1-1-2006

Spatial indexing for scalability in FCA

Benjamin Martin
University of Queensland, bmm265@uow.edu.au

Peter W. Eklund
University of Wollongong, peklund@uow.edu.au

Spatial indexing for scalability in FCA

Abstract

The paper provides evidence that spatial indexing structures offer faster resolution of Formal Concept Analysis queries than B-Tree/Hash methods. We show that many Formal Concept Analysis operations, computing the contingent and extent sizes as well as listing the matching objects, enjoy improved performance with the use of spatial indexing structures such as the RD-Tree. Speed improvements can vary up to eighty times faster depending on the data and query. The motivation for our study is the application of Formal Concept Analysis to Semantic File Systems. In such applications millions of formal objects must be dealt with. It has been found that spatial indexing also provides an effective indexing technique for more general purpose applications requiring scalability in Formal Concept Analysis systems. The coverage and benchmarking are presented with general applications in mind.

Keywords

Spatial, indexing, for, scalability, FCA

Publication Details

Martin, B. & Eklund, P. W. (2006). Spatial indexing for scalability in FCA. In B. Ganter & L. Kwuida (Eds.), International Conference Formal Concept Analysis Conference (pp. 205-220). Berlin: Springer-Verlag.

Spatial Indexing for Scalability in FCA

Ben Martin[1] and Peter Eklund[2]

[1] Information Technology and Electrical Engineering
The University of Queensland
St. Lucia QLD 4072, Australia
monkeyiq@users.sourceforge.net
[2] School of Economics and Information Systems
The University of Wollongong
Northfields Avenue, Wollongong, NSW 2522, Australia
peklund@uow.edu.au

Abstract. The paper provides evidence that spatial indexing structures offer faster resolution of Formal Concept Analysis queries than B-Tree/Hash methods. We show that many Formal Concept Analysis operations, computing the contingent and extent sizes as well as listing the matching objects, enjoy improved performance with the use of spatial indexing structures such as the RD-Tree. Speed improvements can vary up to eighty times faster depending on the data and query. The motivation for our study is the application of Formal Concept Analysis to Semantic File Systems. In such applications millions of formal objects must be dealt with. It has been found that spatial indexing also provides an effective indexing technique for more general purpose applications requiring scalability in Formal Concept Analysis systems. The coverage and benchmarking are presented with general applications in mind.

1 Introduction

A common approach to document retrieval using Formal Concept Analysis is to convert associations between many-valued attributes and objects into binary associations between the same objects and new attributes. For example, a many-valued attribute showing a person's income may be converted into three attributes: low, middle and upper which are then associated with the same set of people. The method of associating binary attributes is called either conceptual scaling [10] or logical scaling [20] depending on the perspective chosen.

This is the approach adopted in the ZIT-library application developed by Rock and Wille [22] as well as the Conceptual Email Manager [8, 6]. The approach is mostly applied to static document collections (such as newsclassifieds) as in the program Rfca [7] but also to dynamic collections (such as email) as in Mail-Sleuth [2] and files in the Logical File System (LISFS) [19]. In all but the latter two the document collection and full-text keyword index are static. Thus, the FCA interface consists of a mechanism for dynamically deriving binary attributes from a static full-text index. Many-valued contexts are used to materialize formal contexts in which objects are document identifiers.

The motivation for the application of spatial structures in this research was for the use of Formal Concept Analysis in a virtual filesystem [11, 19, 16]. In particular the libferris [1] 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. We show that the spatial method proposed in this paper has performance which depends on the number of attributes in each query as well as the density and distribution of the formal context. This paper presents some of the solutions to the problems that arise when integrating Formal Concept Analysis (FCA) [10] into a Semantic File System (SFS) [11, 19, 16] and demonstrates performance improvements that result from the implementation of a spatial indexing structure.

2 Existing Indexing Strategies for Formal Concept Analysis

Two designs dominate current 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) [22] or using inverted files (LISFS) [19]. The libferris design is based on the former with extensions to deal with normalization and the association of emblems [15]. An emblem is a pictorial annotation, usually a small icon, that is associated with an file or directory. An emblem often denotes a category.

Shown in Fig. 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.

