Efficient Browsing and Update of Complex Data Based on the Decomposition of Contexts

Sébastien Ferré

Université de Rennes 1, CNRS
Campus de Beaulieu, 35042 Rennes cedex, France
ferre@irisa.fr

Abstract. Formal concept analysis is recognized as a good paradigm for browsing data sets. Besides browsing, update and complex data are other important aspects of information systems. To have an efficient implementation of concept-based information systems is difficult because of the diversity of complex data and the computation of conceptual structures, but essential for the scalability to real-world applications. We propose to decompose contexts into simpler and specialized components: *logical context functors*. We demonstrate this allows for scalable implementations, updatable ontologies, and richer navigation structures, while retaining genericity.

1 Introduction

Formal Concept Analysis (FCA) [GW99] has been recognized as a good paradigm for browsing data sets [GMA93, CES03, FR04]. Besides browsing (querying and navigation), update is another important aspect of information systems. FCA is defined on binary relations between objects and attributes. Those relations are called *formal contexts*. In practice, data is generally more complex than the simple attributes of formal contexts: e.g., numbers and intervals, strings and patterns, valued attributes [GW89], vectors, trees, graphs [GK01]. Furthermore, logical dependencies may exist in complex data: e.g., if an object has the property $age = 23$, it implicitly has the more general property $age \in [20, 30]$. A first approach to handle complex data in FCA is *conceptual scaling* [GW89], which is a process that takes complex data as an input, and outputs a standard formal context, called the *scaled context*. In the scaled context, the attributes are abstraction of the original data, and the incidence relation reflects their internal logic. For example, Prediger et al [PS99] use description logics to define the meaning a finite set of chosen attributes; and Tane et al [TCH06] use the same description logics to compute scaled contexts as views over a complex knowledge base. A second approach [CM00, FR00, GK01] strives to keep complex data in its original form by generalizing the definition of a formal context as well as other FCA operations (e.g., Galois connection, concept lattice). For example, Logical Concept Analysis (LCA) [FR00] uses logical formulas instead of sets of attributes to represent and manipulate object descriptions, concept intents, queries, and navigation links between concepts. The first approach allows the reuse of FCA algorithms and tools on complex data, while the second approach keeps the original form of complex data, and entails no loss of information.

This paper discusses the efficient implementation of concept-based information systems. In the two above approaches, the existing implementations make no or little use of the specificities of complex data. For example, the data structure used to represent contexts in LCA, called the *logic cache* [FR04], makes use of the logical entailment between formulas, but makes no difference between a string, an interval or a graph. This entails the following problems: (1) the update of the context is not efficient enough to support scalability (10,000 objects at most), (2) any change in the logic requires the complete recomputation of the logic cache, and (3) the set of navigation links is not informative enough.

We propose to benefit from the nature of the original complex data to solve the above problems. For instance, there exist dedicated data structures and algorithms for strings, which can be used to build specialized implementations of contexts where objects are described by strings. The same can be done for other concrete domains, or taxonomies. Now, if objects are described by string-valued attributes, a specialized implementation can be composed from two specialized contexts: one for attributes, and the other for strings. Operations for composing contexts have been defined in FCA [GW99]: e.g., apposition, direct product. However, they apply to formal contexts only, and their implementation is not discussed. We detail in this paper the definition and specialized implementation of both primitive contexts and composition operations. Those are collectively called *logical context functors*, because they are functions (with zero, one or several arguments) from logical contexts to logical contexts. The term *functor* is taken from the domain of functional programming where it denotes functions from modules to modules [Mog89], which are precisely used to implement our functors.

Section 2 recalls the basics of LCA, and introduce the browsing and update operations. Section 3 explains the problems of the logic cache, the existing LCA implementation. Section 4 defines a *logical context functor* as an extension of a *logic functor*, and details five functors: string, taxonomy, product, disjoint union, and *root*. Section 5 illustrates the use of logical context functors on two real examples (string-valued attributes, and user-tag annotations), and demonstrates their efficiency by giving the complexity of operations compared to the logic cache. For instance, the addition of an object into a context of n objects is in $O(1)$ or $O(\ln(n))$ instead of $O(n)$.

