to find the location of the width and height information given a PNG image file.

Queries are submitted to the file system embedded in the path and the results of the query form the content of the virtual directory. For example, to find all documents that have been modified in the last week one might read the directory `/query/(mtime>=begin last week)/`. The results of a query directory are calculated

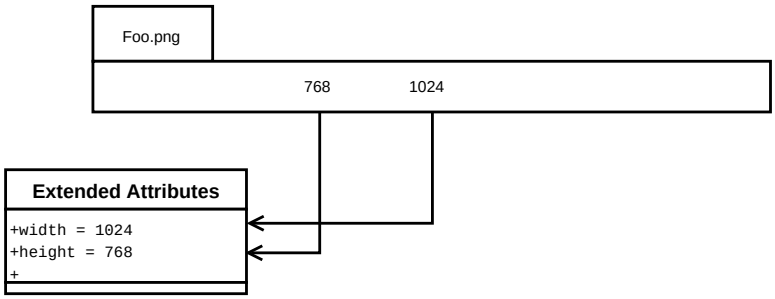

Figure 2.3: The image file Foo.png is shown with it's byte contents displayed from offset zero on the left extending to the right. The png image transducer knows how to find the metadata about the image file's width and height and when called on will extract or infer this information and return it through a metadata interface as an Extended Attribute.

every time it is read. This is a virtual filesystem because the directory does not exist in a persistent form, it is materialized on demand. Any metadata which can be handled by the transducers [39] (metadata extraction process) can be used to form the query.

Another use of the term "virtual filesystem" is the ability to select among various filesystem implementations. The Linux kernel [59, 90] maintains such an arrangement allowing incompatible filesystems such as Ext3 and XFS to be used through a common interface. The implementation used for testing in this thesis fulfills both uses of the virtual filesystem term.

The semantic filesystem used for experimentation and empirical testing during the course of the PhD was libferris [3, 64, 66, 68, 70]. Libferris was chosen because it was already in a stable state at the beginning of the PhD, it offers the ability to mount many data sources as a filesystem, it can extract interesting metadata from many file formats, and as it was written by the PhD author. The later point enables quick applied research because the code base is familiar to the author.

Although there are many filesystems which implement many of the features of a Semantic File System it is extremely difficult to compare them all. For example, the Strigi indexing feature of KDE4 includes support for digging into archives, indexing the individual files that are contained within them. Strigi also includes support for extraction of metadata on a per file basis and searching for information using this metadata. Unfortunately Strigi is not coupled with the KDE4 virtual filesystem interface directly and as such applications need to have explicit support for accessing search results. Thus although Strigi comes close to being a Semantic File System it does not use virtual filesystem interfaces to obtain the data it indexes and does not offer search results primarily through a virtual filesystem interface.

The filesystem implementation in BeOS[1] includes support for both metadata extraction and index and search and can be categorized as an Semantic File System. Relative to the BeOS implementation, libferris includes support for multiple index formats, federated search, RDF, and geotagging among other features.

A recent and widely known attempt to implement a Semantic File System is

[1] http://en.wikipedia.org/wiki/BeOS

WinFS[2] . Although most of the literature does not promote WinFS as a Semantic File System, all of the required features are present in the design. Unfortunately the availability of WinFS being limited and the widespread negative press about its performance does not fair well for its use in the PhD research. The reader will also see in Section 3 that using a fully relational schema for indexing semi structured data, such as a Semantic File System, has performance and extensibility implications.

It is a significant advantage that libferris is open source software. Because the source code is freely available, the indexing structures presented in Chapter 4 were able to be implemented and integrated into the system for empirical verification without any fixed API barriers to closed components to worry about. Moreover, as the entire solution is freely available, third parties can verify the results presented in the thesis and build on this research.

Although libferris was chosen as the implementation for empirical testing in this thesis, the results of the thesis are directly applicable to other Semantic File System like implementations and indeed other Formal Concept Analysis tools. For example, although Strigi does not present itself as a virtual filesystem, the indexing structures presented in Chapter 4 could be used to allow Strigi to return search results based on the application of Formal Concept Analysis.

Some details are now presented about libferris. References to the availability of various data sources which can be exposed as a filesystem by libferris are relevant to allow the reader to see the scope of where Formal Concept Analysis can be applied through the system described in the thesis. Many readers will not consider things like relational databases, XML and RDF [85] as being "filesystems" though they can be seen that way through libferris.

Libferris is a virtual file system. This means that there are many different filesystem implementations and libferris automatically selects which implementation is appropriate for any given URL. For example, given a URL starting with "postgresql://" the PostgreSQL database filesystem implementation is selected. A URL beginning with "file://" will use an implementation based on the operating system's file access system calls.

An innovation that libferris brings is the ability for the filesystem itself to automatically chain together implementations. The filesystem implementation can be varied at any file or directory in the filesystem. For example, in Figure 2.4 because an XML file has a hierarchical structure it might also be seen as a filesystem. The ability to select a different implementation at any directory in a URL requires various filesystems to be overlaid on top of each other in order to present a uniform filesystem interface.

When the filesystem implementation is varied at a file or directory then two different filesystem handlers are active at once for that point. The left side of Figure 2.4 is shown with more details in Figure 2.5. In this case both the operating system kernel implementation and the XML stream filesystem implementation are active at the URL "file://tmp/order.xml". The XML stream implementation relies on the kernel implementation to provide access to a byte stream which is the XML file's contents. The XML implementation knows how to interpret this byte stream and how to allow interaction with the XML structure though a filesystem interface.

Note that because in the above the XML implementation can interact with the

[2] http://en.wikipedia.org/wiki/WinFS

operating system kernel implementation to complete its task this is subtly different to standard UNIX mounting where a filesystem completely overrides the mount point.

The core abstractions in libferris can be seen as the ability to offer many filesystem implementations and select from among them automatically where appropriate for the user, the presentation of key-value attributes that files posses, a generic stream interface [58] for file and metadata content, indexing services and the creation of arbitrary new files.

Filesystem implementations allow one to drill into composite files such as XML, Indexed Sequential Access Method (ISAM) files,[3] databases or tar files and view them as a file system. This is represented in Figure 2.4. Having the virtual filesystem able to select among filesystem implementations in this fashion allows libferris to provide a single file system model on top of a number of heterogeneous data sources[4] .

Presentation of key-value attributes is performed by either storing attributes on disk or by creating synthetic attributes whose values can be dynamically generated and can perform actions when their values are changed. Both stored and generated attributes in libferris are referred to simply as Extended Attributes (EAs). Examples of EAs that can be generated include the width and height of an image, the bit rate of an mp3 file or the MD5 [4] hash of a file. This arrangement is shown in Figure 2.6.

For an example of a synthetic attribute that is writable consider an image file which has the key-value EA `width=800` attached to it. When one writes a value of `640` to the EA `width` for this image then the file's image data is scaled to be only 640 pixels wide. Having performed the scaling of image data the next time the `width` EA is read for this image it will generate the value `640` because the image data is `640` pixels wide. In this way the differences between generated and stored attributes are abstracted from applications.

The Resource Description Framework (RDF) [85] was created by the W3C to allow metadata to be stored, queried and exchanged. The use of an RDF repository in Figure 2.6 allows the Semantic File System to always be able to store metadata. Some filesystems will not naturally allow metadata to be stored, for example, when an FTP server is mounted as a filesystem the user may not have the ability to write to the filesystem. By using a personal RDF repository libferris can allow metadata to be written for files on the mounted FTP server by keeping this information in the personal RDF repository and offering it again for these files in the future. In this way the user of the Semantic File System does not need to be concerned with whether a filesystem supports writing metadata or not.

The use of RDF as a metadata source also allows the Semantic File System to obtain metadata from other users or the Internet about files and integrate that information into the system as a whole.

Another way libferris extends the EA interface is by offering schema data for attributes. Such meta data allows for applicable collations to be set for a data type with a default selected for ordering of values, for example, integer vs. string comparison. Graphical interfaces can also present data in a format which the user will find more intuitive, for example presenting dates with a calendar interface. As will be mentioned later, having the correct method of ordering values of a data type (collation) and for an

[3] eg. B-Tree data stores such as Berkeley db

[4] Some of the data sources that libferris currently handles include; http, ftp, db4, dvd, edb, eet, ssh, tar, gdbm, sysV shared memory, LDAP, mbox, sockets, mysql, tdb and XML.

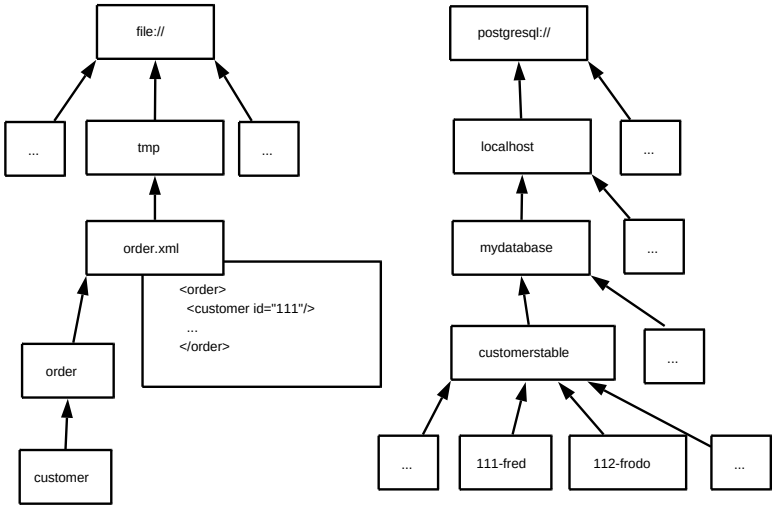

Figure 2.4: A partial view of a libferris filesystem. Arrows point from children to their parents, file names are shown inside each rectangle. Extended Attributes are not shown in the diagram. The box partially overlapped by `order.xml` is the contents of that file. On the left side, an XML file at path /tmp/order.xml has a filesystem overlaid to allow the hierarchical data inside the XML file to be seen as a virtual filesystem. On the right: Relational data can be accessed as one of the many data sources available though libferris.

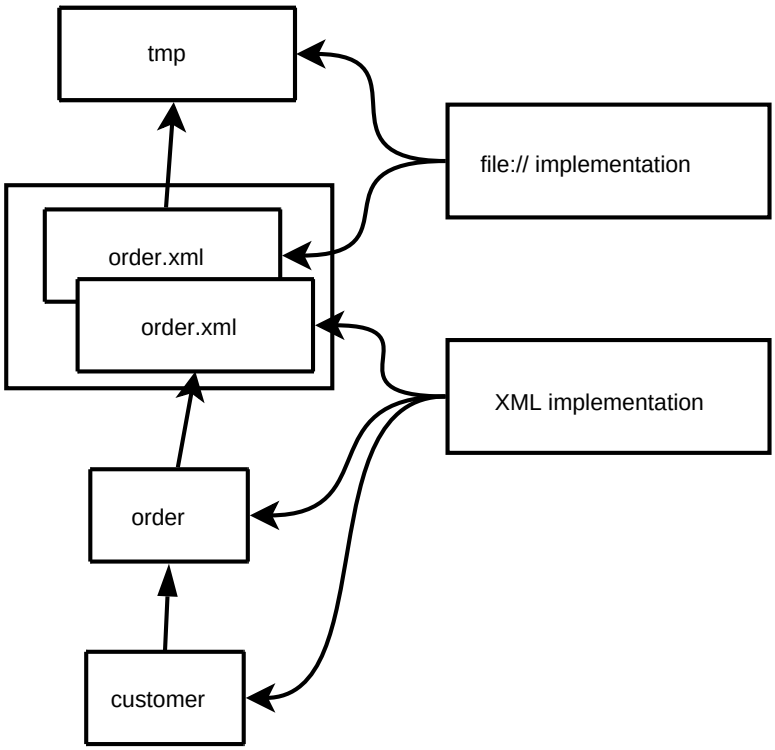

Figure 2.5: The filesystem implementation for an XML file is selected to allow the hierarchical structure of the XML to be exposed as a filesystem. Two different implementations exist at the "order.xml" file level: an implementation using the operating system's kernel IO interface and an implementation which knows how to present a stream of XML data as a filesystem. The XML implementation relies on the kernel IO implementation to provide the XML data itself.

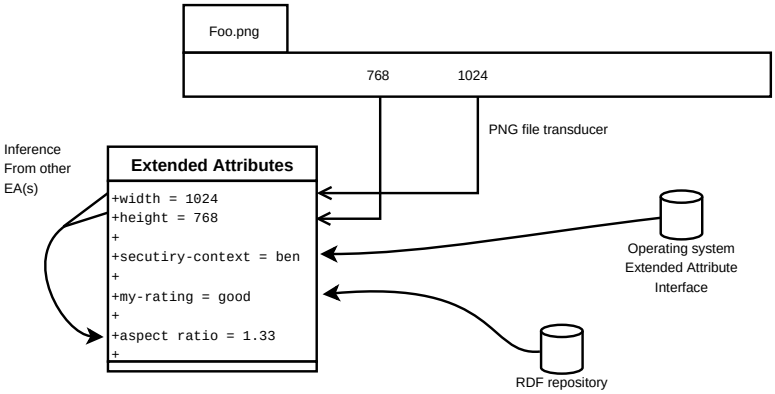

Figure 2.6: Metadata is presented via the same Extended Attribute (EA) interface. The values presented can be derived from the file itself, derived from the values of other EA, taken from the operating system's Extended Attribute interface or from an external RDF repository.

EA is a prerequisite to generating logical scales for use in Formal Concept Analysis.

In order to quickly find the set of files that have a given attribute value indexing is available on both the full text of files [104] and on their EAs. The EA index is maintained in a sorted order to allow the list of files for which a comparative constraint on their EA holds. For example `width<800` can be resolved completely using only the index. Much more detail is presented on indexing and its use in Formal Concept Analysis in Section 3.

The term "indexing" could be used to refer both to the process of adding a new filesystem document to the index as managed by libferris and also to refer to a Relational Database Management System (RDBMS) index. To avoid ambiguity some terminology is introduced here which is used throughout the thesis. An index of document metadata as managed by libferris will be referred to as an `findex`. From an Formal Concept Analysis perspective, one can consider an `findex` as being a large many-valued context from which many formal contexts can be derived. A query against an `findex` will be referred to as an `fquery`. Obtaining the files which match an `fquery` will be referred to as resolving the `fquery`. This helps to avoid ambiguity when the `fquery` is translated into another query language such as SQL by the `findex` engine. The creation of a formal context from the `findex` can be viewed as the resolution of many `fqueries` which define the attributes of the formal context.

It is natural for one to consider the files and directories of a Semantic File System as forming the object set for Formal Concept Analysis. This is more so in a Semantic File System than a traditional filesystem because in many cases a file can appear as a directory in an Semantic File System depending on the context. For example, an archive file like a `tar.gz` file will appear as a file but can also be read directly as though it was a directory. In this way one might wish to use the metadata for their files to form the attribute set for Formal Concept Analysis. If one has some binary metadata about a file, for example `is-character-device`, then its presence can be taken to directly imply a connection in the input binary relation $I \subseteq O \times A$ for Formal Concept Analysis.

The requirement for input to Formal Concept Analysis to be in the form of a binary relation presents challenges when applying it to a Semantic File System in general. This is due to the fact that the metadata attached to the files in a Semantic File System is rarely simple binary values. Also some metadata which at first appears to be binary may have additional structure which should be taken into consideration. For example, the libferris Semantic File System has the notion of tags. A tag allows the user to categorize their files either explicitly or implicitly [61]. This may at first appear to be a simple binary attachment where for a specific tag a file either has that tag associated or it does not. However the tags in libferris themselves form a partially ordered set and as such the association of an tag x also conveys information about a files' association with all the parent tags of x in this partial order.

There are three distinct tasks that must be performed in order to produce a Concept Lattice from a Semantic File System, these are shown in Figure 2.7. Firstly the information from the Semantic File System must be indexed in some format. This is the creation of the `findex` and also what this chapter is concerned with. Secondly the information from the `findex` must be converted into a Formal Context through either logical scaling or conceptual scaling. This is discussed in Section 6.2. Finally the set of Formal Concepts must be identified from the Formal Context and the relations between those concepts explicitly recorded. This is the focus of Section 5.

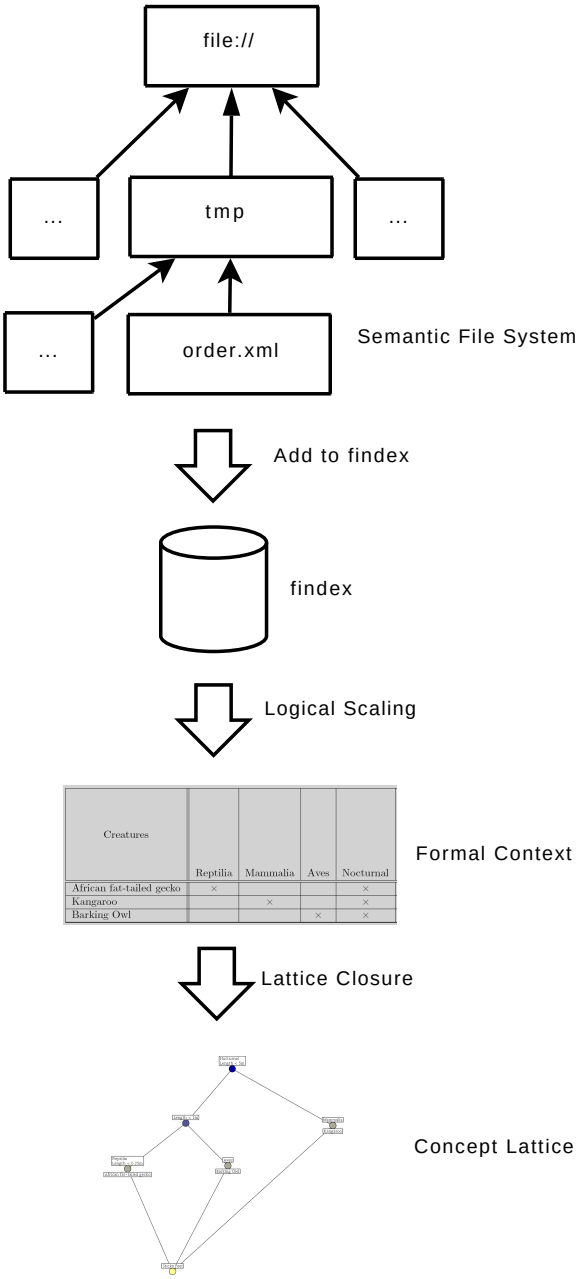

Figure 2.7: The three tasks to get from a filesystem to the result of Formal Concept Analysis: the Concept Lattice.

Chapter 3 focuses on how indexing is performed for the libferris Semantic File System. The discussion focuses on how to arrange indexing for EAs – the key-value metadata pairings. Attention is then turned to how to improve performance for applied Formal Concept Analysis in Chapter 4.

Chapter 3

Indexing Considerations

3.1 Introduction

This chapter describes the building of indexes on Semantic File Systems and the issues related to constructing such indexes. Applying Formal Concept Analysis to file systems will require the filesystem to be indexed in order to possibly achieve an acceptable level of performance. The chapter starts by describing the problem, in Section 3.2 indexing of human language file contents is briefly discussed before attention is turned to the complex issue of indexing filesystem metadata in Section 3.3. To index metadata two different solutions are considered based on inverted files in Section 3.3.4 and relational databases in Section 3.3.5. Storing many versions of file metadata in the index and the performance implications of this is considered in Section 3.3.5.3. Finally attention is turned to empirical performance testing in Section 3.3.6.

One can consider the metadata (EAs) from an SFS to define a collection of tuples as shown in Figure 3.1.

The creation of an index for a human language document or metadata key-value pairs (EAs) requires different data structures and implementation techniques. Indexing must be able to satisfy queries including comparisons on eavalues for given eanames. On the other hand, human language queries are mainly concerned with the presence of terms in resulting documents, for example, finding all documents containing the two words "Semantic XQuery" anywhere in their contents. Although the frequency which terms appear in documents and the document length can be used to perform ranked full text queries the user will not specify term frequencies in the query itself.

It is also desirable for indexes to be able to retain past and present values for metadata on files. It is quite likely that the user will remember some metadata about files from a given point in time and submit a query based on this. The ability to find files which had given metadata during an interval in time allows such queries to succeed even though metadata values have changed. For example, one might like to search for files which they thought to be "important" a year ago. Files might have been subsequently modified to be no longer important but we are interested in finding the files which were important a year ago. As such the query must be evaluated not against the metadata

```
{ file-url, eaname, eavalue }
```

Figure 3.1: Abstract tuple view of Semantic File System metadata.

Boolean Fulltext Query	Alice	Wonderland
gutenberg_alice13a.txt	×	×
Frodo.txt		
Thesis.tex	×	×
gutenberg_frsls10.txt	×	

Figure 3.2: A Formal Context for the two term full text query "alice wonderland".

as it stands now but against the metadata as it was a year ago to return the results which where important a year ago.

Turning again to the size of the index. For a moderate sized Semantic File System there can easily be 500,000 files each having between 100 and 1,000 EAs. A minimal "client" install of Fedora Core 2 Linux will have 57,000+ files after install. This file count will naturally increase once the user places source code, digital images, music files etc. onto the system.

The indexing system currently implemented in libferris is capable of resolving queries involving both full text and metadata components [66]. The following presents some of the issues with creating such indexes, solutions to these issues and empirical testing of the `findex` structure.

3.2 Indexing full text

Full text indexing is handled in the traditional manner using inverted files [104, 56]. For other work on Formal Concept Analysis and full text indexing see [19].

As boolean full text queries can be considered to present a binary relation between terms and documents they are able to be much more mechanically fed into Formal Concept Analysis. The research has focused on the indexing of metadata and how this can be used with Formal Concept Analysis.

As an example, consider the fulltext query "alice wonderland" which defines two Formal Attributes each being defined by the presence of a single query term in the document. A Formal Context which might result from this query is shown in Figure 3.2.

3.3 Indexing metadata

The range of values produced by reading an EA for all files on the filesystem will be different depending on the properties of the EA. Some EA will almost surely be bound to a unique value for each object, for example the MD5 checksum of a file [4]. Some EAs will have a single value shared for a great many files, for example an EA which is `true` if a user can read a file. Some EAs will not have a value for many files, for example the `width` EA will likely only have a value for image and video files. EA which have a value for most files shall be referred to as *dense* EAs. In contrast, EA such as `width` will be referred to as *sparse*. Some EA will be fully dense in that they have a value for all files, these include the file name, the URL and the size of the file.

The indexing structures used to store the EA will have to gracefully handle both dense and sparse value binding as well as the range of values for each EA being small and large.

Three things are required of the EA `findex`:

- Ability to `findex` or re`findex` a document. The re`findex` process presents the issue of invalidation and compaction of the existing data in the `findex` for a file. For example, when a file is already in the `findex` it may be first removed requiring the free space in the `findex` to be reclaimed [33].
- Ability to apply logical scaling to the `findex`. The `findex` for an SFS will have many non single valued attributes, as such it is important for the indexing system to efficiently be able to apply logical scaling to the many valued attributes to generate new single valued attributes. The syntax and abilities for scaling are outlined in Section 3.3.2.
- Ability to quickly resolve queries which involve predicates on many EAs. For example, listing the contingent or extent of a concept. To obtain this goal requires specialized indexing as detailed in Chapter 4.

3.3.1 Reindexing a File

The time which the document is indexed should be recorded by the `findex` to allow subsequent reindexing of a filesystem to avoid needless work. This value will be referred to as the `docidtime`. The name is derived from the fact that every file in the index will have a document identifier: the `docid`. If a file has not been modified in a "meaningful way" since it was last indexed there is no need to reindex the file.

As detailed in Section 2.3 and in particular Figure 2.6 the libferris Semantic File System allows metadata from many sources to be stored and presented for a file. This complicates the procedure to work out if a file has changed in a meaningful way. For example, metadata can be stored in a personal RDF repository for a file which the user does not have modification access too. With such functionality checking if the file's modification or ctime (time of last status change) is newer than the time that the file was last indexed will not detect modification of metadata stored as RDF [14, 18]. The Semantic File System must use a more advanced method to track and update the time that the last modification to any metadata about a file has occurred.

3.3.2 Query Syntax and Semantics

The query syntax and semantics will have a direct effect on the logical scaling which is able to be efficiently performed by the `findex`.

The primary purpose of the syntax is to define a standard implementation independent syntax and semantics for queries on an EA index. Having a single `fquery` syntax allows the logical scales defined for use in FCA to be neutral of the particular implementation of `findex`. Each `findex` implementation is responsible taking queries in this single `fquery` syntax and returning the files which satisfy that `fquery`. The `fquery` syntax must be powerful enough as to allow the logical scales that are desirable to be expressed in the `fquery` syntax.

The user is free to directly use the `fquery` syntax to pose queries directly to the SFS if they wish though this is secondary to the above goal.

The chosen query syntax is designed based on the "The String Representation of LDAP Search Filters" [43]. This is a very simple syntax which provides a small set of

comparative operators to make `key operator value` terms and a means to combine these terms with boolean `and`, `or` and `not`. All terms are contained in parenthesis with boolean combining operators located before the terms they operate on.

The comparative operators have been enhanced and in some cases modified from the original semantics [43]. Syntax changes include the use of `==` instead of `=` for equality testing. Approximate matching `~=` was dropped in favor of regular expression matching using perl operator syntax `=~`. Operators which are specific to the LDAP data model have been removed. The operators and semantics are presented in Table 3.1. Coercion of `rvalue` is performed both for sizes and relative times. For example, "begin today" will be converted into the operating system's time representation for the start of the day that the query is executed.

Operator	Semantics
`=~`	`lvalue` matches the regular expression in `rvalue`
`==`	`lvalue` is exactly equal to `rvalue`
`<=`	`lvalue` is less than or equal to `rvalue`
`>=`	`lvalue` is greater than or equal to `rvalue`

Table 3.1: Comparative operators supported by the libferris search syntax. The operators are used infix, there is a key on the left side and a value on the right. The key is used to determine which EA is being searched for. The `lvalue` is the name of the EA being queried. The `rvalue` is the value the user supplied in the query.

Resolution of the `and` and `or` is performed (conceptually) by merging the sets of matching file names using either an intersection or union operation respectively. The semantics of negation are like set subtraction: the files matching the negated subquery are removed from the set of files matching the portion of the query that the negation will combine with. If negation is applied as the top level operation then the set of files to combine with is considered to be the set of all files. The nesting of `and`, `or` and `not` will define what files the negation subquery will combine with. As an example of negation resolution consider the `fquery` which combines a width search with a negated size search: `(&(width<=640)(!(size<=900)))`. The set of files which have a width satisfying the first subquery are found and we call this set A. The set of files which have a size matching the second part of the query, ie, `size<=900` are found and we call this set B. The result of the query is $A \setminus B$.

3.3.3 Two Designs for the `findex`

Two competing designs for the `findex` have been identified: inverted files and the relational model. An implementation of both methods has been created within libferris to allow empirical testing and to support the work in later chapters of the thesis. Each `findex` implementation allows the user to create multiple `findex`es. The user can also create an `findex` using any of the different indexing implementations.

The designs will have different performance characteristics and perhaps more importantly will make available different extra searching capabilities. Extra searching capabilities are those that are outside of the `fquery` syntax discussed in Section 3.3.2.

With disk prices being so cheap the user may wish to use multiple different indexing options and combine the strengths of each when searching.

The inverted file implementation uses a collection of specially sorted inverted files as detailed in Section 3.3.4 (using Berkeley db [7]). The relational model implementation is detailed in Section 3.3.5 (backed by PostgreSQL [6]).

A Null indexing implementation has also been created to find the amount of time that is spent in reading and inferring the EA values for files. The Null indexing module reads all the attributes for files which are to be indexed and does nothing. All queries against the Null engine return an empty result. The Null engine exists to provide a baseline for how much of the overall time is spent obtaining the values of EAs. Note that reading of all attributes for a file may require significant time [39].

In both designs a file will have a unique identifier. For `findex`es that support temporal analysis a file will have a single `urlid` and many `docid`s for each time it has been observed (reindexed) in the past. For `findex`es that do not support temporal analysis the `docid` is sufficient to uniquely identify a file.

Query resolution is normally two phase, first a `fquery` string is resolved to a obtain a set of distinct integers which are thought of as the Document Identifiers. A Document Identifier is referred to as `docid` in this paper. A second phase takes a `docid` and returns a string which is the URL for that document. This setup allows for both a small number of documents to be fully resolved and shown to the user and the total number of matching documents to be quickly known. The use of `docid`s provides good support for inverted file based indexing engines and does not penalize relational schema. Other indexing schemes may need to generate synthetic `docid`s and cache results so that `docid`s can be resolved from the cache.

The operation of re`findex`ing a document presents two choices:

- Completely remove all references to the document in the `findex` and perform an `findex`ing operation on the document.
- Retain the old `findex` data and simply add the document again as a new document. When a document is added to the `findex` the time it is added is recorded so that normal `fquery` resolution can consider only the latest version of each document's `findex` data.

The second style allows the `findex` to resolve `fqueries` against what the EA values were in the past. The only `findex` engine that supports retaining old document metadata and querying against it is the relational engine.

3.3.4 Specially Sorted Berkeley DB Files

This is a custom designed solution based on the ideas for full text indexing using inverted files [104, 16].

Because different types of data collate in different ways multiple Berkeley db files [7] are maintained. For an example of collation consider that the integer 300 is greater than 40 when compared numerically but using a lexicographical comparison 300 is smaller than 40 (cf. the UNIX ls(1) commands: `ls -l` vs. `ls -lv`). Four collations are maintained: string, case insensitive string, double and integer. Each collation is stored in a different Berkeley db file.

Each key in the Berkeley database has two parts: the EA key and EA value, see Table 3.2.

EA key	EA value	inverted list of document numbers that have a match
width	640	{3,4,5,12,51,200}
width	700	{1,6,9}
width	800	{7,13}
width	801	{8,4000}

Table 3.2: Inverted lists are stored in the order of the EA key and EA value. Partial lookups are possible given just the EA key.

The EA key and EA value taken together are referred to as the inverted list key and uniquely determine a list of matching documents. The value part of an inverted list key is the value that each file in the inverted list has for the Extended Attribute called EA key. Each number in the inverted list references the `docid` in a document name lookup table. The `docid` lookup table allows fast access to the URL of a `docid`.

By storing the inverted files with the keys and values in sorted order queries such as `width<=800` and `size>=50mb` can be resolved by determining an upper and lower bound and merging the inverted lists between these bounds. This resolution relies on the ability to lookup partial keys in the Berkeley database. For example, to resolve the query `width<=800` using the index in Table 3.2 a partial lookup on "width" is done to find the lower bound, followed by a complete lookup using the key "width800", each document list between these two lookups (inclusive) is merged to form the result. Note that "width800" will in reality be the string "width" followed by the fixed width binary value for "800".

Another asset of having this ordering on the keys in the Berkeley db file is that the inverted lists that must be merged should be stored close together on disk. Thus the fetching of the inverted lists and merge process is able to be performed using a disk seek unintensive cosequential process [33]. Assuming the index file is not fragmented such a cosequential merge will operate on a list in memory and move the disk head forward sequentially in a single sweep over the relevant disk sectors.

Resolving regular expression queries with this scheme can be time consuming depending on how many values a given key has in the index. For example, resolving `name=~^my.*foo` would involve a partial lookup on `name` to find the beginning of the range and while the EA key in the index remains the same a sequential process testing the given regular expression against the EA value in the index, merging the inverted lists for each EA value that matches the regular expression.

3.3.4.1 Inverted Lists for Dynamic Index Operation

Inverted lists used in document indexing are normally an ordered list of numbers. Storing such a structure allows aggressive compression to be employed [104]. However as many documents are added to the index the inverted lists become larger making the time to read and decode a list longer. When performing a query for the conjunction of two inverted lists one would like to limit the number of blocks read from disk. If one

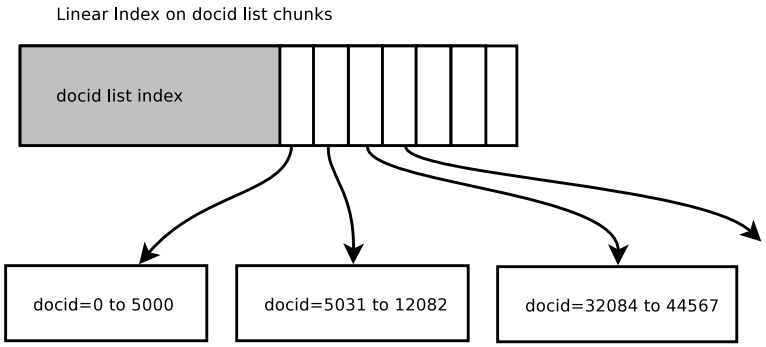

Figure 3.3: An inverted list with a linear index shown above it.

inverted list x has a gap between docid 133 and 56,001 then we would like to be able to quickly skip docids in that range in the other inverted list b. To assist skipping ranges of docids during query resolution the inverted list is normally broken into fixed size chunks with an index recording the maximum and minimum docid in each chunk [104]. An example chunked inverted list is shown in Figure 3.3.

When merging two such chunked inverted lists to form the set intersection only the chunks which might contain matching numbers need be read and decoded.

3.3.5 Relational Database

Semantic File System metadata does not map simply into the relational data model. A primary consideration is whether to attempt to extract and group related metadata into separate relations in the database. For example, the `width`, `height` and `depth` EAs are all related as "image" metadata. One can also consider these three EAs to be related as "animation" metadata when the file is not a single image but a sequence of images of the same size.

Any mapping of Semantic File System metadata into a relational database will require a table relating a document URL to a numeric document identifier: the `docid`. This shall be referred to as the `docmap` table.

It is convenient to include metadata which is expected to exist for all files directly into the `docmap` table. For example, a file's size and modification time are almost always available for storage.

The time which the document was indexed (`docidtime`) is also stored in the `docmap` table to allow subsequent reindexing of a filesystem to avoid needless work as explained in Section 3.3.1.

The base schema of the `docmap` table is shown in Table 3.3.

3.3.5.1 Using Separate Relations

All mimetypes are recorded into a mimetypes relation shown in Figure 3.4.

The `docmap` table shown in Figure 3.3 is extended to contain a integer mimeid field indicating the mimetype of the file. The mimeid field from `docmap` table is then used in a join table (`mimeidjoin`) to select which type specific table contains metadata for the particular file. For example, if a PNG image file is added to the index then its

Column	Type	Extra
url	varchar(4096)	Primary Key
docid	integer	Not Null, Unique
docidtime	timestamp	Not Null
size	integer	
mtime	timestamp	
md5	character(40)	

Table 3.3: Base schema for the `docmap` table.

Column	Type	Extra
mimetype	varchar(40)	Primary key
mimeid	integer	

Table 3.4: The `mimetype` table.

mimetype would be `image/png` with perhaps a mimeid of 43. The `mimeidjoin` table would map the mimeid 43 to the `mdimage` table where metadata about image files can be found (width, height etc). The `mimeidjoin` table would be a many to many relation, each file could have multiple type specific tables associated.

The use of such a relational model carries the following complications:

- Resolving keys in `fqueries` becomes more difficult. For the query `width<800` how does the query engine know that the user is wanting width in the image or animation tables? Perhaps they are looking for the width EA which is not attached to either an image or animation. This can be subverted by forcing the user to enter `image.width` to select table and then attribute. This forces the user to use a different syntax for EA queries against the relational database indexing engine and to know the table schema used for the EA index.
- Query Extensibility: If there are multiple tables in the database then either all tables would have to be checked for matches to queries or the user would have to be made aware of part of the database schema to form acceptable queries.
- EA Extensibility: If EAs are separated into multiple tables then the program responsible for finding an EAs value would need to provide a "semantic group" for that EA so that a relational database table can be created to store that attribute. For example, an EA created to count the number of people in an image would have to assign a table name for this attribute. This then creates the situation that the `people-count` attribute either must be available in the `mdimage` table or there must be a new table `mdpcounttab` with a `people-count` column.

3.3.5.2 Using a less Relational Design

A less relational design can allow for easier expansion of supported metadata and avoid the issues presented in Section 3.3.5.1. The design presented in this section has

been implemented in the libferris filesystem as the core `findex` for performing Formal Concept Analysis on a Semantic File System.

This database design is concerned with two things for each EA: its basic type (size and collation) and whether to denormalize it into the `docmap` table.

The schema has four core concepts: a `docmap` table (Shown in Table 3.5), a `docattrs` table (Shown in Table 3.6) and a fixed collection of value lookup tables. As the docattrs table references an attribute name through a numeric identifier (attrid), a `attrmap` table is needed to find the string representation of the attribute identifier. An overview is presented in Figure 3.4.

Column	Type	Extra
url	varchar(4096)	Primary Key
docid	integer	Not Null, Unique
docidtime	timestamp	Not Null
size	integer	
mtime	timestamp	
md5	character(40)	
width	integer	
height	integer	
bitf	bit varying	

Table 3.5: Extended `docmap` table. The top of figure is identical to the docmap table from Figure 3.3.

Column	Type	Extra
attrid	integer	Not Null, Primary Key
vid	integer	Not Null
docid	integer	Not Null, Primary Key

Table 3.6: docattrs, the lookup table to document map join table.

In the `docattrs` table the `attrid` references an attribute name table `attrmap` and the `vid` references one of many lookup tables: intlookup, doublelookup, timelookup and strlookup.

The lookup tables consist of two columns, one containing the `vid` and the other containing a value of the table's type. For example the intlookup table has a structure vid=integer, value=integer and the value type for the strlookup table will be varchar.

Resolving an `fquery` with this schema will involve a three table join for every EA comparison operator. The three tables involved will be a value lookup table, for example, `intlookup` to find the `vid` for the given EA value, the `attrmap` table to find the `aid` for the given EA name, and the `docattrs` table to join the `aid` with `vid` and obtain the list of matching `docid`s.

As shown in the docmap table in Figure 3.5 there is also the width, height and bitf columns. The width and height EA might only available for 5% of all files and be set to null for all others. Other EA of interest might also be stored directly in the docmap table with the width and height information, ie. in a denormalized form. If

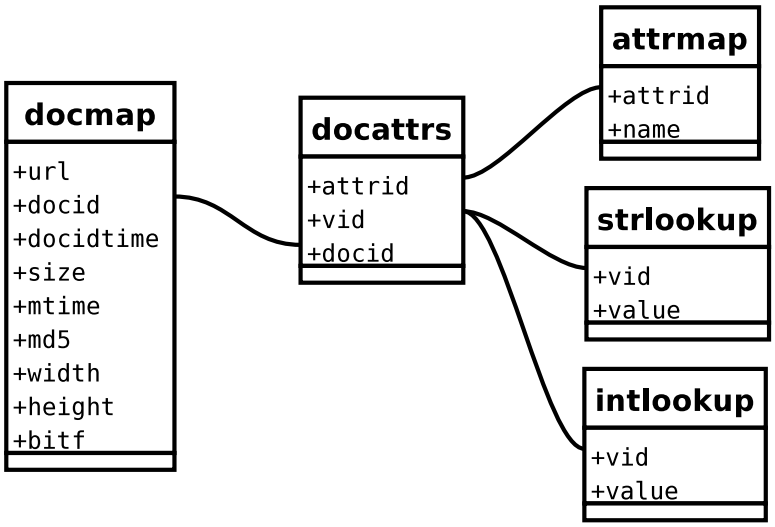

Figure 3.4: Core tables in the relational database schema. A single docid in the docmap table can be associated with many tuples in the docattrs table. A single attrid in the attrmap table can be associated with many tuples in the docattrs table. The attrid and docid in the docattrs table can be considered foreign keys. The vid in docattrs can not be a foreign key because it will reference one of many lookup tables (strlookup, intlookup etc) depending on the type of the eavalue that was indexed.

these EAs are `fqueried` for often enough it might be that denormalization from the docattrs join table for those attributes will be acceptable purely for the performance reasons. Denormalizing an EA into the docmap table requires the `fquery` processor to keep a persistent lookup table and tailor SQL query generation to use either `docmap` only or the three tables depending on if the EA is normalized.

Resolving a normalized query for an EA means consulting the indexes on one of the lookup tables, the docattrs and attrmap table and finding the list of `docid`s which match the query. If the EA is denormalized then only the relational database's index for the EA column need be inspected for the immediate production of the `docid` list.

The best candidate EAs for denormalization are ones for which there is a valid value for the attribute for most files, for which the range of values for the attribute is large and for which the attribute is the subject of many searches. In general an EA which carries only a binary value should be denormalized into the `bitf` field in `docmap`. For EA not meeting these requirements the three table design is preferable to overcome the space lost by denormalization. For example, although the gamma correction EA should be available for most image files, the average user is not likely to search for files based on that EA very often. Also the range of applicable gamma values will probably be low and as such it will occupy few places in the lookup table when stored normalized.

The `bitf` field in `docmap` is used extensively to support geospatial queries and queries involving file tags (See Section 5.5.2 and Section 8.2). Users typically only have a small number of individual tags, for example, ranging from a few hundred to a few thousand. The overhead of storing all tag relations denormalized as part of this `bitf` column is limited. A major advantage of grouping the tag relations in `bitf` is that the spatial indexing detailed in Chapter 4 can substantially assist in resolving queries involving multiple tags.

Selecting which attributes should be denormalized is difficult and perhaps an adaptive strategy based on user searches would be best suited to refining the set of denormalized attributes. The ability for the `findex` to record search history and automatically mutate which attributes are stored denormalized is the subject of future research.

3.3.5.3 Indexed EA Multi Versioning

The files and directories and the metadata assigned to these objects vary over time in a filesystem. For example, a trip itinerary file which was tagged as being important in the week leading up to a trip might have that status stripped after the trip is complete. As a document is edited its summary metadata may automatically change to better reflect new or modified content.

When keeping an `findex` current, if a file already exists in the `findex` and has been modified since it was indexed it might be advantageous to retain the old metadata in the `findex` and simply store a new instance of the file's metadata in the `findex` as the "most current" version. In this way a single file might have many instances of metadata in the `findex` collected at different times.