We now focus on systems using 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 we refer 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 Fig. 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

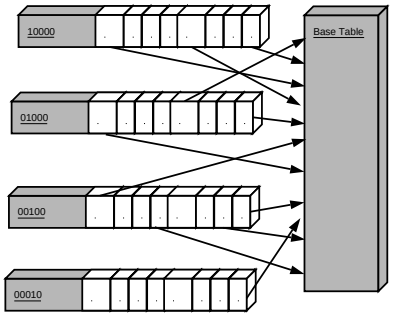


Fig. 1. An inverted file index. For each value of interest there is a list containing all the addresses of tuples which match that value.

	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
...	...	...	...	

Fig. 2. Example base relation containing modification and size data for objects.

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 Fig. 3. For example, from in Fig. 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`[3]. Such an attribute would have an attribute extent containing 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 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* ...".

Current best practice in the Formal Concept Analysis community attempts to assist such queries with B-Tree indexes over subsets of $\{c_1, c_2, ..., c_y\}$. We now discuss how relational databases use B-Tree indexes during query execution.

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

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*
...	...	...

Fig. 3. Ordinal scales on the size, modification and access times of the objects in the base table. Nominal scale on the file-owner.

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 [24]. 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 [9]. 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.

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 [24]. 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.

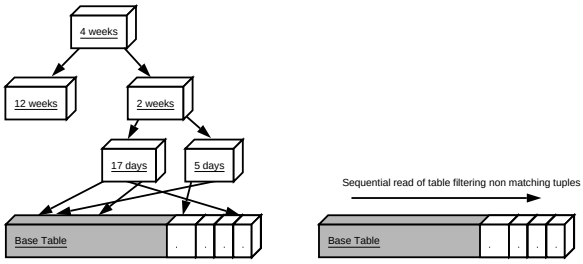


Fig. 4. On the left: B-Tree index on a date column for the base table. Dates in nodes are shown as how long before the current time they represent. The upper nodes are index nodes with the nodes below "12 weeks" omitted. The 17 and 5 days nodes are leaf nodes of the index which point at records in the 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.

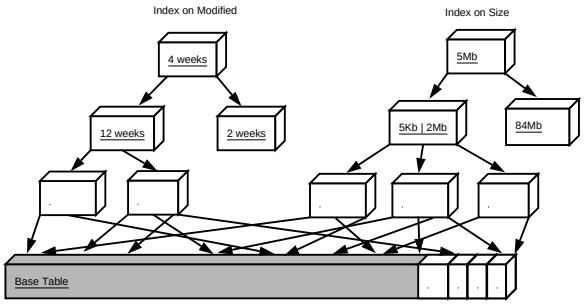


Fig. 5. Two B-Tree indexes on the base table. The left index is on *modified* while the right index is on *size*. Leaf nodes in both indexes point to tuples physically located throughout the base table.

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 first. 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 [24]. 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 cannot 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.

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.

When resolving an SQL query against a base table most relational databases will only consider using a single index [24]. 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.

3 Spatial Indexing for Formal Concept Analysis

We now turn to the application of spatial methods to improve index utilization in query resolution. First we consider using indexes on SQL expressions and then show how spatial methods can be applied to expression indexes to improve performance.

Many relational databases allow the creation of indexes on expressions [3]. 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 Fig. 3. The expression index e_1 on attribute a_1 is shown in Fig. 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 $\cup_1^n e_n$ due to selectivity constraints, highlighted in Section 2. One can however consider indexing structures over the collected $\{e_1, ..., e_n\}$ 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 [14] 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 $S = \{e_1, e_3, e_7\}$.

An indexing structure motivated by the spatial indexing structure, the R-Tree [12, 18], caters for subset queries: the RD-Tree [13, 25]. 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. We first describe 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 Fig. 7). This transitive containment relation is the heart of the R-Tree. R-Trees are not limited to 2 or 3 dimensional data but 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 Fig. 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

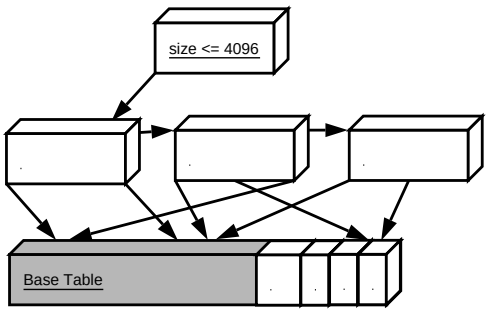


Fig. 6. Expression index on attribute a_1 using f_1, the SQL predicate *size* <= 4096.