2 Logical Contexts and Logical Information Systems

The LCA framework [FR04] applies to logics with a set-valued semantics similar to description logics [BCM$^+$03]. The logic is not fixed *a priori* so that it can be customized to different applications. Examples of logical formulas are binary attributes, attributes valued on different concrete domains (e.g., strings, intervals, dates), terms from taxonomies, and any combination of those such as lists, trees or graphs. It is sufficient here to define a logic (see [FR04] for a detailed presentation) as a pre-order of formulas. The pre-ordering is the logical entailment, called *subsumption*: e.g., an interval included in another one, a string matching some regular expression, a graph being a subgraph of another one.

Definition 1 (logic). *A* logic *is a pre-order $\mathcal{L}_T = (L, \sqsubseteq_T)$, where L is a set of formulas, T is a customizable parameter of the logic, and $\sqsubseteq_T$ is a* subsumption *relation that depends on T. The relation $f \sqsubseteq_T g$ reads "f is more specific than g" or "f is subsumed by g", and is also used to denote the partial ordering induced from the pre-order.*

The parameter T helps to take into account domain knowledge that may change over time: e.g., an ontology, a taxonomy. In the following, for simplicity, we designate this parameter as the "ontology", and consider it is a set of *subsumption axioms* $f \prec g$: e.g., $cat \prec animal$, $Quebec \prec Canada$. In addition to the logic and its ontology, the *logical context* constitutes the third level of knowledge. It defines a set of objects along with their logical description, and a finite subset of formulas, called *vocabulary*, that is used for navigation.

Definition 2 (logical context). *A* logical context *is a tuple $K = (\mathcal{O}, \mathcal{L}_T, X, d)$, where $\mathcal{O}$ is a finite set of objects, $\mathcal{L}_T$ is a logic, $X \subseteq L$ is a finite subset of formulas called the* navigation vocabulary*, and $d \in (\mathcal{O} \rightarrow \mathcal{L}_T)$ is a mapping from objects to logical formulas. For any object o, the formula $d(o)$ denotes the* description *of o.*

Each formula is described by a single formula for genericity reasons. Even if a description is often in practice a set of properties, it can also be a sequence of properties or any other data structure. The definition of a vocabulary is necessary because there is often an infinite set of formulas (e.g., intervals, strings). The choice of a relevant vocabulary depends on both the logic and object descriptions, and a contribution of this paper is precisely to show how it can be automatically generated in a logical context. The elements of the vocabulary are called *features*.

Logical contexts make up the core of Logical Information Systems (LIS) [FR04], so that we need both update and information retrieval operations on them. Update operations apply to the ontology, the objects, and the navigation vocabulary. For every formulas $f, g \in L$:

- $K.axiom(f, g)$ adds the axiom $f \prec g$ to the ontology T, which modifies the behaviour of the subsumption $\sqsubseteq_T$;
- $K.add(o, f)$ adds the new object o to the set of objects $\mathcal{O}$, and sets its description $d(o)$ to f;
- $K.show(f)$ adds the formula f to the navigation vocabulary X.

The *filling of a context* is the successive addition of a set of objects, defining the updatable part of the context. There are also update operations for removing axioms, modifying the description of objects, removing objects, and hiding formulas, but we do not detail them here.

A key feature of LIS, shared by other concept-based information systems [GMA93, DVE06], is to allow the tight combination of querying and navigation. The principle is that, instead of returning a ranking of all the answers to the query, the system returns a set of query *increments* that suggest to users relevant ways to refine the query, i.e., navigation links between concepts, until a manageable amount of answers is reached. They are two information retrieval operations on logical contexts: one to compute the query answers, and another to compute the query increments from those answers.

A query is a logical formula, and its answers are defined as the extent of this formula, i.e., the set of objects whose description is subsumed by this formula.