To support multiple instances of metadata for each file the `docmap` table must be modified by removing the `url` field and adding a `urlid` column. A new `urlmap` table is used to store the `url` to `urlid` mapping. This allows each URL to be associated with multiple `docid`s in the `docmap` table.

```
ferris-current-time >= 2006 and
ferris-current-time <= 2007 and
important = 1 and tokyo-distance <= 50km
```

Figure 3.5: Looking for the `Tokyo.jpg` file by searching for all instances of metadata stored in the `findex` during 2006.

Normal `fquery` resolution will only use the most recent instance of every file's metadata. A new EA, `ferris-current-time`, was added to every file in the Semantic File System. Reading `ferris-current-time` returns the current system time. If a `fquery` includes the `ferris-current-time` in a predicate every instance of a file's metadata is tested during query resolution. Such overloading of an EA is permissible as the user is very unlikely to want to search for files based on the exact time they were indexed. By reserving special EA names which are unlikely to be used by accident for use as special instructions (pragmas) to the `fquery` resolution process the query syntax itself does not require extension.

Queries which mention the `ctime` EA are also evaluated against multiple instances of file metadata. The `ctime` metadata shows the time of the last change to a file's status information. The ctime is updated whenever a file is written too or has a small subset of its metadata changed.

Comparisons against the `ferris-current-time` EA are handled during query resolution by comparing to the `docidtime` in the `findex` for all instances of metadata. As the `docidtime` is created when an instance of metadata for a file is added to the `findex` it records the time that the instance was current. As the `ferris-current-time` is part of the standard query syntax it can be used twice to express a desired range or can delineate multiple discrete time ranges.

Consider the following scenario: the user wishes to find all documents which at one stage were tagged as being important and are geographically relevant to Tokyo. Let us say that one such file is `Tokyo.jpg`. In this scenario the user keeps only a selection of files which are currently important tagged as "important". The trip to Tokyo occurred in March 2006 and it is now June 2007. A query using abstract syntax might look like the following:

```
important = 1 and tokyo-distance <= 50km
```

This query will not return `Tokyo.jpg` because the file is no longer tagged as being important and the most recent instance of `Tokyo.jpg`'s metadata in the index will not match the first part of the query.

If a time interval is specified as shown in Figure 3.5 then all instances of `Tokyo.jpg`'s metadata in the `findex` which were added in the year 2006 will be considered. As `Tokyo.jpg` was important during this time interval it will be returned as a result.

To support time interval queries the query translator must detect a time range which is using the `ferris-current-time` EA and modify the query translation semantics to allow documents which once matched the query to be returned. If there is no mention of `ferris-current-time` or `ctime` in the query string then only the most recently added instance of metadata for a file should be considered in the query resolution.

```
SELECT d.docid as docid,
       d.urlid as urlid
FROM docmap d,
  ( select max(docidtime) as ddtime, urlid
    from docmap
    group by urlid
   ) dd
WHERE d.urlid=dd.urlid
AND d.docidtime=dd.ddtime
AND ...
```

Figure 3.6: With multiple instances of metadata directly stored in `docmap` and `docattrs` the query must include a subselect to limit consideration to only the most recently added metadata instance for a urlid.

It was found that storing multiple instances of metadata directly in the `docmap` table gave poor performance. This was discovered when testing `fqueries` against an `findex` with $1,000,000$ files and 5 instances of metadata for each.

In order to limit the results to only include the most recent instance a query would have to be constructed as shown in Figure 3.6. The "select max(docidtime)" subselect query would degenerate to a sequential scan of the `docmap` table which completely dominated query resolution time.

Resolving an `fquery` against all versions of metadata does not require the grouping subselect statement. As resolving an `fquery` against only the most recent instance of a file's metadata is the most common `findex` use some changes to the `findex` structure were made to improve performance.

Two new tables, `docmap_multi` and `docattrs_multi` were created with identical indexes and schema to the non `_multi` tables. After files are added to the `findex` any instances in `docmap` which are not the most recent instance are retired into `docmap_multi`. A similar procedure is performed for the `docattrs` table. Such a migration adds a fixed overhead in the single seconds time frame when dealing with an index of a million files.

To resolve an `fquery` against only the most recent instance of metadata only the `docmap` table need be considered without the expensive subselect statement shown in Figure 3.6. To resolve an `fquery` against all instances of metadata the reference to `docmap` must be changed to the union of `docmap` and `docmap_multi`. It was found that the query optimizer is able to handle such a union statement well.

The semantics of negation in a multi versioned `findex` are the subject of further research. Consider the following query shown in Figure 3.7. The inner `fquery` is looking for any documents in the time window from the start of the year to the start of this month which have a small width. It may be the case that document x existed between these two time frames and was originally a small image but then we obtained a more detailed version of that image with a width of 1600. The issue then is given that x will match both the inner `fquery` and the negation of it because it has a version which matches both of those queries should x be included in the results. This issue is currently ignored and x will appear in the result if any version of its metadata in the index matches

```
(!(&(mtime>=begin last year)
    (mtime<=begin month)
    (width<=200)
))
```

Figure 3.7: Multiversioned query using negation has undefined semantics.

the final query.

3.3.6 Index Performance

Empirical testing was performed using a Intel Q6600 quad core processor, 4 Gb of DRR2 800Mhz RAM and the database stored on a model ST380024A seagate 7200RPM 80Gb disk. The operating system was Fedora 7 develop with PostgreSQL version 8.2.4 and libferris version `1.2.0`. The specially sorted Berkeley DB `findex` detailed in Section 3.3.4 used stldb4 version `0.4.7` with its embedded build of Berkeley db `4.1.25`.

The input data was a set of $1,000,000$ files. A thousand directories were created each containing a thousand files. In each directory each file was named from 1 to 1000. Each file was created with an `id` EA which was set to the file number. This means that every integer x between 1 and 1,000 would have 1,000 files where `id`=x.

The same set of files was used for testing the multi instance indexing detailed in Section 3.3.5.3. First the files were added as per other `findex`es then each file was touched to have a modification date a year later and reindexed. For subsequent reindexing of files each time they were touched adding a year to the previous version. The entire million files was added five times to the `findex`.

A total of 5 indexes where created from the test data set: a null index which always returns no result, a Berkeley db index which shall be referred to as the stldb4 `findex` (Section 3.3.4), and three indexes using PostgreSQL (Section 3.3.5).

The PostgreSQL indexes include a `findex` with `id` denormalized into the `docmap`, a `findex` with `id` normalized through `docattrs` and a denormalized `findex` with 5 instances of metadata for each of the million files. These three indexes will be referred to as standard, normalized, and multi instance respectively.

Some of the tests were performed using both a hot cache and a "hard cold" cache. A hot cache means that the operating system kernel and or the database server itself might hold some information in memory that is relevant to the `fquery` resolution. A hard cold cache means that the database server was stopped, the filesystem containing the files for the database was unmounted and remounted and the database server restarted. This forces both the database and operating system kernel to have no cache at all for any information in the `findex`.

The rationale for denormalizing an EA in the `findex` is to gain speed at the expense of disk space. The speed of `fquery` execution against both the stldb4 `findex` and the three relational database `findex`es is shown in Table 3.7. Although normalization plays no role in the null and stldb4 `findex` they are are shown for comparison purposes.

The cold cache query time for the normalized relational database `findex` is relatively poor because the query planner decides to do a bitmap based sequential scan of the `docmap` table. This is driven by the fact that the number of rows to be returned

singleversion `findex`	id= 1	id<= 5	id=< 10	id<= 50	id<= 100
null	0.47	0.47	0.47	0.47	0.47
stldb4	0.68	1.23	1.29	1.35	1.53
/ hot	0.59	0.61	0.62	0.72	0.83
standard relational	4.06	5.19	6.10	6.01	5.85
/ hot	0.60	0.62	0.64	0.74	0.74
normalized relational	22.9	69	28	39	46
/ hot	0.61	0.67	1.1	1.1	1.4

Table 3.7: Time in seconds to run a query on the `id` EA on an `findex` with the `id` EA normalized or inlined in the `docmap` table.

by even the `id=1` query is large relative to the whole table size. The `id=1` query will return on average one in a thousand rows from the table of a million rows. Each row is relatively small so many rows will fit into a disk page. The cost of non sequentially reading the number of needed disk blocks is greater than the cost of sequentially reading the `docmap` table.

The data in the relational database `docmap` table will have been added in a manner that would have spread the tuples matching `id=1` fairly evenly throughout the table. This is a pathological worst case data distribution for the relational database because the database indexes do not allow much of the `docmap` table to be skipped.

The stldb4 `findex` does not suffer as badly from having a completely cold disk cache because the list of documents matching each `id` value is stored sequentially. To resolve the `id=1` query the stldb4 `findex` will only need to read a few blocks from disk. As the query includes more values for `id` the stldb4 `findex` will need to read in more lists from disk and the query time for cold cache degenerates linearly.

The worst performance was for the normalized relational database `findex` on a cold disk cache. This can be explained by the need to read more disk blocks to perform the join through `docattrs`.

Notice that the performance on a hot cache between the normalized and denormalized relational `findex`es for `id=1` and `id<=5` is similar. The `id<=5` `fquery` will return $5,000$ rows or 0.5% of the entire `findex`. Thus the performance for the normalized and denormalized `findex` on all but the hard cold cache case should be comparable where the number of files returned is less than 0.5% of the entire `findex`.

In order to test the relational database driven `findex`es in a less synthetic manner 100 files of the million each had 29 bytes written to them and were reindexed. Both the standard and normalized `findex` took 1.2 seconds on a hard cold cache to resolve an `fquery size==29`. The stldb4 `findex` took 1.0 seconds on a cold cache to resolve the same `fquery`. Hot cache times were 0.67, 0.94 and 0.97 seconds for the stldb4, standard relational and normalized relational `findex`es respectively.

The queries in Table 3.8 were devised to test the effects on the query resolution time of keeping many instances of metadata for each file in the index.

restriction on `findex`	actual `fquery`	seconds required
No time restriction	`(id==40)`	0.6
1 year	`(&(id==40)(ferris-current-time<=begin 2009))`	1.6
3 years	`(&(id==40)(ferris-current-time<=begin 2011))`	1.7
5 years	`(&(id==40)(ferris-current-time<=begin 2013))`	1.7
all time	`(&(id==40)(ferris-current-time>=begin 1970))`	2.1

Table 3.8: Benchmarks of running the same base query ($id == 40$) against the many instance `findex` with varying time restrictions. For each benchmark the suitable time restrictions were anded to the base query to limit which instances were considered during `fquery` resolution.

3.4 Conclusion

The creation of an `findex` for the metadata from a Semantic File System brings many challenges. The `findex` structure for metadata will be necessarily different than that chosen for indexing a human language due to the requirement to store a key-value pair in the `findex`. The metadata index must be able to service queries involving comparisons on the value for a given metadata key.

The metadata from an Semantic File System also has varying types for the metadata values which have to be taken into consideration when designing the index. Different types bring different collations and these effect query resolutions involving order such as `width<=800`. The diverse distributions of values for each metadata key further complicate index design because metadata keys with many files that match a single value are better suited to particular index structures than metadata keys where each value appears for only a few files.

The core of this chapter was Section 3.3.3 where issues with metadata `findex`es and solutions where discussed. The ability for an `findex` to resolve queries against metadata as it stood in the past was detailed in Section 3.3.5.3.

Empirical testing of the two competing `findex` designs was conducted in Section 3.3.6. A million files were added to the `findex` and various queries lodged at the `findex` on both hot and cold hard disk caches. Some of the queries presented a pathological worst case for the relational database design. This is because the amount of data sought was too large a percentage of the base table size and the relational database resorted to a sequential scan of the table producing poor results on a cold disk cache.

To remove the synthetic test that was performing very poorly on a hard cold cache relational database a test where the query would only return 100 files from the million was conducted. In this case the relational database chose to use its indicies and the resulting query times were comparable with the pure inverted file design.

In practice if the relational database design is being used to perform a selection of queries which are going to return a large percentage of the base table (the worst case above) for Formal Concept Analysis then each query will not be executed against a hard cold disk cache and thus query performance will be much more acceptable.

Chapter 4

Formal Concept Analysis and Spatial Indexing

4.1 Introduction

This chapter describes improvements to indexing over traditional B-Tree index used by relational database to improve query performance. The improvements that can be achieved by using tailored indexing can enable Formal Concept Analysis to be applied to larger data stores than was previously tractable.

Section 4.2 describes why traditional B-Tree indexing is ineffective for typical Formal Concept Analysis queries. Spatial indexing is introduced in Section 4.3 as a method of resolving Formal Concept Analysis queries in a more timely manner. Empirical testing is performed in Section 4.3.1 to validate that spatial indexing can vastly improve Formal Concept Analysis query performance for a number of data sources. Motivated by this improved performance, Section 4.4 investigates improvements to the spatial indexing structure itself. The two new spatial index structures are then empirically tested in Section 4.4.4 and found to give superior performance in a number of important settings.

The motivation for the application of spatial structures was initially for the use of Formal Concept Analysis on a filesystem [39, 82, 62], in particular the libferris [3] Semantic File System. Spatial indexing has been found to bring similar performance improvements to more general Formal Concept Analysis applications: sometimes referred to as Toscana-systems. The spatial method proposed in this thesis has performance which depends on the number of attributes in each query as well as the density and distribution of the formal context.

Typical Formal Concept Analysis queries seek all index entries which either (a) exactly match a given key or (b) are a super set of the given key. As an example of (b) consider Formal Concept Analysis on animal species: one concept might contain the attributes {has-tail, has-fur}, to find the objects which match this concept we will want all known objects which have **at least** these attributes but may include other attributes as well. Both of these common query types, (a) and (b), can be vastly aided with spatial indexing as this chapter explores and presents. Note that even exact match queries (a) present problems for conventional B-Tree indexes due to attribute ordering in index creation [75].

4.2 Why Conventional Indexing is Ineffective

Prior to this research introducing spatial indexing to FCA applications, two designs dominated Formal Concept Analysis implementations for the indexing of data: either a single large table in a relational database where objects are rows and their attributes form columns (Toscana) [91] or using inverted files (LISFS) [82]. The Toscana design appears to be more widely implemented. The Toscana design is usually implemented using a relational database system [52, 78].

The libferris design is most closely affiliated with the Toscana design with extensions to deal with normalization [64, 89] and the association of tags [61, 67]. A tag is a pictorial annotation, usually a small icon, that is associated with an file or directory. A tag often denotes a category.

Shown in Figure 4.1 is an example inverted file index. With an inverted file index values of interest each have a list of the address of the tuples from the origin of the base table. For example, an inverted file index on a **name** column would have a list for the value "peter" with pointers to all the tuples where the name column was "peter". Inverted files work well when there are a limited number of values of interest. Given an inverted file defined such that the values of interest are formal attributes and a concept with intent $\{10000, 01000\}$ one must combine the lists for 10000 and 01000 to list the extent of that concept.

The focus is now turned exclusively to systems using the Toscana design and relational databases for data storage and indexing. Assuming, without loss of generality, that the many-valued context is available – denormalized in a single relation which shall be referred to as the base table. This base table having columns $\{c_1, c_2, ...c_y\}$. As a concrete example, consider a base table with 4 numeric columns c_1 = **size** and c_2 = **modified**, c_3 = **accessed** and c_4 = **file-owner**. Although the modified and accessed columns are numeric they are presented here in a human readable form. As an example consider three ordinal scales on the columns c_1, c_2 and c_3 and a nominal scale on c_4 (see Figure 4.3).

More generally for the base relation we consider a formal attribute $\{a_j\}$ to be defined through possible values for one or more columns $\{c_i, ...c_u\}$. It can be convenient to consider the definition of an attribute $\{a_j\}$ as an SQL condition f_j on the values of one or more columns $\{c_i, ...c_u\}$. Thus for all $i \in \{1...j\}$ the formal attribute a_i is defined by the SQL expression f_i on the base table. The convenience of using SQL expressions f_j to define the formal attributes a_j is due to the SQL expression returning a binary result. Note that there is a one-to-one correspondence between A and F, every formal attribute is defined by an SQL expression. The number of attributes $|A|$ can vary from the number of columns $|C|$ in the database. The a_x, f_x and c_y are shown in Figure 4.3. For example, from in Figure 4.3 an attribute a_x might be defined on the columns $\{c_2, c_3\}$ using the SQL expression f_x `= modified < last week AND accessed > yesterday`[1] . The attribute extent for f_x would contain all files which have been accessed today but not modified this week.

Due to the generality of the terms attribute and value some communities use them to refer to specific concepts which are related to the above uses. For example the term attribute in some communities would more naturally refer to the c_i. The above

[1] date values represented as human readable strings in this example

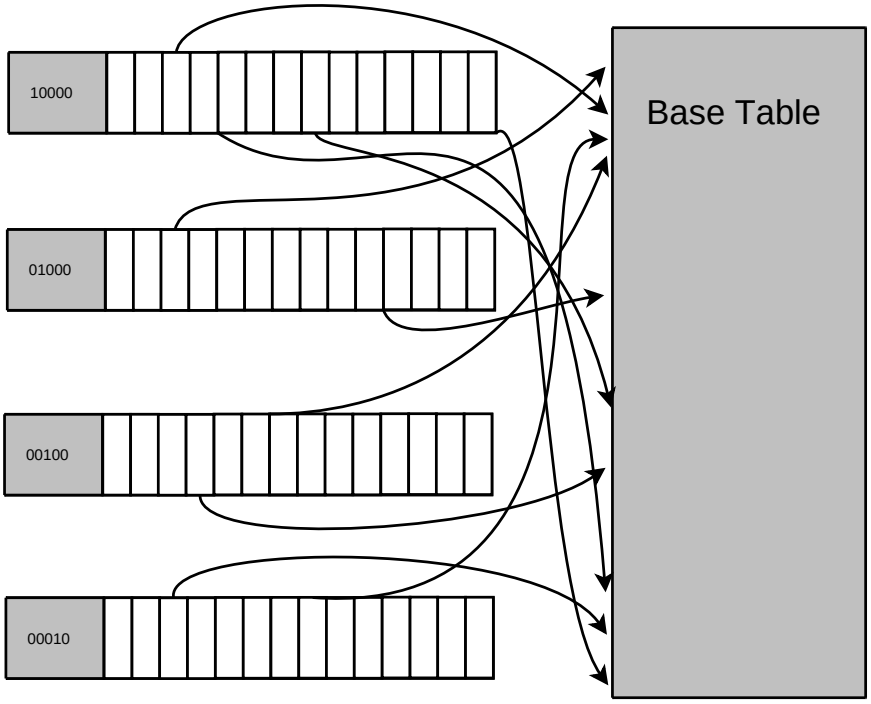

Figure 4.1: An inverted file index. For each value of interest there is a list containing all the addresses of tuples which match that value.

	c_1	c_2	c_3	c_4
object-ID	size	modified	accessed	file-owner
1	4096	today	today	ben
2	800	yesterday	today	peter
3	400k	1 year ago	last week	ben
...	...	...	...	

Figure 4.2: Example base relation containing modification and size data for objects.

Attribute	Columns involved	SQL predicate (f_x)
a_1	c_1	**size <= 4096**
a_2	c_1	**size <= 1Mb**
a_3	c_2	**modified <= this week**
a_4	c_2	**modified <= yesterday**
a_5	c_2	**modified <= today**
a_6	c_3	**accessed <= last week**
a_7	c_3	**accessed <= yesterday**
a_8	c_3	**accessed <= today**
a_9	c_4	**file-owner = ben**
a_{10}	c_4	**file-owner = peter**
a_{11}	c_4	**file-owner = foo**
...	...	...

Figure 4.3: Ordinal scales on the size, modification and access times of the objects in the base table. Nominal scale on the file-owner.

terminology was selected to more closely model Formal Concept Analysis where the formal attributes are binary. Thus the (formal) attributes are modeled as the a_i.

Consider finding the extent of a concept which has attributes $\{a_1, a_3, a_7\}$. The SQL query is formed with an SQL `WHERE` clause as "... where f_1 and f_3 and f_7 ...". For our concrete example, the SQL predicate will be "... where **size** <= 4096 and **modified** <= **this week** and **accessed** <= **yesterday** ...".

Previous best practice in the Formal Concept Analysis community was to attempt to assist such queries with B-Tree indexes over subsets of $\{c_1, c_2, ..., c_y\}$. The discussion is now turned to how relational databases use B-Tree indexes during query execution.

A common implementation of relational database queries is to check to see if the use of an index is estimated at returning a percent of the base table which is below a given internal threshold [98]. For example, if the use of an index results in 30% of the tuples in the base table being fetched then the database elects not to use that index. If there are no other indexes available for the query then it will sequentially scan the base table to resolve the query. When fetching a large proportion of the base table a sequential scan is usually faster than using the index because the table can be read in order [33]. The estimated ratio of matching tuples is called the **selectivity**. The key to efficient query execution is therefore for the query to be able to use an index which will sufficiently narrow the number of tuples fetched to make index usage attractive.

The selectivity of an index is estimated for the values given in the SQL predicate using statistics of how many tuples will match the given value or value range. For example, if 60% of column c_3 has values below 20 and the SQL predicate is $c_3 < 20$ then an index on column c_3 would be considered unattractive in the resolution of the query because it is not selective enough on average to outperform a sequential scan. The selectivity can be more formally defined as 100×estimated tuple count/size of base table. Thus lower numeric selectivity values are considered "better" in retrieval terms. A relational database's query planner will prohibit the use of all indexes which have an estimated selectivity beyond a predetermined sequential scan cutoff value.

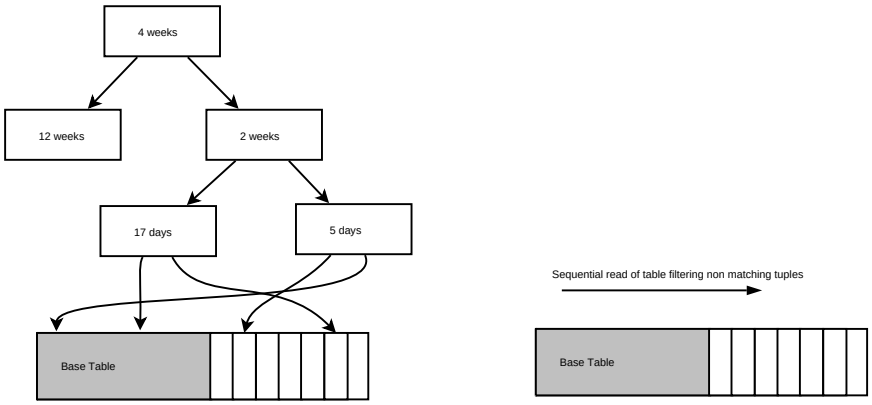

Figure 4.4: On the left: B-Tree index on a date column for the base table. Dates in nodes are shown as how long before the current time they represent. The upper nodes are index nodes with the nodes below "12 weeks" omitted. The 17 and 5 days nodes are leaf nodes of the index which point at records in the base table. The B-Tree has a restricted branching factor of two children for illustration purposes. On the right: Resolving the query by a sequential scan filtering out non matching tuples.

Consider a query against the index shown in Figure 4.4. If one is seeking all dates more recent than 4 weeks ago they will likely have to scan half the index [95]. Because the base table is not stored in the order of the index key the records sought will likely be scattered throughout the base table. Given the high percentage of the base table's tuples sought and the scattered positioning of these tuples on disk a sequential scan of the base table will likely provide the most efficient resolution of the query. Seeking disk heads is a very slow operation relative to contiguous sequential disk access [33]. Although some tuples are read and subsequently filtered out using the sequential scan method the lack of disk head seeks makes sequential scan faster overall for queries seeking a substantial amount of the base table. This is because data can be transferred from disk in a contiguous read using the sequential scan much faster than individual pages of the base table required by the index.

When there are two predicates in the **where** clause commonly the predicate which has an available index with the best selectivity is chosen first. After this initial index selection the other predicate is used as a filter on the tuples as they are read from the base table [98]. This query design strategy works ineffectively on typical Formal Concept Analysis SQL queries because there is usually more than one predicate joined with a logical **AND**. In the normal case, the selectivity of either predicate will be beyond the query planner's sequential scan cutoff.

Again, referring to our example, say we are looking for $size <= 4096$ and **modified <= this week**. Assume an index on **size** and one on the **modified** column as shown in Figure 4.5. For a large database most files will be modified before the current week. This would give an extremely poor selectivity for the **modified** subpredicate. This would rule out the use of an index on the modified column for this query. Thus, the only chance of using an index would be on the column **size**. Assuming the base table was created using metadata from a filesystem the selectivity of the index depends on that metadata, $size <= 4096$ results in selectivity good enough to use the index but

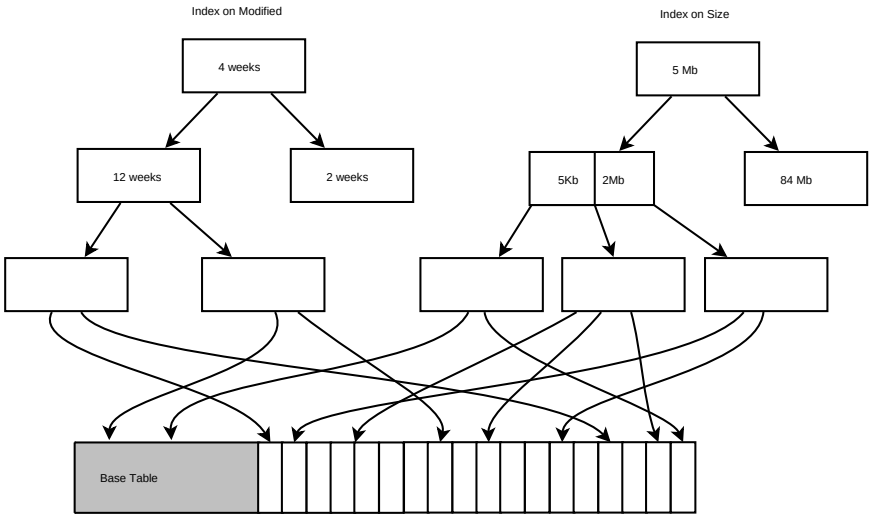

Figure 4.5: Two B-Tree indexes on the base table. The left index is on *modified* while the right index is on *size*. Leaf nodes in both indexes point to tuples physically located throughout the base table.

$size <= 4Mb$ would not offer attractive selectivity.

Considering an ordinal scale $Z = \{a_m, S_{m+1}, ...S_n\}$ on the size column it is likely that only a fraction of Z would result in a good enough selectivity to utilize the above index on the size column. For example, if Z was defined as **size** $<= 4096$, **size** $<= 400kb$, **size** $<= 4Mb$, **size** $<= 40Mb$ and **size** $<= 400Mb$, it would be likely that only the first one or two predicates could use the size index with acceptable selectivity.

When both predicates are considered together a single index over multiple columns may be used in an attempt to achieve better selectivity. For an index created over multiple columns only the leading columns specified in a predicate are considered when computing the number of matching tuples using the index.

Consider an example taken from the formal attributes in Figure 4.3. When we seek the size of the concept with intent $\{a_1, a_5, a_{11}\}$ we will have 3 predicates **size** $<= 4096$, **modified** $<=$ **today** and **file-owner** $=$ **foo** respectively. These SQL predicates are operating on the columns $\{c_1, c_2, c_4\}$. Assume that an index is created over $\{c_1, c_2, c_3, c_4\}$ to assist this query. Most relational databases do not consider any terms from the predicate which are not contiguous leading terms in the index when calculating the selectivity of an index [98]. Nothing in the query makes reference to c_3 so only the predicates **size** $<= 4096$, **modified** $<=$ **today** will be used to compute selectivity. For this example the index can't take advantage of the file-owner predicate which may in this case offer a significant improvement to selectivity. Given that the use of the column c_2 will not significantly improve selectivity the use of the whole index deteriorates to the selectivity of c_1 alone.

More generally, if the B-Tree index is created over $\{c_1, c_2, ..., c_y\}$. Assume that the f_1 and f_4 and f_5 refer to columns c_1 and c_4 and c_5 respectively. For our target concept, the critical leading index terms $\{c_2, c_3\}$ are left unspecified, implicitly defining them as matching any value. The query planner will likely only consider the selectivity using

only c_1 and decide that the B-Tree index $\{c_1, ..., c_y\}$ is an unattractive access path. This is because it will have to scan large parts of the index before c_4 or c_5 can contribute to the selectivity. In practice many relational databases do not consider any terms from the predicate which are not contiguous leading terms in the index when calculating the selectivity of an index.

This situation deteriorates further the more columns are available in the relation due to the probability that leading index terms are not present in the query predicate. For example, for a concept with a handful of attributes in its intent, say $\{\{o_1, o_2\}, \{a_1, a_2, a_3, a_4\}\}$, the chance of having at least one attribute a_x, which happens to have a f_x referring to a column in the index's leading terms, is low. Even with a reference to a leading index term it is unlikely that the reference will be very selective by itself. It is a particular strong point of spatial access methods that they gracefully handle such unreferenced columns on a many column index.

If for each $i \in \{1..x\}$ an index on only column c_i is created most relational databases will only consider using one of these indexes for resolving a expression f_i which refers to multiple columns [98].

When resolving an SQL query against a base table most relational databases will only consider using a single index [98]. If one considers the possibility of creating a custom index to assist queries for each concept, there are potentially $|C| = 2^{|A|}$ concept intents for a formal context. Given that many f_x will reference the same column, the number of unique combinations of columns from the base table will be less than this number. However, as discussed, the ordering of the columns in the index may have to be taken into account to improve performance. This ordering of columns in indexes will raise the number of indexes needed back towards $|C|$, however, the number of attribute combinations makes it is infeasible to create custom B-Tree indexes for each concept intent or column order.

4.3 Spatial Indexing

This section discusses the application of spatial methods to improve index utilization in query resolution. First using indexes on SQL expressions is considered followed by how spatial methods can be applied to expression indexes to improve performance.

Many relational databases allow the creation of indexes on expressions [6]. For example, given a column **name** an expression index can be created on **lower(name)** to help case insensitive searches. Turning to Formal Concept Analysis one can define an expression index e_x for each respective SQL predicate f_x. Consider again our example from Figure 4.3. The expression index e_1 on attribute a_1 is shown in Figure 4.6. In an expression index tuples which do not satisfy the index expression are not added to the index.

Turning to the application of expression indexes to Formal Concept Analysis. The indexes $\{e_1, e_2, ..., e_n\}$ having been defined by scales $\{f_1, f_2, ..., f_n\}$ are an implementation artifact which is equivalent to the formal attributes $\{a_1, a_2, ..., a_n\}$ of the formal context. Thus queries on an attribute a_x become queries against the respective index e_x. This allows the materialization of binary attributes from the base table using indexes alone. Creating expression indexes on the f_i expressions does not change the problem of the query planner ignoring such indexes $\{e_1, ...e_n\}$ due to selectivity constraints, highlighted in Section 4.2. One can however consider indexing structures over the collected

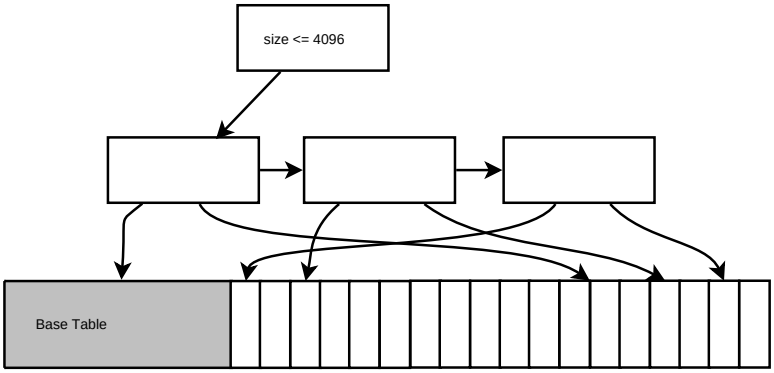

Figure 4.6: Expression index on attribute a_1 using f_1, the SQL predicate $size <= 4096$. The B-Tree structure is degenerate because there is only one value indexed. At the leaf page level, pages continue to overflow and the B-Tree approximates in inverted file structure.

$\{e_1, ..., e_n\}$ indexes.

The use of spatial indexing structures over $\{e_1, ..., e_n\}$ can provide substantial increases in FCA query performance. If expression indexes are created for each attribute then above queries such as $Q = \{e_1, e_5, e_{11}\}$ can be classified as a subset query [51] on the expression indexes. In a subset query over $\{e_1, ..., e_n\}$ the objective is to seek all objects with a given subset $S \subseteq \{e_1, ..., e_n\}$ of specified values. For example, the query seeking "... **size** <= 4096 and **modified** <= **this week** and **accessed** <= **yesterday** ..." specifies objects matching $S = \{e_1, e_3, e_7\}$.

Note that it is an implementation detail as to whether the expression indexes themselves must be created to support the higher level spatial indexing. The implementation used for empirical testing [3] in fact performs the spatial indexing directly on database 4GL functions [6] thus alleviating the need for the spurious expression indexes.

An indexing structure motivated by the spatial indexing structure, the R-Tree [44, 81], caters for subset queries: the RD-Tree [50, 106]. A particular strong point of these structures is that they index multiple columns in arbitrary order and gracefully handle lookups given a subset of the indexed columns. The subsequent presentation first describes the R-Tree followed by the RD-Tree.

The internal nodes in an R-Tree structure contain entries of the form; `(bounding n-dimensional box, page pointer)`, where pages in the subtree reached by page pointers are within the given bounding n-dimensional box (see Figure 4.7). This transitive containment relation is the heart of the R-Tree. R-Trees are not limited to 2 or 3 dimensional data and typically use page sizes allowing branching factors much closer to B-Trees than shown in the example.

Searching for a spatial object in the R-Tree starts at the root node and considers all children whose bounding box contains the query object. Searching for the query object in Figure 4.7 begins at the root node (R) – the left node (C1) has a bounding box not containing the query object so only the right child (C2) is followed. In turn, the new left node (C2.1) contains the query object and will be followed whereas (C2.2) is not. At the lowest level (the children of C2.1) many nodes may contain the query object and these are followed to retrieve tuples in the base table.

The RD-Tree operates similarly by treating input as an n-dimensional binary spatial area. The R-Tree notion of containment is replaced by set inclusion and the bounding n-dimensional box replaced by a bounding set. The union of a collection of sets forms the bounding set. The bounding set of a child is thus defined as the union of all the elements in the child. The bounding set defined in this way preserves the "containment" notion of the R-Tree during search as a subset relation. When seeking an element which might be in a child it is sufficient to test if the sought element is a subset of the bounding set for the child to know if that subtree should be considered.

An example RD-Tree is shown in Figure 4.8. Searching for the query object `01100` starts at the root node discarding (C1) because it does not contain the query object and only following the (C2) child. At (C2) the node (C2.2) is not followed because it does not contain the query object and only (C2.1) is followed. The child (C2.1.2) has a bounding set `00110` which does not contain the query object and is not considered. Only (C2.1.1) matches this query and its contents are tested against the query object to retrieve the results from the base table.

The two main Formal Concept Analysis queries that an RD-Tree can improve are subset and overlap queries [51, 50]. As described above a subset query seeks all objects

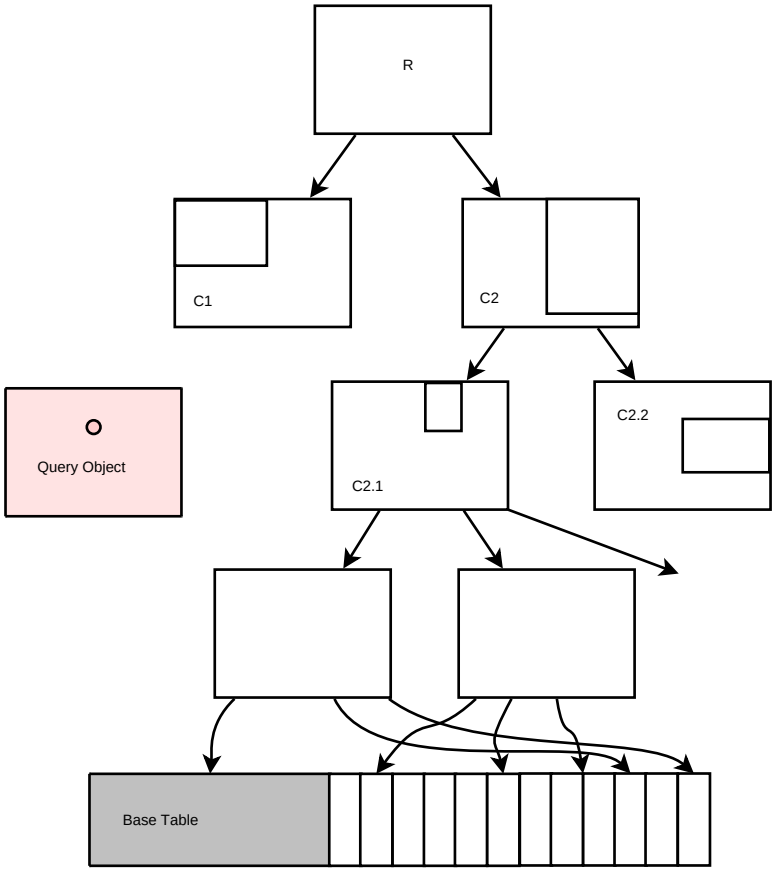

Figure 4.7: An example R-Tree with a query object on the left. Each node has a bounding box which fully contains all objects in its child nodes. An implementation stores the bounding box for each child in the parent node. Note the example is limited to 2 dimensional space with a low branching factor for presentation purposes.

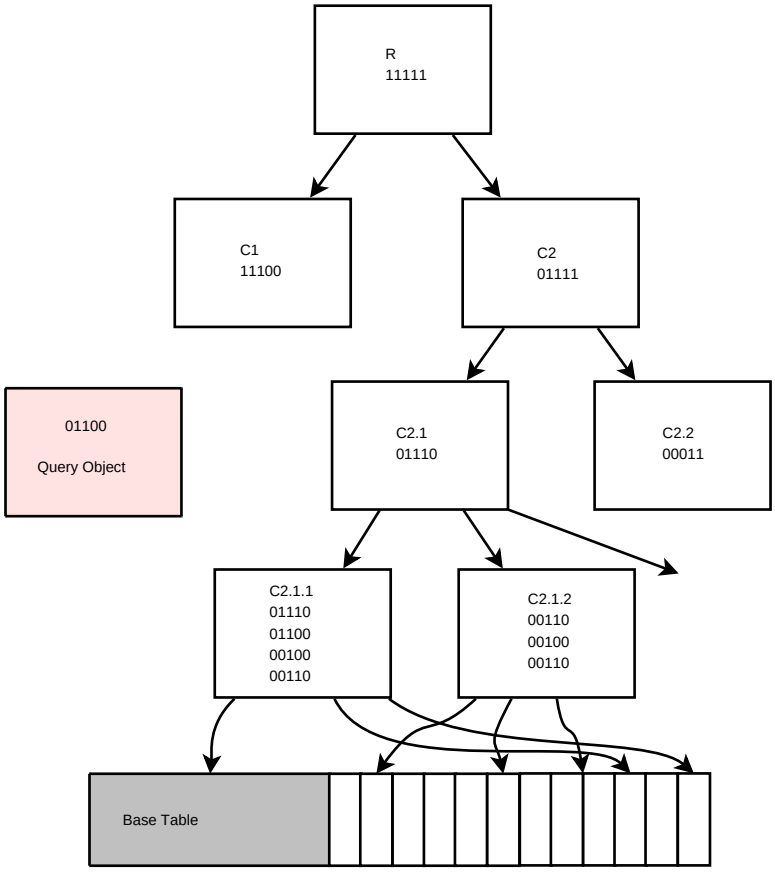

Figure 4.8: An example RD-Tree with a query object on the left. Each node has a bounding set associated which fully contains all objects in its child nodes. An implementation would store the bounding set for each child in the parent node. Note that the example is limited to only a small set size with a low branching factor in the tree for presentation.

Normal query	$a < 10$ and $a < 20$ and not $(a < 30)$ and not $(a < 40)$
Simple translation	rd-tree contains 10,20 and not rd-tree contains 30 ...
Custom translation	rd-tree contains 10,20 and hamming-weight(rd-tree) = 2

Figure 4.9: Translating queries involving negation to take advantage of the RD-Tree. This assumes that the attributes 10, 20 and 30 stand for the predicates $a < 10$ and $a < 20$ and $a < 30$ respectively. The weight function returns the number of RD-Tree predicates a tuple contains. So in the above, the third query doesn't need to negate the 30 and 40 predicates because the weight test will already ensure that 30 and 40 are not set.

for which the query object is a subset. For example the query object might be $Q = \{e_1, e_3, e_7\}$ and a matching object $o_i \in O = \{a_1, a_3, a_6, a_7\}'$. For a given set of attributes $A = \{a_1, a_2, ..., a_n\}$ defined by their respective index expressions $E = \{e_1, e_2, ..., e_n\}$ a bitset can be derived $\{b_1, b_2, ..., b_n\}$ such that b_x is set to true when $e_x \in E$ is true. Thus for the example in Figure 4.8 we are seeking the query object `01100` which means we want all objects where e_2 and e_3 are true, which is the same as having the formal attributes $\{a_2, a_3\}$.

To resolve a subset query the RD-Tree is walked from the root eliminating any branches with a bounding set which is a subset of the query set. It is apparent that the more items from $\{e_1, ..., e_n\}$ specified in the query the less of the index structure will be searched. The trend for RD-Trees is the inverse of that of inverted files. To resolve the above with inverted files the lists for each e_x would have to be fetched and merged. For our same query object `01100` we would have to fetch the lists for `01000` and `00100` to form the set intersection and finally fetch the records from the base table (see Figure 4.1).