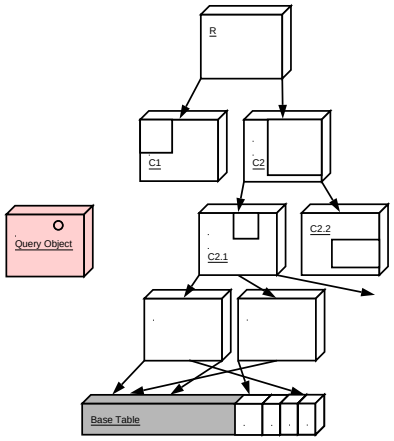


Fig. 7. An example R-Tree with a query object on the left. Each node has a bounding box which fully contains all objects in its child nodes. An implementation stores the bounding box for each child in the parent node. Note the example is limited to 2 dimensional space with a low branching factor for presentation purposes.

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 Fig. 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 [14, 13]. As described above a subset query seeks all objects 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 Fig. 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

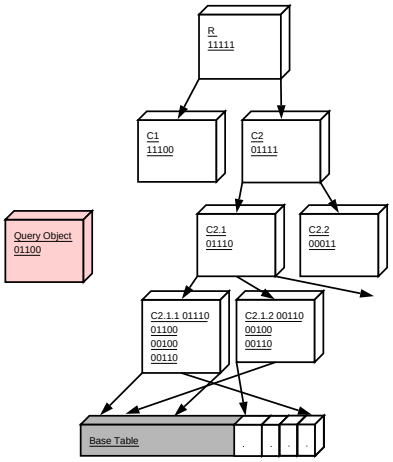


Fig. 8. An example RD-Tree with a query object on the left. Each node has a bounding set associated which fully contains all objects in its child nodes. An implementation would store the bounding set for each child in the parent node. Note that the example is limited to only a small set size with a low branching factor in the tree for presentation.

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 Fig. 1).

An overlap query seeks objects which have more than a given number of attributes in common with the query [25]. To efficiently find the contingent size the RD-Tree index must also contain the hamming weight of the binary expression indexes $\cup_1^n e_n$ (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 Fig. 9.

4 Performance Analysis

The benchmark system is 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 include 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.

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

Fig. 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.

Testing was completed on 3 different input data sets: various synthetic formal contexts generated with the IBM synthetic data generator [21], the mushroom and covtype databases from the UCI dataset [4] and a formal context derived from the metadata of 67,000 document files [1]. 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.

4.1 Performance on the UCI Mushroom dataset

The attributes used from the mushroom table are shown in Fig. 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 sets modeling the use case of someone interactively creating and modifying

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

Fig. 10. Selected attributes for the mushroom table and the number of tuples which have the given attribute-value combination.

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

Fig. 11. Times with hot and cold caches to complete queries for 8 attribute list context. Times are in seconds.

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 Fig. 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 $\cup_1^n e_n$ as described in Section 3.

4.2 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 Fig. 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.

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

Fig. 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.

4.3 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. 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 Fig. 13. Without the use of a special purpose index structure the database degenerated to a sequential table scan for almost every query.

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

Fig. 13. Times in seconds with hot and cold caches to complete queries.

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.

4.4 Performance on Synthetic data

The following use synthetic data generated with the IBM synthetic data generator [21]. 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

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

Fig. 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.

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 [23, 17]. 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 Fig. 14 and graphically in Fig. 15. Size statistics for the DBMS tables and indexes are shown in Fig. 16.

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 Fig. 16. The results with 128 items per itemset are the same as those in Fig. 14.

5 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 [5].