Definition 3 (extent). *Let K be a logical context, and $q \in \mathcal{L}$ be a query formula. The* extent *of q in K is defined by $K.ext(q) = \{o \in \mathcal{O} \mid d(o) \sqsubseteq_T q\}$.*

The increments of a query q are the features that are frequent in the extent of q, i.e., the features whose extent shares a pre-defined minimum m of objects with the extent of q: $\{y \in X \mid |K.ext(q) \cap K.ext(y)| \geq m\}$. Those increments are partially ordered by subsumption, and should be presented so to users because this gives a more comprehensive view. For instance, if the vocabulary contains continents, countries, and regions, it is better to display them as a tree rather than a flat list. Moreover, it is not necessary to compute and display all of them at once; continents should be displayed first, and could then be expanded on demand to display countries, etc. So, the navigation operation takes an increment x and returns its lower neighbours that are also increments. For technical reasons, we prefer to compute increments w.r.t. a set of objects $O \subseteq K.ext(x)$ instead of a query. Starting with a query q, that set O is defined as $K.ext(q) \cap K.ext(x)$. Each increment is returned with the extent of the concept that would be reached by using it as a navigation link.

Definition 4 (children increments). *Let K be a logical context, $x \in X$ be an increment, O be a set of objects s.t. $O \subseteq K.ext(x)$, and m be a frequency threshold. The* children increments *of x, called the* parent increment*, w.r.t. O at threshold m is defined by $K.incrs(x, O, m) =$*

$$Max_{\sqsubseteq_T}\{(y, O') \mid y \in X,\ y \sqsubseteq_T x\ ,\ x \not\sqsubseteq_T y\ , O' = O \cap K.ext(y), |O'| \geq m\},$$

where $\sqsubseteq_T$ is trivially extended to pairs (y, O').

These increments provide feedback on the current query and answers, as well as several forms of navigation [Fer09]. LIS have been applied to many kinds of data, and most noticeably to the management of a collection of > 5000 photos [Fer09], which can be browsed and updated in terms of time, location, event, persons, and objects.

3 Problems with Logic Caches

Logical information systems were designed to be generic, and so can make no assumption on the logic, except for the existence of a decision procedure for subsumption (decidability). Because this subsumption test is costly for some expressive logics, the choice was made to minimize its use in browsing operations, which are more frequent than update operations. Therefore the cost of computing subsumption is moved to update operations, in the form of an incremental preprocessing, whose result is called a *logic cache* [FR04].

There are a number of problems with logic caches as a generic implementation. The first problem is that efficient browsing has been achieved at the cost of a quadratic complexity for the building of a logic cache. This is more acceptable to users than lengthy response times in browsing operations, but this strongly limits the scalability of LIS. An appropriate complexity would be $O(n)$ or at most $O(n \ln(n))$. The second problem is the operation $K.axiom$ that requires a complete recomputation of the logic cache. This is because the impact of new axioms on the subsumption is only known by the logic, and not by the generic logic cache. In order to take real profit of changing ontologies, the ability to handle them incrementally is crucial. The third problem is in the design of the vocabulary (the set of scale attributes in conceptual scaling). There is a conflict between having a rich and progressive navigation that requires a large vocabulary, and the efficient filling of contexts that requires a small vocabulary. Consider the example where objects are documents, and object descriptions are titles, i.e. strings. If the vocabulary contains only full titles, the navigation structure is a flat list of titles, which is not very interesting. If the vocabulary contains all words occuring in the titles, the navigation structure is more interesting, but: (1) it still misses composed keywords such as “formal concept analysis”; (2) it contains unique words that are specific to one title, and in this case it is better to show the full title; (3) it is costly compared to the small vocabulary.

Consider another context where documents are described by a set of pairs (user, tag), and users/tags are organized in two taxonomies. The taxonomy of users defines various groups over users (possibly overlapping), and the taxonomy of tags defines a generalization ordering on tags. This context enables different kinds of queries: Which documents have been given this kind of tags by this kind of users ? Which tags have been given by this kind of users on this set of documents ? Which users have given this kind of tags on this set of documents ? Even if each taxonomy has a reasonable size, say 1000, the vocabulary will contain all pairs (user, tag) that have a non-empty extent, which can go up to 1 million pairs. Intuitively, it should be possible to keep each taxonomy on its side as a kind of partial logic cache, and to combine them on the fly, thus bounding the size of the total logic cache to the sum rather than the product of taxonomies sizes. In the next section, this decomposition is formalized by *logical context functors*.