An overlap query seeks objects which have more than a given number of attributes in common with the query [106]. To efficiently find the contingent size the RD-Tree index must also contain the hamming weight of the binary expression indexes $\{e_1, ...e_n\}$ (ie. formal attributes) which are indexed. The hamming weight for a bitset is the number of bits which are not zero. This is so objects that are in the extent (but not the contingent) can be quickly filtered from the result using the index alone.

The specialized overlap query Q contains the attributes $Q \subseteq \{e_1, e_2, ...e_n\}$ which define the exact attributes sought in the result set. The above subset query would not return object $o_i \in O = \{a_1, a_3, a_6, a_7\}'$ for $Q = \{e_1, e_3, e_7\}$ because attribute a_6 was not specified in the query. It can be seen that o_i would be in the extent of a concept with intent $Q = \{e_1, e_3, e_7\}$ but not in the contingent. An example query translation is shown in Figure 4.9.

4.3.1 Performance Analysis

The benchmark system was an AMD XP-Mobile running at 2.4GHz with 200Mhz FSB, 1Gb of RAM at 400Mhz dual channel cas222. The software versions which may effect performance were Linux kernel 2.6.11rc3, gcc 4.0.0 20050308,
PostgreSQL 8.0.1, libferris 1.1.50, ToscanaJ 1.5.1 and Java 1.5.0_01.

Testing was completed on 3 different input data sets: various synthetic formal

Column	Value	Selectivity (count)	Selectivity (% of table)
bruises	NO	10080	59.9
bruises	BRUISES	6752	40.1
capshape	KNOBBED	1680	10.0
capshape	CONVEX	7592	45.1
capshape	FLAT	6584	39.1
capshape	BELL	904	5.4
capshape	SUNKEN	64	0.4
capshape	CONICAL	8	0.05

Figure 4.10: Selected attributes for the mushroom table and the number of tuples which have the given attribute-value combination.

contexts generated with the IBM synthetic data generator [88], the mushroom and covtype databases from the UCI dataset [17] and a formal context derived from the metadata of 67,000 document files [3]. Also, all columns in the databases had single column B-Tree indexes created on them for every column that might be relevant to query resolution. The mushroom database has 16,832 tuples and the covtype table has 581,012 tuples.

A test consists of lodging a collection of SQL queries against the database as a single batch job. Unless otherwise stated these batch tests were completed after the database was shutdown, the filesystem with the database information was unmounted (and remounted) and finally the database started again. This process flushes internal database buffers and the kernel's disk cache. Where tests were not performed under these cases, the terms "cold cache" refer to a setup where all buffers were flushed and the database restarted as above while "hot cache" mean that the queries were performed with no such flushing or database restart.

For various tests the SQL **explain** was used on each query in the batch to see how many sequential table scans were planned for the batch execution. For small datasets a sequential table scan might prove the fastest method to resolve a query (although performance is bound to be linear and thus will not scale well to larger data sets). Other statistics are shown as well such as the selectivity, mean and standard deviation of a column or table. In order to demonstrate how the spatial indexing performs on various densities of formal context for the synthetic datasets the distribution statistics of the attributes in the formal context is shown.

4.3.1.1 Performance on the UCI Mushroom dataset

The attributes used from the mushroom table are shown in Figure 4.10. The two columns **bruises** and **capshape** were used in an attribute list in ToscanaJ. As there are eight binary attributes when each distinct value for these two columns is considered there are a total of 256 SQL queries generated.

As the relation is relatively small this test was also conducted with explicitly hot caches. This should give an indication of performance differences on small data

Test type	Cold cache	Hot cache	Sequential scans
B-Tree only	30	18	4
RD-Tree index simple query translation	10	4	1
RD-Tree optimized query translation	1.6	0.4	0

Figure 4.11: Times with hot and cold caches to complete queries for 8 attribute list context. Times are in seconds.

sets modeling the use case of someone interactively creating and modifying scales. The benchmark was obtained by executing a test multiple times in a row and only taking the last batch time. The results are shown in Figure 4.11.

There are 2 versions using an RD-Tree to speed execution: a simple translation and a customized query. The simple translation just substitutes SQL operations to consult the RD-Tree and leaves all other query structure identical. This translation is fairly mechanical and does not fully take advantage of the RD-Tree for query resolution. The custom translation version takes advantage of adding to the RD-Tree the additional information of the hamming weight of the index expressions $\{e_1, ...e_n\}$ as described in Section 4.3.

4.3.1.2 Performance on UCI covtype dataset

The UCI covtype database consists of 581,012 tuples with 54 columns of data. For this paper two ordinal columns were used: the slope and elevation. Tests were performed by nesting an ordinal scale on elevation inside an ordinal scale on slope using ToscanaJ 1.5.1. This nesting produced a total of 378 queries against the database. Given that the primary table is 987Mb tests against a hot cache were not performed. The results are shown in Figure 4.12. The RD-Tree index takes 2m:17s to create. As there were no explicit negations there was no gain in producing an RD-Tree optimized SQL query as was done for the mushroom database.

4.3.1.3 Performance on Semantic File System Data

An index for part of the libferris filesystem was created for 66,936 files. A formal context based on file name components contains 886 formal attributes for these objects.

Test type	Cold cache (mm:sec)	Sequential scans
B-Tree only	56:16	90
RD-Tree index simple query translation	0:42	0

Figure 4.12: Times to complete nested scale queries against the covtype database. The nesting is obtained by generating a nested line diagram in ToscanaJ placing an ordinal scale on elevation inside an ordinal scale on slope.

Test type	Cold cache	Hot cache	Sequential scans
B-Tree only	80	80	487
RD-Tree index	5	1	0

Figure 4.13: Times in seconds with hot and cold caches to complete queries.

Formal attributes were packed into a single SQL **bit varying** field making the total size of the formal context only 10Mb. For this formal context there are 488 concepts. Benchmarks for querying the size of the extent of each context is shown for both hot and cold caches in Figure 4.13. Without the use of a special purpose index structure the database degenerated to a sequential table scan for almost every query.

To find the contingent sizes without using an RD-Tree using an SQL query per concept is slower overall than using a single table scan and handling the logic in the client. Using a single table scan from a relational database client takes 70 seconds to find every concept's contingent size. Using an RD-Tree index the same operation takes 2.2 seconds. The use of RD-Trees implies a cost of creating the index, for the above example this is 27 seconds. Index creation can happen faster than a client table scan because it is being done inside the database server process and avoids formatting and copying overhead. So the index can be created and used faster than any other method for finding contingent counts.

4.3.1.4 Performance on Synthetic data

The following use synthetic data generated with the IBM synthetic data generator [88]. Parameters include the number of transactions (ntrans), the transaction length (tlen), length of each pattern (patlen), number of patterns (npat) and number of items (nitems). The number of items was fixed at its minimal value of 1000. The tlen, patlen and npats can be varied to change the density of the resulting formal context while the ntrans is useful for testing the scalability of the query resolution.

Only the first 32 items were imported into the database. Five values of **tlen** were tested, 256, 128, 64, 32 and 16 at various database sizes ranging from 1,000 to a million transactions. The query sets were constructed by mining the Closed Frequent Itemsets for the 1,000 transaction database with a minimum support value of 0.01%. The Closed Frequent Itemsets provide the concept intents for an iceberg lattice [96, 79]. This generated 556 concept intents. A query was generated to find the size of the extent of each concept. This produced a distribution of 28 single attribute, 224 two attribute, 284 three attribute and 20 four attribute SQL queries. Benchmarks against these datasets are presented in Figure 4.14 and graphically in Figure 4.15 and Figure 4.16. Size statistics for the DBMS tables and indexes are shown in Figure 4.18.

The efficiency of using RD-Trees degenerates as the density of the formal context increases. To measure this effect the number of items per itemset was varied with all other parameters static. Results are shown in Figure 4.17. The results with 128 items per itemset are the same as those in Figure 4.14.

The performance of spatial indexing for Formal Concept Analysis in various settings has been examined and shown to provide substantial improvements in many cases.

Query Type	Thousands of trans	Time (128)	Time (64)	Time (32)	Time (16)
B-Tree	1	0.8	0.8	0.7	0.7
RD-Tree	1	0.7	0.6	0.6	0.5
B-Tree	10	3.3	3.3	3.1	3.1
RD-Tree	10	2.4	1.4	0.9	0.7
B-Tree	100	27.6	26.8	26.4	26.2
RD-Tree	100	19.2	10.4	6.7	4.5
B-Tree	1000	6:28	6:14	5:50	5:35
RD-Tree	1000	5:56	2:32	1:30	1:18

Figure 4.14: Times for query sets against synthetic databases. SQL Explain shows the B-Tree method always electing to disregard all indexes and perform a sequential scan. The RD-Tree query plan always includes zero sequential scans. The number in brackets below the Time column header is the tlen.

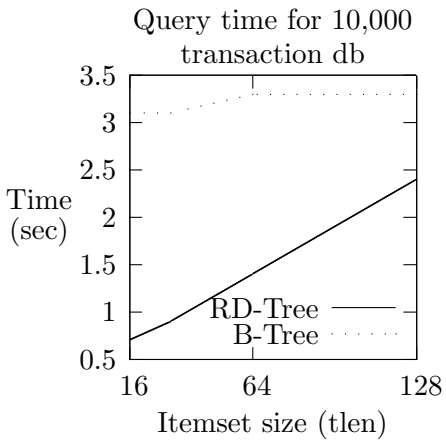

Figure 4.15: Execution times for queries using either B-Tree or RD-Tree indexing against databases of varying density with 10,000 transactions.

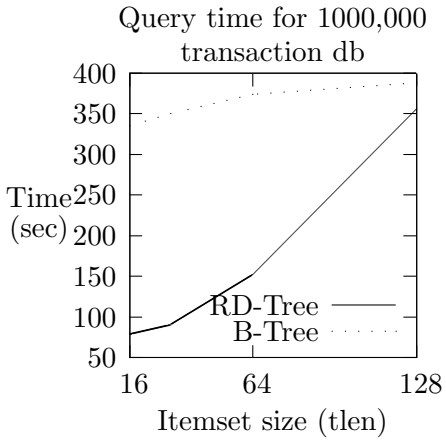

Figure 4.16: Execution times for queries using either B-Tree or RD-Tree indexing against databases of varying density with 1,000,000 transactions.

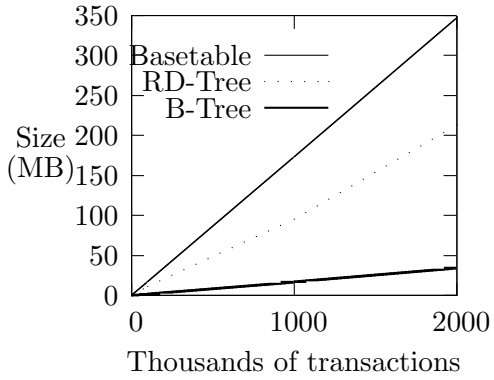

Figure 4.17: Statistics for the base table and indexes of the synthetic databases. Note that the B-Tree index size is only for a single column whereas the RD-Tree covers all 32 columns.

Query Type	Items/ pattern	Max (%)	Mean (%)	Std Dev	Time (sec)
B-Tree	256	64	26	19	29.3
RD-Tree					38.9
B-Tree	128	36	13	10	27.6
RD-Tree					19.2
B-Tree	64	19	6	5	26.4
RD-Tree					10.2
B-Tree	32	10	3	3	25.7
RD-Tree					6.7
B-Tree	16	5	1.6	1.4	25.9
RD-Tree					4.8

Figure 4.18: Effect of formal context density on RD-Tree performance for 100,000 transaction database. The number of items per pattern was reduced in increments from 128 to 16 giving a max, average and standard deviation of set bits in the formal context as shown.

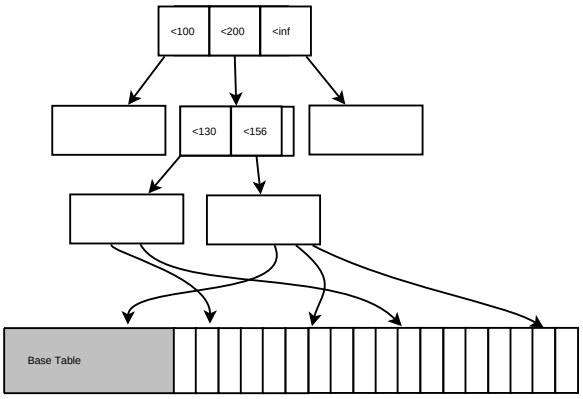

Figure 4.19: Basic Generalized Index Search Tree structure. There are four internal nodes (shown at the top) and two leaf nodes (just above the base table) which contain links to the actual information that is indexed. The page size for this tree is illustrative and would normally contain hundreds/thousands of entries.

Performance gains from RD-Trees are very effective for sparse formal contexts where queries can be resolved five times faster on large data sets as shown in Section 4.4.4.2. The largest benchmark results were found when applied to a large dataset from the UCI collection where the formal context was generated by nesting one conceptual scale inside another. In such an environment queries can be executed over 80 times faster using an RD-Tree than without.

4.4 Asymmetric Page Split Generalized Index Search Trees for Formal Concept Analysis

In order to improve again on the RD-Tree results presented in Section 4.3.1 one must first step back from the problem, considering a more generalized index structure and how the RD-Tree relates to it. It is in the context of this more generalized indexing that one can see how the index tree is formed and in particular the direct reliance of the overall index structure and depth on how overfull index pages are split.

A framework for building secondary storage search trees was recently introduced as the Generalized Index Search Tree [49, 10]. A Generalized Index Search Tree abstracts the core operations of a tree index structure into a small well defined collection of functions. Both the R-Tree and RD-Tree can be considered as specific Generalized Index Search Trees. The page splitting and propagation of page splits toward the tree root as described above directly transfers into a Generalized Index Search Tree.

A Generalized Index Search Tree is constructed from a collection of pages. A page is usually much larger than individual keys and may contain many keys. Page sizes are generally selected based on the hardware that the system runs on. A typical page size is 8Kb which are usually also referred to as nodes [33]. Some nodes contain links to other "child" nodes and thus form a tree. Nodes with child links are referred to as internal nodes. Children without links to other nodes – leaf nodes – contain links to the indexed data itself. An example tree is shown in Figure 4.19.

The process of adding an entry to a Generalized Index Search Tree can be consid-

ered as two parts: finding the appropriate leaf node in the index, and adding an entry to that leaf. Finding the leaf node occurs in a similar manner to normal search. The difference between search and insert being when a node has multiple children which could contain the new entry a `penalty` function is used to decide which child node to insert the entry into. An entry can only be inserted into one leaf node. Typically `penalty` will select the child node which is the most closely related to the new entry in order to aid future searches.

When data is added to a Generalized Index Search Tree eventually a leaf node will become overfull. When a node is overfull it is typically split into two nodes: the original node and a new node. Entries are then redistributed between the original node and the new node. The creation of the new node necessitates a new entry in the original node's parent linking the new node into the tree structure. By adding an entry to the parent node the parent itself may become overfull and thus the process of splitting nodes will continue up the Generalized Index Search Tree to the root while overfull nodes still exist.

The process of splitting a node in a Generalized Index Search Tree is delegated to the `picksplit` function. This function decides for each of the entries in the overfull node whether to store that entry into the original node or new node and provides the updated and new keys for the parent node.

It can be seen from the above description that the functions which will decide the shape of a Generalized Index Search Tree are the `penalty` and `picksplit` functions. The following presents customizations of the RD-Tree `picksplit` function for better Formal Concept Analysis performance.

As the RD-Tree's `picksplit` is based on that of the R-Tree the presentation will begin with the R-Tree. The R-Tree [44] is a data structure that was created to allow spatial objects to be indexed effectively. Keys in an R-Tree are n-dimensional bounding boxes. The R-Tree `picksplit` first selects the two elements who's bounding box are furthest apart as keys to be placed into the parent. These new parent keys are updated as children are distributed into each child in order to be the bounding box of their respective child node. The remaining elements are then distributed as children of one of these new parent keys. A child is distributed into the node to which it causes the least expansion of that node's bounding box.

The RD-Tree [50] operates similarly by treating input as an n-dimensional binary spatial area. The R-Tree notion of containment is replaced by set inclusion and the bounding n-dimensional box replaced by a bounding set. The union of a collection of sets forms the bounding set. The bounding set of a child is thus defined as the union of all the elements in the child. The bounding set defined in this way preserves the "containment" notion of the R-Tree during search as a subset relation. When seeking an element which might be in a child it is sufficient to test if the sought element is a subset of the bounding set for the child to know if that subtree should be considered. The standard RD-Tree `picksplit` is based on the spatial R-Tree index structure's `picksplit`.

There is a major distinction between the standard R-Tree index keys and the RD-Tree index keys: the R-Tree index key is based around n-dimensional bounding boxes whereas the RD-Tree is based around a "bounding set". For a given page the bounding set is simply the union of all the keys in the page. The bounding set faithfully serves the same purpose as bounding box from which it was derived: both can be treated in a

similar manner as a "container" in which all keys of a child page must reside.

Any n-dimensional bounding box can always be represented as $2 \times n$ coordinates: those coordinates in n-space on two opposite sides of the bounding box. A bounding set has no such fixed representation and can require an arbitrary number of elements to represent it (up to the set cardinality). This distinction is very important, an RD-Tree based index structure needs to focus on keeping its keys small, particularly those closer to the root of the tree. Both a voluminous bounding box and a bounding set with many elements are less effective in limiting the amount of a Generalized Index Search Tree that must be searched. However, a large bounding set will also consume more of a page, limiting the branching factor of the tree.

For an index on the same information, a tree with a lower branching factor will be a deeper tree [33]. The limited branching factor index will also have more internal nodes. The efficient caching of internal nodes in a computer's RAM is critical to index performance [33]. Having more internal nodes will decrease the effectiveness of such RAM caches.

Another critical factor in the selection of small bounding sets is that once a bounding set is selected for a parent that bounding set has an extremely hard time becoming smaller again. Further compounding this issue is that poorly chosen bounding sets at the leaf node level will eventually promote overly large bounding sets towards the root of the Generalized Index Search Tree.

Minimization of the number of elements in any bounding set should be a priority given that variation in size of the bounding set effects the overall tree branching factor. At times a `picksplit` function should favour making one of the new bounding sets slightly larger if it means a that the other bounding set can become substantially smaller.

If the page split is too asymmetric then one of the resulting pages may contain only a single element. Such an index structure may lead to many leaf pages being drastically under full and degrade overall performance. To counter this situation a minimum page fill ratio can be selected. There is a balance between maintaining this minimum ratio and the extension of the bounding set required to do so.

Two methods to achieving an asymmetric page split are considered; a custom distribution function to completely replace the standard Guttman function [74] and the application of the Guttman distribution followed by a redistribution using Formal Concept Analysis.

4.4.1 Complete replacement of Guttman

This customized asymmetric `picksplit` algorithm pre-allocates elements to pages where they will not expand the page key, tries to minimize the expansion to one of the bounding sets, incrementally takes into account any expansion of bounding sets while distributing elements and attempts to leniently maintain a minimum page fill. The basic algorithm is shown in Figure 4.20.

Note that the complexity of the core algorithm after the initial left and right keys are selected and before post split shuffling is linear in the number of keys to distribute. The shuffle process can range from linear to k^2 where k is the number of keys. There are areas of the algorithm which invite variation. The major one being how best to handle a drastically asymmetric page split.

Though it would be simple to have the final shuffle move elements until a prede-

fined minimum page fill was achieved, it is better to try to get closer to this ratio while still minimizing the number of new elements that have to be added to the under full page's bounding set.

4.4.2 Guttman distribution followed by Formal Concept Analysis

In this method the standard Guttman generalization [50] is applied followed by the use of Formal Concept Analysis to improve the cardinality of one of the bounding sets.

The algorithm is shown in Figure 4.21. Abstractly the algorithm is mainly concerned with selecting which keys from the source page to move to the target page based on information from the concept lattice of the source page. As the final step of updating the extent sizes for the THeap is a computationally intensive task it is made optional and leads to two implementations for later benchmarking.

Shown in Figure 4.22 is the an example concept lattice for a page after the Guttman algorithm has been applied. Notice that the concept nodes are coloured depending on the number of keys which exactly match their concept or any downward transitively connected concept. One can immediately see that concepts with intent "c" and "d" are less strongly connected to the page. In the following concepts will be identified by their intent, for example, from the above we shall simply say concept "c" and "d". In particular there is a low overlap between "c" and "a" or "b".

Considering Figure 4.22, assuming that the fixed cuttoff of 40% allowed 14 of the 40 keys to be moved, the algorithm in Figure 4.21 would first move the key matching the "d" concept from the source to the target page. Following this movement of 1 key the next smallest movement would be for concept "c". The movement this time is for 5 keys making a running total of 6 keys moved. By elimination, the next candidate would be "a" or "b". This is where the updating of THeap makes a difference. If THeap is not updated then it is luck if "a" or "b" are selected because their initial counts are identical. If "a" is selected then the algorithm will terminate rather than move 12 more keys making a total of $18 > 14$. If "b" is selected (which is guaranteed when updating THeap) then the 8 keys matching "b" are moved. Note that there are only 8 matches because the previous move of "c" also moved the keys in common with "b and c".

4.4.3 Customized Key Compression

If the input bounding sets are generated from a known structure then compression can take advantage of that structure. The use of Formal Concept Analysis makes available both full and partial implication information. Consider the application of Formal Concept Analysis to a numeric input using a linear ordinal scale: given a set of objects O such that each $o \in O$ has a numeric value $v \in V$ associated with it such a scale will generate a set of (binary) formal attributes $a \in A$ associated with the set O. For example, if the object set was formed using the planets and V was the number of moons of the planet then perhaps $A = \{some, few, many\}$ where planets which have few moons also have some moons by implication.

Such ordinal scaling is typical in many applications of Formal Concept Analysis [38].

If we are representing a bounding set as a bitset and the first 63 bits are the result of such a linear ordinal scale then there is a direct implication between a bit and its

(1) Using the standard RD-Tree method select the initial left L and right R sets as new parent keys.

(2) For all sets yet to be distributed, preallocate any set which is a non strict subset of either parent key $\{L, R\}$.

(3) For all unallocated keys

   (a) Test if the current key is a subset of either updated parent key, if so then allocate that key to the child page of the respective parent key.

   (b) Attempt to minimize expansion of either parent key, when expansion has to occur prefer to expand the right parent's bounding set.

   (c) If both parents would have to expand the same amount to cater for a key then distribute as per the normal RD-Tree method.

(4) If either page is drastically under full shuffle keys into it from the other page. A page is under full if it is less than 10% utilized. Do not expand the under full page's bounding set by more than x elements. For our testing $x = 1$.

Figure 4.20: Pseudo code for asymmetric page split. Preallocation will require a traversal over all keys to be distributed and set union with each key and L and R. The next step is the central part of the algorithm and only loops over keys once. The central distribution will require set unions with each key and both L and R. These can't be cached from the values computed during preallocation because L and R are incrementally expanded during this phase. The final shuffle phase potentially touches most of the keys to be distributed.

(1) Apply the standard guttman algorithm to obtain the initial two page distribution.

(2) Calculate the size of the bounding sets of both pages. The page with the smaller bounding set is the source page and the other the target page.

(3) Calculate the maximum number of keys to move k from as the number of keys in the source page multiplied by a cutoff percentage x.

(4) For the source page:

  (a) Find the Intent (T_I) of the top formal concept of the page.
  (b) Find the lower covers of (T_I) and sort them by their extent size. Store this in THeap.
  (c) Initialize CumulativeKeysMoved = 0
  (d) While less keys than k have been moved:
    (i) select the next lower cover of (T_I) that has yet to be considered working from the lower cover with the smallest extent to the largest.
    (ii) Set CumulativeKeysMoved = CumulativeKeysMoved + number of keys in (T_I).
    (iii) If CumulativeKeysMoved $> k$ then exit
    (iv) Move the keys in (T_I) to the target page
    (v) **optionally** Remove (T_I) from THeap and recalculate the extents in THeap and resort it by extent size.

(5) If the sum of the size of the final bounding sets for the source and target page are larger than the initial sum of the size of the bounding sets then ignore the results of Formal Concept Analysis. Otherwise apply the asymmetric page split.

Figure 4.21: Pseudocode for asymmetric page split using guttman and an Formal Concept Analysis postprocess to achieve superior bounding set sizes.

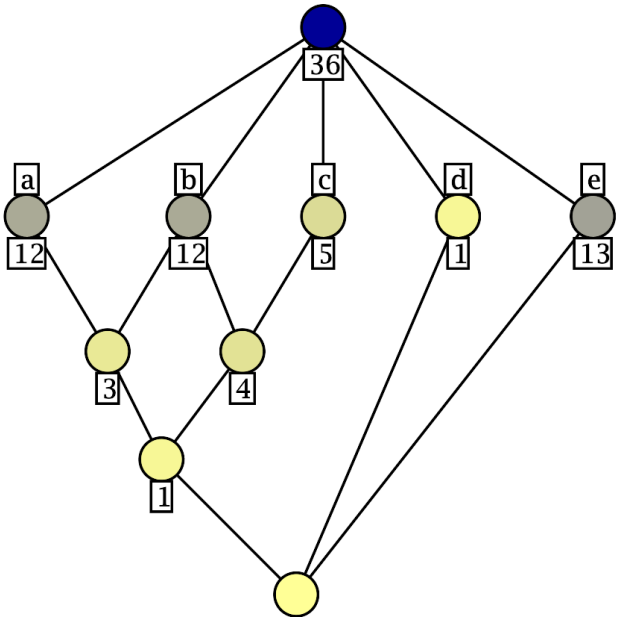

Figure 4.22: Concept lattice for the source page after Guttman's algorithm has been applied to obtain the initial distribution. The letters above the nodes indicate which attributes are introduced at that concept. When an attribute is introduced at concept x all concepts connected below concept x in the diagram also have that attribute. The numbers below the nodes indicate how many keys match that concept or any connected below it. For example, there is one key with attributes $\{d\}$, four keys with at least $\{b, c\}$ and one with $\{a, b, c\}$.

Original set	compressed	stored bitset	physical size (bits)
$\{1,2,3,4,5\}$	N	11111000...0	63
$\{1,2,3,4,5\}$	Y	000101	6
$\{1,2,3,...35,36\}$	N	111000...001100...0	63
$\{1,2,3,...35,36\}$	Y	100100	6

Figure 4.23: Compression of a bitset representing a linear scale.

predecessors. If there is an implication between bits then compression is possible.

Instead of using 63 bits to store this scale we can compress this to just 6 bits by storing only the position of the highest set bit in the first 63 bits. Some examples are shown in Figure 4.23.

If one considers a Generalized Index Search Tree created using two ordinal scales then the bounding set can always be compressed to just two integers. It is much more difficult to take advantage of cross implications between the two ordinal scales.

If the compression used by the Generalized Index Search Tree is to change then expensive updates to the index would be required. Potentially, every bounding set which is compressed would have to be loaded, decompressed, recompressed and saved again. Thus any implications between two ordinal scales which are to be factored into the Generalized Index Search Tree compression would need to be asserted by a domain expert.

The storage of a single integer for each ordinal scale in no way changes the mechanics of the Generalized Index Search Tree. Due to the storage of two integers it may be tempting to think that the tree is in some way more closely related to the B-Tree [33]. This is not the case, the compression is simply an implementation artifact to enable more bounding sets to be stored in internal nodes.

4.4.4 Performance Analysis

The benchmark system is an AMD XP running at 2.4GHz with 1Gb of RAM. The database is stored on a single 160Gb 7200RPM PATA disk. The implementations use the PostgreSQL Generalized Index Search Tree system. Testing was performed on the Covtype database from the UCI dataset [17] and a synthetic formal context generated with the IBM synthetic data generator [88].

Where an implementation includes the `NC` postfix there is no compression of bounding sets. Where an implementation includes the `Comp` postfix in its name it employs compression of bounding sets. Note that the compression is only ever applied to the bounding sets on internal pages, never to leaf nodes.

There are two compression techniques used – a generic compression `Comp` and compression relying on Formal Concept Analysis knowledge `FCAComp`.

The `RD` and `RD-Comp` use the standard Guttman `picksplit`.

The implementations are as follows: The `Asym-NC` uses the asymmetric page split algorithm from Section 4.4.1, `Shuf-NC` builds on `Asym-NC` by performing a shuffle after the initial allocation in an attempt to subvert the creation of drastically under full pages. Finally `Shuf-Comp` builds on `Shuf-NC` by including compression of bounding sets.

These implementations use an allowed cardinality expansion of 1 (see Section 4.4.1). To demonstrate gains for the compression outlined in Section 4.4.3 the `Shuf-FCAComp` takes advantage of the ordinal nature of the scales used to generate the set elements.

The `GuttFCA` implementations use the standard RD-Tree generalized Guttman distribution followed by Formal Concept Analysis to provide an asymmetric page split as detailed in Section 4.4.2. The "R" postfix indicates that the THeap is recalculated after keys from a concept are moved to the target page. The 30p and 50p postfixes indicate a 30% and 50% target for the number of keys to distribute from the source page respectively. All the `GuttFCA` implementations use compression – either generic compression when the `Comp` postfix is used or Formal Concept Analysis specific compression when the `FCAComp` postfix is used.

Two major metrics of interest are the tree depth and the number of internal nodes in the Generalized Index Search Tree. It is unlikely that an entire Generalized Index Search Tree will be resident in RAM. Normally some of the tree can be cached in RAM for a successive search. By minimizing the number of internal nodes and the overall tree depth we can increase the chances that successive searches can find an internal node in the RAM cache.

4.4.4.1 Performance on UCI Covtype dataset

This section examines the implementations in the setting of the application of Formal Concept Analysis on a large data source. This selected application is particularly difficult due to it having 512 formal attributes as well as each formal object having a relatively large number of attributes.

The UCI covtype database consists of 581,012 tuples (formal objects) with 54 columns of data (many-valued attributes). Two ordinal columns were used: the aspect and elevation. A scaled table scaledcov was created from the original data source with a bit varying field containing a single bit for each formal attribute. Formal attributes were created for the most frequent 256 values for both aspect and elevation resulting in a 512 bit column. An example formal attribute would be created from a predicate like "elevation < 3144". A smaller table called mediumscaledcov was created with only the first 10,000 tuples.

The static structure of the produced index for various Generalized Index Search Tree implementations is shown in Figure 4.24. As seen in Figure 4.24 using an asymmetric page split can reduce the number of internal nodes in the tree by over 30% (Comparing `Asym-NC` with `RD-NC`). Notice that `GuttFCA-50p-Comp` contains 76% and 81% the number of internal nodes when compared to `RD-Comp` and `Shuf-Comp` respectively.

Although the mean leaf fill goes down for the `Shuf-Comp` tree compared with the `RD-Comp`, there are also fewer leaf nodes in all. Considering the statistic of: leaf node count $\times$ mean leaf fill, the `Shuf-Comp` tree has an overall reduction of over 20% compared with `RD-Comp`. As to be expected the custom compression in `Shuf-FCAComp` significantly reduces the number of internal nodes in the tree. By allowing more bounding sets to be stored per internal node the tree itself is reduced in depth. Note that compressing the bounding sets for internal nodes has a significant effect on reducing the tree depth (compare `RD-Comp` and `RD-NC`).

Queries for the extent size of each single attribute value were executed against

Index	tree depth	index size (Mb) (Mb)	leaf node count count	internal node count count	mean leaf free free
RD-NC	17	66.1	3883	4582	5247
Asym-NC	14	52.2	3581	3100	4799
Shuf-NC	12	47.1	3353	2680	4539
RD-Comp	7	37.7	3846	977	4946
Shuf-Comp	7	33.1	3321	915	4565
GuttFCA-30p-Comp	6	34.3	3652	741	4681
GuttFCA-50p-Comp	6	34.3	3648	739	4677
GuttFCAR-30p-Comp	6	34.3	3652	741	4681
GuttFCAR-50p-Comp	6	34.3	3648	739	4677
Shuf-FCAComp	3	23.4	2912	35	3651
GuttFCA-30p-FCAComp	3	25.3	3207	35	4056
GuttFCA-50p-FCAComp	3	25.3	3207	35	4056
GuttFCAR-30p-FCAComp	3	25.3	3207	35	4056
GuttFCAR-50p-FCAComp	3	25.3	3207	35	4056

Figure 4.24: Overall statistics for various Generalized Index Search Tree implementations on the scaled UCI covtype database mediumscaledcov.

each index. During query execution the number of internal keys read was recorded. The results for the three uncompressed index structures are shown in Figure 4.25. It can be seen that using asymmetric page splits and post distribution shuffling lowers the number of internal keys touched.

Testing for two attribute queries was conducted by selecting two primary attributes and querying pairings of those primary attributes with each 16th attribute. For this test only the `RD-Compress` and `Shuffle-Compress` Generalized Index Search Tree are considered. Results are shown in Figure 4.26.

Queries for the extent size of 32 single attributes were executed against each index. The 32 attributes were selected for query x as the $16x^{th}$ formal attribute. Two attribute queries were formed using the 25, 248 and 293rd attributes in combination with every 16th attribute. The results are shown in Figure 4.27. It can be seen that using asymmetric page splits and post distribution shuffling lowers the number of internal keys touched. The higher branching factor of the `FCAComp` Generalized Index Search Tree does require more leaf keys to be touched on average.

The two attribute queries saw a reduction in the number of internal keys touched with no significant changes in leaf keys touched. Of particular note, the `GuttFCA-50p-Comp` Generalized Index Search Tree only touched 85% the number of internal keys when compared with `RD-Comp`.

4.4.4.2 Performance on Synthetic data

The following use synthetic data generated with the IBM synthetic data generator [88]. Parameters include the number of transactions (ntrans), the transaction length (tlen), length of each pattern (patlen), number of patterns (npat) and number of items (nitems). The parameters were as follows ntrans=100,000 and ntrans=1,000,000, nitems=1000, tlen=32, patlen=7, npats=10000.

The output of the IBM synthetic data generator is a list of **ntrans** transactions. Each transaction contains a number of items. Each item is represented by a unique integer in the range $\{1, ..., nitems\}$. Transactions were imported into an int array field in a PostgreSQL table including only the first n items for $n \in \{16, 32, 1024\}$. Such an arrangement allows an RD-Tree to easily be created on the input data of varying dimensionality.

The static index structure is presented in Figure 4.28. Gains can be seen for both the 32 and 64 dimensional RD-Tree `picksplit` customizations. For the 1,024 dimensional data the standard guttman `picksplit` remains superior. Notice that for the `rd32` the `GuttFCA-30p-Comp` is 1.6% larger on disk with about 2% more leaf nodes though has only 82% the number of internal nodes when compared to `RD-Comp`.

As can be seen from the 100,000 transaction table shown in Figure 4.29 the `Shuffle-Compress` index has significantly fewer internal keys touched in most queries. There are only two queries where `Shuffle-Compress` touches more keys than `RD-Compress`. For a million transaction table `Shuffle-Compress` always touches fewer internal keys. For the million transaction table the mean of the number of internal keys touched for all 32 attributes for `Shuffle-Compress` is 36% less than the same mean for `RD-Compress`.

The `Shuffle-Compress` has more leaf nodes than `RD-Compress` for both Generalized Index Search Tree. However the `Shuffle-Compress` only touched 38% the number of leaf keys that `RD-Compress` did overall. This can be explained because the

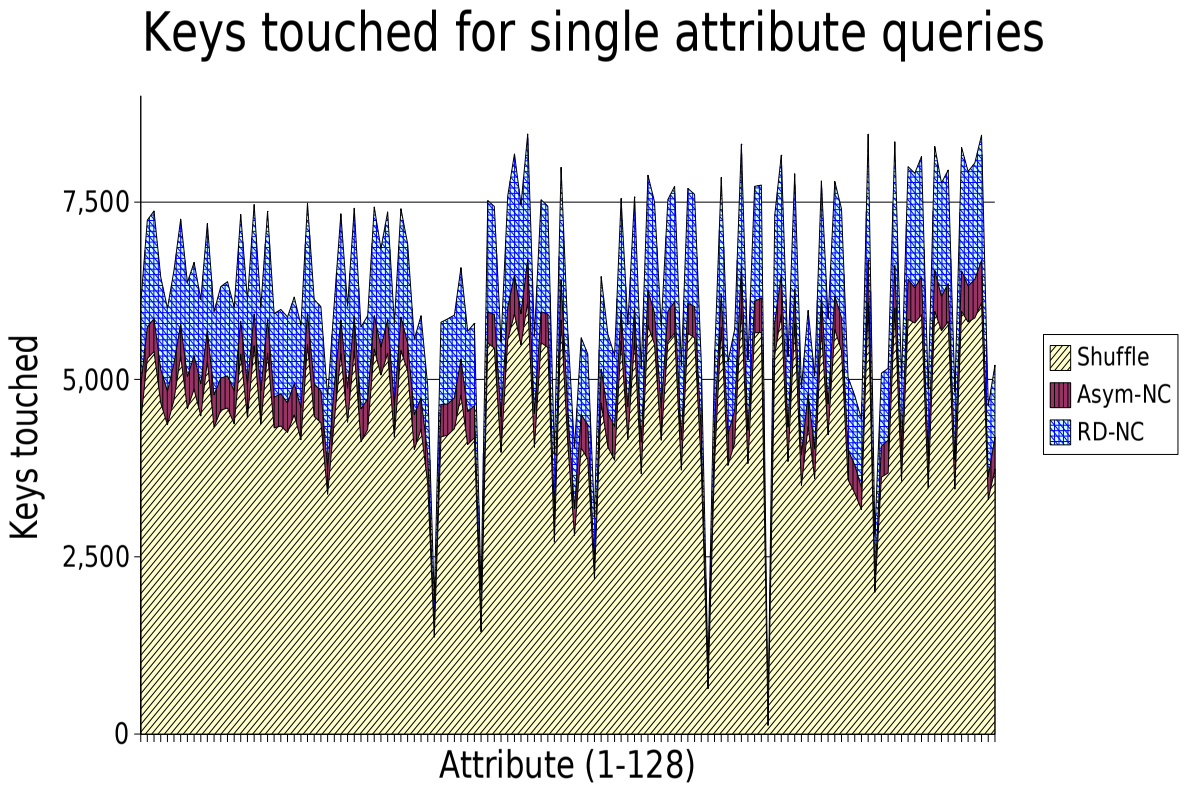

Figure 4.25: Number of keys touched during single attribute extent size queries for the first 128 attributes on an uncompressed index structures. The lowest number to touched keys is always for `Shuffle`. From there upwards are: Asym-NC and RD-NC, in that order.

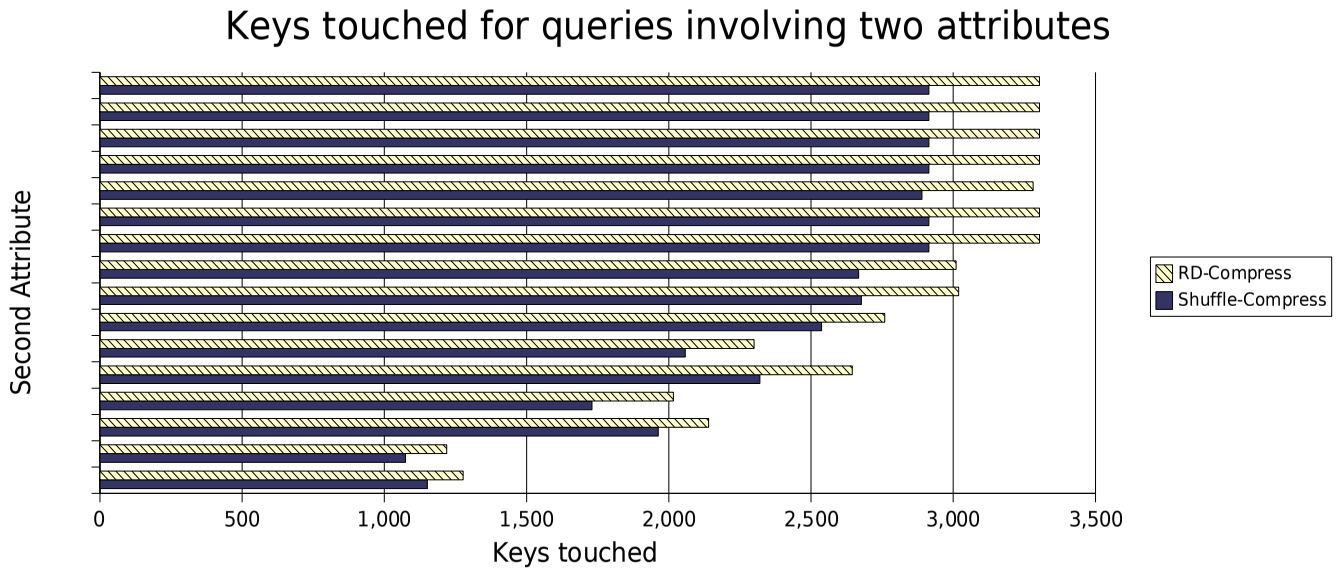

Figure 4.26: Number of internal keys touched while querying two attributes against the `RD-Compress` and `Shuffle-Compress` Generalized Index Search Tree for every 16th other attribute with a primary attribute 25.

Index	single attribute **internal**	single attribute **leaf**	two attribute **internal**	two attribute **leaf**
RD-NC	5875	5184	3908	3795
Asym-NC	4800	5188	3139	3841
Shuffle-NC	4401	5218	2946	3815
RD-Comp	3491	5260	2362	3885
Shuf-Comp	3107	5249	2083	3873
GuttFCA-30p-Comp	3171	5194	2014	3881
GuttFCA-50p-Comp	3167	5194	2010	3881
GuttFCAR-30p-Comp	3171	5194	2014	3881
GuttFCAR-50p-Comp	3167	5194	2010	3881

Figure 4.27: Mean number of keys touched for single and double attribute queries.