Although the use of spatial indexing was first adopted and implemented to allow Formal Concept Analysis to be applied specifically to the special circum-

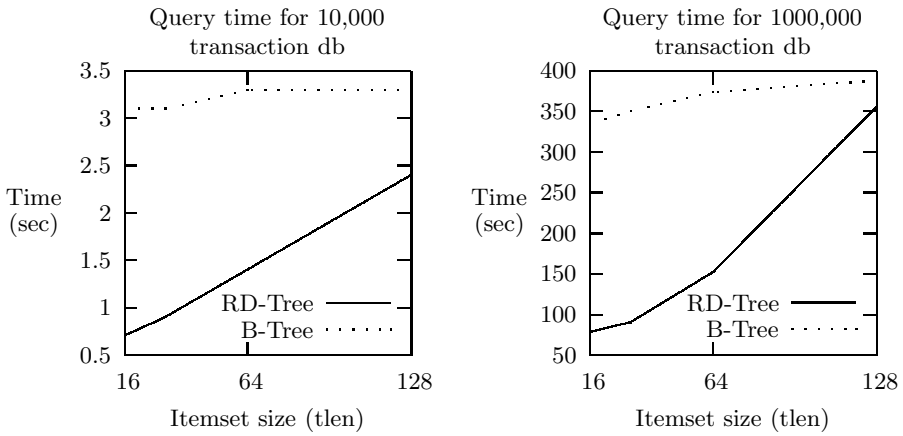


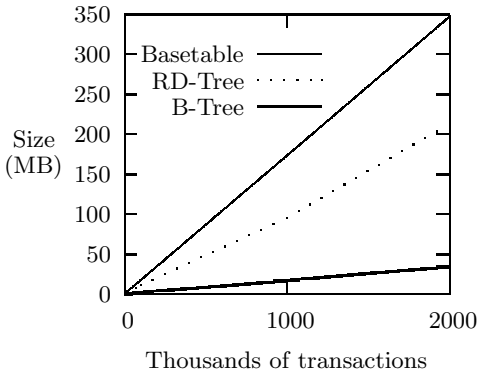
Fig. 15. Execution times for queries using either B-Tree or RD-Tree indexing against databases of varying density.

stances 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. 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.

References

1. libferris, http://witme.sourceforge.net/libferris.web/. Visited Nov 2005.
2. Mail-sleuth homepage, http://www.mail-sleuth.com/. Visited Jan 2005.
3. Postgresql, http://www.postgresql.org/. Visited June 2004.
4. 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.
5. Claudio Carpineto and Giovanni Romano. *Concept Data Analysis*. Wiley, England, 2004.
6. Richard Cole and Peter Eklund. Analyzing an email collection using formal concept analysis. In *European Conf. on Knowledge and Data Discovery, PKDD'99*, number 1704 in LNAI, pages 309–315. Springer Verlag, 1999.
7. Richard Cole and Peter Eklund. Browsing semi-structured web texts using formal concept analysis. In *Proceedings 9th International Conference on Conceptual Structures*, number 2120 in LNAI, pages 319–332. Springer Verlag, 2001.



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

Fig. 16. (Left) 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. (Right) 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.

8. Richard Cole and Gerd Stumme. Cem: A conceptual email manager. In *7th International Conference on Conceptual Structures, ICCS'2000*. Springer Verlag, 2000.
9. Michael J. Folk and Bill Zoelick. *File Structures*. Addison-Wesley, Reading, Massachusetts 01867, 1992.
10. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer–Verlag, Berlin Heidelberg, 1999.
11. 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.
12. Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston Mass, 1984.
13. Joseph M. Hellerstein and Avi Pfeifer. The RD-Tree: An Index Structure for Sets, Technical Report 1252. University of Wisconsin at Madison, October 1994.
14. S. Helmer. Index structures for databases containing data items with setvalued attributes, 1997.
15. Ben Martin. File system wide file classification with agents. In *Australian Document Computing Symposium (ADCS03)*. University of Queensland, 2003.
16. 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.
17. 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.

18. 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.
19. Yoann Padioleau and Olivier Ridoux. A logic file system. In *USENIX 2003 Annual Technical Conference*, pages 99–112, 2003.
20. Susanne Prediger. Logical scaling in formal concept analysis. In *International Conference on Conceptual Structures*, pages 332–341. Springer, 1997.
21. 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.
22. 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.
23. 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.
24. Dan Tow. *SQL Tuning*. O'Reilly & Associates, Sebastopol, California, 2004.
25. 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.