4 Logical Context Functors

We introduce in this section *logical context functors*, i.e. functions that build logical contexts and their operations from simpler parts. We show how they solve the problems presented in the previous section: efficiency, ontology evolution, and selection of the navigation vocabulary. The problem of efficiency comes from the well-known trade-off between genericity and efficiency. Efficiency requires specific data structures and algorithms, which is *a priori* incompatible with the need for genericity. A solution, that has already been done and validated on logics [FR04], is to define specific components, called *functors*, and to allow their composition in more and more complex components, in which the correction of operations w.r.t. semantics is automatically verified [FR06]. The efficiency comes from the specificity of functors, and the genericity comes from the ability to compose them. This is similar to languages where a finite number of different words can be combined in an infinity of sentences. Of course, the expressivity is limited by the available functors, but new functors can always be added to the set. Another benefit of logic functors is to permit the choice of the *right* logic for each application, instead of having an all-purpose logic that is costly to use, and cannot cover the specific needs of all applications.

Definition 5 (logic functor). *A* logic functor *is a function $\mathcal{F}$ that takes logics $\mathcal{L}_1, ..., \mathcal{L}_n$ as arguments ($n \geq 0$), returns a composed logic $\mathcal{L}_T = \mathcal{F}(\mathcal{L}_1, ..., \mathcal{L}_n)$. As for any logic, one has $\mathcal{L}_T = (L, \sqsubseteq_T)$, but L (resp. $\sqsubseteq_T$) is function of the sets of formulas (resp. subsumption) of arguments logics.*

Examples of logic functors are *String* that takes no argument, and returns the set of all strings ordered by the subsumption relation "contains"; and $\mathit{Prod}(\mathcal{L}_1, \mathcal{L}_2)$ that takes two arguments, and returns the logic where formulas are pairs (f_1, f_2) of formulas from $\mathcal{L}_1$ and $\mathcal{L}_2$, and the subsumption test is naturally decomposed in the two subsumption tests. These examples and others are more formally defined below, along with logical context functors. Logical context functors are defined similarly to logic functors, as functions from contexts to contexts.

Definition 6 (logical context functor). *A* logical context functor *is a function F that takes logical contexts $K_1, ..., K_n$ as arguments ($n \geq 0$), returns a composed context $K = F(K_1, ..., K_n)$. As for any context, one has $K = (\mathcal{O}, \mathcal{L}_T, X, d)$, but each part of this context is function of the respective parts of argument contexts.*

Logical context functors and logic functors are implemented as *functors*[1] in the Objective Caml programming language. In order to reuse code from logics into contexts, logical context functors *inherit* (in the object-oriented sense) from their respective logic functors.

In the following we detail a few common logical context functors. For each functor we define the set of objects, the set of formulas, the subsumption, the vocabulary, and the description mapping as a function of argument contexts. Besides this mathematical point of view, each functor is also given data structures and algorithms for the implementation of logical context operations. Complexities are given in function of the number n of objects, the size x of the vocabulary, and the number i of children increments.

[1] Similar structures exist in other programming languages: e.g., parameterized modules (ML), generic classes (Java), templates (C++).

4.1 String Context

The logical context functor *String* represents the concrete domain of strings and substrings ordered by string inclusion. It has no argument, so that it constitutes a logical context that can be used alone or as part of a more complex logical context. This logical context could also be defined as *Seq*(*Char*), where *Seq* and *Char* would be two functors respectively about sequences and characters.

The logic is simply defined as $\mathcal{L} = (\Sigma^*, \supseteq)$, where Σ^* stands for finite strings over an alphabet Σ, and $\supseteq$ is the "contains" relation on strings. It is not parameterized by an ontology because it is a concrete domain. It enables to describe objects by strings, and to query them with patterns like `contains "FCA"`. A previous work [Fer07] has shown that a concise and complete vocabulary X can be efficiently and automatically extracted from such a context: the set of *maximal substrings*. This vocabulary is complete because the set of formal concepts it generates is the same as when considering all possible substrings. It is concise because its size is bounded by the cumulated size kn of strings describing the n objects in K, where k is the average length of strings. It can be computed in $O(kn \ln(kn))$. In practice, however, the number of maximal substrings is generally much lower than kn (e.g., 3,816 instead of 52,360 [Fer07]).