Index	tree depth	index size (Mb)	leaf node count	internal node count	mean leaf free
rd16					
RD-Comp	3	63.5	8033	95	3950
Shuf-Comp	3	64.9	8233	77	4051
GuttFCA-30p-Comp	3	63.9	8097	80	3982
GuttFCAR-30p-Comp	3	64.1	8118	81	3993
GuttFCA-50p-Comp	3	64.5	8175	77	4021
GuttFCAR-50p-Comp	3	64.5	8181	79	4025
rd32					
RD-Comp	4	67.1	8462	131	3749
Shuf-Comp	4	69.5	8789	103	3910
GuttFCA-30p-Comp	4	68.2	8620	107	3827
GuttFCAR-30p-Comp	4	68.8	8698	104	3864
GuttFCA-50p-Comp	3	69.5	8797	100	3913
GuttFCAR-50p-Comp	3	69.7	8820	99	3924
rd1024					
RD-Comp	5	81.5	9739	697	2804
Shuf-Comp	5	142.5	17577	669	5117
GuttFCA-30p-Comp	5	83.5	9982	710	2928
GuttFCAR-30p-Comp	5	84.8	10169	679	3020
GuttFCA-50p-Comp	5	91.4	11003	695	3384
GuttFCAR-50p-Comp	5	95.2	11599	582	3600

Figure 4.28: Overall statistics for various Generalized Index Search Tree implementations on the IBM data mining synthetic database.

`Shuffle-Compress` better clusters like keys into leaf nodes than `RD-Compress`. Also

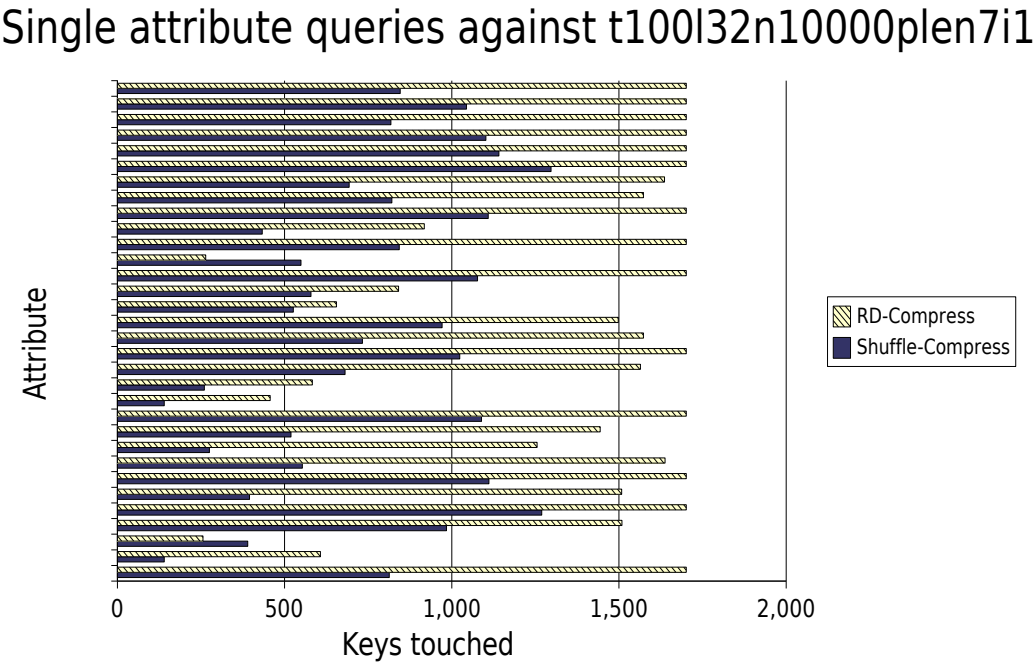

Figure 4.29: Single attribute queries for the first 32 attributes in t100l32n10000plen7i1. This data set involves 100,000 transactions, a total of 10,000 different patterns, a transaction length of 32, a pattern length of 7 items and a total of 1000 different items.

`Shuffle-Compress` being a 7% larger index size overall it has less congested leaf nodes.

A series of subset queries was posed to the database for one, two and three attributes to test the effectiveness of the index structure. Given that the three indexes contain 16, 32 and 1,024 attributes the single attribute queries were for each of the attributes that the index contains. As nitems=1,000 there are $C_2^{1,000} = 499,500$ combinations of two attribute queries possible. As such, for each of the n-attribute queries and index on the first r attributes, only $r - n$ queries are posed. These are for $i \in \{1, 2, ..., r - n\}$ for the attributes $\{a_i, ..., a_{i+n}\}$. As the input data should not be biased toward such queries they should when averaged be fairly representative of any such n-attribute query against the data. Shown in Figure 4.30 are the mean number of internal and leaf keys touched while performing these queries.

Note that the Formal Concept Analysis `picksplit` both yield superior results for the 16 and 32 attribute indexes. In particular the `GuttFCA-50p-Comp` index touches less than 16% the number of internal keys than `RD-Comp` for 3 attribute queries against the 16 attribute index.

4.5 Conclusion

This chapter has explained why conventional B-Tree indexing is ineffective for resolving Formal Concept Analysis queries in Section 4.2. This is also the pathological worst case empirical timings that were found in the previous chapter in Section 3.4.

Spatial Indexing can be employed in order to drastically reduce query times produced by ineffective use of relational database indices. The spatial designs presented in Section 4.3 are shown to be effective in the empirical testing performed in Section 4.3.1. Depending on the distribution of the data being indexed it was found that spatial indexing could resolve typical Formal Concept Analysis queries from five to 80 times faster than conventional indexing.

Further gains are obtained if the method that the spatial index uses to handle over full pages is modified as shown in Section 4.4. An over full page must be split into two pages and the old information distributed between these two pages. By employing an asymmetric distribution when splitting an index page the overall structure of the index tree can be modified to produce a more efficient index. The two index pages will require two keys to be placed into the parent index page, keeping these keys as small as possible is essential to allow more keys to fit into the parent page leading to an index tree with a reduced height. Each page in the height of an index tree will likely require an expensive disk head seek and so the index height plays a very large role in the overall index performance.

The empirical testing on the asymmetric distribution page splits is presented in Section 4.4.4. The best results can be obtained if the spatial index has more intimate knowledge of the data it is indexing. This knowledge can be exploited to compress index keys and as a consequence of smaller index keys, reduce the overall height of the index structure itself. This is detailed in Section 4.4.3. When employing customized compression tailored to Formal Concept Analysis only 16% the number of internal keys are touched relative to a standard RD-Tree for a 16 attribute index. For the IBM data mining synthetic input the Formal Concept Analysis based page split with Formal Concept Analysis tailored index compression could actually reduce the index tree height from 4 to 3 which will directly reduce the number of disk seeks required for each query.

Index	in1	lf1	in2	lf2	in3	lf3
rd16						
RD-Comp	6147	1851	4515	3362	3772	123
Shuf-Comp	2013	597	1041	88	712	31
GuttFCA-30p-Comp	2587	745	1197	116	869	43
GuttFCAR-30p-Comp	2550	750	1101	117	670	35
GuttFCA-50p-Comp	1634	554	789	76	586	22
GuttFCAR-50p-Comp	1677	548	766	67	525	20
rd32						
RD-Comp	6584	1926	5045	455	4129	168
Shuf-Comp	3868	905	1658	188	1044	60
GuttFCA-30p-Comp	3887	1051	2000	214	1375	77
GuttFCAR-30p-Comp	3503	1002	1846	202	1282	69
GuttFCA-50p-Comp	2822	820	1413	158	921	51
GuttFCAR-50p-Comp	2543	761	1242	137	819	43
rd1024						
RD-Comp	8933	4622	7586	2156	6451	1083
Shuf-Comp	8387	2578	4229	889	2432	384
GuttFCA-30p-Comp	9507	4580	8409	2206	7425	1156
GuttFCAR-30p-Comp	9218	4197	7703	1894	6394	947
GuttFCA-50p-Comp	9382	4097	7529	1862	6234	958
GuttFCAR-50p-Comp	8619	3457	6209	1383	4666	652

Figure 4.30: Average number of internal and leaf keys touched for single, two and three attribute queries on various Generalized Index Search Tree implementations. Note that i3 is the internal mean and l3 is the leaf mean. Internal counts are exact, leaf counts are expressed as figures rounded to the nearest hundred. ie. a leaf count in the table of n is for a reading of $n \times 100$ leaf keys.

This impact will be more again when one considers that a relational database will cache the root index node and quite possibly the direct children of the root node. This leads to the Formal Concept Analysis based index requiring one disk seek compared to the standard RD-Tree requiring two.

Chapter 5

Lattice Closure In A Timely Manner

5.1 Introduction

Attention will now be turned to methods for obtaining the concept lattice from the formal context. As this process is a computationally expensive operation the selection of algorithm and how it interacts with its input data are important factors contributing to how interactive the system as a whole can be.

Section 5.2 discusses the issue of finding the set of all concepts for a given formal context. This is followed by a new Border algorithm for explicitly recording the covering relation among the set of all concepts in Section 5.3. Empirical testing is carried out in Section 5.5 to verify that the border algorithm provides acceptable performance for up to a few thousand concepts.

The process of obtaining a concept lattice can be seen as two distinct subtasks: finding the set of all concepts and finding the covering relation of the concepts. Finding the set of all concepts is equivalent to finding the lattice closure. The set of concept intents uniquely determines the set of concepts.

A recent paper [96] presented both the Titanic algorithm and highlighted the link between finding the lattice closure in FCA and the problem of locating Closed Frequent Itemsets (CFI) in Data Mining [45]. The Titanic algorithm is based on the Apriori Data Mining algorithm [9].

This paragraph establishes the link between CFI in Data Mining and finding the set of concepts (intents) from a formal context in FCA. An itemset in Data Mining is a set of formal attributes in the formal context. The support of an itemset is the number of objects which have at least that itemset in their row in the formal context. This is shown in Figure 5.1.

For the set of all attributes A, a closed itemset is an itemset $X \subseteq A$ for which there exists no attribute $y \in A \backslash X$ such that the itemset $X \cup \{y\}$ has the same support as X. That is, any itemset is closed if it can't be expanded to contain any other attribute and not have a lesser support.

The "Frequent" part of the Closed Frequent Itemset relates to a cut off threshold in the Data Mining process. For a given minimum support value, any itemset with a higher support than the minimum is considered frequent. Thus a CFI has a support above the minimal threshold and can not be expanded to contain any other attribute without modifying its support. Much computational complexity can be avoided if the minimum support is set slightly higher than zero. For example, a minimum support in the range of 2% to 5%.

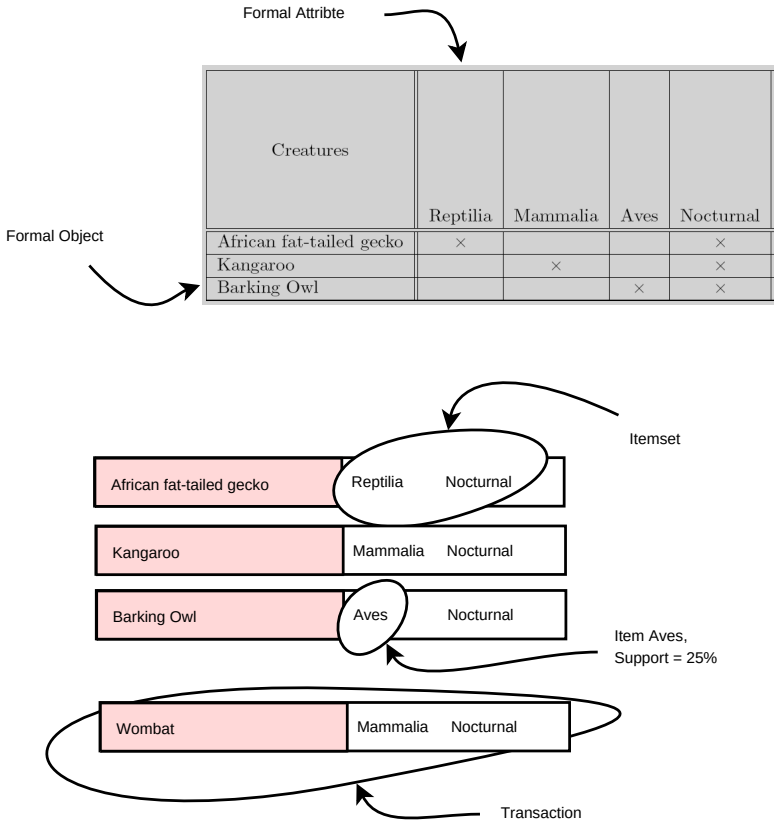

Figure 5.1: At the top of the figure is a Formal Context with one of its Formal Attributes and one of its Formal Objects highlighted. The Data Mining perspective is shown below. Formal Objects are seen as Transactions, a group of Formal Attributes is an Itemset. The support for a given Item or Itemset y is the number of transactions which contain a non proper superset that itemset y.

Finding the set of all intents in FCA is equivalent to finding the set of all Closed Frequent Itemsets (CFI) in Data Mining using a minimum support of zero. When the minimum support is above zero the side effect is that some concepts at the bottom of the concept lattice are not discovered. Such partial concept lattices are called iceberg lattices [96].

Algorithms for finding closed frequent itemsets in the Data Mining community are the subject of much research and recently an annual workshop to benchmark both algorithms and their implementations has emerged [40].

The set of CFI for a given dataset implicitly contains the concept lattice for that dataset. In order to display that concept lattice one must make explicit the covering relation of the CFI. Titanic and CHARM-L [107] explicitly record this covering relation whereas Data Mining algorithms normally stop after finding the CFI themselves.

The process of obtaining a concept lattice using Data Mining algorithms can be seen as two distinct subtasks: finding the CFI and finding the covering relation of the CFI. The active community research on the first step is discussed in Section 5.2 then the focus is turned to addressing the covering relation from CFI step in Section 5.3 with empirical testing in Section 5.5.

5.2 Finding the Closed Frequent Itemsets

There are various aspects of the algorithms for finding CFIs which will impact the design of the `findex`. These aspects include whether the algorithm expects its data in a vertical or horizontal orientation, the number of passes over the `findex` in various cases and how well the algorithm can be used in an iterative fashion for a fixed support value. Using the CFI algorithm in an iterative manner allows one to obtain the CFIs for the top n layers of the lattice while computing the next layer in the background.

The Titanic and Apriori algorithms perform passes over the `findex` inspecting each object in turn and working on the attributes which each object has before moving on to the next object. Such a visiting strategy is based on a horizontal index representation in which each object has the list of attributes that the object has. The common alternative to the horizontal representation is to consider each attribute to have an associated list of object identifiers which identify the objects that have that attribute. Such a representation is known as the vertical representation. For a vertical representation the closure computation forms the set intersection of many such lists. Some of the algorithms which operate on the vertical representation include Eclat and Clique [108] and implicitly the Logic File System [82] and Docco [11].

Many CFI algorithms form a Frequent-Pattern Tree (FP-Tree) which conceptually consists of a tree which includes the number of times each itemset occurs [46, 40]. After such a tree is built from the `findex` then CFI discovery can be performed using just the FP-Tree [46]. The FP-Tree has been represented using either a trie [46], an array [42] or a patricia tree [84].

One advantage of using an Apriori based algorithm is that for each iteration of the algorithm it generates the next set of intents always moving from the intents consisting of single attributes to those with a larger intent. This can be considered a breadth first search of the FP-Tree. The breadth first search has the advantage that for a fixed minimum support value for a session the lattice can be generated iteratively.

Attempts to adopt depth first FP-Tree searching algorithms such as FPClose [46]

to a breadth first iterative implementation suffer from the need to keep intermediate state available for the next iteration. For FPClose, once an item x is selected to recurse on a new FP-Tree is created which is conditional on item x in the database. Thus to perform an iterative FPClose many FP-Trees would have to be stored in memory or recalculated.

One key issue with using an Apriori algorithm is that it will take at least as many parses over the `findex` as the size of the largest intent in the concept lattice. In contrast the FPClose algorithm will parse over the `findex` generating an FP-Tree which is considerably smaller than the `findex` and from then on recursively mine the FP-Tree generating smaller FP-Trees. In some cases the FPClose can significantly outperform the Apriori algorithm [40].

When the concept lattice is to be presented as a diagram to the user the Apriori based algorithm would be advantageous. This allows the lattice to be computed while it is incrementally displayed. The limited number of concepts that can be shown in a diagram will limit the number of database passes required by Apriori. When the lattice is not directly shown in its entirety, such as in a command line lattice interface running the FPClose algorithm may be better suited.

For further details of Data Mining CFI algorithms see [40, 80, 83].

5.3 A border algorithm

Although mathematically the ordering relation among concepts is implicitly available once the set of all concepts is found, in a computer system this ordering must be explicitly recorded. This section presents a new algorithm for quickly making the covering relation explicitly available. A later section will provide empirical testing to verify the performance of the algorithm for the setting it was designed to operate in – making the covers explicit for up to a few thousand concepts. The major advantage of the border algorithm is that it does not require the formal context at all. This makes it efficient to apply where the data source is large and dynamic and simply reading the formal context sequentially implies a large performance cost.

The Titanic [96] algorithm is based on the Apriori Data Mining algorithm [9] and groups the finding of the concepts and recording the covering relation into a single algorithm. A major advantage of separating the recording of the covering relation from the finding of the set of concepts is that the latter can be freely varied. As mentioned, finding the set of concepts for an iceberg concept lattice is the same as finding the Closed Frequent Itemsets (CFI) in Data Mining. There is a great deal of active research being performed to improve the performance of finding the CFI (that is, the set of concepts for an iceberg concept lattice). Thus it is highly advantageous to separate the process of recording the covering relation on concepts (the border algorithm) from the finding of the set of concepts (CFI) in order to take advantage of newer algorithms for CFI discovery.

We turn now to the Border algorithm for explicitly recording the covering relation for a set of concepts.

In the following discussion C will represent the set of all concepts. A naive algorithm for finding the covering relation among concepts would inspect each concept as a potential parent for every concept. The complexity to find the parent-child relations between the concepts for this algorithm is proportional to $|C|^2$.

Various order theoretic properties of a concept lattice can be used to reduce the search space for covers and thus handle a larger $|C|$. The following border algorithm maintains a border set B as the concept lattice is inspected in a top down manner. If the mean size of B is y the complexity becomes proportional to $y \times |C|$.

The border algorithm is shown in Figure 5.2. The algorithm in Figure 5.2 will be referred to as the "intents only" algorithm in the following. The Maxima function used by Figure 5.2 is shown in Figure 5.3.

The algorithm only requires the set of all intents F as input. The algorithm requires that the intent set be a partial order of least intent attribute cardinality to maximum intent attribute cardinality. For example, all intents with only one attribute will be sorted before all intents with two or more attributes. As each intent will uniquely define a concept the discussion will also simply mention "the concept" for $a \in F$ instead of the concept with the intent $\{a\}$.

There may be many concepts $D \subseteq F$ sharing the smallest intent cardinality. Any intent of D may be the intent of the top concept of the lattice. As such a new intent is created t which will act as the intent of the top concept of the lattice. After the algorithm has completed, if t has only a single child then it can be removed from the lattice. This allows the algorithm to quickly know what concept is the top concept of the concept lattice. The top concept is used to initialize the border set with so it must be known.

The algorithm works by starting with a border of the top lattice node and sequentially working through concepts in the ordered intent set $a \in F$ and forming the intersection with each intent in the border set to find parents of a. After each concept is checked against the border set any new parent-child relations are explicitly linked and the border set is updated.

5.3.1 An Application Example of the Border Algorithm

The concept lattice shown in Figure 5.5 is used to present an example of the application of the border algorithm.

The steps that the border algorithm performs to find the concept lattice are shown in Figure 5.4. The current concept that is being worked on, the current border set as well as the Intents set from line 12 in Figure 5.2 before and after Maxima is called on it. All edge additions are shown at the time when they are performed on line 14 of Figure 5.2.

5.4 A Baseline Algorithm

The algorithm shown in Figure 5.2 will be referred to as the "intents only" algorithm during benchmarking. The Covering Edges [20] algorithm was implemented with a minor change to work against ice berg lattices. The modified algorithm is shown in Figure. 5.6. This modified algorithm will be simply referred to as "covering edges" in the following.

5.5 Performance Analysis

The benchmark system is a dual core AMD X2 running at 2.2GHz with 2Gb of RAM at DDR400. The database is stored on a single 80Gb 7200RPM PATA disk. The

(1) IntentsOnly(F)
(2)
(3) set $t := \{\}$
(4) $F := F \cup \{t\}$
(5) Border := $\{t\}$
(6)
(7) for each $a \in F$
(8) Intents := $\{\}$
(9) for each $b \in$ Border
(10) $y := b \cap a$
(11) Intents := Intents $\cup \{y\}$
(12) Intents := $Maxima($ Intents)
(13) for each $y \in$ Intents
(14) Add edge $y \rightarrow a$ to E
(15) Border := Border $\setminus \{y\}$
(16) Border := Border $\cup \{a\}$
(17)
(18) if $|children(t)| = 1$
(19) for each $c \in children(t)$
(20) Remove edge $t \rightarrow c$ from E
(21) F := F$\setminus\{t\}$

Figure 5.2: Algorithm to make the order relation between concepts explicit. Input: F the set of concept intents partially ordered on the cardinality of the Intent size from smallest intent size to largest. Output: E an edge mapping from parent concept intent to child concept intent forming the covers for the concept lattice of F. The Intents set introduced on line 8 is also partially ordered from intents with the smallest cardinality to intents with the largest cardinality.

```
(1)  Maxima( Intents )
(2)
(3)  Ret := {}
(4)  for each y ∈ reverse(Intents)
(5)       ismin := 1
(6)       for each r ∈ Ret
(7)            if {y} ∩ {r} = {r}
(8)                 ismin := 0
(9)       if ismin
(10)           Ret := Ret∪{y}
(11)
(12) return Ret
```

Figure 5.3: The Maxima function returns the set of intents which are maximal from the given set of intents. The Intents set used as input is ordered from smallest intent cardinality to largest intent cardinality. Line 4 indicates that the ordered Intents poset is to be inspected in reverse order, from largest intent cardinality to smallest.

(1) Current Concept = $\{\{a\}\}$
Border = $\{\{\}\}$
Intents = $\{\{\}\}$
Maxima(Intents) = $\{\{\}\}$
Add edge $\{\} \rightarrow \{a\}$

(2) Current Concept = $\{\{c\}\}$
Border = $\{\{a\}\}$
Intents = $\{\{\}\}$
Maxima(Intents) = $\{\{\}\}$
Add edge $\{\} \rightarrow \{c\}$

(3) Current Concept = $\{\{d\}\}$
Border = $\{\{c\}, \{a\}\}$
Intents = $\{\{\}, \{\}\}$
Maxima(Intents) = $\{\{\}\}$
Add edge $\{\} \rightarrow \{d\}$

(4) Current Concept = $\{\{b\}\}$
Border = $\{\{d\}, \{c\}, \{a\}\}$
Intents = $\{\{\}, \{\}, \{\}\}$
Maxima(Intents) = $\{\{\}\}$
Add edge $\{\} \rightarrow \{b\}$

(5) Current Concept = $\{\{ae\}\}$
Border = $\{\{b\}, \{d\}, \{c\}, \{a\}\}$
Intents = $\{\{a\}, \{\}, \{\}, \{\}\}$
Maxima(Intents) = $\{\{a\}\}$
Add edge $\{a\} \rightarrow \{ae\}$

(6) Current Concept = $\{\{ac\}\}$
Border = $\{\{ae\}, \{b\}, \{d\}, \{c\}\}$
Intents = $\{\{c\}, \{a\}, \{\}, \{\}\}$
Maxima(Intents) = $\{\{a\}, \{c\}\}$
Add edge $\{c\} \rightarrow \{ac\}$
Add edge $\{a\} \rightarrow \{ac\}$

(7) Current Concept = $\{\{ad\}\}$
Border = $\{\{ac\}, \{ae\}, \{b\}, \{d\}\}$
Intents = $\{\{d\}, \{a\}, \{a\}, \{\}\}$
Maxima(Intents) = $\{\{a\}, \{d\}\}$
Add edge $\{d\} \rightarrow \{ad\}$
Add edge $\{a\} \rightarrow \{ad\}$

(8) Current Concept = $\{\{bc\}\}$
Border = $\{\{ad\}, \{ac\}, \{ae\}, \{b\}\}$
Intents = $\{\{b\}, \{c\}, \{\}, \{\}\}$
Maxima(Intents) = $\{\{c\}, \{b\}\}$
Add edge $\{b\} \rightarrow \{bc\}$
Add edge $\{c\} \rightarrow \{bc\}$

(9) Current Concept = $\{\{bd\}\}$
Border = $\{\{bc\}, \{ad\}, \{ac\}, \{ae\}\}$
Intents = $\{\{d\}, \{b\}, \{\}, \{\}\}$
Maxima(Intents) = $\{\{b\}, \{d\}\}$
Add edge $\{d\} \rightarrow \{bd\}$
Add edge $\{b\} \rightarrow \{bd\}$

(10) Current Concept = $\{\{bcd\}\}$
Border = $\{\{bd\}, \{bc\}, \{ad\}, \{ac\}, \{ae\}\}$
Intents = $\{\{bc\}, \{bd\}, \{c\}, \{d\}, \{\}\}$
Maxima(Intents) = $\{\{bd\}, \{bc\}\}$
Add edge $\{bc\} \rightarrow \{bcd\}$
Add edge $\{bd\} \rightarrow \{bcd\}$

(11) Current Concept = $\{\{abcde\}\}$
Border = $\{\{ad\}, \{ac\}, \{ae\}, \{\}, \{bcd\}\}$
Intents = $\{\{bcd\}, \{ae\}, \{ac\}, \{ad\}, \{\}\}$
Maxima(Intents) = $\{\{bcd\}, \{ad\}, \{ac\}, \{ae\}\}$
Add edge $\{ae\} \rightarrow \{abcde\}$
Add edge $\{ac\} \rightarrow \{abcde\}$
Add edge $\{ad\} \rightarrow \{abcde\}$
Add edge $\{bcd\} \rightarrow \{abcde\}$

Figure 5.4: Steps performed by the border algorithm to find the covers of the concept lattice shown in Figure 5.5.

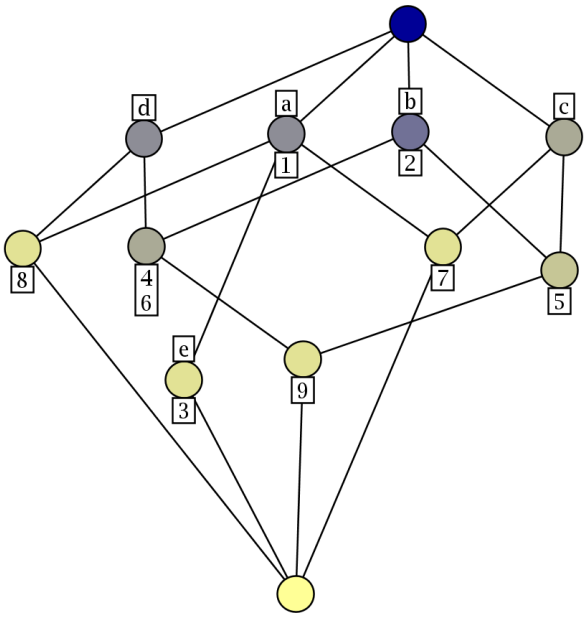

Figure 5.5: Concept lattice used as example of border algorithm application.

(1) IceBergCoveringEdges(C, (G, M, I))

(2) for each(X,Y) $\in C$

(3) Set count of any concept in C to 0

(4) for each $m \in M \backslash Y$

  (a) inters := $X \cap \{m\}$'

  (b) Find $(X_1, Y_1) \in$ C such that X_1 = inters

  (c) **if((X_1, Y_1) exists) then**

    (i) count(X_1, Y_1) := count(X_1, Y_1) + 1

    (ii) if($|Y_1|$ - $|Y|$) = count(X_1, Y_1) then

      (ii.a) Add edge $(X_1, Y_1) \rightarrow$ (X,Y) to E

Figure 5.6: Modified CoveringEdges using the same syntax as in Concept Data Analysis [20]. As the iceberg lattice does not contain all concepts, the modified version must check that the concept (X_1, Y_1) exists before proceeding.

implementations use the PostgreSQL database to read the CFI. Testing was performed on a synthetic formal context generated with the IBM synthetic data generator [88] databases from the UCI dataset [17] and two filesystem examples. The filesystem examples include a formal context derived from the metadata of 67,000 document files [3] and 2,000 geospatially tagged digital pictures. The digital picture context contains over 90 formal attributes.

Tests were run multiple times and the result of the final run was taken. This reduces the impact of the relational database, disk speed and other non relevant implementation details because most information will be coming directly from RAM cache in the relational database itself. This "hot caching" is acceptable as we are mostly interested in the speed of the core algorithm which makes explicit the covering relations of the CFI. The core of the implementation was compiled with gcc 4.1.1 using the `-O4` flag to turn on code optimization.

The implementation itself is single processed and single threaded. At times where the database server and implementation can run concurrently then advantages in the dual core CPU can be seen. Note that these are limited to the data fetching portions of execution, the actual finding of covers is completely contained in the implementation and makes no use of the second CPU core.

In order not to disadvantage either algorithm in empirical testing variants were created of each. As the intents only algorithm makes use of many temporary small sized bit sets it was implemented around `boost::dynamic_bitset<>` bit container.

As the covering edges algorithm makes extensive use of intersecting very large sets, see line 4a from Figure 5.6, the covering edges uses a bitset implementation which makes use of gap encoding and SIMD execution. Modern processors support Single Instruction Multiple Data (SIMD) execution which allows simultaneous processing of 4, 8, 16 or more data items with the same instruction in the same processor clock cycle. The general SIMD concept is also known by specific implementations such as MMX, SSE, SSE2 etc. These two optimizations greatly reduce execution time for covering edges on large data sets.

The use of SIMD carries a setup overhead and as such can actually be a disadvantage for smaller bitsets. The choice of bit set implementation being advantageous for each algorithm was tested by using both bit set implementations to verify empirically that the choice was optimal.

For sets with 64 or less elements an implementation of intents only was created using a single 64 bit integer instead of a `boost::dynamic_bitset<>`. For covering edges a version which operates on the row reduced context was created. This implementation will be referred to as covering edges reduced.

There should be somewhat more efficient ways to perform the covering edges "setup" that is shown in the following. Regardless of potential improvements this setup time can not be avoided and the relative differences in speed shown will remain.

5.5.1 Performance on Synthetic data

The following use synthetic data generated with the IBM synthetic data generator [88]. Parameters include the number of transactions (ntrans), the transaction length (tlen), length of each pattern (patlen), number of patterns (npat) and number of items (nitems). The parameters were as follows ntrans=10,000, ntrans=100,000 and

ntrans=1,000,000, nitems=1000, patlen=7, npats=10000. For each ntrans value, the tlen was varied between 32, 128 and 256. Some combinations of tlen and ntrans could not be generated because the implementation ran into RAM shortages.

The output of the IBM synthetic data generator is a list of **ntrans** transactions. Each transaction contains a number of items. Each item is represented by a unique integer in the range $\{1, ..., nitems\}$. Transactions were imported into a bit field column type in a PostgreSQL table for $n \in \{32, 128, 256\}$. An expression index was then created on the corresponding int array for the bit field column by using a conversion function to generate an int array from a bit field. Such an arrangement allows an RD-Tree to easily be created on the int array using the same implementation from Section 4.4.4.2. The base data type being a bit field allows the fastest possible bulk transfers of data.

The Apriori [9] algorithm was used to generate the CFI for each dataset. Parameters for support were varied in order to obtain a desired number of CFI for each data set. The target CFI count was 1,000, 3,000 and 10,000. Results were not recorded for datasets which do not support the target number of CFI.

Shown in Table 5.1 is the number of CFI for each input data set along with the size of the row reduced formal context. The results of running the two algorithms against the datasets in Table 5.1 is shown in Table 5.2. Where the number of CFI is less than 3,000 the intents only method has a clear advantage. However once the number of CFI is at 10,000 the covering edges based algorithms perform much better. The border algorithm functions very poorly on the 10,000 CFI input because the average number of concepts in the border set is very high, as shown in Table 5.3. As the border algorithm, shown in Figure 5.2, visits each member of the border set for each concept having the average border size almost half the number of concepts makes the overall complexity of the border algorithm unacceptable.

ntrans (x1000)	tlen	$\|CFI\|$ 1,000	$\|CFI\|$ 3,000	$\|CFI\|$ 10,000	row reduced object count (x1000)
1.0	128	931	2867	9989	1.0
10.0	32	1118	.	.	1.0
10.0	128	991	2983	9885	10.0
10.0	256	1155	3009	9767	10.0
100.0	32	906	2909	.	3.8
100.0	128	955	2880	9494	100.0

Table 5.1: The number of CFI for each configuration. The reduced count is the number of transactions in an object row reduced formal context. The reduced count plays a role in the Covering Edges implementation. As can be seen the reduction process has no bearing for formal contexts with $tlen > 32$. Where the data does not support the requested number of CFI the table has blank cells.

5.5.2 Performance on a Filesystem data

5.5.2.1 Personal Photo Collection

The first dataset is for 2,000 photographs with metadata describing their location both semantically and quantitatively. Semantic metadata is captured by associating

ntrans (x1000)	tlen	$\|CFI\|$	algorithm	covers time (mm:ss.d)	$\{m\}'$ time (mm:ss.d)	setup X (mm:ss.d)
1	128	931	intents only	2.5		
			covering edges	4.5	0.02	1.1
		2867	intents only	13.4		
			covering edges	8.2	0.02	3.5
		9989	intents only	3:22.3		
			covering edges	30.6	0.02	12.3
10	32	1118	intents only	1.2		
			covering edges	1.4	0.01	0.8
	128	991	intents only	1.7		
			covering edges	2.6	0.2	7.0
		2983	intents only	15.9		
			covering edges	8.3	0.2	18.1
		9885	intents only	3:24.2		
			covering edges	28.9	0.2	57.7
	256	1155	intents only	2.6		
			covering edges	7.1	0.4	12.5
		3009	intents only	16.0		
			covering edges	19.1	0.4	31.0
		9767	intents only	3:20.1		
			covering edges	1:21.2	0.4	33.8
100	32	906	intents only	0.9		
			covering edges	1.4	0.9	2.8
			covering edges R	0.8	0.04	1.1
		2909	intents only	10.4		
			covering edges	5.6	0.9	6.2
			covering edges R	3.1	0.04	3.2
	128	955	intents only	1.6		
			covering edges	3.2	2.0	57.3
		2880	intents only	15.3		
			covering edges	10.5	2.0	41.5
		9494	intents only	3:12.8		
			covering edges	34.6	2.1	30.6

Table 5.2: Time taken by various algorithm implementations to make the covering relations between CFI explicit.

ntrans (x1000)	tlen	$\|CFI\|$	average border size
10	256	3009	956
		9767	3399
100	32	906	331
		2909	1004
	128	955	432
		2880	1162
		9494	3991

Table 5.3: Average border size for various CFI data sets.

place names with each file. Quantitative metadata comes from both the metadata automatically recorded by a digital camera or film processing machine and also indirectly from the place name tags in the form of longitude and latitude information.

Using the border algorithm takes 0.03 seconds whereas the covering edges requires 0.49 seconds plus 0.2 seconds to setup data structures. Note that although this data was only for 2,000 images since the border algorithm is dependent only on the CFI this advantage would only increase proportionally with the number of images in the collection.

5.5.2.2 Two scales on 200,000 files

An findex was created on a Fedora Core 4 Linux machine using libferris 1.1.54 of 201,759 files in `/usr/share/`. For this findex a nominal scale was created on mimetype and a data driven scale on file modification time. A total of 141 formal attributes was created with these two scales. This findex is the same as that used in Section 6.2.

With these two scales, the support was set to give two datasets with 955 and 5463 concepts. For the 955 concepts the intents only algorithm took 1.6 seconds whereas the covering edges required 7.5 seconds plus 14.8 seconds of setup time. For the larger set of 5463 concepts the intent only algorithm needed 1:07 whereas the covering edges completed with 42.5 and 25.5 seconds of setup time.

5.5.3 Performance on UCI Covtype dataset

This section examines the implementations in the setting of the application of Formal Concept Analysis on a large data source. This selected application is particularly difficult due to it having 64 formal attributes as well as each formal object having a relatively large number of attributes.

The UCI covtype database consists of 581,012 tuples (formal objects) with 54 columns of data (many-valued attributes). Two ordinal columns were used: the aspect and elevation. Formal attributes were created using 32 formal attributes per ordinal scale. An example formal attribute would be created from a predicate like "elevation < 3144".

A subsample with only 100,000 tuples was created for testing. A support cutoff was selected to generate datasets with roughly 1,000 and 3,000 CFI. The results are shown in Table 5.4. It can be seen that the intents only algorithm suffers when the number of CFI increases. The intents only algorithm does not require to touch the base table in order to find X for all CFI which presents a significant setup cost for the covering edges algorithm on such a large input dataset.

5.6 Conclusion

The border algorithm can suffer poor performance as the mean border set size grows. Some testing has revealed border set averages in the order of half the concept set size.

This border set issue does not significantly impact the border algorithm where the size of the CFI set is small ($< 1,000$)). The use of the border algorithm is generally advantageous when the number of concepts in the CFI is relatively small ($< 3,000$)).

$\|CFI\|$	average border size	intents only (seconds)	covering edges core (seconds)	covering edges setup (minutes : seconds)
1,000	522	1.8	1.7	4:08
3,000	1376	13.5	4.8	10:42

Table 5.4: Performance of intents only and covering edges algorithms on CFI drawn from 100,000 objects with 64 attributes.

The efficiency of the border algorithm is independent of the number of formal objects. Only the CFI (that is, the set of concepts of an iceberg concept lattice) are required for the border algorithm.

For the covering edges algorithm, as the number of formal objects increases the burden of inspecting the whole database during algorithm setup increases. The setup for the covering edges algorithm includes finding the extent of each concept and the attribute extent for every attribute in preparation for step 4a of Figure 5.6.

For a large formal context where formal objects have few attributes and the set of all formal attributes A is relatively large the covering edges algorithm can be very slow. This is because step 4a of Figure 5.6 will be executed on average $|C| * |A|$ times each involving a $|O|$ length bit intersection operation.

Chapter 6

Formal Concept Analysis and Semantic File Systems

6.1 Introduction

Previous chapters have sequentially focused on building up a system in which Formal Concept Analysis can be applied to a Semantic File System. First an underlying system was detailed which can create an index of a Semantic File System with acceptable query performance (Chapter 3), then additional index structures were described allowing the common queries that Formal Concept Analysis will lodge at such an index to be resolved in an acceptable time frame (Chapter 4) and finally attention was turned to obtaining the concept lattice from the set of concepts reasonably quickly (Chapter 5). At this stage we are finally in a position to apply Formal Concept Analysis to a Semantic File System to create a query and navigation system.

This chapter is concerned with the application of Formal Concept Analysis to Semantic File Systems. In particular how to select the Formal Attributes. This includes the creation of scales on numeric ranges such as geospatial information, number ranges which present high clustering of values and numeric ranges that represent time values. A brief investigation into SELinux which will be followed up in Chapter 7. The structuring of a concept lattice based on file URLs and the use of Wordnet. The chapter concludes with a look at context based navigation for handling the presentation and interaction with large concept lattices for the user.

The chapter starts by describing various scaling methods for nominal data in Section 6.2.1, geospatial data in Section 6.2.2, numeric data in Section 6.2.3, temporal data in Section 6.2.4 and metadata in file paths in Section 6.2.6. The issue of navigating a large concept lattice is then investigated in Section 6.2.7 presenting a context based and file system based method of interacting with a concept lattice.

6.2 Application of Formal Concept Analysis

It has been found that in many cases some pre-analysis for a Semantic File System is needed in order to best expose the Semantic File System without generating a cluttered output.

The standard scale types of Formal Concept Analysis: nominal, ordinal and inter-ordinal can be used though some extensions are useful. Using a file's URL as a source of metadata for use in FCA can also be useful. Together with the applications described below there is a method of restricting which documents from an findex are potentially useful. This allows areas of the findex to be negated from query results **en masse**. For

example, one might consider only documents under `/usr/local` to be of interest for a particular analysis and so restrict all results to also satisfy this condition.

To demonstrate, an findex was created on a Fedora Core 4 Linux machine using libferris 1.1.54 of 201,759 files in `/usr/share/`.

6.2.1 Scaling nominal orders

In addition to the standard treatment of nominal scaling [38, 20], two new capabilities for handling ordering over nominal attribute creation have been found useful.

The first ordering capability is to handle MIME type like strings such as `image/png` by allowing the values of the distribution to be split into distinct parts and have common parent attributes created. Following the MIME example, a common parent for all image files would be the new `mime.image` attribute. Using this form of nominal scaling an ordering can be introduced based on the values of the distribution which will help to generate a taller, narrower concept lattice [20].

The second ordering capability is to take advantage of the ordering over the user defined tags when performing nominal scaling via an tag. The ordering on the tags is a partial order allowing reasonable flexibility in how one designs tag categories. The ability to handle entire downsets relative to the tag ordering when generating a formal context allows one to see their lattice including the influence of their tag ordering. Given an ordered set P and $Q \subseteq P$ then Q is a downset iff $x \in Q, y \in P$ and $y \leq x$ then $y \in Q$.

6.2.2 Scaling Geospatial information

Geospatial metadata is exposed through two cooperating interfaces in libferris. These are the latitude and longitude EA and the tag system. Geospatial tags are those which are a child of the `libferris-geospatial` tag in the tag partial ordering. Interaction with the filesystem for tagging and retrieval is usually simpler when tags with city or place names are used instead of world coordinates.

Associating geographical information with files is often referred to as geotagging in the literature. The tags with digital longitude and latitude information associated and their hierarchy are normally defined by the user. The association of geotags with files is performed either explicitly by the user or as an automated process using a GPS device that continuously collects location and timestamp information. The collected GPS and timestamp information is then automatically merged with the filesystem, for example, using the creation time of a file and interpolating the GPS coordinates that indicate roughly where the user was located when that file was created.

As the tag system is employed the scaling methods of Section 6.2.1 are also applicable for geospatial values. A major advantage of the geotagging system coexisting with the latitude and longitude system is the ability to handle geospatial regions. The tag partial order can be used to define geospatial regions that expand from point locations to physically containing regions. For example, the Sydney Opera House might be given a specific tag with Sydney as its parent. The Sydney tag may have Australia which itself has `libferris-geospatial` as its parent. If less specific tags in the partial order define containing geospatial regions then the downset handling in Section 6.2.1 can be used to introduce geospatial refinements into the concept lattice.

Without the ability to represent geospatial regions through the tag partial ordering in this way one would have to explicitly define the boundaries using bounding box constraints on the latitude and longitude for the region. Consider the difficulty in defining the boundary of the city of Sydney using only equality constraints on latitude and longitude.

6.2.3 Scaling numeric ranges

Three commands exist for creating formal contexts from numeric data in libferris. These are `numeric-ordinal`, `numeric` and `gf-numeric`.

The `numeric` client can create many binary attributes each exposing a numeric interval of the input data as is standard for Formal Concept Analysis. For example, consider scaling a numeric range of $\{1, 2, ..., 20\}$ into four attributes at an even interval of 5 using $\leq$ as is standard in Formal Concept Analysis. This will produce four attributes with higher successive values having less matches due to the $\leq$ relation.

The standard application of ordinal scaling preserves both linearity **and** density for the input [**?**]. Due to the intermixing of other attributes in a concept lattice it is hard to take advantage of the preservation of density information. When one places these four attributes alongside another ten attributes and generates a concept lattice the relation between ≤ 10 and ≤ 15 is not so immediately obvious from the concept lattice. One can see the order of the two attributes but the density information is lost due to the introduction of the other attributes.

For some value distributions using a linear interval for the range is ineffective. For example, if one is to scale the values for the file size metadata then the distribution of values may be very ineffectively presented when split into a low number of linear intervals. To overcome this issue the data-driven-scale option was added to allow numeric distributions to be scaled taking the value distribution into account. This option will make an output which will have smaller intervals where many files have similar values and larger intervals where few files match the interval.

The std-deviations option has been added to handle simpler value distributions by allowing output to be generated based on the mean and variance of the input distribution.

One can manually select where intervals begin and end using the GTK+ `gf-create` client. Figures 6.1-6.4 were generated with `gf-create`. For value distributions which neither fit direct data frequency nor standard deviation models the ability to explicitly choose where intervals begin and end on a value frequency plot can generate a small number of meaningful attributes. For this purpose an interactive graphical client was create allowing intervals to be selected with the mouse.

The plot for the modification time (mtime) of the findex is shown in Figure 6.1 and the metadata status change time (ctime) in Figure 6.2. One can see that although modification was more frequent in recent times the ctime plot has explicit natural clusters of values. Such clusters are likely due to large scale system administration activities such as distribution release upgrades. Using the graph a small number of meaningful attributes can be created based on major system update activities.

An EA was added to the libferris system to support the ability for many versions of a file's metadata to exist simultaneously in an findex [64]. This EA returns the current system time when it is read. As expected, the plot for this attribute gives

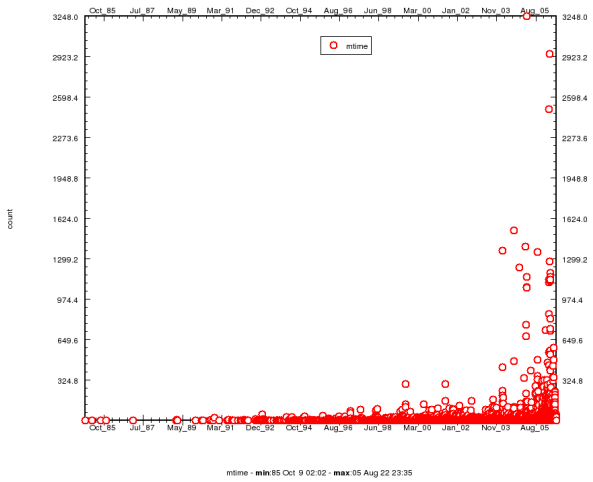

Figure 6.1: Plot of the modification time of 201,759 files from /usr/share/. Horizontal axis shows time from October 1985 to present day with almost 2 years between graduations. Vertical axis ranges from 0 to 3248 with around 235 files separating each graduation.

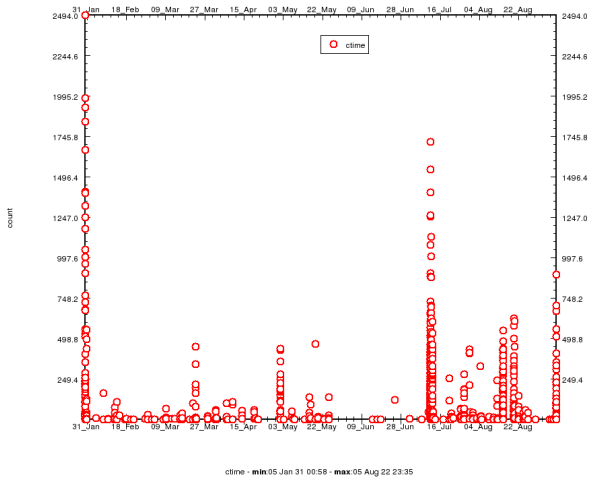

Figure 6.2: Plot of the ctime of 201,759 files from /usr/share/. The ctime for a file changes whenever any of its metadata (except atime from lstat) changes. Horizontal axis shows time from 31st January to 05 August 2005 with two and a half weeks between graduations. Vertical axis ranges from 0 to 2494 with around 250 files separating each graduation.

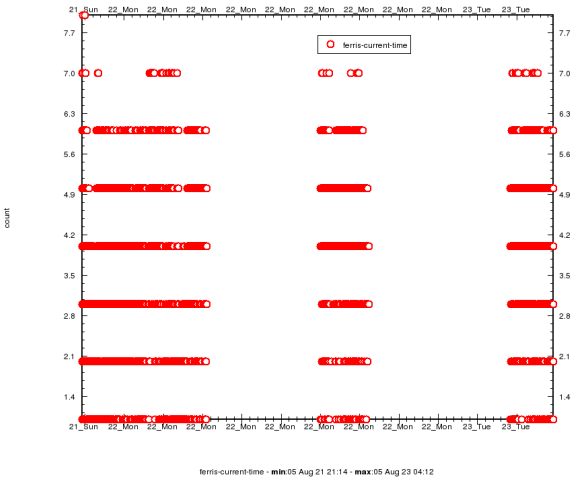

Figure 6.3: Plot of the ferris-current-time EA of 201,759 files from /usr/share/.

valuable information about when indexing sessions were held as shown in Figure 6.3. Looking at Figure 6.3 one would be lead to create three formal attributes, one for each of the major groupings of matching files.

The width EA presents the width in pixels of a file. For many systems this EA must be handled explicitly because a small number of extremely large images can easily distort simpler methods of splitting the value distribution. In this case two images stand out, sgvol.png from the kdemultimedia package is 7,140 pixels wide and sunclock_huge_earthmap.jpg from the sunclock_huge_earthmap package is 10,800 pixels wide. All other image files in the findex are below 3,500 pixels wide. The width plot is shown in Figure 6.4. One can also start from the megapixel count of images as more generalized overview of image size to generate formal attributes. As can be seen from Figure 6.5 there is a similar trend as to the width plot.

Two concept lattices were generated using the width EA and the modification time for the examples 201,759 files. Both scale the width and modification time metadata using 7 formal attributes for each. The first one shown in Figure 6.6 uses the standard linear ranges to generate the formal attributes. Shown in Figure 6.7 is the concept lattice that results when dividing the input ranges based on value density.

Because the formal attributes in Figure 6.7 are data driven there is much more interaction between concepts in the resulting concept lattice.

For some numeric EA such as: group-owner-number, user-owner-number the user may wish to explicitly specify the range for each formal attribute based on knowledge of the computer system. For example, on many Linux installations the numeric user and group identifiers above 500 are used for normal user accounts.

6.2.4 Natural Groups of Time

Some applications such as browsing digital picture collections will offer a natural clustering around a time attribute. Consider a filesystem tree with digital pictures taken

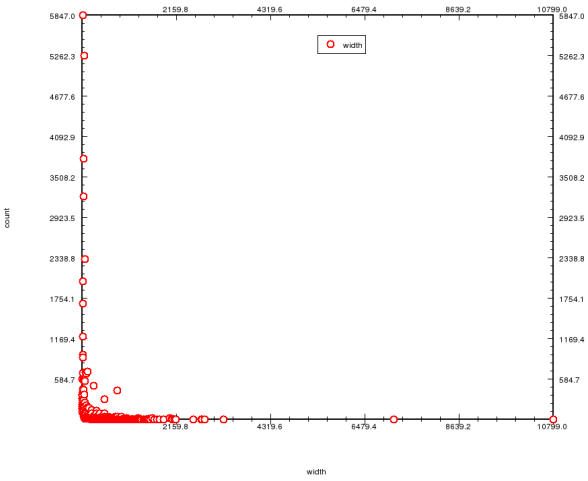

Figure 6.4: Plot of the the width of image files from /usr/share/.

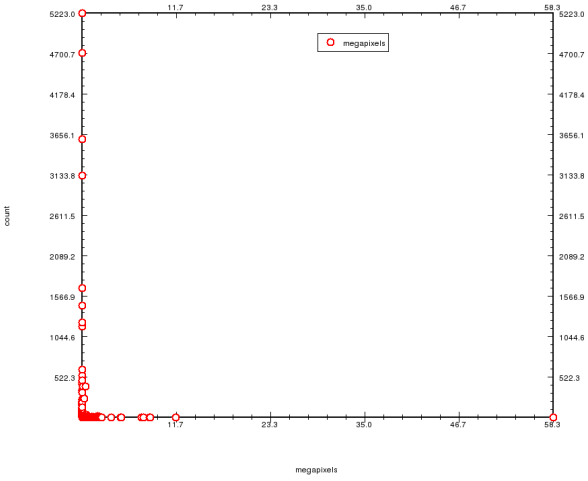

Figure 6.5: Fewer plot points but a similar overall trend to the width plot. Plot of the megapixels of image files from /usr/share/.

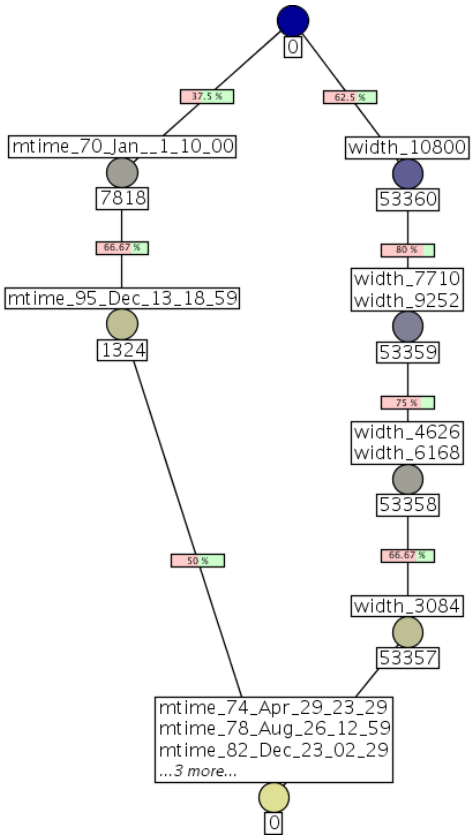

Figure 6.6: 7 formal attributes for each of mtime (modification time) and width using a standard linear range division. Concepts are represented as circles. Labels above a concept show the formal attribute which is introduced by that concept and labels below a concept show the number of filesystem objects which match that concept or one of its refinements. An introduced formal attribute is a formal attribute for which this concept is the highest one in the lattice with that attribute. Thus, where a concept has an introduced formal attribute all concepts reachable transitively downwards will also have this formal attribute.

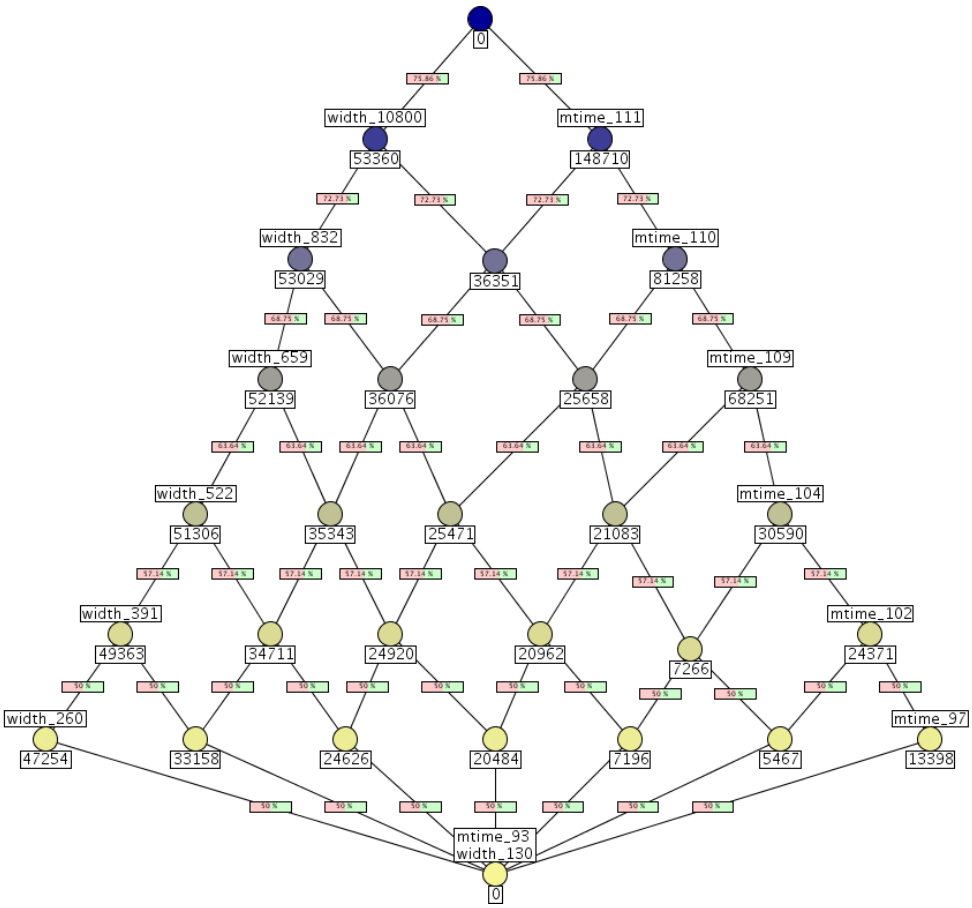

Figure 6.7: 7 formal attributes for each of mtime (modification time) and width. Formal attributes are generated based on the density of the input metadata.

on three different journeys; January 2006, September 2006 and February 2007. For the purposes of making the example concise we shall only have two other formal attributes – 1 and 2. The object set will be $O = \{a, b, c, d, e, f, g, h, i\}$. Three of the objects will be from each trip and the other two attributes will be assigned randomly to the objects of each trip.

If the time groupings are discerned by looking for the numeric groupings as outlined in Section 6.2.3 then the concept lattice will appear like Figure 6.8. This is less than optimal because at times like from concept "1" the user is forced to select from refinements with many temporal options. Three different time options doesn't seem so bad but considering that there are only three time attributes in the whole of the formal context such refinements will not scale well to larger image collections with more natural groupings.

If a simple nominal scale is used when performing Formal Concept Analysis on this data then the concept lattice will look something like Figure 6.9. The options from the "1" concept are much more sane than with the concept lattice in Figure 6.8. As one moves down the lattice they add more time restrictions and can nominate to loosen those restrictions and move closer to the lattice top.

A combination of both the numeric group detection and ordinal scaling is shown in Figure 6.10. In this case the user can either lock the time attribute at the Jan06Trip time frame or decide to place further restrictions on the time and move down the lattice that way.

Note that although the lattice style of Figure 6.10 has more concepts and overall appears more congested the refinements offered at each concept are very intuitive. For applications where the concept lattice is not designed to be directly interpreted as a whole but instead navigated using a context and the parent-child relation the clean semantics of navigation of the time dimension in Figure 6.10 will outweigh the additional (mostly unseen) complexity of the concept lattice as a whole.

6.2.5 SELinux

Security Enhanced Linux (SELinux) [77] allows modern Linux installations to offer Mandatory Access Control (MAC) as well as the more familiar Discretionary Access Control (DAC). Under DAC, file access is granted or denied based on the user running an application. Assume that my user account has read and write access to my thesis and read access to my music collection. Under DAC a music player has the ability to overwrite my thesis just as xemacs can read my music files. With MAC programs can be allowed access only to the files that are required for them to operate. For example, using MAC I can disallow my media player access to any files relating to my thesis. It should be noted that my user account will still be the owner of my music files and thesis though the media player run by me will be disallowed access to some files owned by my user account.

SELinux information which is attached to files is comprised of three datum: the identity, role and type. The identity is a SELinux user account, the role is ignored for files and the type is the primary security attribute for making authorization decisions.

In a minimal installation one has an SELinux user_u account which is shared by all users in a similar category and a system_u for daemon usage. The example 201,759 files have three identities: root, user_u and system_u. Also there are nine types: etc_t,

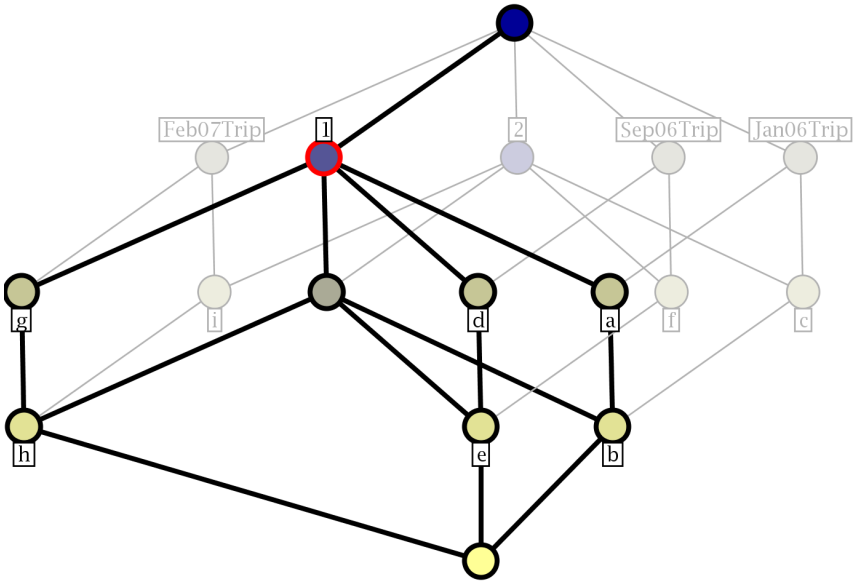

Figure 6.8: Numeric group based scaling on time. The concept "1" is selected offering four direct refinements one including the "2" attribute and the other three offering a more restrictive time attribute.

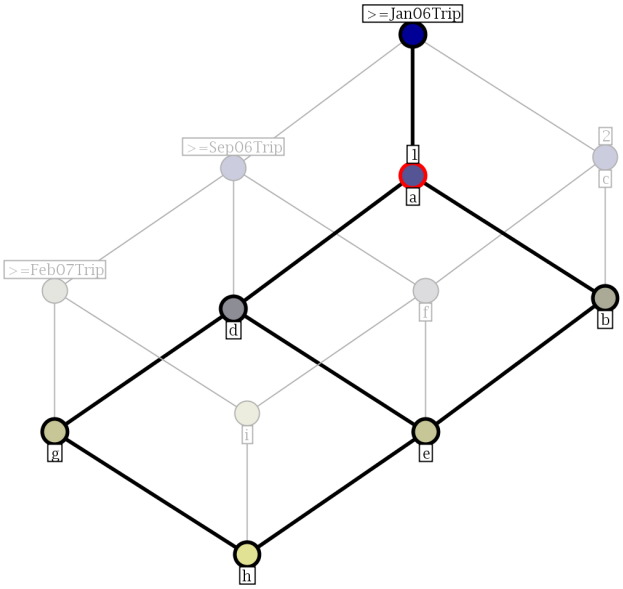

Figure 6.9: Nominal scaling on time. The concept "1" is selected offering direct refinements to include "2" or a more restrictive time attribute.

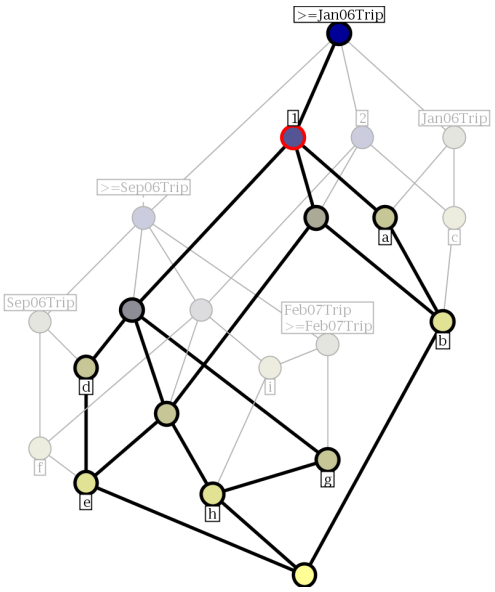

Figure 6.10: Combination of nominal scaling with ordinal scaling applied to group the nominal time attributes. The concept “1” is selected offering direct refinements to include “2” as well as the option of locking the time at Jan06Trip or adding a further restriction to the time attribute to be equal or latter than September 2006.

fonts_t, httpd_sys_content_t, lib_t, locale_t, man_t, shlib_t, snmpd_var_lib_t, and usr_t.

A very high level view of how access is granted or denied follows, for details see [77]. Each process also has an associated SELinux context. Access is granted or denied based on the SELinux context of the process and the file together with the operation requested to be performed. As such viewing only the SELinux context for files provides an incomplete picture of overall security policy.

Using the SELinux type and identity of the example 201,759 files the concept lattice shown in Figure 6.11 is generated. The concept 11 in the middle of the bottom row shows that user_u identity is only active for 3 fonts_t typed files. Many of the links to the lower concepts are caused by the root and system identities being mutually exclusive while the system identity combines with every attribute that the root identity does.

6.2.6 Structuring with URLs

Often the URL for a file is comprised of metadata forming an ad-hoc hierarchy [12]. To put such metadata into the URL itself requires arbitrary decisions about the ordering of such metadata. For example, one must decide if they are to first classify a file by its conference name or conference year in the URL `.../adcs/2005/martin-eklund/...`.

The `scale-urls` client creates a formal context from the directory components in URLs. Additional processing can be performed to present a more attractive and useful concept lattice. For example, heuristics can be used to strip version information from directory names such as `java-1.5.0`.

Wordnet [37] is also employed to explicitly allow the generation of formal concepts for hypernyms of common directory names. Explicit hypernym concepts are generated as follows: each URL is divided into its directory name components with a number of the rightmost path components being dropped (normally just one, the filename), each directory name is then stripped of version information and added to the set D. Many such preprocessed directory names $d \in D$ are then candidates for use with Wordnet. If d can be found in Wordnet then its synonym set X is found and all the hypernyms for X are collected. When two or more d have the same synonym set X then the hypernyms for X are emitted into the formal context with a prefix "wn_" to denote that they have been mechanically added.

The semantic commonalities between directory names are made more explicit in the output concept lattice using the Wordnet hypernym associations. Another advantage is that because the "wn_" attributes effectively form the join of many existing attributes they are closer to the top of the concept lattice. Assuming that the concept lattice is being read in the usual way from top to bottom having these wordnet concepts toward the top of the concept lattice can greatly assist understanding and navigation to the desired concept.

Shown in Figure 6.12 is an example concept lattice that does not take advantage of wordnet. In this case there are no hints towards the top of the lattice that papers and features are available. This semantic information is only offered in the second layer of the lattice from the top. Figure 6.13 includes some augmentation from wordnet, in particular the wn_article formal attribute has been added based on the paper and feature attributes. The new wordnet wn_article attribute is one of the first specializations offered

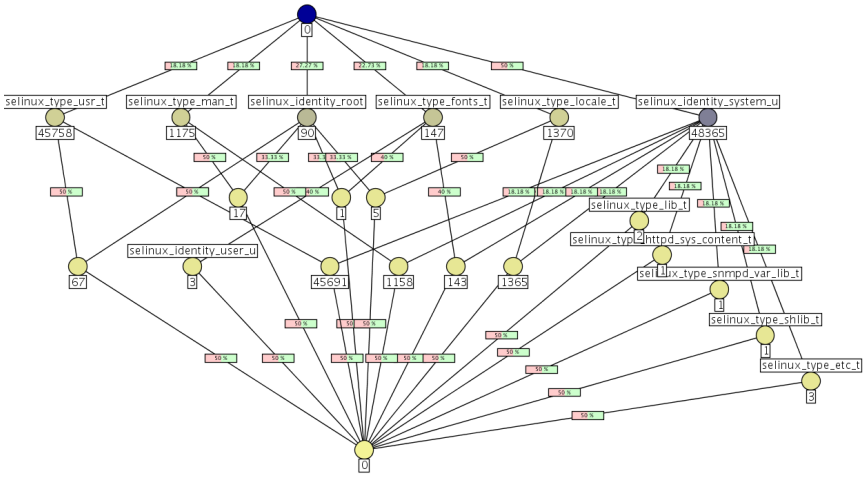

Figure 6.11: Concept lattice for SELinux type and identity of files in /usr/share/ on a Fedora Core 4 Linux installation. The Hasse diagram is arranged with three major sections; direct parents of the root are in a row across the top, refinements of SELinux_identity_system_u are down the right side with combinations of the top row in the middle and left of the diagram.

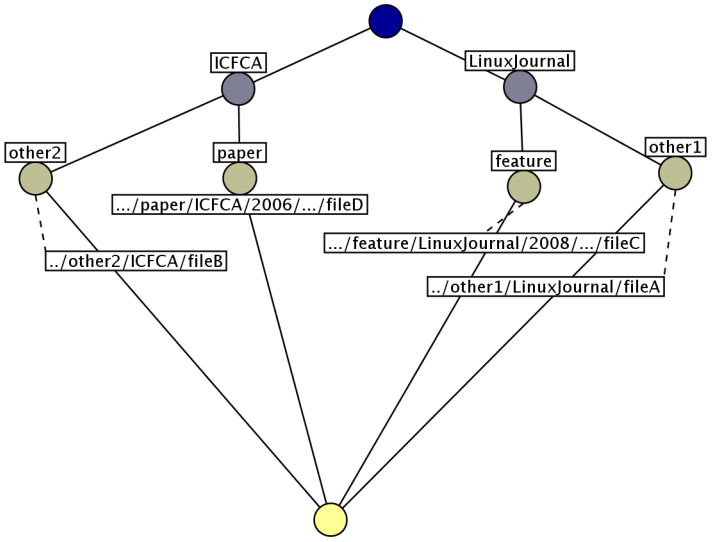

Figure 6.12: Example lattice with no wordnet augmentation.

when reading the lattice top down and a user who is seeking written articles would have a navigation path offered without having to dig into the lattice.

For the example findex of a filesystem from a standard Fedora install, dropping only the final directory component (i.e., filename) from each URL the wordnet scheme above generated 403 new "wn_" attributes. An example of one such attribute is wn_article which is equal to `feature` $\vee$ paper.

Shown in Figure 6.14 is a small example concept lattice where the two files of interest are both masked from the top concept by attributes which one does not immediately consider related to the child concept. Consider that both the concepts labeled with $a1$ and $a2$ may have many more child concepts than those shown. The introduction of a semantically more general attribute form wordnet in Figure 6.15 may allow the user greater ease in locating their desired concept.

Unfortunately the use of wordnet is not a free lunch as can be seen in Figure 6.15, the use of the term "feature" is more likely to be one of the homonyms that is not related to articles but rather in the sense that describes a characteristic of something.

As in any venture into human language processing there are issues when automatic processing is used and the machine performing the task does not understand the semantics of the information is it processing.

Modern file systems support Extended Attributes (EAs) which allows arbitrary key-value data to be attached to files and directories. Additional APIs have been provided for both UNIX[1] and Microsoft Windows[2] operating systems for associating a key-value pair with a file. With EAs an application can store meta data about a file with the file itself on disk. One can abstractly consider the EAs for a file f as a subdirectory which can not contain directories but only files containing meta data about the file f. In this light, a directory can be seen as a many valued formal context. Assume that a directory with content G forms the objects g using its contents. A file $g \in G$ may have

[1] EA and ACL for Linux website, http://acl.bestbits.at/
[2] http://linux-ntfs.sf.net/ntfs/attributes/ea_information.html

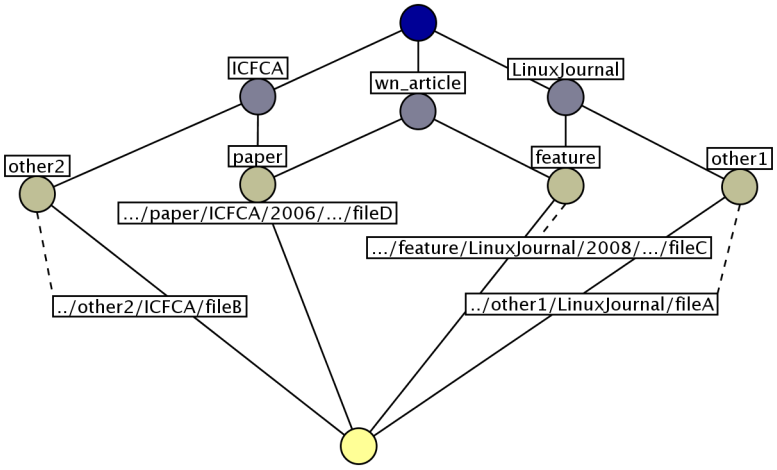

Figure 6.13: Example lattice using wordnet augmentation, notice how the wn_article concept is the common parent of both feature and paper and is also closer to the top of the lattice than either hyponym.

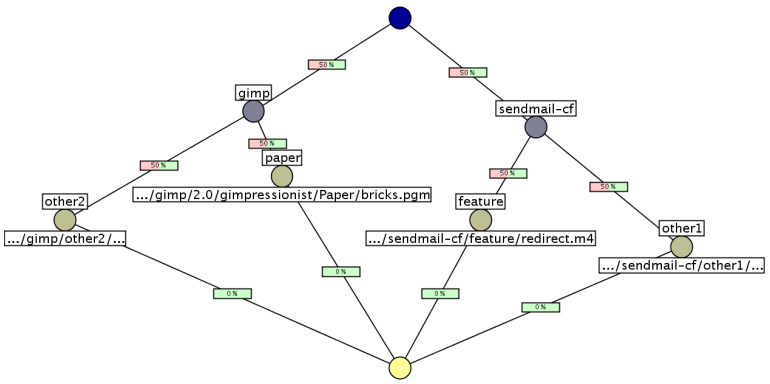

Figure 6.14: Second example lattice with no wordnet augmentation drawn from an findex of a standard Fedora install.

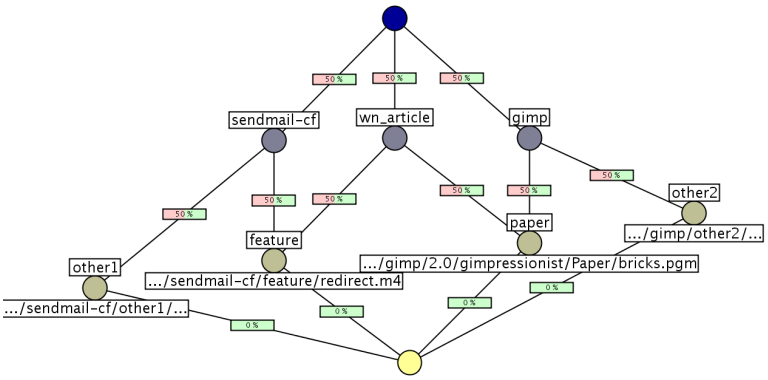

Figure 6.15: Example lattice using wordnet augmentation, notice how the wn_article concept is the common parent of both feature and paper and is also closer to the top of the lattice than either hyponym. Unfortunately in this case the "feature" drawn from the redirect.m4 file is a homonym and not the sense that is related to articles.

an EA $m \in M$ with value $w \in W$ where m and w are strings. Then w is functionally dependent on g and m.

6.2.7 Context Based Navigation

Consider a relatively small dataset of 2,000 photographs with metadata describing their location both semantically and quantitatively. Semantic metadata is captured by associating place names with each file. For example, marking an image as being of the Königsalle in Düsseldorf. Quantitative metadata comes from both the metadata automatically recorded by a digital camera or film processing machine and also indirectly from the semantic metadata in the form of longitude and latitude information. For example, once an image is semantically tagged as of the Königsalle the digital longitude and latitude range for this site can be inferred by the Semantic File System.

The combined metadata for these 2,000 photographs generates 92 formal attributes giving rise to a modest iceberg concept lattice of 168 concepts. An overview of the structure of this concept lattice is shown in Figure 6.16. Although there are relatively few concepts the direct visualization of the whole lattice becomes difficult.

While the concept lattice shown in Figure 6.16 contains less than 200 concepts the techniques discussed in previous chapters allow for in the order of 3,000 concepts to be present in the concept lattice. The layout of concepts in large concept lattices is the subject of ongoing research in the community [97].

For larger concept lattices drawn from a Semantic File System the set of concepts and covering relation can be found as detailed in previous chapters. Using a specially augmented concept based search and navigation interface can enable simpler interaction with such large lattices. In the following discussion the interface is concerned primarily with showing the user the current concept and the concepts which are nearby.

Shown in Figure 6.17 is the top concept from the full concept lattice of Figure 6.16. There are six lower covers and no upper covers for the top concept. Selecting the Germany lower cover modifies the current concept and results in the interface shown in Figure 6.18. At this stage we have a single upper cover pointing in to the Germany center node. There are six lower covers from Germany. Two of the lower covers offer a refinement in the time dimension, one in the Flash (lighting) used to create the images and three are geographical refinements on the current concept.

Most digital pictures contain information about the exact time they were taken. The time related refinements are generated by searching for clustering in this photo generation time metadata as discussed in Section 6.2.3. This is to avoid flooding the lattice with many formal attributes which would be generated by a linear numeric scale on the photo generation time. The use of clustering on photo generation time allows the system to present time refinements in the lattice in a manner which will usually have semantics for the user. For example, the digital pictures generated in two distinct holidays will cluster together to generate the two formal attributes shown as refinements in Figure 6.18.

Further navigation brings the user to Figure 6.19. The current concept node is at the Königsalle in Düsseldorf. From the current concept there are two time refinements and a Flash refinement. As Düsseldorf geographically contains the Königsalle there is only the one geographical generalization offered.

An overall perspective is lost when using a concept based interface as shown in

Figures 6.17, 6.18 and 6.19. Due to the loss of information about the lattice as a whole, the ability to jump to a concept of interest is also lost. To help the user to quickly search the lattice an interface exists to allow the user to select formal attributes that are of interest and jump directly to a concept that contains those attributes.

A concept lattice can also be navigated as a Semantic File System. This is shown in Figure 6.20. Navigation of a concept lattice as a tree has been presented in the past [23] as well as presentation as virtual filesystem [82].

As well as presenting a concept lattice as a virtual filesystem some new virtual directories were created to allow a system that is both responsive in an interactive environment and allows semantically valid batch mode processing. For example, the user might wish to see a sample of files which match the top concept but will normally not be interested in manually scanning through half a million files matching the extent of the top concept.

As shown in Figure 6.20, four virtual directories exist at each concept shown through the filesystem interface to allow inspection of the extent and contingent of a concept. The `-all` virtual directory shows all files in the extent of a concept. The `-self` virtual directory shows all the files in the contingent of the concept. These directory names were chosen as they use a lexicon which is more accessible to users who are not familiar with Formal Concept Analysis. The `-all-s` and `-self-s` directories operate similarly to the directories without the `-s` postfix but only show a sample of the extent and contingent respectively. This allows the user to get an idea of the files contained in a concept without a longer wait that would be required as the size of the list increases.

The sample virtual directories work with the spatial indexing detailed in Chapter 4 to allow the Formal Concept Analysis system to be interactively explored without substantial delays.

The all virtual directories allow all of the files from the contingent of a concept to be systematically operated on from a command line tool. For example, copying all the digital pictures of the Königsalle that were taken on a given holiday to a removable hard disk.

6.3 Conclusion

The application of Formal Concept Analysis typically involves a "Domain expert" who creates the scales which are applicable to a domain or a data set. When applying Formal Concept Analysis to a desktop Semantic File System such a domain expert will not be available and that role will have to be pushed into the computer system or the system user. Some predefined scales can be shipped with the Formal Concept Analysis system but the most interesting application of Formal Concept Analysis to a user's Semantic File System will involve metadata which was not known to the designer of the computer system and thus scales can not be shipped for this metadata with the system.

In some cases such as scaling time values an approach combining both nominal and ordinal scaling in a single concept lattice can generate refinements which may be more relevant to the system user. This is shown in Section 6.2.4.

For large concept lattices, presentation either through a context interface or as a virtual filesystem may allow the user both an easier route to using an Formal Concept Analysis based interface and also the ability to navigate larger concept lattices which would not be directly understandable (Section 6.2.7).

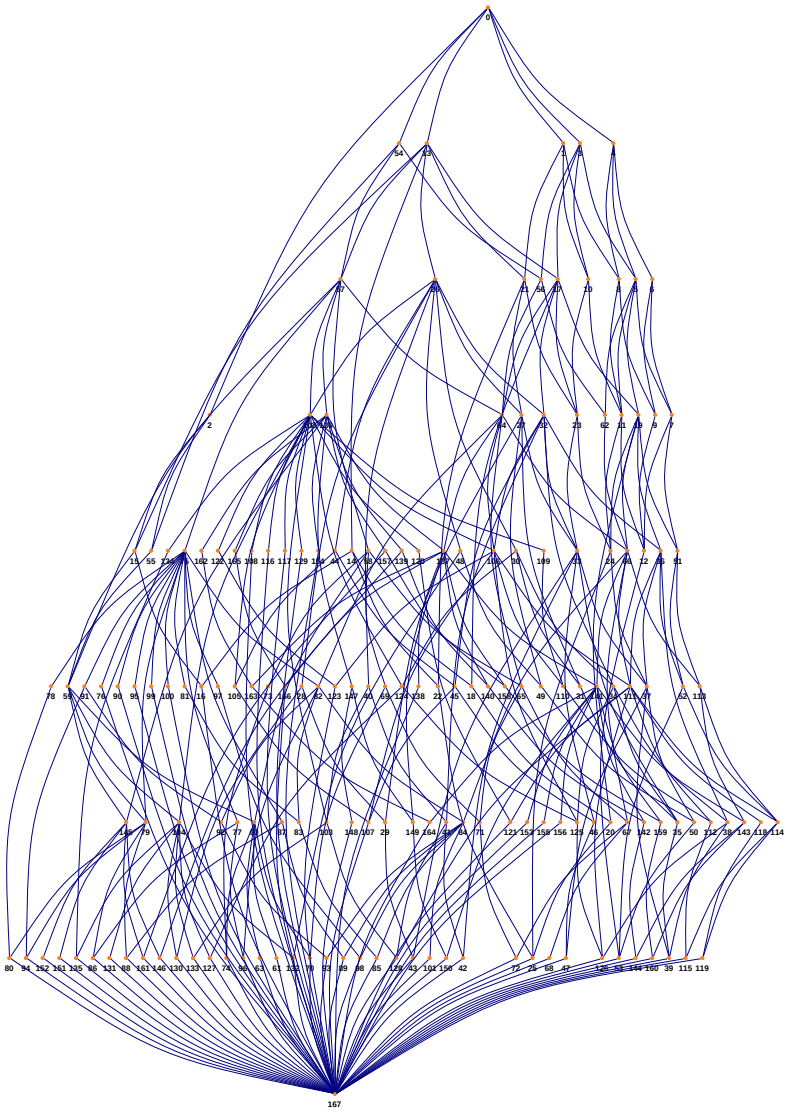

Figure 6.16: An iceberg concept lattice showing the 168 Concepts of some geographically tagged digital photographs. The formal context has 92 attributes and 2000 objects.

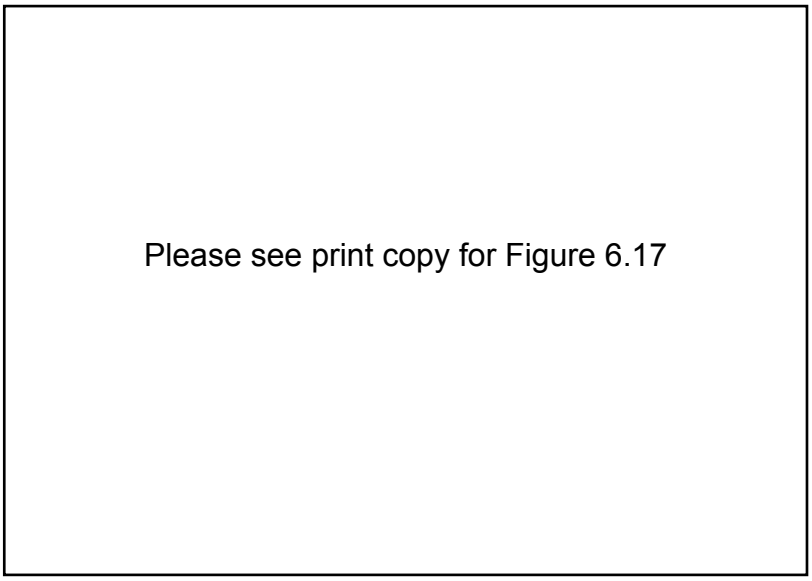

Figure 6.17: A context based viewer showing the top node of the iceberg lattice from Figure 6.16.