The data structure that permits to store the string-description of objects, and to compute the vocabulary, is an extended *suffix tree* [Fer07]. Its complexity in space is in $O(kn \ln(kn))$. It naturally provides the efficient incremental addition of a string, and hence the addition of an object in the context: $K.add(o, s)$ takes $O(|s| \ln(kn))$ time, where $|s|$ is the length of the string s. Other operations can also be directly performed on the extended suffix tree. For the operation $K.show(s)$ one reads the string s down the suffix tree, in $O(|s|)$ time, and marks the reached node as visible. For the operation $K.ext(s)$, one also reads the string s, and then collects the objects below the reached node: $O(|s| + n)$ time. For the operation $K.incrs(x, O, m)$, one again reads the string s, then follows links from the reached node to find the smallest maximal substrings below, and finally filters those that are frequent in O: $O(|s| + ni)$ time.

The most visible advantage of the functor, compared to the logic cache, is the computation of a rich, yet concise, vocabulary that provides a much better navigation feeling because it dynamically adapts to the context contents. Another advantage is the efficiency in the filling of the context: $O(kn \ln(kn))$ instead of $O(k^4 n^2)$. Even when the vocabulary is restricted to the strings describing objects, the complexity of the logic cache is in $O(kn^2)$, still quadratic in the number of objects. This makes a huge difference in practice when the number of objects gets high, and this efficiency comes with a much larger and useful vocabulary.

4.2 Taxonomy Context

Taxonomies are very helpful in the organization of a collection of documents [Sac00]. They are a simple kind of ontologies in which an axiom states that a term is more specific than another. Examples of such axioms are $inQuebec \prec inCanada$, or $cat \prec mammal$, $mammal \prec animal$. The logical context functor *Taxo* produces contexts in which each object is described by one term, and implicitly has all more general terms. For instance, an object described as a *cat* is also an instance of *mammal*, and *animal*. The logic of those contexts is naturally derived from the taxonomy T that parameterizes it.

Definition 7 (taxonomy logic). *Let* $T = (L, A)$ *be a taxonomy, where* L *is the set of terms, and* A *is a set of taxonomic axioms* $f \prec g$. *The logic of functor Taxo is defined by* $\mathcal{L}_T = (L, \sqsubseteq_T)$, *where the subsumption* $\sqsubseteq_T$ *is defined as the reflexive and transitive closure of the relation* $\prec$.

Because the set of terms of a taxonomy is finite, the vocabulary can be defined as $X = L$, ensuring it to be complete w.r.t. the generation of concepts. The data structure used in the implementation of *Taxo* is the taxonomy seen as a graph: nodes are terms from X, edges are the taxonomic axioms from T, and terms are labelled by their extent. We call *ancestors* of a term f all terms that subsume f (including f), which can be found in the graph by transitively collecting successors.

The addition of a new object, $K.add(o, f)$, consists in adding the term f into X with an empty extent if it is not yet in X, then in adding o to the extent of every ancestor of f. This operation is in $O(a)$ with a the number of ancestors. The operation $K.show(f)$ is unnecessary because every relevant formula is already in the vocabulary. The addition of an axiom $K.axiom(f, g)$ consists in adding an edge from f to g in the graph, and adding the extent of f to the extent of every ancestor of g. This operation is in $O(na)$. It must be noted here that, contrary to a logic cache, it is not necessary to completely recompute the data structure to add an axiom; new axioms can be processed incrementally. This is possible because the functor *Taxo* has knowledge about its logic, and so can update its data structures in parallel to updates in the logic. The operation $K.ext(f)$ consists in a simple access to the label of f in the graph, and is therefore in $O(1)$. The operation $K.incrs(x, O, m)$ consists in filtering among the predecessors of x in the graph those that are frequent in O. This operation is in $O(ni)$.

The shape of the taxonomy has an influence on the complexity of operations. If we assume the taxonomy is a balanced tree with arity k ($a = \ln_k(x)$), the filling of a context is in $n \ln_k(x)$, and the computation of increments is in nk. This suggests to find a compromise between flat taxonomies (small a) and deep taxonomies (large a).