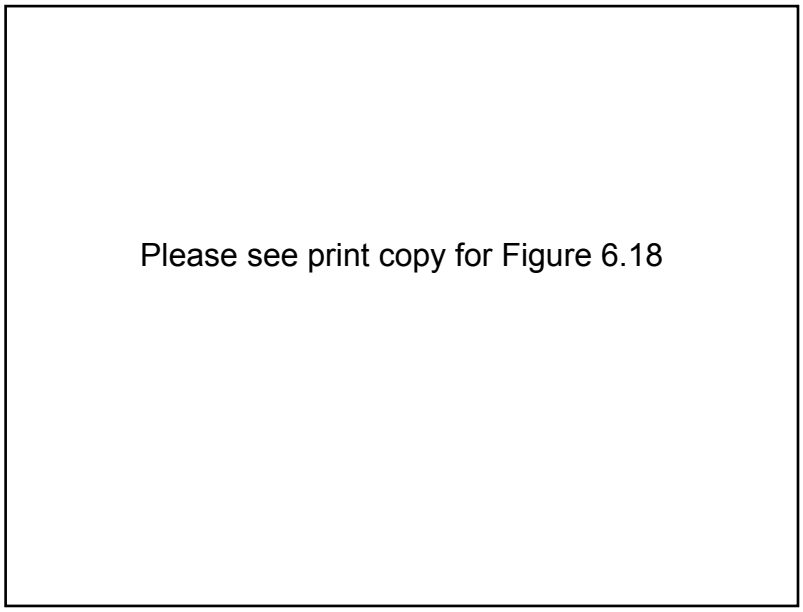

Figure 6.18: A context based viewer showing the Germany of the iceberg lattice from Figure 6.16. This figure is obtained by selecting the Germany node in Figure 6.16. There is only the top node as an upper cover and six lower covers. Three of the lower covers are for geographical refinement and have their arrows marked with an "X". Two lower covers are for Time refinements and have their arrow marked with a "T". There is an exposure related refinement marked with an "F".

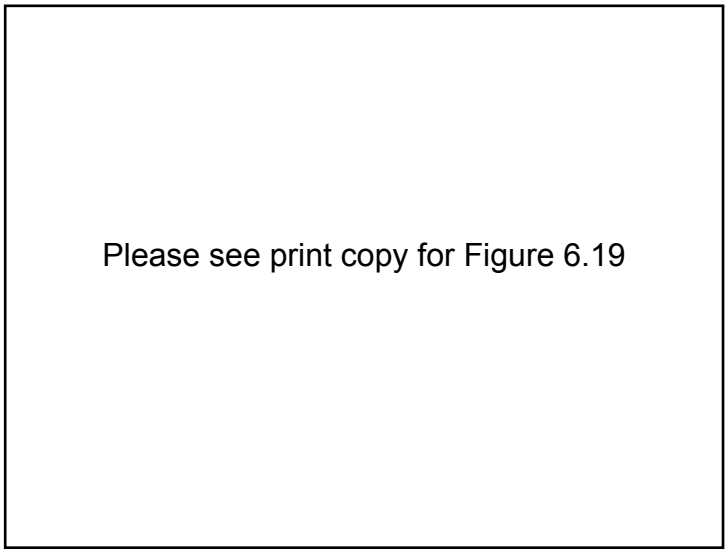

Figure 6.19: The iceberg concept lattice from Figure 6.16 centered on the Königsalle in Düsseldorf. There are two time related refinements and a single exposure refinement (marked with a "F" on the arrow).

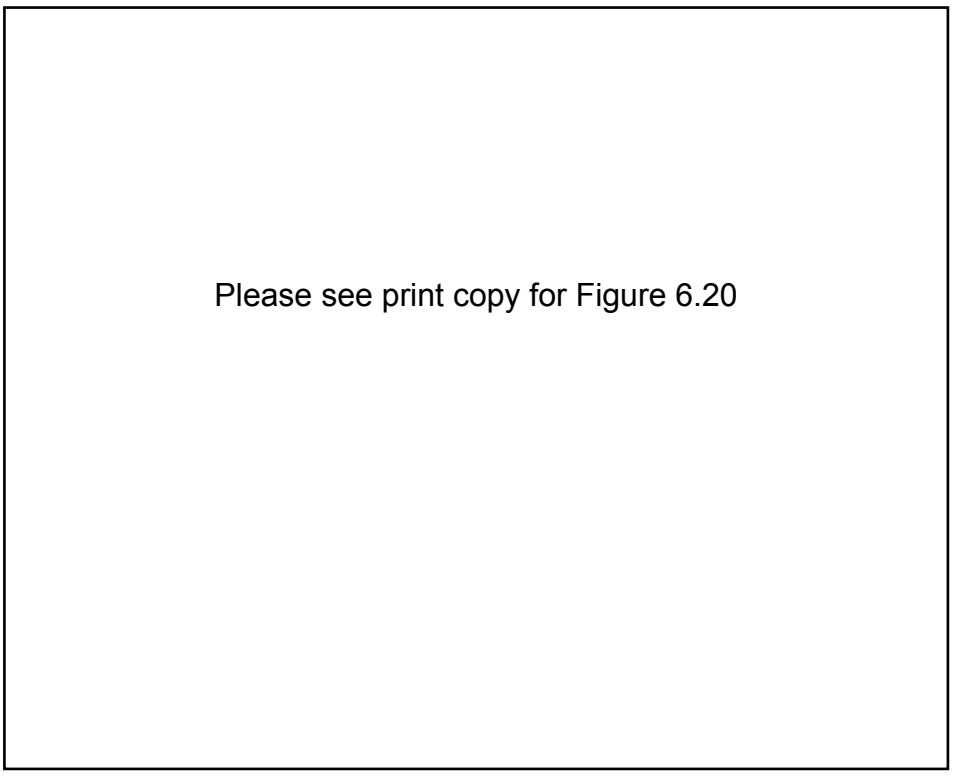

Figure 6.20: Navigating the concept lattice from Figure 6.16 as a Semantic File System. The Düsseldorf concept is selected and lower covers are shown in both the file browser on the right and as children in the left tree list. The four virtual directories `all` and `self` allow the user to view the extent or contingent at any concept. The size of the extent of each concept is shown explicitly in the left tree as an EA.

This chapter presents ongoing research into how to best allow a user to interact with Formal Concept Analysis where the user might hold great domain knowledge but be largely ignorant of the terminology and theory of Formal Concept Analysis itself.

Chapter 7

System Security and Access Control

7.1 Introduction

This chapter investigates computer security, how lattices have been used in the past for analysing system security and finally how Formal Concept Analysis can be used to assist the field.

The research in this chapter is not intended to vindicate that lattices provide a better representation than the more traditional usage of graphs to investigate system security. Rather the idea is that Formal Concept Analysis can be used to provide a better understanding and auditing of system security and access control policy.

An introduction to access control is given in Section 7.2 with differentiation between a Discretionary (Section 7.2.1) and Mandatory (Section 7.2.2) Access Control systems. This explanatory material is followed by a look at SELinux and how it relates to discretionary and mandatory access control in Section 7.3. Prior work on applying lattices to access control and information flow is given in Section 7.4. This is followed by an investigation into how Formal Concept Analysis can be applied to an SELinux system in Section 7.5 both in a holistic manner in Section 7.5.1 and from a fixed security type in Section 7.5.2. Single user access control is considered in Section 7.5.3.

In order to limit the scope of the research while remaining relevant to current systems focus is primarily given to the SELinux environment. SELinux [77, 34] was chosen for its ability to handle multiple security policies allowing it to scale from handling a personal computer to a networked military installation [60]. SELinux is a modern access control system and security enforcement environment which is both widely available and well maintained making the research both immediately applicable to systems and relevant for future years.

Although SELinux was originally deployed to protect operating system and filesystem objects it has recently been added to relational databases such as SE-PostgreSQL and print servers such as CUPS as well as the XWindow system. The research presented in this chapter remains largely applicable to these more recent applications of SELinux.

As computer security is a large topic in itself the presentation is limited to an overview of only the concepts which are necessary for the research. Many details are omitted or only presented at a conceptual level due to space constraints. Interested readers should see [77] for more details.

This section makes reference to "security context" and "filesystem object" in a sense that is not related to Formal Contexts and Formal Objects.

7.2 An Introduction to Access Control

In order to discuss certain points it is necessary at times to mention how a Linux system operates at a fairly low level. For example, in SELinux understanding the typical manner in which new processes are created is essential to understanding how the security context of executing applications can be modified. As such the discussion in this section at times must mention some lower level operations performed by the operating system in order to discuss how security information is maintained for a running system.

7.2.1 Discretionary Access Control

Access control systems have been integrated the operating systems for decades. The traditional access control system on POSIX operating systems such as early Linux and BSD Unix has been Discretionary Access Control (DAC).

A DAC system enforces security based on the notion of "users". A user in the system normally has a one to one correspondence with a human who is accessing the system. For example, the person Ben Martin might log onto a computer as the user "ben". A group notion generally exists which can have a list of one or more users who are members of the group. In modern POSIX systems a user can be a member of one or more groups simultaneously.

In the following discussion, files, special files and directories will be referred to as filesystem objects. POSIX operating systems normally expose many operating system abstractions as special files. For example, network sockets and device files representing physical disks and audio cards are normally available as special files. This enables an access control system on filesystem objects to also handle access control for operations on these operating system abstractions.

A SELinux system can also enforce security policy on operating system calls. Given that many operating system primitives are also available as filesystem objects, for the purpose of this discussion such enforcement of operating system call access can be though of as a request for a special operation on a filesystem object.

Each filesystem object in a DAC system is owned by both a user and a group as well as having permissions for "other" users. A set of permission bits is assigned to each filesystem object allowing read, write and execute access to the filesystem object. If the user attempting read access to a file is the same as the owner of the file then the read bit is checked to see if that operation should proceed. If the read bit is set then the operation is permitted otherwise it is denied. If the user is not the owner but they are in the group that owns the file then the group-read permission bit is tested. If the user is neither the owner of the file or in the group that owns the file the "other" read bit is tested to see if people with no association with the file should be granted read access.

For example see Table. 7.1. Assuming that the user ben is the owner of the file `/tmp/junk.txt` then he can read and write the file because he is the owner of the file and these protection bits are set. If the group owning `/tmp/junk.txt` is `tmptxt` and the peter user is a member of this group then he can read the file but is not permitted to make any changes to it. If the `/root/secret.txt` is owned by the user root and is in the rootonly group, and we assume that the only member of the rootonly group is the root user then only the root user can read or modify the secret.txt file. This is because there is only the root user in the rootonly group and there is no permission to read or

write the secret.txt file granted for "other" users.

Filename	user			group			other		
	read	write	exe	read	write	exe	read	write	exe
/tmp/junk.txt	×	×		×					
/root/secret.txt	×	×							
.../bash.man	×						×		

Table 7.1: Filesystem objects in a DAC system have read, write and execute bits assigned for the owner of the file, those in the owning group and "other" people with no association as either the owner or being in the owning group.

The group notion as described here fulfills the requirements of NIST Flat Role Based Access Control (RBAC) [93]. Flat RBAC is the basic role base access control outlined by NIST. Further enhancements to RBAC as defined in [93] include support for hierarchies of roles, separation of duties allowing the security policy to be more holistic and protected and symmetric RBAC. With symmetric RBAC the permission-role assignments must be more efficiently reviewable than traditional POSIX systems allow. It is important to note that the group model described for DAC does fulfill the least requirements to be considered a role based access control system by NIST [93].

The DAC system seems to offer some protection from one user reading or modifying another user's filesystem objects. For example, files which are in Peter Eklund's home directory would be owned by the user "peter" and likely belong to the group "peter". These files would not allow any access to other people. Given this policy, attempts made by the ben user to access files in peter's home directory would fail.

As the above discussion has highlighted there is a lot of security policy that is contingent on user prudence. If peter makes a file in a directory that ben has access to, such as `/tmp`, which has the read bit set for other users then ben will be able to read that file. This sort of information leak can occur due to unintended programming errors allowing wider access to files than is strictly necessary.

One example of where such a security model can fail was given in [94] where peter might trust another user on the system, say Larry. Peter allows a file `NOT-FOR-BEN` to be read by Larry under the assumption that Larry will not leak this information to the ben user. Larry has permission to read the file `NOT-FOR-BEN` and ben does not. However Larry can create a new file `COPY-OF-NOT-FOR-BEN` in a directory that ben has access to and allow ben permission read `COPY-OF-NOT-FOR-BEN`. Larry can set the permission for `COPY-OF-NOT-FOR-BEN` to whatever he likes because he created that file and will be the owner of it. Then as long as Larry keeps the contents of `COPY-OF-NOT-FOR-BEN` reasonably up to date with changes from `NOT-FOR-BEN` then ben can read the `NOT-FOR-BEN` file through this proxy. Such a loophole is allowed because Larry has digression over who is allowed to access his filesystem objects.

Such information leaks can also be performed by Larry without him being complicit with ben. These leaks are explicitly created by using trojan horse applications. A trojan horse application is a modified executable which appears to perform one task but is also designed to allow unwanted information disclosure.

Under a DAC system when the user executes an application then the application

runs with the user's login credentials (it assumes their user and group for the purposes of filesystem object access control). This means that any file that the user has read access too will be available to the application to read. If the user runs a trojan horse audio player the operating system will allow that application to read a text file called `secret.txt` in their home directory as well as any audio tracks that the user has stored. From the access control point of view if the user owns these files and their read bit is set then the operating system allows both a text editor and a possible trojan horse audio player the ability to read `secret.txt`.

Once a file has been read there are many methods for a trojan horse application to communicate that possibly confidential data to unauthorized parties. These methods extend beyond the use of the filesystem as described above in the `COPY-OF-NOT-FOR-BEN` example. For example, the trojan horse application can allocate and free memory or perform some other task that effects the system in a manner that ben can detect. Such communications are referred to as Covert Channels and the details and mitigation of such practices are beyond the scope of this thesis.

7.2.2 Mandatory Access Control

The above issues called for Mandatory Access Control (MAC) [15] systems to be created. In a MAC system each file has a security context assigned to it and applications that the user executes are given certain permissions to certain types of files. In the above case the audio playing trojan would have access to an audio track but since it has no business reading `secret.txt` then permission will be denied for files of that security context.

On an SELinux MAC system each file has a security context associated. Each application that is executed also has a security context associated. A set of security context transitions are defined which allow the security context to change when an application is executed.

In this case the user might want a different security context associated with the text editor, for example allowing them to edit the `secret.txt` file but not to allow any access to their multimedia files to the text editor and to deny all network and global clipboard access to the text editor. Such denials make it harder for sensitive information to flow out of the text editor application to unintended readers. A type transition can be setup to automatically make the text editor change from the security context that was associated with the user's shell (that executed it) into a new security context tailored for the text editor.

For example, when the user logs in to the system they might be at a command line interface presented by a shell program. The shell program will have an associated security context which might be quite liberal, allowing the user to list many of the directories which are available to them. When the user executes a text editor an operating system `fork()` and `exec()` sequence is executed. The `fork()` function creates a copy of the user's shell process and the `exec()` function replaces the shell executable in the new process with the text editor application. The effective result of this is that a child process of the shell is created which is executing the text editor. When an `exec()` function is executed the SELinux system might modify the security context that the new application will be running under. This security context change is called a type transition in SELinux. In this example, the system is setup to automatically perform a

type transition from the shell security context to the text editor security context when an `exec()` function executes the text editor binary and the current security context is the shell security context.

The security context of the application, the operation requested and security context of the filesystem object determine whether an operation is allowed in SELinux. The decision as to what operations are to be permitted is referred to as Type Enforcement in SELinux. This makes the type transitions of applications a very important component in an SELinux system. Part of a security policy is shown conceptually in Table 7.2. In practice such a representation is not used because it would make for a very large sparse matrix consuming more resources than necessary.

File security context	Application security context					
	bash_t			audioplayer_t		
	read	write	getattr	read	write	getattr
user_home_t	×	×	×			
user_home_audio_cfg_t	×	×	×	×	×	×
audio_collection_t	×		×	×		×
text_t	×	×	×			
secure_text_t			×			

Table 7.2: The security context of a running process, the operation requested and the security context of the file determine if an operation will be permitted in SELinux.

For the security policy shown in Table 7.2, the user's shell will have a great deal of power in the system. If the `secret.txt` file is in the secure_text_t security context then the shell itself will not be given access to read or modify the file. A different application security context would have to be in use to read the `secret.txt` file. This might be achieved by running a special text editor which has been audited and will transition into a secure_text_editor_t application security context. Files which are in the audio_collection_t security context can be read by an application in the audioplayer_t security context. Notice that an application in the audioplayer_t context can not access files in the user_home_t or secure_text_t contexts. This is expected as audio players have no business reading files which are not either their configuration files (user_home_audio_cfg_t) or audio files for playback (audio_collection_t).

By restricting the audioplayer_t application security context and defining a security policy where audio playback applications automatically run in the audioplayer_t application security context the user's home directory is protected from violation by the audio playing applications.

Security policies for MAC systems generally aim to provide applications with access to only the filesystem objects that they require in order for the user to perform their task.

A great deal of attention is paid to the ability for information that is contained in a filesystem object of one security context to move to a filesystem object of another security context in a MAC security policy. For example we would not like any application to be given the ability to read a secure_text_t security context file and also write a audio_collection_t type file. Such a permission would effectively allow the application to read a secure_text_t file and write it's contents to a new file with the security

context of audio_collection_t. This sort of information flow might not be considered acceptable because we would like secure_text_t files to have a higher confidentiality than audio_collection_t files.

7.3 SELinux – DAC and MAC

Access control in an SELinux environment uses both DAC and MAC to enforce system security. Firstly, checks are performed using traditional DAC. If access is not granted for an operation based on these checks then it will fail immediately. If access is granted by DAC then the MAC system is tested to see if it will allow the operation as well. The operation will only succeed if both DAC and MAC allow it to be performed. This is in the spirit of the original MAC proposal [15].

Only a few users are allowed to modify the SELinux security policy of a computer. The security policy itself will define which users will be able to make adjustments to the system's security policy and what adjustments they are allowed. For typical users on the system the security policy will be immutable and the user will have absolutely no discretion over the policy.

There is no strict distinction between the information contained in the security context of an application and the security context of a filesystem object in SELinux. This allows one to consider both of these security contexts as subsets of a single set of all security contexts in the system.

7.4 Prior work on Lattices and Access Control

Prior work on using lattices with access control is detailed in [94]. This work centered on information flows and protecting the confidentiality and integrity of filesystem objects in a MAC system. The confidentiality of a file is concerned with the disclosure of information whereas integrity is concerned with the modification of information. This section presents the work both in it's original notation and from a Formal Concept Analysis point of view.

As defined in [28, 94] an information flow policy is a triple $\{SC, \rightarrow, \oplus\}$, where SC is a set of security classes, $\rightarrow$ is a binary can-flow relation, and $\oplus$ is a binary class combining function. The $\rightarrow$ function defines which information can flow from one security class to another. For example, restricted $\rightarrow$ top-secret means that information can flow into top-secret from the restricted class. The combining function determines that security class will be given to information derived from two security classes.

It was shown by Denning [28] that under certain assumptions the above triple will define a finite lattice. These assumptions are that:

(1) SC is finite

(2) the $\rightarrow$ can-flow operation is a partial order on SC

(3) SC has a lower bound with respect to $\rightarrow$

(4) $\oplus$ is a totally defined least upper bound operator.

When the can-flow relation is a total order we can obtain the famous classification used by the government and military. Many cinema fans will be familiar with this very

Confidentiality	Top Secret	Secret	Classified	Unclassified
Top Secret	×	×	×	×
Secret		×	×	×
Classified			×	×
Unclassified				×

Figure 7.1: The can-flow relation for a four class Confidentiality information flow policy. Information flows are permitted from the attribute label to any object listed under that label. For example, information in the Secret class can-flow to the Secret and Top-Secret class.

simple confidentiality order. This assigns four classes from the most confidential class Top-Secret, through Secret, Classified to Unclassified. The can-flow function is shown in Figure. 7.1 with the resulting Concept Lattice in Figure. 7.2. In the literature, information flow lattices are typically presented with more restrictive states at the top and permitted information flows heading upwards. The can-flow relation has been presented in the manner more familiar to the Formal Concept Analysis reader, with information flows leading down the lattice and more specialized states toward the bottom of the lattice. In this case the $\oplus$ would need to be the greatest lower bound. For example, Top-Secret $\oplus$ Classified = Top-Secret. The Formal Concept Analysis reader can see that the Top-Secret class can have information from the Secret class and this is an acceptable condition.

A partially ordered classification lattice example might assign two categories: medical (M) and payroll (P). To fulfill the Denning properties there would be a security class Top T for the join of M and P which would contain both types of information such that $M \oplus P = T$. There would also be a bottom lattice node B which contains neither M or P information. The concept lattice is shown in Figure 7.3. Notice that the introduced top and bottom node are reversed in the concept lattice. As information can-flow from both medical and payroll into the top node it serves as the bottom node in the concept lattice.

Although the core of the simple medical and payroll example is equivalent to a nominal scale in Formal Concept Analysis the security classes may combine a discrete arbitrary number of times before meeting the top node.

As noted in [94] totally and partially ordered lattices of the above types may be combined into a single lattice. The classic example being the combination of the four level confidentiality lattice (Top-Secret etc) with a classification showing which types of information are contained in the document. For example, now allowing a user who has Top-Secret clearance for physics documents to access a file rated as Top-Secret and also containing information about biotechnology.

For further examples of security policy and examples of lattices which are the product of one or more sub lattices see [57, 48, 103].

The notion of Dominance is defined in the security literature as $A \geq B$ (read as A dominates B) if and only if $B \rightarrow A$.

The presentation here follows the Bell-LaPadula (BLP) model presented in [94]. For a given running application for a filesystem object we assume that the security context is obtained using the λ function. Setting aside functions such as filesystem

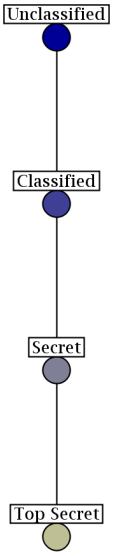

Figure 7.2: The can-flow function from Figure.7.1 as a Concept Lattice.

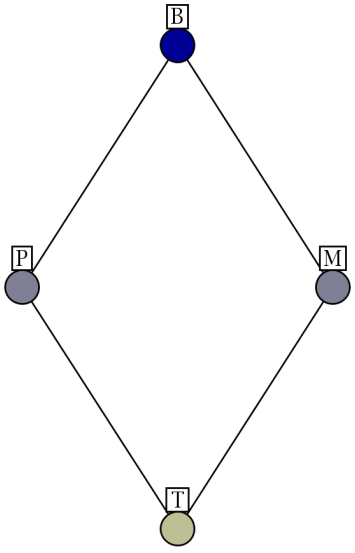

Figure 7.3: A concept lattice for the information flows of two unrelated information classes: medical and payroll data.

object creation and destruction, enforcement in the BLP is carried out such that;

(1) Simple-Security property: Application a can read filesystem object o only if $\lambda(a) \geq \lambda(o)$.

(2) $\star$-Property: Application a can write filesystem object o only if $\lambda(a) \leq \lambda(o)$.

As noted in Section. 7.3 the security context obtained using λ on both applications and filesystem objects are drawn from the same set of all security contexts.

The main purpose of the $\star$-Property is to prevent undesired information flow. An application which has Top-Secret clearance should not be able to write to a file which is Unclassified in order to prevent either explicitly malicious or unintended (trojan horse) information flows which allow information to flow to a lower clearance level.

The Bell-LaPadula model is primarily focused on preserving the confidentiality of information by explicitly denying insecure information flows. The Biba model is concerned only with protecting the integrity of filesystem objects. It has been shown that although the Biba model was initially presented to work in the opposite manner to the Bell-LaPadula model one may take the lattice dual of a Biba model and evaluate it under a system designed to enforce the Bell-LaPadula model [94]. Given that both models can be evaluated using the same enforcement machinery it is also common to build product lattices of both models to gain both confidentiality and integrity enforcement.

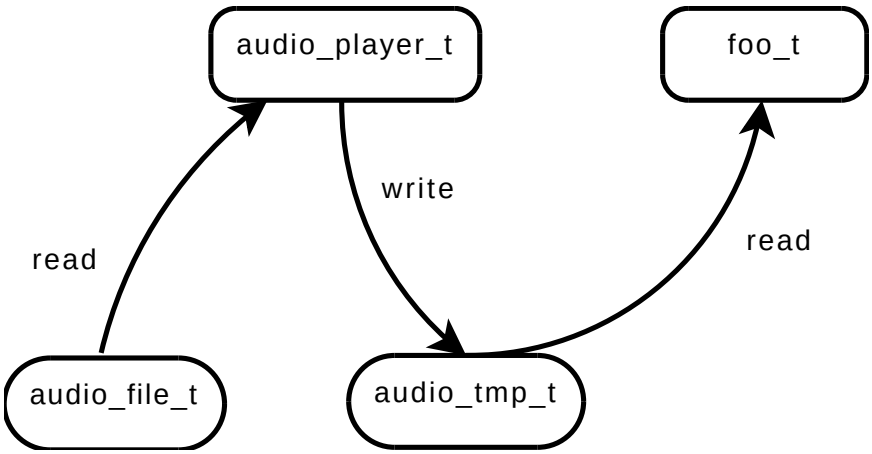

Figure 7.4: Information flow from a file in audio_file_t to an application with type foo_t.

7.5 SELinux and Formal Concept Analysis

This section describes new research performed on existing SELinux security policies using Formal Concept Analysis. The security policies used were from the Fedora 7 Linux distribution, in particular the targeted and strict policies which were both of version 2.6.4-30.fc7.

The targeted policy is designed to add as much security to a system while having little to no impact on system usability. Higher security and low impact on users are to a great degree conflicting goals. For example, to limit which files an audio playing application is allowed to read requires the system users to be aware of the security context of the files they wish to play. As such the targeted policy mainly offers protection to many system daemons where file access patterns are well known. Very little protection is offered by default to user's normal application usage in order to not effect system usability. The strict policy allows some impact of running in SELinux to be seen by regular system users in order to offer greater security.

In SELinux, types can be used as the security context for both applications and filesystem objects. Type enforcement rules describe which types can access which types and under what conditions. An example of a type enforcement rule might conceptually say that an application running as type `audio_player_t` can read a file with the type `audio_file_t`. The targeted policy has $1,671$ types with $101,051$ type enforcement rules. The strict policy has $2,064$ types and $160,493$ type enforcement rules.

Each operation permitted by a type enforcement rule might allow an information flow to be performed. Following from the above example, if `audio_player_t` can also write a file which has a type `audio_tmp_t` then information could possibly flow from `audio_file_t` to `audio_tmp_t` when any application which runs in the security context `audio_player_t` is executed. If a second application running as type `foo_t` is not permitted to read `audio_file_t` files directly but can read `audio_tmp_t` filesystem objects there is the potential of either collaborative or trojan activity allowing `foo_t` applications to read `audio_file_t` via the information flow permitted in the security policy for `audio_player_t` applications. This arrangement is shown in Figure 7.4.

In a real security policy such as the SELinux policies chosen for this research there are a multitude of operations which might be permitted or denied by a security policy. These include operations which are the more direct "overt" information flow candidates such as reading and writing files and also some operations which can be used to convey information in a slightly less direct manner such as the ability to see how large a file is or when it was modified. For example, a trojan application can easily communicate information by continually changing the size of a file to create a channel for information flow to an application which can not read the file's content but is permitted to see the file's size.

The SETools policy analysis tools include some support for performing both direct and transitive information flow analysis on security policies. Operations are assigned a weight to indicate how easily they may be used to perform an information flow. For example, in the case of writing to a file the information flow weight will be large because writing to a file is a high bandwidth manner for an application to output information. In contrast the monitoring of a file's size information is given a relatively lower weight because it is harder to exploit for information flow and is more likely to be discovered by the system administrator if an attempt is made to use it for high bandwidth information flows.

7.5.1 Holistic Transitive Information Flow

The information flow out of all of the $1,671$ types T for the targeted policy was used to generate a formal context of direct information flow $D = (T, T, I)$. The relation tIy is set where there is a outward flow from a type $t \in T$ to a type $y \in T$. As the set of objects and attributes is the same this formal context is also a directed graph. The transitive closure J of I is calculated to find how a system running that security policy might converge over time. When analysing information flows one must assume that the presence of a flow will be used by an attacker or complicit user. As such the transitive closure of all the information flows allows one to see if there are any security classes where information can not flow from one class to another. The concept lattice of the formal context (T, T, J) is shown in Figure 7.5.

The concept lattice of the transitive information flow shows that there are two major security groups with a very limited set of types that permit information flow between these groups.

The structure of the concept lattice of the transitive flow for the $2,064$ types in the strict policy is identical to the targeted policy shown in Figure 7.5. There is a change in the number of formal objects mainly in the top and second top concept for the strict policy lattice. Thus Formal Concept Analysis has shown that the two security policies do not offer any difference as far as strictly layered security is concerned [35, 36].

7.5.2 Transitive Information Flow from a Fixed Type

The password database for a computer contains very sensitive information. A filesystem object security context is used to protect this information and it is imperative that access to this security context is properly audited. The password database was originally stored in `/etc/passwd` but in modern systems the hashed password is stored in `/etc/shadow` instead. This allows many more applications to have access to the passwd

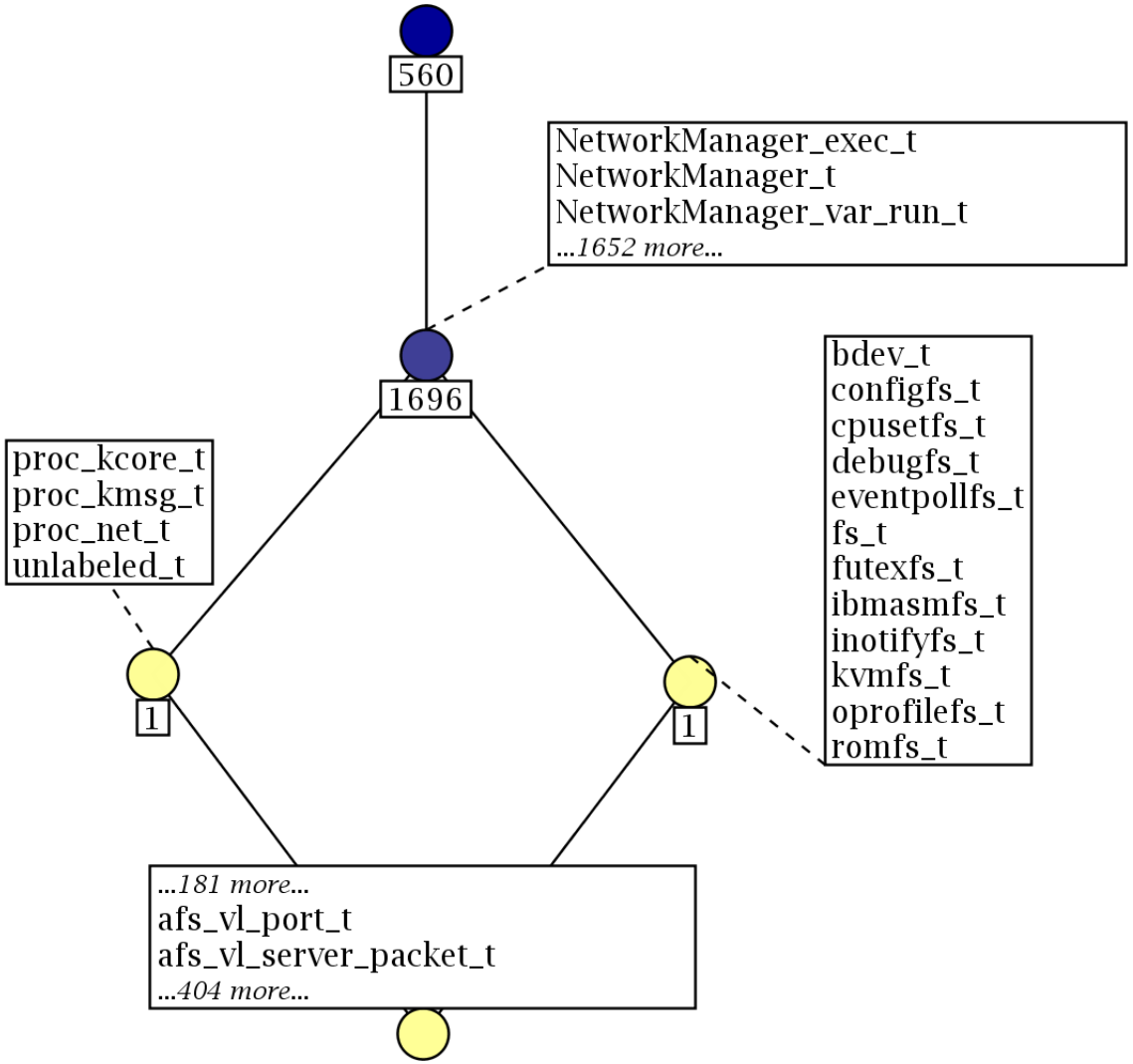

Figure 7.5: Concept Lattice of the transitive closure of all direct information flows in the SELinux targeted policy version 2.6.4-30.fc7 for Fedora 7.

file while only exposing the hashed passwords to applications which really require access to them.

The `shadow_t` is used in the strict policy for the `/etc/shadow` and `/etc/gshadow` files along with a lock file to protect these files from simultaneous access. These files are the only ones on the system that will belong to the `shadow_t` security context.

To view the information flow out of the `shadow_t` type the direct information flows from a type x to type y are obtained. First the direct information flows out of `shadow_t` form a set $B = \{y_1, y_2, ..., y_n\}$ and then for each type $y \in B$ the direct information flows out of y are obtained. This process can be repeated many times to obtain the set of all information flows that might originate at `shadow_t`.

Information flows originating directly out of `shadow_t` are referred to as degree 0 flows. Information flows that originate at `shadow_t` and have a single intermediate type are referred to as degree 1 flows. An so on, where the degree specifies the number of intermediate types that are flowed through to get to the destination type.

The direct information flows out of the `shadow_t` type up to degree 1 are shown in the concept lattice in Figure 7.6. The formal context (O, A, I) was obtained by considering the set of security types T to form both the object and attribute set. Where $T = A = O$, the incidence relation I is formed such that for $o \in O$ and $a \in A$ we have oIa iff there exists an information flow from type o to type a.

As the formal context of the direct information flows has $A = O$ the formal context can be represented as a digraph. As an information flow from a to b to c also allows by indirection a flow from a to c the transitive closure of the digraph of direct information flows is also a semantically valid formal context.

The concept lattice of the formal context of the transitive closure of degree 1 or less direct information flows is shown in Figure 7.7. As can be seen by comparing Figure 7.6 with Figure 7.7 viewing the concept lattice of the transitive closure of direct information flows produces a much simpler concept lattice.

Shown in Figure 7.8 is the concept lattice of the transitive closure for degree two or less information flows. The number of incident relations in I for the various formal contexts of the information flows shown in Figures 7.6, 7.7, and 7.8 is shown in Table 7.3.

The transitive closure can produce a much more enlightening concept lattice when considering first and second degree information flows. Considering the transitive closure of information flow gives a higher level view of how information could possibly flow through a system. As a computer system can produce millions of filesystem operations per second the formal context of the transitive closure of information flows provides a good high level view of the relations of security types.

As more degrees of information flow are added to the formal context the tendency is for the concept lattice to to converge to having only two concepts. How many degrees can be used to construct the transitive closure of the formal context before it converges to produce a two concept lattice gives an indication of how freely information can flow away from a particular security type.

The concept lattice of the transitive closure of the digraph of degree one or less information flows shown in Figure 7.7 allows the security auditor a clearer view of the information flow that the system permits from shadow_t. The shadow_t attribute is introduced in the concept at the bottom of the lattice. This means that for information to flow to the concept numbered 2, it must either first pass through any of the concepts

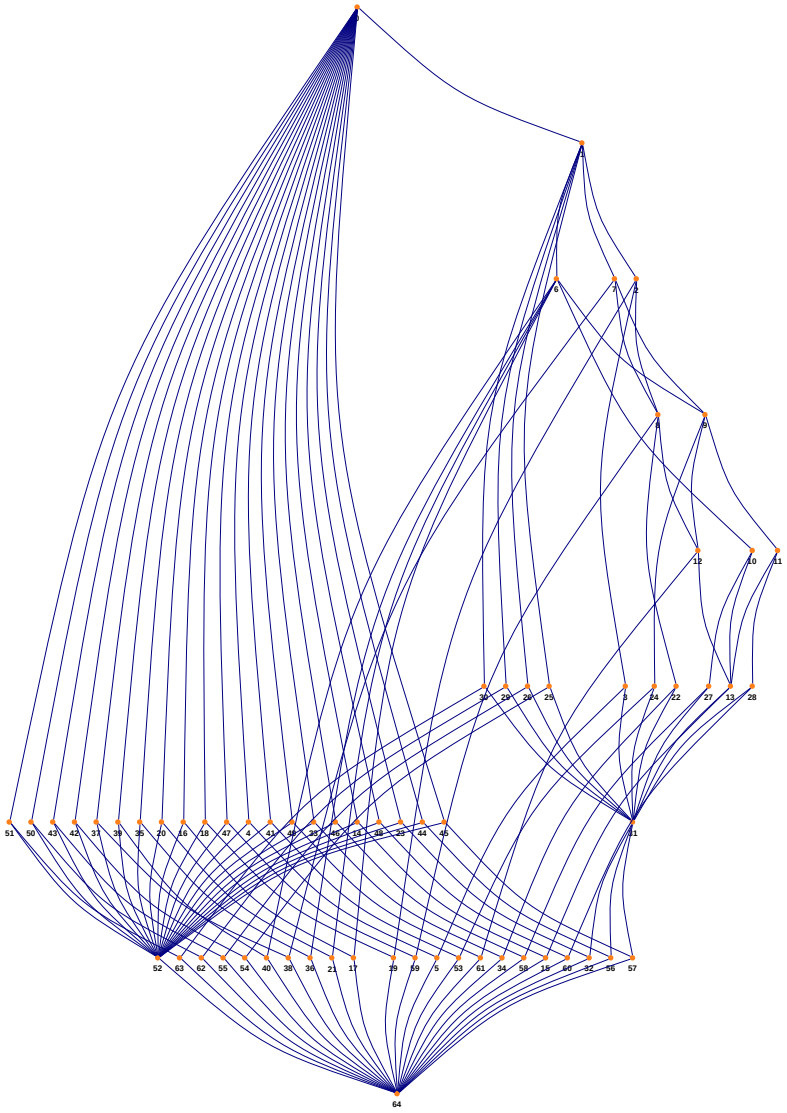

Figure 7.6: Formal Concept Analysis of transitive information flow out of the shadow_t security context. The top concept, number 0, has no attributes. Moving down and to the right, concept number 1 has the introduced attributes cardmgr_t. Concept number 2 introduces etc_t and shadow_t, concept 6 introduces var_auth_t and concept 7 introduces security_t.

3,4,5,8,9 or 13 or a degree three information flow is required. The transitive closure of information flows together with the natural clustering offered by Formal Concept Analysis work together to allow a clean concept lattice to convey information about how closely coupled two security labels are.

The concept lattices of the transitive closure of multi-degree information flows is of great interest to the auditor. As mentioned above, as the degree of information flow increases there is a tendency for the concept lattices of the transitive closure to converge to a two node lattice. This makes the auditing process iterative, the auditor is looking for a transitive closure concept lattice that still has many nodes but not to the level that they will be overwhelmed as would likely be the case if the lattice was medium sized and highly connected as in Figure 6.16.

Context	Max Degree	Transitive Closure	$\|I\|$	$\|C\|$
shadow-d1	1		3,756	65
shadow-d1-tc	1	×	12,281	18
shadow-d2	2		32,625	13,835
shadow-d2-tc	2	×	1,956,185	8
shadow-d3	3		47,594	15,187
shadow-d3-tc	3	×	2,558,020	2

Table 7.3: The number of incident relations in I for the formal contexts of direct information flows from shadow_t.

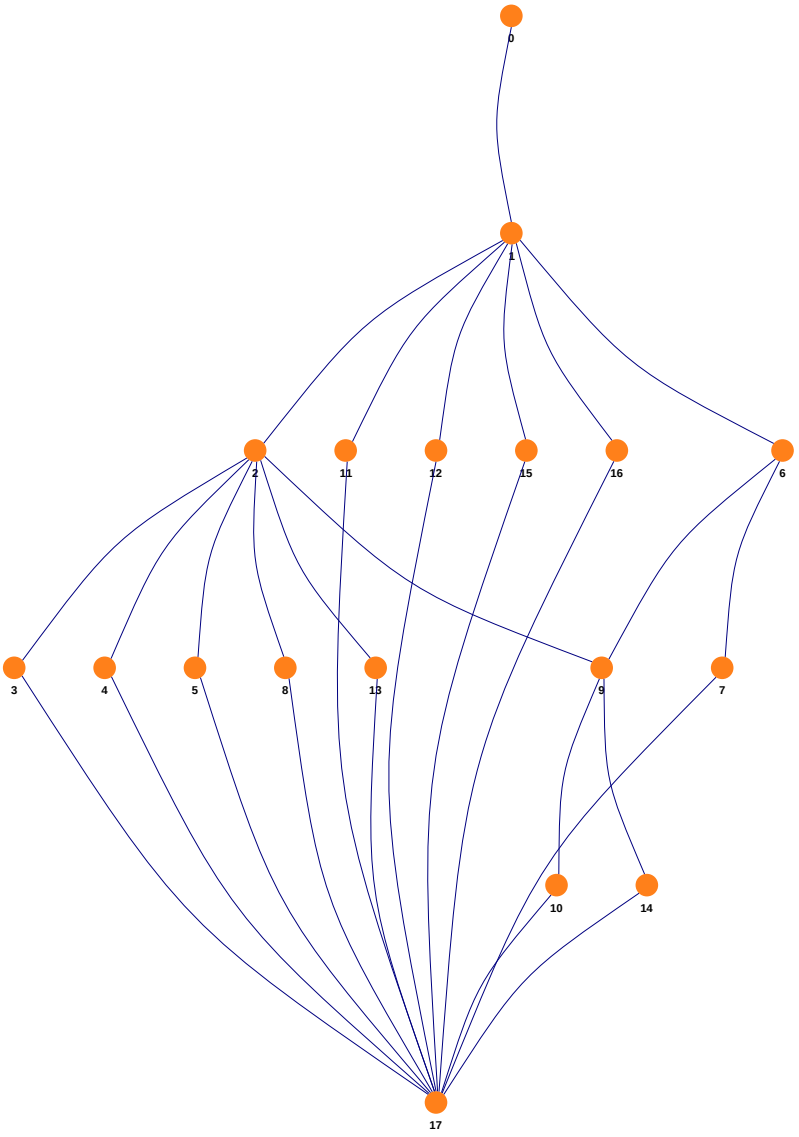

Figure 7.7: Concept lattice of the transitive closure of the digraph of degree one or less information flows from shadow_t. The shadow_t attribute is introduced by concept 17. Concept 0 has no attributes. Concept 1 introduces attributes NetworkManager_t and cardmgr_t. Concept 14 introduces attributes nscd_log_t, nscd_t, nscd_var_run_t and samba_log_t. Concept 10 introduces attributes saslauthd_t, saslauthd_tmp_t and saslauthd_var_run_t.

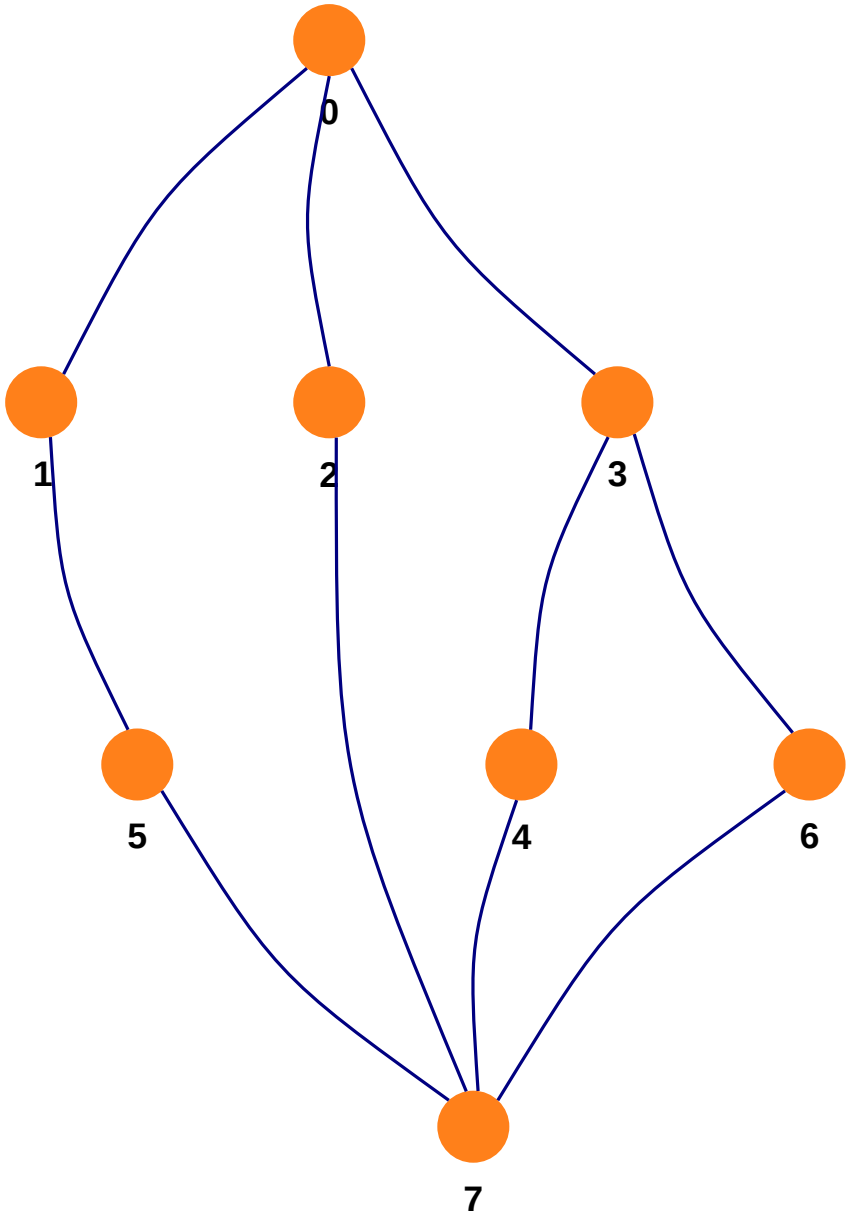

Figure 7.8: Concept lattice of the transitive closure of the digraph of degree two or less information flows from shadow_t. The shadow_t attribute appears on Concept number 7. Concept 5 introduces attributes automount_t and irqbalance_t. Concept 6 introduces insmod_t and kudzu_t. Concept 4 introduces iptables_t, staff_xserver_t, sysadm_xserver_t, user_xserver_t and xdm_xserver_t. Concept 0 has no attributes.

7.5.3 Single User Access Control

Formal Concept Analysis can be applied to single user access control either in a standard POSIX setting or on an SELinux environment. As outlined in Section 7.2.1 the security permissions offered by the system for a single user on a POSIX discretionary access control system are very limited. In contrast a mandatory access control system such as detailed in Section 7.2.2 offers many more permissions with a much finer granularity for each.

A concept lattice derived from standard POSIX protection bits is shown in Figure 7.9. There are only 9 main protection bits with a few more possibly used to indicate very simplistic user identity transitions (setuid) and file "sticky" bits where new files created in a directory will be inherited by the group owning the directory. As there are only around twelve attributes in a POSIX DAC system concept lattices will not be as interesting as for the finer grained formal attributes available on a MAC system.

Filesystem operations which can be explicitly controlled with SELinux include the following; append, create, execute, execute_no_trans, getattr, ioctl, link, lock, read, relabelto, relabelfrom, rename, setattr, unlink and write. The security labels of files in the system and the security context which processes are running under will effect which of these operations will be permitted by the system.

As shown in Table 7.2 all three of the above mentioned concepts must be taken into account on an SELinux system to decide if an action should be permitted. In concrete terms, for a process executing with a given security context, on a file with a given security context, an action (open, read etc) is either explicitly permitted or denied.

A formal context with the same structure as Table 7.2 but including all the types in the SELinux targeted policy has $|O \bigcup A| = 3,637,058$ and $|I| = 2,843,053$.

As the formal context grows extremely large when considering many types at once it can be more informative to consider again one or more single types and their explicit relation to the system as a whole. Consider the `shadow_t` and all the types that can perform any operation on this object. This subtly is different to Section 7.5.2 and Figure 7.6 in that we are considering not information flow but direct filesystem level operations and how certain process types relate to other process types.

Shown in Figure 7.11 is the concept lattice for the security types which can perform various operations on the `shadow_t` type. The concept towards the top left with `shadow_t-relabelto` as an attribute shows that many of the types require to know if files of the `shadow_t` type exist on a system. As one would expect, types which can read a `shadow_t` file are also able to invoke `getattr` on `shadow_t` files.

Shown in Figure 7.12 are all the types and attributes which can perform any operation on the SELinux `shadow_t` in the targeted policy. Some of the SELinux policy statements which construct the formal context for Figure 7.12 are shown in Figure 7.13.

The types that can access files with the `user_home_t` are shown in Figure 7.14. The interaction between the `shadow_t` and `user_home_t` is shown in Figure 7.15. The formal context for Figure 7.15 was created by considering the types that could perform various operations on either `shadow_t` or `user_home_t`.

Considering the concept lattice shown in Figure 7.15, we can see an interesting interaction between the `shadow_t` and `user_home_t` types. The interaction occurs in concept 16, this concept gets the ability to relabel the security context of files with

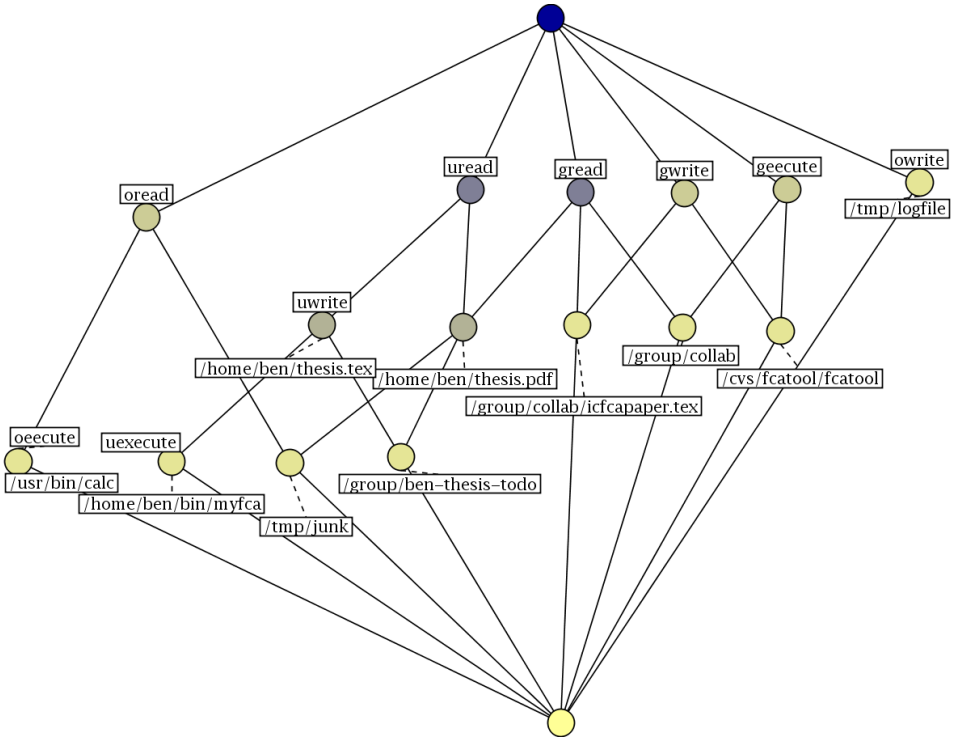

Figure 7.9: Concept lattice of standard read, write and execute POSIX protection bits for a sample of files. Three sets of read, write and execute bits exist, one for user, one for group and one for other access. For example, on the far left side of the figure the `/usr/bin/calc` program can be read and executed by everybody on the system.

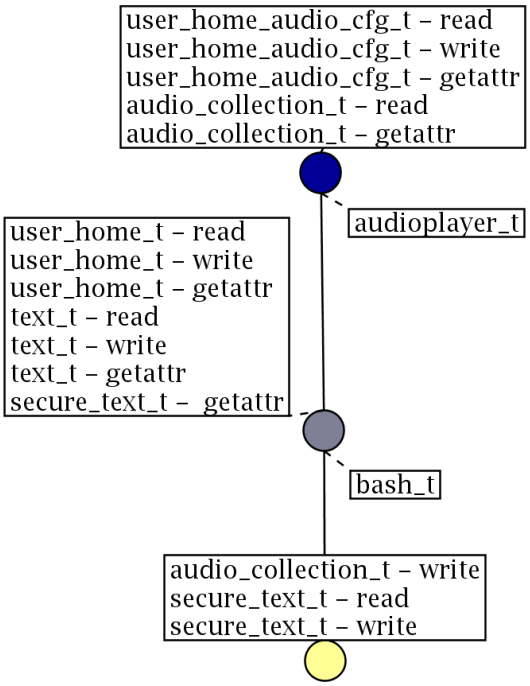

Figure 7.10: The concept lattice of the formal context from Table 7.2. It can be easily seen in Table 7.2 that audioplayer_t can be completely overlayed onto bash_t and thus bash_t is a subconcept and there are only the three concepts.

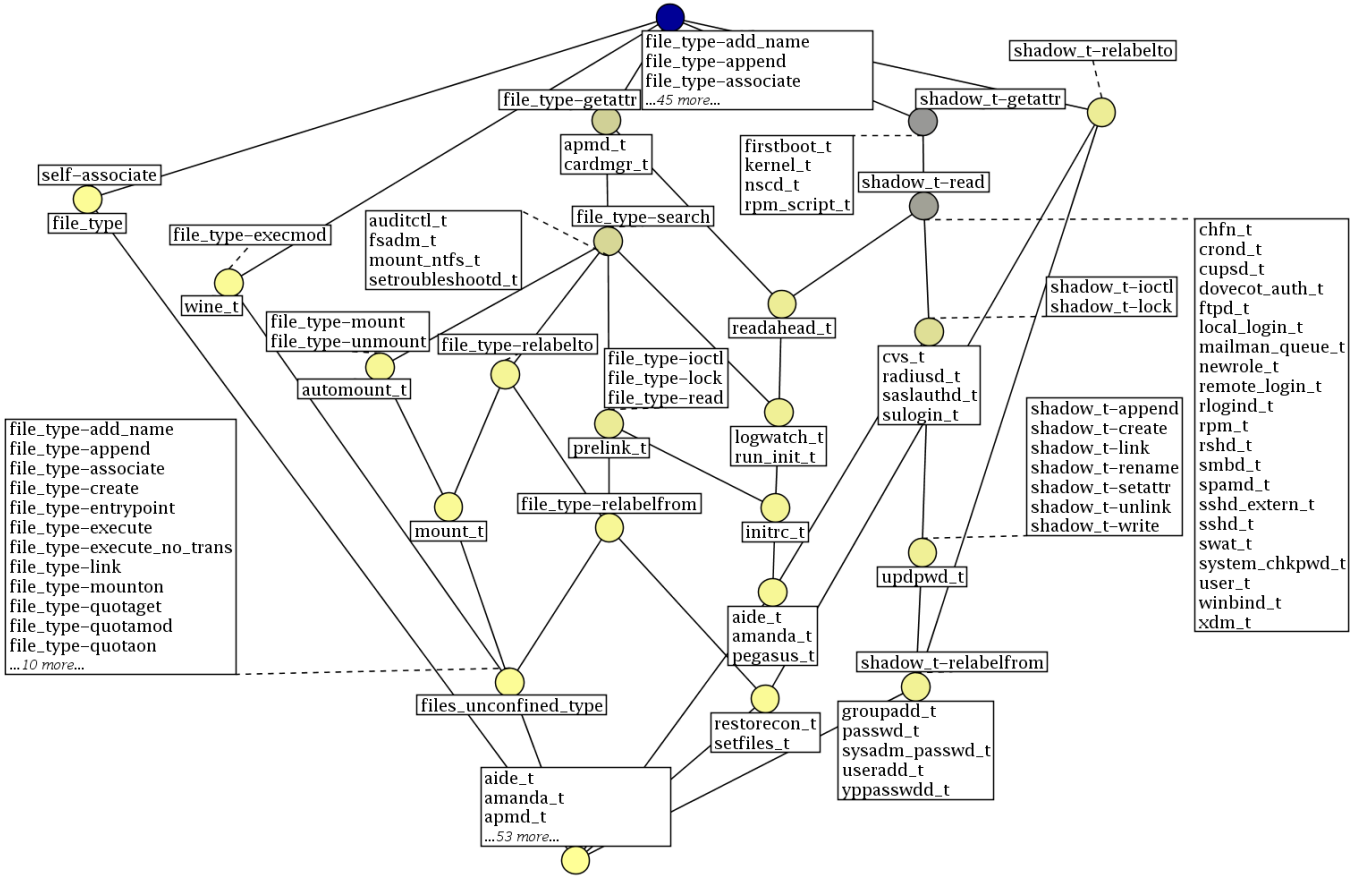

Figure 7.11: The concept lattice of all security types and attributes which can perform operations on the `shadow_t`.

the shadow_t type and also to to write to files with the `user_home_t` type. Such a combination is highly security sensitive as new files can be created and relabeled to be in this highly security sensitive type (`shadow_t`) or conversely the information in the `shadow_t` can be directly relabeled to another less secure type. The extent of concept 16 only contains useradd_t which is the security context that only sees very limited use when the system has a new user account added to it.

In this case we have discovered a concept that contains security types that have a large amount of power in the system because it can infect secure types or directly cause information to be leaked out of secure files.

It is most instructive if the concept lattice for two types allows the user to select from a list of interesting filesystem operations and have the related concepts highlighted on the concept lattice. For example, being able to see the sublattice related to the filesystem read operation to quickly pinpoint where possible interactions occur. Such a system made the highly sensitive concept found in Figure 7.15 much more readily apparent.

The targeted security policy from Fedora 7 used in this testing has $1,671$ types. Using Formal Concept Analysis directly on two types in this manner will only be feasible for a fraction of the overall policy. Such two type analysis is not intended to be holistic or provide full coverage. The aim is to detect and remove superfluous interactions between two security types.

7.6 Conclusion

This chapter has focused on the application of Formal Concept Analysis to a computer system running Mandatory Access Control in the form of SELinux. Focus has not been turned to a networked environment, complex multi role configurations or multi-level security (MLS).

The topic of Mandatory and Discretionary access control is extensive and a whole PhD thesis could be written covering improving such systems with Formal Concept Analysis.

Previous work on applying lattices to investigate information flows in MAC systems was presented in Section 7.4. The use of lattices in previous work is quite different to the application of Formal Concept Analysis to direct and transitive information flows represented as a formal context.

The core of the chapter is Section 7.5 where the application of Formal Concept Analysis to SELinux is presented. Surprisingly the strict and targeted SELinux security policy both offer the same level of transitive compartments. One would hope that using the strict policy there would be more overall nodes in the concept lattice. This would represent more states in the system which would have effectively identical security given enough time for information flows to be performed.

A single starting security label is then considered and transitive information flow reachability considered from that starting type. It is effective to consider the transitive closure of the formal context when analysing information flows from a single type in order to see a much simpler lattice which represents the security of the system under more sophisticated attacks that require one or more intermediate security labels to be traversed in order to achieve a breach.

Access control from a single user perspective is then investigated with Formal

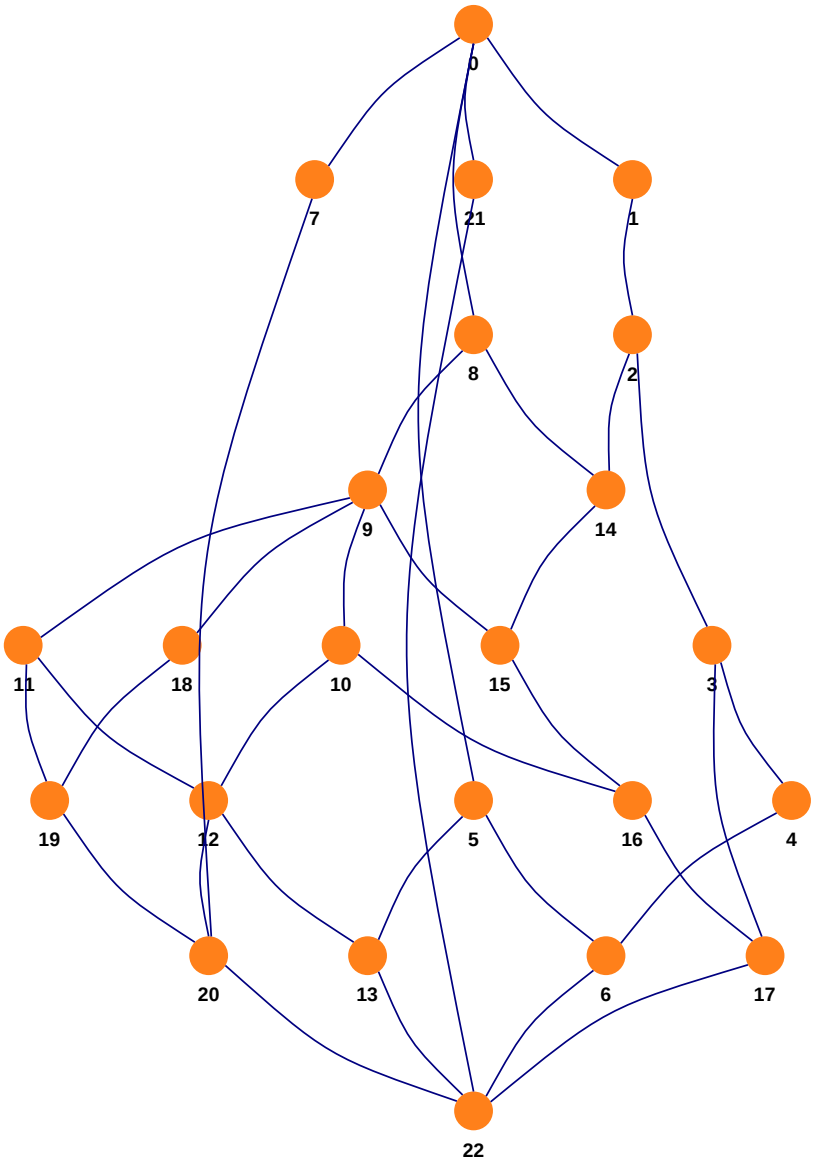

Figure 7.12: The concept lattice of all security types and attributes which can perform operations on the `shadow_t`. The attribute labels for each concept are given in Table 7.4.

Concept	Introduced Attributes
1	shadow_t-getattr
2	shadow_t-read
3	shadow_t-ioctl shadow_t-lock
4	shadow_t-append shadow_t-create shadow_t-link
5	shadow_t-relabelto
6	shadow_t-relabelfrom
7	file_type-execmod
8	file_type-getattr
9	file_type-search
10	file_type-ioctl file_type-lock file_type-read
11	file_type-relabelto
12	file_type-relabelfrom
18	file_type-mount file_type-unmount
20	file_type-add_name file_type-append file_type-associate file_type-create file_type-entrypoint file_type-execute file_type-execute_no_trans file_type-link file_type-mounton file_type-quotaget file_type-quotamod file_type-quotaon file_type-remount file_type-remove_name file_type-reparent file_type-rmdir file_type-setattr file_type-transition file_type-unlink file_type-write file_type-rename file_type-swapon
21	self-associate

Table 7.4: The attribute labels for the concept lattice of all security types and attributes which can perform operations on the `shadow_t`. The concept lattice is shown in Figure 7.12.

Concept Analysis both in a Discretionary and Mandatory Access Control system. Security states which are shared between two security labels are shown in the concept lattice in Figure 7.15. From such a concept lattice the interaction of two security labels is made available and concepts which are not necessary can be removed from the system in order to heighten security.

```
allow system_chkpwd_t shadow_t : file { read getattr } ;
allow amanda_t file_type : dir { getattr search } ;
allow files_unconfined_type file_type
      : { file chr_file } { ioctl read ... } ;
```

Figure 7.13: SELinux policy lines that are used to construct the formal context for the concept lattice in Figure 7.12. The first line means that the `system_chkpwd_t` type can access files of type `shadow_t` if performing a read or a getattr operation. All other types of access to `shadow_t` files for `system_chkpwd_t` will be blocked unless another policy rule explicitly allows it. The last rule uses an SELinux attribute `file_type` to allow `files_unconfined_type` to access all files in general with a broad range of operations. This policy rule affects the `shadow_t` because files of type `shadow_t` will also have the SELinux attribute `file_type` to indicate that they are filesystem files.

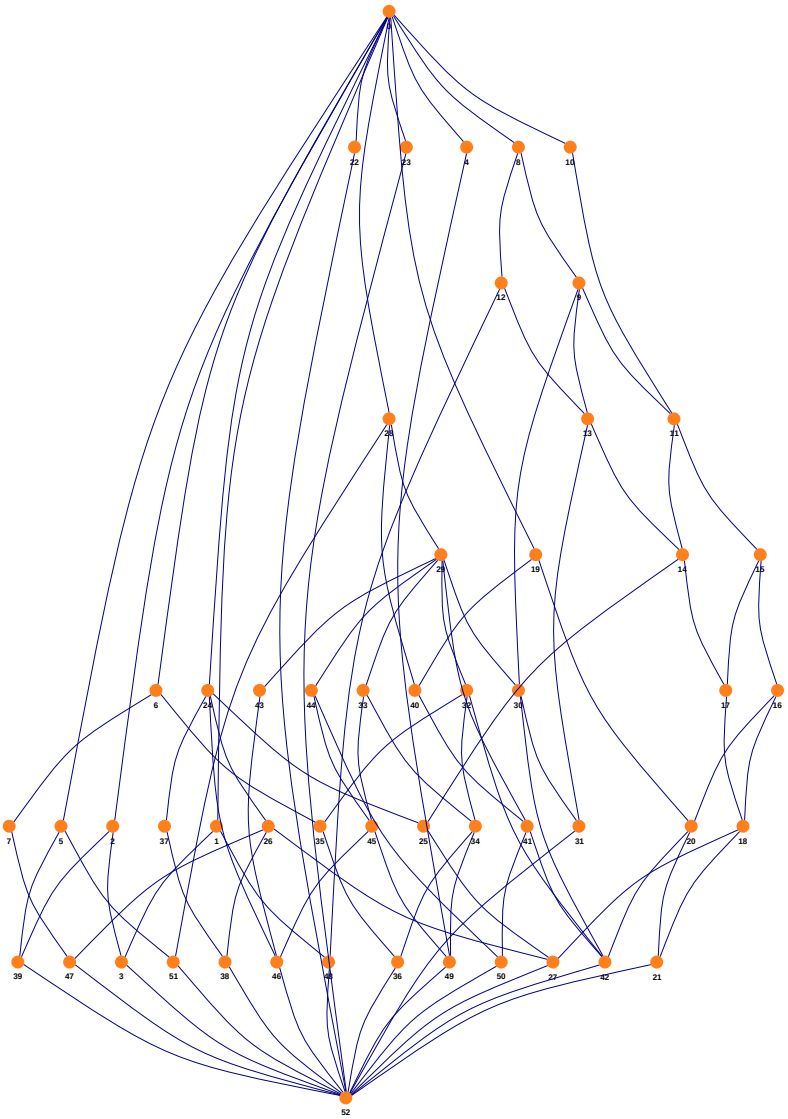

Figure 7.14: The concept lattice of all security types and attributes which can perform operations on the `user_home_t`.

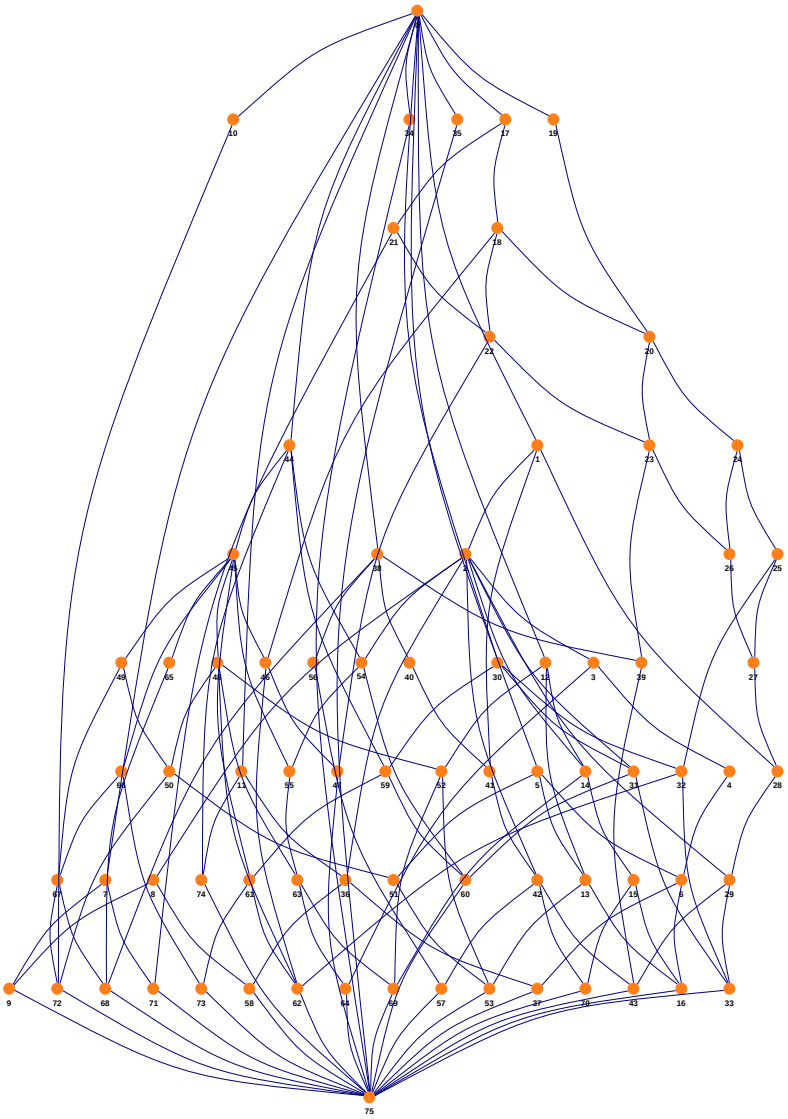

Figure 7.15: The concept lattice of all security types and attributes which can perform operations on either the `user_home_t` or `shadow_t`. Concept number 2, which is in the middle of the lattice, introduces the shadow_t-read attribute. Concept number 4, which is second from the right in the third row up from the bottom, introduces the shadow_t-append, shadow_t-create and shadow_t-link attributes. Concept 6, which is directly below concept 4, introduces the shadow_t-relabelfrom attribute. Concept 6 also inherits shadow_t-relabelto from Concept 5 which is its other direct cover. Concept 15, which is directly left of concept 6, introduces home_type-write among other attributes. Concept 16 which is the meet of concept 15 and concept 6 and just below them, introduces no attributes and has useradd_t in its extent. Concept 75, the bottom concept, introduces no attributes and has an empty extent.

Chapter 8

Advances to Semantic File Systems

8.1 Introduction

The research performed during the PhD candidature was focused on enabling Formal Concept Analysis to be applied to a large dynamic data source such as a Semantic File System makes available.

As the author has a long background in Semantic File System development there was also some research performed pertaining purely to the advancement of Semantic File Systems. Some of this work can be combined with Formal Concept Analysis though it also stands on its own merits.

The research includes integrating supervised machine learning for automatic information classification in Section 8.2. The later two sections of the chapter are related and investigate the relationship of Semantic File Systems to the XML data model and other XML technologies.

8.2 Supervised Machine Learning and Automatic File Classification

Although attaching and interacting with typed arbitrary key-value data on files is very convenient it leaves applications open to interpret the data how they choose. For this reason specific key-value pairs have been defined for semantic categorization of files on a system wide basis.

These reserved key-value pairs allow one to associate files with many tags to show what categories those files are in. The set of all tags T is maintained in a partial order $(T, \leqslant)$. The relation for $\mu, \varphi \in T$ of $\mu \leqslant \varphi$ means that μ logically `is-a` φ.

Consider the example of marking an image file: one may attach the tag "sam" to the image file. One of the parents of the tag "sam" may be "my friends" to indicate that sam is one of my friends. It follows that the image file is also of one of my friends. This is due to the partial ordering on the tags imposing transitivity on tag assignment[1] . It follows that only the most specific tags applicable to a file need be associated explicitly.

The collection of tags that are associated with a file is stored in a single reserved key-value pair per user per file. This serialization of tags is called a medallion and it is not recommended that applications read medallions directly. Each user is expected to have their own partially ordered tag set and so medallions are attached to files on a per user basis. For application access the medallion is split into many EA as shown in Table 8.1.

[1] The assignment of an tag μ to a file will also assign all the parent tags of μ to that file

attribute	value	type
medallion.600	`<xml...>`	binary
e:has-cat	false	boolean
e:has-fuzzy-cat	0.0	double
e:has-animal	false	boolean
e:has-interesting	true	boolean
e:has-burnt	true	boolean
e:has-burnt-cd5	true	boolean
e:list	has-interesting, has-burnt has-burnt-cd5	string list
e:list-ui	has-interesting has-burnt	string list
e:upset	has-interesting has-burnt has-burnt-cd5	string list

Table 8.1: A medallion is broken down into its individual tags at run-time. The "e:" prefix is a name space prefix which is abridged for presentation purposes.

The assignment of a tag to a file is called a belief and logically collects the: file, tag, time of assertion or retraction, sureness of this belief and who holds this belief. To represent the holder of the belief a portion of the tag set is used to define "personalities". There can be one belief held for each personality for a file and tag combination.

As seen in Table 8.1 the tag attachment, when attachment was done and the fuzzy belief for a tag are all exposed as EA for each file[2] . By exposing this data as simple typed information it can be added to the EA inverted file index and files can be quickly found based on tag association.

Because many beliefs about tag attachment can exist for any given file a belief resolution system was introduced to show an overall picture about a file-tag association. This is expressed in the form of a floating point number ranging from `-100.0` for full retraction to `100.0` for absolute assertion. The default belief resolution gives the "user" personality veto status and if the user doesn't have a belief it presents the mean of all beliefs for the file-tag association. An example of belief resolution is shown in Figure 8.1.

At an abstract level supervised Machine Learning [92, 53, 8, 1] involves two steps: training and classification. Of the many kinds of machine learning we are most interested in binary classification algorithms which can tell if a file is in a single category or not.

During training two lists are presented to the machine learning algorithm: a list of desired documents and a list of non desired documents. The machine learning algorithm then builds a model to use in its classification mode when it will tell if a presented document fits into the desired, undesired or unknown category.

The accuracy of the classifications that machine learning offers depends on the quality and size of its training data. Some machine learning algorithms can perform relatively better when offered limited training data.

This is very important when attempting to use machine learning for file system

[2] All EA in the "e:" namespace are derived EA that are generated from the medallion for each file. The "e:" EA are not stored separately for each file.

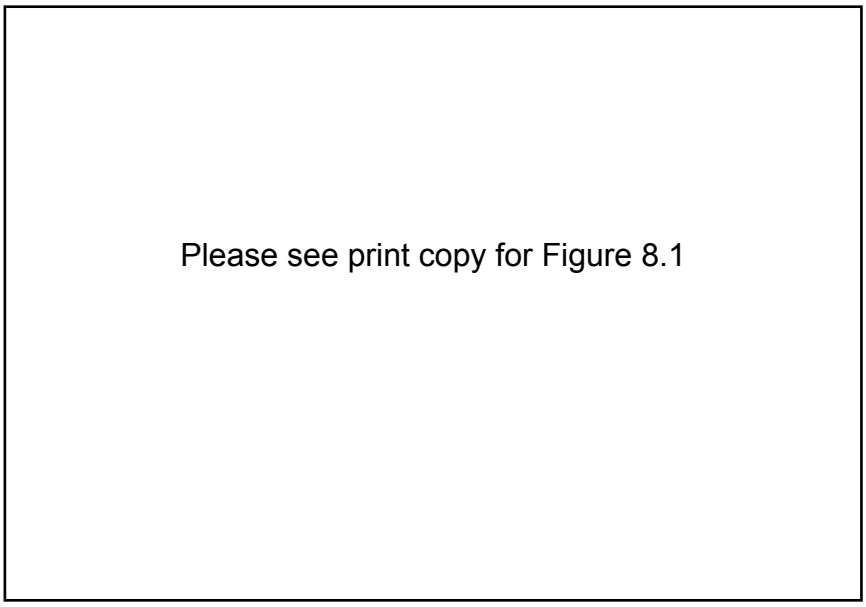

Figure 8.1: Viewing the tags associated with file: docs is fully asserted, exe is fully retracted, agents have offered partial retraction on travel and partial assertion on waffle. Assertion is shown in green extending from left to right, retraction is shown in red extending right to left. For readers using a non coloured medium, the tags with ID 10 and 22 are the only green ones.

classification where the user is not inclined to manually construct a large initial training collection.

To adapt to the less complete initial model the system needs to be able to add and remove test cases and retrain agents quickly to support a more evolutionary model building process. The user can then enhance the model from a small initial one as they see agents as being a valuable part of the system.

To connect machine learning with libferris one can use tag associations to train on and then in turn the machine learning can be used to obtain a fuzzy assertion or retraction for if an tag is applicable for a new file.

There are many design choices when integrating machine learning into the file system. Firstly there is what machine learning algorithms to use. Usually the algorithm has different requirements for how much state is required during training, the size of the model produced during training, the ease with which the model can be updated, speed of execution in training and classification and relative quality of results.

Two algorithms have been chosen for initial testing: Bayesian [1] and Support Vector Machine (SVM) [53]. Bayesian filtering is very efficient in terms of speed and its model allows individual documents to be added or removed without having to retrain on all test cases.

SVM operates on the Feature Sets of its input files. A file's feature set is a map of each term to its relative importance μ to the document. The value of μ is usually calculated as the frequency of that term (TF) in the document multiplied by the Inverse Document Frequency (IDF) for that term. The IDF is the reciprocal of the number of times a term appears in all documents. Many SVM implementations require a collection of positive and negative Feature Sets to train on and can not "add" a new case to an existing model.

We will now take a closer look at the SVM Agent Implementation (SAI).

Generated Feature Sets are cached for two reasons: true belief capturing and efficiency. True belief capturing reflects that when the user highlights a document as a training example they are making assertions about that document at that point in time. For example, when the user asserts that they like a web page they are really talking about the page as it stands at assertion time. Calculating Feature Sets is a costly act and must be avoided in order to support timely addition of training examples when the number of existing Feature Sets is high.

The SAI reuses a lexicon implementation from the fulltext indexing code to map unique terms to unique integers. These integers are subsequently used in Feature Sets to identify the terms. The IDF values are stored in a Feature Set which is maintained by the SAI and updated by the SAI Trainer (SAIT).

To classify a file it needs to be tokenized and a Feature Set μ generated for it. Then μ is multiplied by the IDF and normalized by dividing it by the euclidean length of the entire feature vector for that document. For example, given the initial Feature Set $\mu = (x_1, ..., x_N)$ and the IDF Feature Set $(\beth_1, ..., \beth_n)$ the result $\phi = (y_1, ..., y_N)$ is calculated as $y_i = x_i * \beth_i / \sqrt{(\sum_{t=1}^{N}(x_t^2))}$.

It is acceptable for the training of the agent to be a more costly operation than its use for predictions. The SAIT maintains a Feature Set τ which is the total number of times each term appears in the Training Data (TD). When a new Training Case (TC) is presented it is tokenized and a Feature Set of the documents term frequencies is created while at the same time τ is updated. The maintenance of τ is due to the many possible

formulae for calculating the IDF [104]. If one is calculating the IDF as shown above then τ is not required and should be the element wise reciprocal of the IDF.

The two major efficiency requirements for the SAIT are the caching of Feature Sets and the ability to generate a current IDF Feature Sets quickly. The storage and update of τ achieves the latter goal while allowing flexibility in how the IDF is generated. The SAIT applies the same IDF multiplication and normalization as the SAI before training on those Feature Sets.

Tests were conducted in order to test both the implementation's correctness and the utility of the agent based classification under a relatively low training data size. A subset of the Reuters-21578 test collection[3] was used with only 179 positive and 191 negative cases. The Reuters-21578 files are a collection of news-feed stories and have been assigned zero or more categories based on their content. For this test the positive cases are documents that are about corporate acquisitions and negative cases are documents that aren't.

When trained on this data the SVM agent correctly predicted 5 of the 6 classification examples with the incorrect result's prediction value being closer to a zero than full assertion or retraction. These examples were chosen at random from the news-feed documents.

The SAIT storing all Feature Sets does consume more storage than is strictly required. For the above training example the SVM SAI consumed $1.7M$ of disk for all of its state, of which the lexicon was $508K$ (uncompressed in XML format) and the cached Feature Sets $399K$. The svm_light model was $474K$. Given that hard disk costs are well under a dollar per gigabyte it should not be considered unreasonable for agents to consume tens of megabytes for state information.

[3] http://www.daviddlewis.com/resources/testcollections/reuters21578/

8.3 Data models: Semantic Filesystems and XML

Taking an abstract view of the data model of a semantic filesystem one arrives at; files with byte contents, files nested in a hierarchy and arbitrary attributes associated with each file. This is in many ways similar to the data model of XML; elements with an ordered list of byte content, elements nested in an explicit ordering with attributes possibly attached to each element.

Many differences are insignificant, for example, the fact that (extended) attributes in a semantic filesystem may be derived at run time rather than stored.

The differences between the data models may raise issues and require explicit handling. The differences have been found to be;

- XML elements can contain multiple contiguous bytes serving as their "contents". Files may have many sections of contiguous bytes separated by holes. Holes serve to allow the many sections of contiguous bytes to appear in a single offset range. For example, I could have a file with the two worlds "alice" and "wonderland" logically separated by 10 bytes. The divergence of the data models in this respect is that the many sections of contiguous bytes in an XML element are not explicitly mapped into a single logical contiguous byte range.

- XML elements are explicitly ordered by their physical location in the document. For any two elements with a common parent element it will be apparent which element comes "before" the other. Normally files in a filesystem are ordered by an implementation detail – their inode. The inode is a unique number (across the filesystem itself) identifying that file. Many tools which interact with a filesystem will sort a directory by the file name to be more palatable to a human reader.

- The notions of file name and element name have different syntax requirements. A file name can contain any character apart from the "/" character. There are much more stringent requirements on XML element names – no leading numbers, a large range of characters which are forbidden.

- For all XML elements with a common parent it is not required that each child's name be unique. Any two files in a directory **must** have different names.

The differences are shown in Figure 8.2.

The identification of this link between data models and various means to address the issues where differences arise helps both the semantic filesystem and XML communities by bringing new possibilities to both. For example, the direct evaluation of XQuery on a semantic filesystem instead of on an XML document.

The file name uniqueness issue is only present if XML is being seen as a semantic filesystem. In this case it can be acceptable to modify the file name to include a unique number as a postfix. In cases such as resolution of XPath or XQueries file names should be tested without consideration of the unique postfix so that query semantics are preserved.

As XML elements can not contain the "/" character exposing XML as a semantic filesystem poses no issue with mapping XML element names into file names. Unfortunately the heavy restrictions on XML element name syntax does present an issue.