4.3 Product of Contexts

Section 3 presents two examples of contexts, where the formulas can be decomposed as a pair of subformulas. The first example shows valued attributes made of an attribute and a (sub)string; the second example shows pairs (user,tag), where each part is placed into a taxonomy. Instead of developping two new logical context functors, it is more useful to define a logical context functor $Prod(K_1, K_2)$ that produces the product of two simpler contexts. Then, the first example is simply composed as $Prod(Taxo, String)$, and the second example as $Prod(Taxo, Taxo)$. We recall that in logical contexts, each object is described by a single formula, here a pair.

Definition 8 (product of contexts). *Let* $K_1 = (\mathcal{O}, \mathcal{L}_1, X_1, d_1)$ *and* $K_2 = (\mathcal{O}, \mathcal{L}_2, X_2, d_2)$ *two logical contexts sharing a same set of objects. The* product $\mathit{Prod}(K_1, K_2)$ *of these two contexts is defined as a context* $K = (\mathcal{O}, \mathcal{L}, X, d)$, *where* $L = L_1 \times L_2$, $(f_1, f_2) \sqsubseteq (g_1, g_2)$ *iff* $f_1 \sqsubseteq_1 g_1$ *and* $f_2 \sqsubseteq_2 g_2$, $X = X_1 \times X_2$, *and* $d(o) = (d_1(o), d_2(o))$.

The term *product* is justified by the fact that the logical language and the vocabulary are defined as products. There is no equivalent operation on formal contexts [GW99] (i.e., same set of objects, product of attribute sets). The *apposition* uses set union for composing attribute sets, which would entail a loss of the user-tag connection. The *direct product* uses set product to compose object sets, and uses a disjunction that would entail indeterminacy in the operation $K.\mathit{add}$.

The data structure of $K = \mathit{Prod}(K_1, K_2)$ is reduced to the union of the data structures of K_1 and K_2, and there is no proper data structure in the functor *Prod*. Hence, the size of K is the *sum*, and not the product, of the size of the argument contexts. The implementation of *Prod* looks like apposition [GW99], but that is not an apposition because $L \neq L_1 \cup L_2$. This means that X is not explicitely represented, i.e, a pair (f_1, f_2) is only known to belong to X because each part f_i is known to belong to the respective vocabulary X_i. This makes a big difference with a logic cache or conceptual scaling, where the vocabulary $X = X_1 \times X_2$ would be represented explicitly.

Because *Prod* has no proper data structure, operations on K are reduced to a composition of the operations on argument contexts. The operation $K.\mathit{add}(o, (f_1, f_2))$ decomposes itself in the sequence $K_1.\mathit{add}(o, f_1)$; $K_2.\mathit{add}(o, f_2)$. The operations $K.\mathit{show}$ and $K.\mathit{axiom}$ decompose similarly. The time complexity of those three operations are the *sum* of the time complexities in the two argument contexts. Because of the definition of the subsumption $\sqsubseteq$, an object is in the extent of (f_1, f_2) in K if it is in the extent of f_1 in K_1, and in the extent of f_2 in K_2.

$$K.\mathit{ext}((f_1, f_2)) = K_1.\mathit{ext}(f_1) \cap K_2.\mathit{ext}(f_2).$$

The computation of the increments requires the computation of the lower neighbours of a parent increment (x_1, x_2). As the vocabulary X is not explicitely represented, we have to make use of the vocabularies X_1 and X_2, and the respective increments operations. The lower neighbours of (x_1, x_2) are the pairs (y_1, x_2) where y_1 is a lower neighbour of x_1 in X_1, and the pairs (x_1, y_2) where y_2 is a lower neighbour of x_2 in X_2.

$$\begin{aligned} &K.\mathit{incrs}((x_1, x_2), O, m) = \\ &\quad \{((y_1, x_2), O') \mid (y_1, O'_1) \in K_1.\mathit{incrs}(x_1, O, m), O' = O \cap O'_1, |O'| \geq m\} \\ &\quad \cup \{((x_1, y_2), O') \mid (y_2, O'_2) \in K_2.\mathit{incrs}(x_2, O, m), O' = O \cap O'_2, |O'| \geq m\}. \end{aligned}$$

This definition is justified by the definition of the extent in K that makes the extent of a pair (f_1, f_2) in K be a subset of the extent of f_1 and f_2 in their respective context. The time complexity of those operations is equal to the sum of the time complexities in the argument contexts plus additional set intersections: $O(n)$ for $K.\mathit{ext}$, and $O(ni)$ for $K.\mathit{incrs}$.