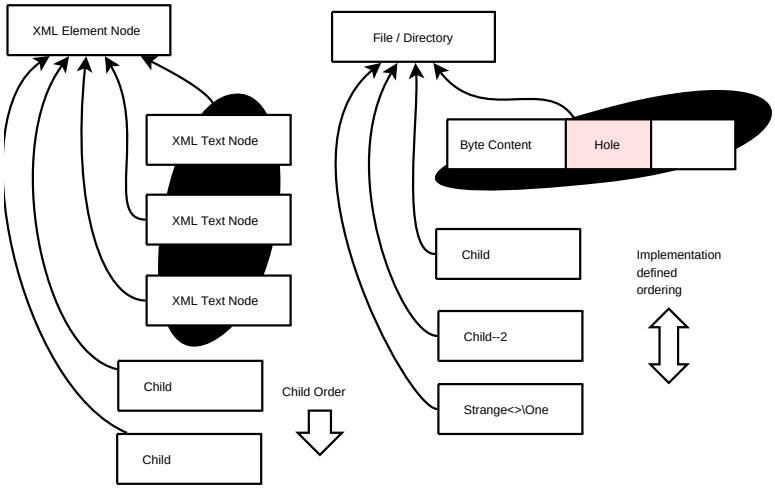

Figure 8.2: On the left an XML Element node is shown with some child nodes. On the right a filesystem node is shown with some similar child nodes. Note that XML Text nodes can be considered to provide the byte content of the synonymous filesystem abstraction but metadata about their arrangement can not be easily communicated. Child nodes in the XML side do not need to have unique names for the given parent node and maintain a strict document order. Child nodes on the filesystem side can contain more characters in their file names but the ordering is implementation defined by default.

The most convenient solution has been found to be mapping illegal characters in filenames into a more verbose description of the character itself. For example a file name "foo<bar>.txt" might be canonicalized to an XML element name of "foo-lt-bar-gt.txt". The original unmodified file name can be accessed through a name spaced XML attribute on the XML element.

XML element ordering can be handled by exposing the place that the XML element appeared in document order. For example, a document with "b" containing "c,d,e" in that order the "c" file would have a place of zero, and "e" would be two. With this attribute available the original document ordering can be obtained through the semantic filesystem by sorting the directory on that attribute. As there is no (useful) document ordering for a filesystem this is not an issue when exposing a filesystem as XML.

There is no simple solution to the fact that XML elements can have multiple text nodes as children. In cases where XML with multiple child nodes exist they are merged into a single text node containing the concatenation in document order of the child text nodes. Files with holes are presented as though the hole contained null bytes.

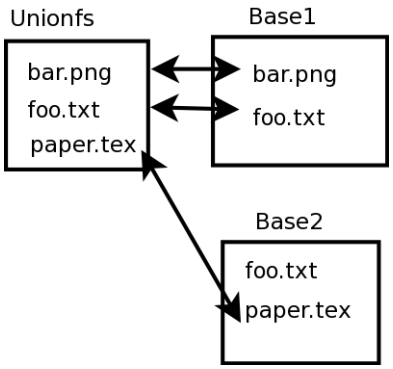

Figure 8.3: Union filesystem. The two base filesystems, Base1 and Base2 have their contents combined by set union. Normally there is a linear precedence relation between the base filesystems to explicitly resolve name clashes. In this case Base1's foo.txt will always be selected over the file with the same name in Base2.

8.4 Arbitrary Translation Semantic File Systems

Modern filesystems have become virtual filesystems: many sources of information are presented through a filesystem interface. Such filesystems allow direct interaction with the operating system kernel through a filesystem interface (`/proc`, `/sys`) [59].

Metadata interfaces have been introduced for both stored [59, 5] and derived file information [39]. Metadata is associated on a per file or directory basis though an Extended Attribute (EA) interface [62, 61]. EAs are key-value pairs, for example, `size=25` to indicate that a file is 25 bytes in length. This additional information interface further broadens the possible variations of how a virtual filesystem's schema will be designed. For example, the same information can be made available as a single directory with many subfiles or a single file with many EAs.

In the past virtual filesystems have evolved to include the layering of virtual filesystems on top of themselves. Such constructs include the unionfs [105] which performs a set theoretic union on many underlying virtual filesystems to filtering and sorting filesystems [3]. A union virtual filesystem is shown in Figure 8.3 and a filtering virtual filesystem in Figure 8.4. This section (and the paper it is based on) advances the notions of these fixed simple transforms and morphisms to include dynamically changeable bidirectional filesystem morphisms.

When one considers filesystems as files and directories (elements) along with metadata interfaces allowing key-value pairings (attributes) to be associated with elements then the boundary between modern filesystems and XML becomes minimal. Differences do exist, such as XML having multiple text nodes per entity vs a file having a single byte stream, or as another instance, filesystems mandating a unique name for each subobject of a directory where XML allows subentities to be repeated with identification relying on document ordering. The libferris virtual filesystem [62, 3] allows virtual filesystems to be accessed as XML Document Object Models (DOM)s and DOMs to be mounted as virtual filesystems. Such a relation is shown in Figure 8.5.

The blurred distinction between XML and filesystems allows one to use a powerful tree morphism language to succinctly describe filesystem morphisms. The XSLT

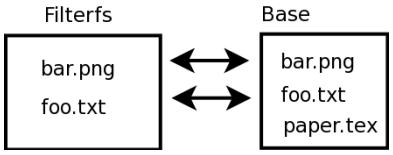

Figure 8.4: A filtering filesystem. Only files matching a given predicate (in this case *.png and *.txt) are exposed from the base filesystem.

```
$ fls -l d1
-rw-rw---- ben ben 29   06 Jun  7 20:29 datefile.txt
-rw-rw---- ben ben 1338 06 Jun  7 20:30 snack.jpg
$ fls --xml d1
<ferrisls>
<ferrisls url="file:///.../d1"  name="d1"  >
 <context  protection-ls="-rw-rw----"
    user-owner-name="ben" group-owner-name="ben"
    size="29" mtime-display="06 Jun  7 20:29"
    name="datefile.txt"  />
 <context  protection-ls="-rw-rw----"
    user-owner-name="ben" group-owner-name="ben"
    size="1338" mtime-display="06 Jun  7 20:30"
    name="snack.jpg"  />
</ferrisls>
</ferrisls>
```

Figure 8.5: A virtual filesystem can be seen equally as a virtual XML document. Notice that the same EAs are shown by both commands, files and directories naturally map to XML entities, EAs map to XML Attributes.

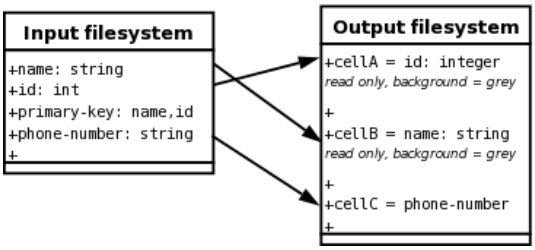

Figure 8.6: An abstract representation of a schema morphism from a virtual directory to a spreadsheet file.

language is a declarative XML grammar to describe tree transformations.

Modern office document formats are XML files. As described above, some virtual filesystems can also be exposed as XML. If a virtual filesystem has an office document's schema then one can use their office applications directly on a virtual filesystem. With filesystem morphisms one can freely modify the schema of their filesystem to manufacture a virtual filesystem with the desired office document schema.

The libferris virtual filesystem [3] makes a great deal of schema information available. Examples include what basic type an EA is, such as the `size` attribute being of type integer. Other schema constrains which can span many EAs are also available such as the set of EA which comprise a primary key for a directory. Normally the file name takes on the properties of a primary key: unique across all files in the directory and not a null value. When a relational database is mounted as a filesystem the primary key may be comprised of many EAs.

The availability of type information in the filesystem can also be taken advantage of in the morphic filesystem. For example, if generating a morphic office document schema, numeric data taken from a filesystem can be marked as such in the output to allow a office spreadsheet to correctly process that information. For example, collation can be set correctly for a column in the spreadsheet.

As the virtual filesystem allows access to schema and type information for its files it is possible to define morphisms between these schemas and a desired filesystem format.

As an example consider defining a morphism between the schema of a directory and the schema of a spreadsheet file. In this case one might wish to ensure that the EA which comprise the primary key of the directory are presented in read only cells on the left side of the spreadsheet file. This definition is between the schema of the directory, in particular which EA comprise the primary key, and the schema of the spreadsheet, in particular that columns $\{a, b, ..., n\}$ are read only cells to be painted with a darker grey background. The morphism also records which EA in the directories schema will map to which columns in the spreadsheet file. This morphism is based only on the schema of the input directory and the desired schema of the output, for example, converting to the schema of a Microsoft Excel 2003 XML file.

This example is shown in Figure 8.6. In this case the primary key is used to define the morphism onto spreadsheet cells of the EA from the input filesystem. EA which comprise the primary key are marked as read only in the schema of the office document.

msgs Column	Type	Modifiers
id	integer	primary key, serial
msg	varchar(.)	
foo	varchar(.)	

Figure 8.7: Schema for msgs table. The id column is the primary key with a default sequential next value if id is not given for a new tuple.

8.4.1 Relational Models and OpenOffice morphisms

One method for exposing a relational database through a virtual filesystem is to consider each tuple as defining metadata about its primary key. Another would be to create directories for each tuple with a file for each column. It is unlikely that either of these methods will be fully satisfactory for all users.

Consider a relational database with a table `msgs` which has a numeric public key `id` and two varchar columns `msg` and `foo`. Shown in Figure 8.7.

Assuming that a tuple is exposed as a file named using the public key with metadata attached to the file for each column one might view the virtual filesystem as shown in Figure 8.8.

The manner in which the relational database is exposed in Figure 8.8 is biased toward both command line and file manager interaction. The chosen schema for this filesystem makes coarser level document computing such as opening this relation in a spreadsheet more difficult.

A relational database has good schema metadata and type availability. Given a desired schema such as an office spreadsheet file one can mechanically define a schema morphism to expose this filesystem as a virtual spreadsheet file.

The general data flow to create a virtual office document from a mounted relational database is shown in Figure 8.10. The XSL file defining the morphism is created a priori by inspecting the mounted relational database and inferring the morphism required to generate an office spreadsheet file schema. When the virtual office document is read, data is taken from the mounted relational database by the morphism engine for the XSL file to be applied too in order to generate a new virtual filesystem. This resulting filesystem is then taken as the source for the XML file of the office document. Also, when the office document is written from OpenOffice the morphism engine applies a reverse morphism in order to save the information back into the relational database.

Note that although in this case we are expecting to save the data back to the

```
# fls --show-ea=name,msg,schema:msg,foo,primary-key pg://localhost/play/msgs
1       msg1value  schema://xsd/attributes/string foo1value id
2       msg2value  schema://xsd/attributes/string foo2val   id
```

Figure 8.8: Viewing a mounted PostgreSQL relational database as a virtual filesystem. As can be seen the primary key for this directory is the id metadata. The schema for the msg metadata is a string.

same place it originated we do not seek *iso*morphisms. The morphism engine is free to ignore part of the input filesystem, calculate aggregate, derivative or totally unrelated information for the output virtual filesystem. In this case columns in the spreadsheet are named based on the schema of the input filesystem. Such names are ignored in the reverse morphism. To facilitate non isomorphic translations both the forward and reverse morphism must be explicitly defined.

In this case the overall appearance of the office document can be left for the user to decide. What is important in the morphism is to translate the schema for a given relational table into a spreadsheet cell block schema. A simple morphism for a table would indicate to the user that some cells form the primary key for the table and others are information relating to that key. This would allow the user to edit information without risk of violating key constraints by only editing non primary key cells.

An example schema morphism could expose each file (tuple) as a row in the spreadsheet. Each column in the table would form a cell in each row. A suitable schema morphism for this example would generate an XSL file which has two core aspects: each column is named in a header by generating cells regardless of the input filesystem and an `xsl:template` which would match input files and translate them into output cells.

The result of these morphisms is the ability to edit a relational database table as shown in Figure 8.9.

8.4.2 Document Computing and The Semantic Web

At the base of the semantic web is the RDF [85, 14] technology with its schema language RDFS. The core of RDF is a very simple model based on triples. A triple consists of a subject, predicate and object. All three of these can be Unique Resource Indicators (URI)s. In addition objects can be literal values. This model allows one to define entities with properties expressed on them using predicates which link to literal values or other entities. Because objects can be URIs they can be used to link to other subjects and thus form a graph structure.

Due to the triple format being rather different to most past document computing methods, authoring and editing RDF data presents new issues. By their very nature URIs are intended to be unique and in practice mostly resemble URLs. Although the length of URIs can help to avoid the implicit collision of identifiers it also makes the direct expression of information in RDF more cumbersome.

Using schema morphisms at the filesystem level existing document computing methods are again applicable to the new Semantic Web data formats.

The main contention with expressing RDF as a virtual filesystem is the lack of a "root" node in the graph. This can be solved by explicitly nominating a URI to serve as the root of the virtual filesystem. The URI to use as the root of the RDF mount can be either specified within the RDF store itself or as an EA on the RDF file. By allowing the root URI to be an extended attribute of the file, many views of the RDF store can be made starting at different root URIs by making softlinks to the same RDF store, each of which nominates a different root URI through an EA on the softlink.

Once the root URI has been identified a virtual filesystem rooted at that URI can be created. Triples which feature a literal value as their object can be mapped to Extended Attributes (metadata) in the filesystem. Where a triple features a URI as the object then the predicate can serve as a directory name and the triple pointed to be the

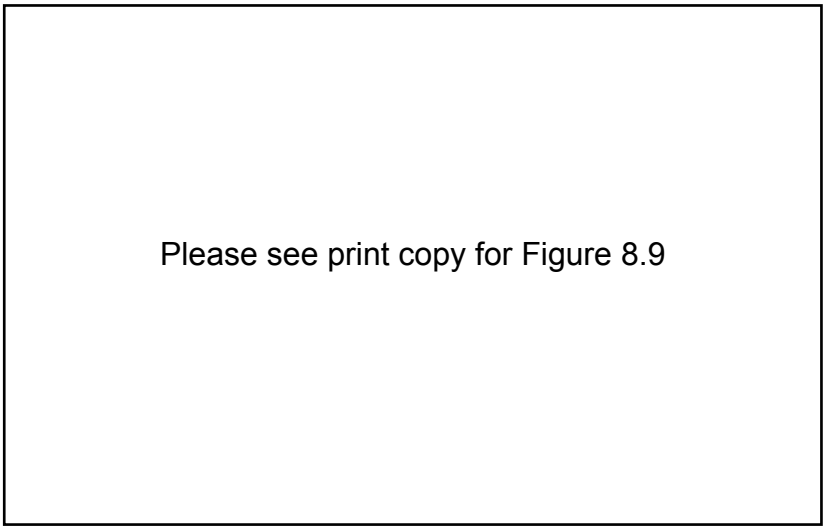

Figure 8.9: OpenOffice editing a virtual office document created from the current contents of the msgs table in a PostgreSQL database.

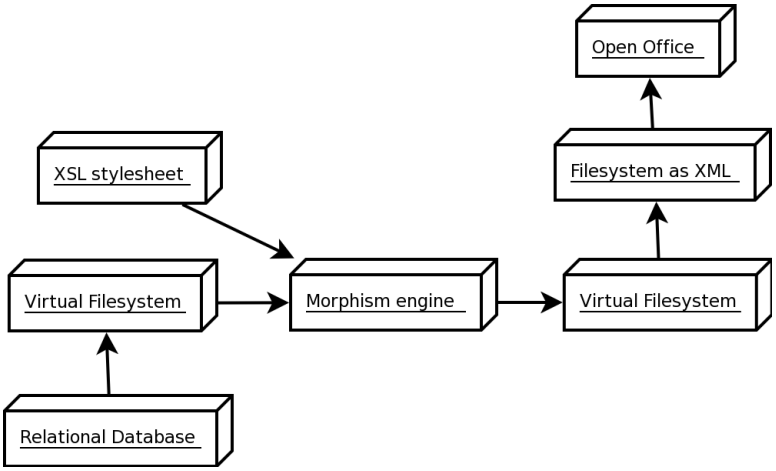

Figure 8.10: Data flow in the creation of a virtual office document from a mounted relational database.

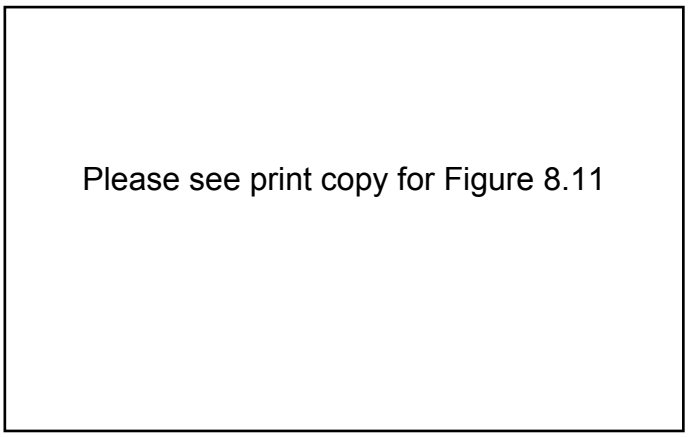

Figure 8.11: OpenOffice editing a virtual office document created from the current contents of a mounted RDF graph.

object URI resolved as the directory contents. Thus a directed graph is exposed from the given subject URI.

In Figure 8.12, a small RDF graph is shown in n-triples format and a virtual filesystem mounting it from a virtual root element with children explicitly named in the RDF graph itself. The RDF graph is shown visually in Figure 8.13. The resulting office document is shown in Figure 8.11.

A similar method to that employed in exposing relational data can then be used to automatically find a schema morphism for RDF. The RDFS defines the schema for the RDF graph and as such can be used to automatically generate a schema morphism from the RDF graph into a desired schema.

This enables direct bidirectional document computing on RDF stores.

8.5 Conclusion

Automatic classification of files is directly advantageous when applying Formal Concept Analysis to a Semantic File System because the classifications can be directly used to create a concept lattice. The use of an Semantic File System as the data source for running Supervised Machine Learning immediately unlocks the ability to run such Machine Learning on a myriad of data sources such as online web forums and message groups.

Union filesystems have existed in one form or another for a long time in the filesystem world. The use of arbitrarily complex forward and reverse translations allows not only the union of underlying data sources into a single filesystem but also data model mutation. For example, the rigid data model of a relational database can be augmented with data from a source with a looser schema such as RDF to create a virtual document which obeys the schema of an office document. Having support for both forward and reverse translations allows the mutated filesystem to be modified and effects translated back to the original data source.

The flexibility provided by the bidirectional arbitrary translations from within the virtual filesystem itself allows data to be made available in a desired schema, in a

```
# rdfproc foo serialize ntriples
<http://foo.com/child1> <foo1> "Value1.1" .
<http://foo.com/child1> <http://foo.com/num> "100" .
<http://foo.com/child1>
    <http://witme.sf.net/libferris-core/ns/unknown/foo:num> "" .
<http://foo.com/child2> <bar2> "no namespaces here" .
<http://foo.com/child2> <foo1> "value2.A" .
<http://foo.com/child2> <http://foo.com/num> "200" .
<http://foo.com/child2> <http://foo.com/bar1> "This is fun" .
<http://foo.com/child2>
    <http://witme.sf.net/libferris-core/ns/unknown/foo:num> "" .
<http://witme.sf.net/libferris-core/0.1/rdf-root-subject-uri>
    <root> <http://foo.com/child1> .
<http://witme.sf.net/libferris-core/0.1/rdf-root-subject-uri>
    <root> <http://foo.com/child2> .
# fls -0h ./foo-po2s.db
root
# fls -0h ./foo-po2s.db/root --show-ea=name,foo1
http://foo.com/child1   Value1.1
http://foo.com/child2   value2.A
```

Figure 8.12: Mounting RDF as a filesystem. The RDF graph is stored in a collection of Berkeley db files for fast query processing. These files are named with a foo prefix.

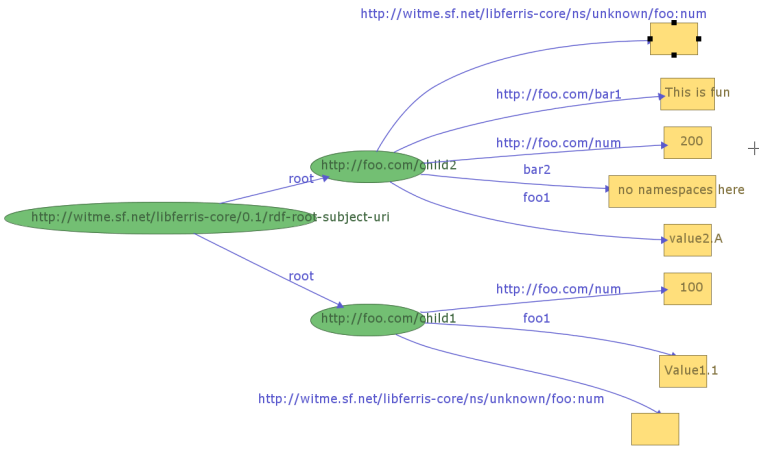

Figure 8.13: Graphical representation of RDF from Fig 8.12.

read and write format with changes effecting the original data source. Such flexibility has historically been provided by an application developer where the exact schema of a filesystem does not meet their requirements.

Directly considering the link to the XML data model does raise some areas where major differences between filesystem and XML data models exist. For many cases these differences can be smoothed over to allow interaction with a Semantic File System as though it was actually an XML document. An example of the power of seeing a Semantic File System as an XML document is in the use of XQuery to query for data stored in an XML file. The XML file can be converted into a B-Tree file (mountable as a Semantic File System) and the XQuery will resolve against the B-Tree file taking substantially less time than the XML file [71].

8.6 Semantic File System Future Directions

Allowing the Semantic File System to protect and maintain its consistency by asserting a constraints system. Future work will include research on integration with existing constraints [29] and correctness systems such as XML schema enforcement [99, 100, 101] and the Object Constraint Language [102, 22] with extensions for dynamic behaviour [32].

Given the vast pool of metadata made available by the Semantic File System the use of deductive query techniques [26] and formal logic systems [54, 21] could also enhance the Semantic File System experience as an information resource.

Chapter 9

Conclusion

Special indexing structures are essential to FCA systems with large data sets like those encountered in semantic file systems. An index structure derived from spatial indexing for accelerating subset queries has been found to be productive. When the user wishes to list the files matching a concept an RD-Tree permits this within an acceptable time frame for interactive use. The link to spatial indexing structures have not been reported in current best practices elsewhere in the FCA literature [20].

Although the use of spatial indexing was first adopted and implemented to allow Formal Concept Analysis to be applied specifically to the special circumstances encountered by Semantic File Systems, the data structures have also been found to be an advantageous structure for general purpose FCA applications: such as those supported by TOSCANAJ.

The performance of spatial indexing for Formal Concept Analysis in various settings has been examined and shown to provide substantial improvements in many cases. Performance gains from RD-Trees are very effective for sparse formal contexts where queries can be resolved five times faster on large data sets as shown in Section 4.4.4.2. The largest benchmark results were found when applied to a large dataset from the UCI collection where the formal context was generated by nesting one conceptual scale inside another. In such an environment queries can be executed over 80 times faster using an RD-Tree than without.

The custom Generalized Index Search Tree presented offers at times a 36% drop in internal node count and in many cases a reduction of the depth over an RD-Tree. These two metrics directly impact the query performance of such a tree index [33]. It should be noted that in many typical applications of Formal Concept Analysis many hundred queries are rendered against the index [75] further compounding the above performance advantages.

The application of Formal Concept Analysis as a postprocess to the standard Guttman distribution at page splitting time can help to improve the clustering in the index structure itself.

The application of compression tailored to take advantage of knowledge of how Formal Concept Analysis is being applied to the input data can significantly reduce the size of the index structure. In particular the number of internal nodes can be reduced to just 4% of that required by generic compression. This is reflected directly in the depth of the Generalized Index Search Tree going from 7 to just 3 levels.

The ability to resolve queries in a more timely manner facilitates the scalability of FCA to applications with many more objects and attributes than is presently considered

feasible. For example, application to data sets the size of a filesystem becomes possible. Due to the indexing structure's ability to more efficiently handle high dimensional data Formal Concept Analysis can also be applied with a wider scope. For example, the ability to consider more attributes simultaneously than was previously tractable.

Though the work on applying Formal Concept Analysis to obtain superior clustering post page split is still in its infancy, research contained in this thesis has demonstrated superior results for indexes with 16 and 32 dimensions.

The exploitation of the link to Data Mining opens up a range of new algorithms for finding the set of Formal Concepts for a given Formal Context. This is especially true when only concepts with at least a given number of objects is required. The publication of the titanic algorithm brought the apriori algorithm from Data Mining into the Formal Concept Analysis community. The titanic algorithm makes use of how the apriori algorithm works in order to deliver not only the set of Formal Concepts but also make the ordering among them explicit at the same time. However, there is no explicit requirement to use the titanic process and the covering relation between formal concepts can be made explicit with reasonable time complexity. This opens the door to using any Closed Frequent Itemset algorithm to find the set of Formal Concepts.

The thesis has focused in large part on improving the state of the art of applied Formal Concept Analysis. During the course of the research various articles have also appeared in the Linux Journal [64, 66, 70, 68] and xml.com [63] directly relating to the research. The implementation has been seen to be well crafted and technically advanced enough to merit acceptance of the author at the Linux Kongress 2005 [65] and the Ottawa Linux Symposium 2007 [69]. The combined thesis and implementation of indexing structures were furthermore interesting enough for the author to present both at the Linux Kongress 2008 [72]. This reinforces that the applied research has been performed using an implementation which is of sufficient quality to garner such wide community interest.

The work in Section 8.2 appeared in ADCS 2003 [61]. Section 1.2 appeared in ICFCA 2004 [62]. Section 2.3 contains information from ICFCA 2004 [62]. Section 6 contains information from ADCS 2005 [73]. Section 4.2 and Section 4.3 contain information from ICFCA 2006 [75]. Section 4.4 contains information from ISMIS 2006 [74] and ICFCA 2007 [76].

Bibliography

[1] bogofilter homepage, http://bogofilter.sourceforge.net/. Visited Sep 2003.

[2] Ea and acl for linux website, http://acl.bestbits.at/. Visited Sep 2003.

[3] libferris, http://witme.sf.net/libferris.web/. Visited Apr 2007.

[4] Md5 hash function rfc, http://www.ietf.org/rfc/rfc1321.txt. Visited Sep 03.

[5] Ntfs extended attributes, http://linux-ntfs.sf.net/ntfs/attributes/ea_information.html. Visited Sep 2003.

[6] Postgresql, http://www.postgresql.org/. Visited Apr 2007.

[7] Sleepycat berkeley db, http://www.sleepycat.com. Visited June 2004.

[8] Svmfu svm project website, http://five-percent-nation.mit.edu/svmfu/. Visited Sep 2003.

[9] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, Proc. 20th Int. Conf. Very Large Data Bases, VLDB, pages 487–499. Morgan Kaufmann, 12–15 1994.

[10] Paul M. Aoki. Implementation of extended indexes in POSTGRES. SIGIR Forum, 25(1):2–9, 1991.

[11] Peter Becker. Docco: Document retrieval with formal concept analysis. In Engelbert NGuifo et al., editor, Supplementary Proc. of the 3rd Int. Conference on Formal Concept Analysis. University of Artois, 2005.

[12] Peter Becker and Richard Cole. Querying and analysing document collections with formal concept analysis. In Australian Document Computing Symposium (ADCS03). University of Queensland, 2003.

[13] Peter Becker and Peter Eklund. Prospects for formal concept analysis in document retrieval. In Australian Document Computing Symposium (ADCS01), pages 5–9. University of Sydney, Basser Department of Computer Science, 2001.

[14] Dave J. Beckett. The design and implementation of the redland RDF application framework. In World Wide Web, pages 449–456, 2001.

[15] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations and model. M74-244, Mitre Corporation, Bedford, Massachusetts, 1975.

[16] Michael W. Berry. Understanding search engines : mathematical modeling and text retrieval. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[17] Blake, C., Merz, C. UCI Repository of Machine Learning Databases. [`http://www.ics.uci.edu/~mlearn/MLRepository.html`]. Irvine, CA: University of California, Department of Information and Computer Science, 1998.

[18] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information, 2001.

[19] C. Carpineto and G. Romano. Order-theoretical ranking. Journal of the American Society for Information Science (JASIS), 51(7):587–601, 2000.

[20] Claudio Carpineto and Giovanni Romano. Concept Data Analysis. Wiley, England, 2004.

[21] J. Chomicki and G. Saake, editors. Logics for Databases and Information Systems. Kluwer, 1998.

[22] Tony Clark and Jos Warmer, editors. Object Modeling with the OCL: The Rationale behind the Object Constraint Language, volume 2263 of LNCS. Springer, 2002.

[23] Richard Cole. Document Retrieval using Formal Concept Analysis. PhD thesis, School of Information Technology, Griffith University, 2001.

[24] Richard Cole, Peter Eklund, and Don Walker. Constructing conceptual scales in formal concept analysis. In Procedings of the 2nd Pacific Asian Conference on Knowledge Discovery and Data Mining, number 1394 in LNAI, pages 378–379. Springer Verlag, 1998.

[25] Richard J. Cole, Peter W. Eklund, and Gerd Stumme. Document retrieval for email search and discovery using formal concept analysis. Journal of Applied Artificial Intelligence (AAI), 17(3):257–280, 2003.

[26] Robert M. Colomb. Deductive Databases and Their Applications. Taylor & Francis, Inc., Bristol, PA, USA, 1998.

[27] Brian A. Davey and H.A. Priestley. Introduction to Lattices and Order 2nd Edition. Cambridge University Press, Cambridge, UK, 2002.

[28] Dorothy E. Denning. A lattice model of secure information flow. Commun. ACM, 19(5):236–243, 1976.

[29] S. M. Embury and P. M. D. Gray. Compiling a Declarative High-Level Language for Semantic Integrity Constraints. In R. Meersman and L. Mark, editors, Proceedings of the 6th IFIP TC-2 Working Conference on Data Semantics (DS-6), pages 188–226. Chapman & Hall, 1996.

[30] Sebastien Ferré and Olivier Ridoux. A file system based on concept analysis. In Vernica Dahl et al. John W. Lloyd, editor, Proceedings of the First International Conference on Computational Logic, number 1861 in LNAI, pages 1033–1047, London, UK, 2000. Springer-Verlag.

[31] Sebastien Ferré and Olivier Ridoux. A logical generalization of formal concept analysis. In B. Ganter and G. W. Mineau, editors, 8th International Conference on Conceptual Structures, LNAI, pages 371–384, Heidelberg-Berlin, August 2000. Springer-Verlag.

[32] S. Flake and W. Mueller. Expressing property specification patterns with ocl, 2003.

[33] Michael J. Folk and Bill Zoelick. File Structures. Addison-Wesley, Reading, Massachusetts 01867, 1992.

[34] David Caplan Frank Mayer, Karl MacMillan. SELinux by Example: Using Security Enhanced Linux. Prentice Hall PTR, 2006.

[35] Timothy Fraser. Lomac: Low water-mark integrity protection for cots environments. In SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

[36] Timothy Fraser. Lomac: Mac you can live with. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, pages 1–13, Berkeley, CA, USA, 2001. USENIX Association.

[37] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, K. J. Miller. Introduction to wordnet: An on-line lexical database. In Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, pages 112–119, 1990.

[38] Bernhard Ganter and Rudolf Wille. Formal Concept Analysis — Mathematical Foundations. Springer–Verlag, Berlin Heidelberg, 1999.

[39] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. Jr O'Toole. Semantic file systems. In Proceedings of 13th ACM Symposium on Operating Systems Principles, ACM SIGOPS, pages 16–25, 1991.

[40] Bart Goethals and Mohammed Javeed Zaki. Advances in frequent itemset mining implementations: Report on fimi'03. In Bart Goethals and Mohammed Javeed Zaki, editors, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, volume 90 of CEUR Workshop Proceedings, 2003.

[41] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchial file systems. In Proceedings of third symposium on Operating Systems Design and Implementation, USENIX Association, pages 265–278, 1999.

[42] Osta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Bart Goethals and Mohammed J. Zaki, editors, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations. Ceur, 2003.

[43] Network Working Group. Rfc 2254 - the string representation of ldap search filters, http://www.faqs.org/rfcs/rfc2254.html. Visited Sep 2003.

[44] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Proc. ACM-SIGMOD International Conference on Management of Data, Boston Mass, 1984.

[45] Jiawei Han. Data mining : concepts and techniques. Morgan Kaufmann Publishers, San Francisco, Calif., 2001.

[46] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, 2000 ACM SIGMOD Intl. Conference on Management of Data, pages 1–12. ACM Press, 05 2000.

[47] Darren R. Hardy and Michael F. Schwartz. Essence: A resource discovery system based on semantic file indexing. In USENIX Winter, pages 361–374, 1993.

[48] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998, pages 365–377, New York, NY, USA, 1998. ACM Press.

[49] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, Proc. 21st Int. Conf. Very Large Data Bases, VLDB, pages 562–573. Morgan Kaufmann, 11–15 1995.

[50] Joseph M. Hellerstein and Avi Pfeifer. The RD-Tree: An Index Structure for Sets, Technical Report 1252. University of Wisconsin at Madison, October 1994.

[51] S. Helmer. Index structures for databases containing data items with setvalued attributes, Technical Report 2/97, Universitat Mannheim, 1997.

[52] Alan R. Simon Jim Melton. SQL:1999 : understanding relational language components. Academic Press, San Francisco, CA, 2002.

[53] T. Joachims. Making large-scale support vector machine learning practical. In A. Smola B. Schlkopf, C. Burges, editor, Advances in Kernel Methods: Support Vector Machines. MIT Press, Cambridge, MA, 1998.

[54] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. Technical Report TR-90-003, 1, 1990.

[55] M. Kim and P Compton. Formal concept analysis for domain-specific document retrieval systems. In The 13th Australian Joint Conference on Artificial Intelligence (AI'01), pages 179–184. Springer-Verlag, 2001.

[56] Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997.

[57] Butler W. Lampson. A note on the confinement problem. Communications of the ACM, 16(10):613–615, 1973.

[58] Angelike Langer and Klaus Kreft. Standard C++ IOStreams and Locales: Advanced programmer's Guide and Reference. Addison Wesley, Reading, Massachusetts 01867, 2000.

[59] Robert Love. Linux Kernel Development. Novell Press, 2005.

[60] Wen-Pai Lu and Malur K. Sundareshan. A model for multilevel security in computer networks. In IEEE Transactions on Software Engineering archive, volume 16, pages 647–659, Piscataway, NJ, USA, June 1990. IEEE.

[61] Ben Martin. File system wide file classification with agents. In Australian Document Computing Symposium (ADCS03). University of Queensland, 2003.

[62] Ben Martin. Formal concept analysis and semantic file systems. In Peter W. Eklund, editor, Concept Lattices, Second International Conference on Formal Concept Analysis, ICFCA 2004, Sydney, Australia, Proceedings, volume 2961 of Lecture Notes in Computer Science, pages 88–95. Springer, 2004.

[63] Ben Martin. Using libferris with xml, March 2004.

[64] Ben Martin. Filesystem indexing with libferris. Linux Journal, 2005(130):7, 2005.

[65] Ben Martin. A virtual filesystem on steroids: Mount anything, index and search it. In Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress 2005). GUUG e.V. / Lehmanns / Ralf Spenneberg, 2005.

[66] Ben Martin. Federated desktop and file server search with libferris. Linux Journal, 2006(152):8, 2006.

[67] Ben Martin. Geotagging files with libferris and google earth (linux.com), April 2006.

[68] Ben Martin. The world is a libferris filesystem. Linux Journal, 2006(146):7, 2006.

[69] Ben Martin. Everything is a virtual filesystem: libferris. In Proceedings of the 10th Ottawa Linux Symposium, 2007.

[70] Ben Martin. Virtual filesystems are virtual office documents. Linux Journal, 2007(154):8, 2007.

[71] Ben Martin. Xquery, libferris, and virtual filesystems, July 2007.

[72] Ben Martin. Semantic filesystems and formal concept analysis: a phd 5 years in the making. In W. Stief, editor, Proceedings of the Linux-Kongress 2008 in Hamburg. GUUG e.V. / UpTimes, 2008.

[73] Ben Martin and Peter Eklund. Applying formal concept analysis to semantic file systems leveraging wordnet. In Australian Document Computing Symposium (ADCS05). Sydney University, 2005.

[74] Ben Martin and Peter Eklund. Asymmetric page split generalized index search trees for formal concept analysis. In ISMIS06, Proceedings, Lecture Notes in Computer Science, page FIXME. Springer, 2006.

[75] Ben Martin and Peter W. Eklund. Spatial indexing for scalability in fca. In Rokia Missaoui and Jürg Schmid, editors, ICFCA, volume 3874 of Lecture Notes in Computer Science, pages 205–220. Springer, 2006.

[76] Ben Martin and Peter W. Eklund. Custom asymmetric page split generalized index search trees and formal concept analysis. In ICFCA, 2007.

[77] Bill McCarty. SELinux: NSA's Open Source Security Enhanced Linux. O'Reilly & Associates, Sebastopol, California, 2004.

[78] Jim Melton. Advanced SQL, 1999 : understanding object-relational and other advanced features. Morgan Kaufmann Pub., Boston, Mass., 2003.

[79] Mohammed J. Zaki, Nagender Parimi, Nilanjana De, Feng Gao, Benjarath Phoophakdee, Joe Urban, Vineet Chaoji, Mohammad Al Hasan and Saeed Salem. Towards generic pattern mining. In Bernhard Ganter and Robert Godin, editors, Concept Lattices, Third International Conference on Formal Concept Analysis, ICFCA 2005, Proceedings, Lecture Notes in Computer Science, pages 1–20, Lens, France, 2005. Springer.

[80] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, Departure of Computer Science, University of Maryland, College Park, MD, 1995.

[81] Hans-Peter Kriegelm Ralf Schneider Norbert Beckmann and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Proc. ACM-SIGMOD International Conference on Management of Data, Atlantic city, N.J., 1990.

[82] Yoann Padioleau and Olivier Ridoux. A logic file system. In USENIX 2003 Annual Technical Conference, pages 99–112, 2003.

[83] Jian Pei. Pattern-growth methods for frequent pattern mining, ph.d. thesis, computing science, simon fraser university, 2001.

[84] Andrea Pietracaprina and Dario Zendolin. Mining frequent itemsets using patricia tries. In Bart Goethals and Mohammed J. Zaki, editors, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations. Ceur, 2003.

[85] Shelley Powers. Practical RDF. O'Reilly & Associates, Sebastopol, California, 2003.

[86] Susanne Prediger. Logical scaling in formal concept analysis. In International Conference on Conceptual Structures, pages 332–341. Springer, 1997.

[87] Susanne Prediger. Symbolic objects in formal concept analysis. In Proceedings of the Second International Symposium on Knowledge, Retrieval, Use and Storage for Efficiency, 1997.

[88] R. Agrawal, H. Mannila, R Srikant, H. Toivonen and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad et al., editor, Advances in Knowledge Discovery and Data Mining, pages 307–328, Menlo Park CA, 1996. AAAI Press.

[89] Shamkant B. Navathe Ramez Elmasri. Fundamentals of database systems. Pearson/Addison Wesley, New York, 2004.

[90] Marc Rochkind. Advanced Unix Programming. Addison Wesley Professional, 2004.

[91] T. Rock and R. Wille. Ein TOSCANA-erkundungssytem zur literatursuche. In G. Stumme and R. Wille, editors, Begriffliche WissensveraRbeitung: Methoden und Anwendungen, pages 239–253, Berlin-Heidelberg, 2000. Springer-Verlag.

[92] Mark Rosen. E-mail classification in the haystack framework, 2003. MIT, Masters thesis, describes the design and implementation of a text classification framework for the Haystack project.

[93] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: Towards a unified standard. pages 47–64.

[94] Ravi S. Sandhu. Lattice-based access control models. IEEE Computer, 26(11):9–19, 1993.

[95] Robert Sedgewick. Algorithmn in C++ third edition. Addison-Wesley, Reading, Massachusetts 01867, 1998.

[96] Gerd Stumme, Rafik Taouil, Yves Bastide, Nicolas Pasquier, and Lotfi Lakhal. Computing iceberg concept lattices with titanic. In J. on Knowledge and Data Engineering (KDE), volume 42, pages 189–222, 2002.

[97] Tim Hannan, Alex Pogel. Spring-Based Lattice Drawing Highlighting Conceptual Similarity. In Missaoui and Schmid, editors, Concept Lattices, Third International Conference on Formal Concept Analysis, ICFCA 2006, Proceedings, LNAI, pages 264–279. Springer, 2006.

[98] Dan Tow. SQL Tuning. O'Reilly & Associates, Sebastopol, California, 2004.

[99] Eric van der Vlist. XML Schema. O'Reilly & Associates, Sebastopol, California, 2002.

[100] Eric van der Vlist. Relax-NG. O'Reilly & Associates, Sebastopol, California, 2003.

[101] Eric van der Vlist. Schematron. O'Reilly & Associates, Sebastopol, California, 2003.

[102] Jos Warmer and Anneke Kleppe. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1998.

[103] Horst F. Wedde and Mario Lischka. Composing Heterogenous Access Policies between Organizations. In Proceedings of the IADIS International Conference e-Society 2003, Lisbon/ Portuagal, June, 3-6 2003. International Association for Development of the Information Society.

[104] Ian H. Witten, Alistar Moffat, and Timothy C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, 340 Pine Street, San Francisco, CA 94104-3205, USA, 1999.

[105] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. Linux Journal, (128):24–29, December 2004.

[106] Woo Suk Yang, Yon Dohn Chung, and Myoung Ho Kim. The rd-tree: a structure for processing partial-max/min queries in olap. Inf. Sci. Appl., 146(1-4):137–149, 2002.

[107] Mohammed J. Zaki and Ching jui Hsiao. Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Transactions on Knowledge and Data Engineering, 17:2005, 2005.

[108] Mohammed Javeed Zaki. Scalable algorithms for association mining. In Knowledge and Data Engineering, volume 12, pages 372–390, 2000.