4.4 Disjoint Union of Contexts

Suppose we want valued attributes whose values belong to one of several value domains, e.g. taxonomic terms and strings. This means we want to define a value domain that is the disjoint union of these value domains. The logical context functor *DisjointUnion*(K_1, K_2) produces a context whose logic is the disjoint union of the logics of K_1 and K_2: e.g., *DisjointUnion*(*Taxo*, *String*).

Definition 9 (disjoint union of contexts). *Let* $K_1 = (\mathcal{O}_1, \mathcal{L}_1, X_1, d_1)$*, and* $K_2 = (\mathcal{O}_2, \mathcal{L}_2, X_2, d_2)$ *be two logical contexts such that* $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$*, and* $L_1 \cap L_2 = \emptyset$*. The* disjoint union (K_1, K_2) *of these two contexts is defined as a context* $K = (\mathcal{O}, \mathcal{L}, X, d)$*, where* $\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2$*,* $L = L_1 \cup L_2$*,* $(\sqsubseteq) = (\sqsubseteq_1) \cup (\sqsubseteq_2)$*,* $X = X_1 \cup X_2$*, and* $d(o) = d_1(o)$ *if* $o \in \mathcal{O}_1$*, and* $d(o) = d_2(o)$ *otherwise.*

The condition $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$ is required because an object of K can have only one description, and the condition $L_1 \cap L_2 = \emptyset$ is required to avoid confusion on the meaning of a formula. The term *disjoint union* is justified by the fact that the logical language and the vocabulary are defined as a disjoint union. There is an equivalent operation on formal contexts (i.e., union of objects, union of attributes), also called the disjoint union.

Like the functor *Prod*, the functor *DisjointUnion* has no proper data structure, and only relies on the data structures of its argument contexts. It behaves like a switch, redirecting each operation to one argument context, depending on the parameter formula. For instance, the operation $K.show(f)$ is defined as $K_1.show(f)$ if $f \in L_1$, and as $K_2.show(f)$ if $f \in L_2$. The operation $K.axiom(f, g)$ is defined only when the two formulas f, g belong to the same logic L_1 or L_2. Hence, the worst case time complexity of those operations is the *maximum* of the worst case time complexities in the argument contexts.

4.5 The Root Context Functor

There remains a gap between the previous logical context functors, and the need in most LIS to describe objects with several properties, and to combine features by boolean connectives in queries. This gap is filled by a functor that is called *Root*(K_1) because it is designed to be used as the outermost functor in a composition of functors. Another function of that logical context functor is the application of the *Closed World Assumption* (CWA) that says that every object that is not instance of a property f is an instance of its negation $\neg f$. This makes querying more intuitive as boolean connectives then match set operations: e.g., the extent of a disjunction $q_1 \vee q_2$ is the union of the extents of q_1 and q_2. The functor *Root*(K_1) may look artificial, but it is in fact an idiomatic composition of more primitive functors (defined in [FR06]): *Prop*(*Bottom*(*Single*(*Multiset*(K_1)))), where *Prop* applies the Boolean closure, *Single* applies the CWA, and *Multiset* allows for (multi)sets of properties on objects.

Definition 10 (root logic). *Let* $\mathcal{L}_1 = (L_1, \sqsubseteq_1)$ *be a logic. The* root logic *Root* $(\mathcal{L}_1)$ *is defined as a logic* $\mathcal{L} = (L, \sqsubseteq)$*, where:*

- *$L = L_d \cup L_q$, where*
  - *L_d is the set of finite subsets $\{f_1, \dots, f_p\} \subseteq L_1$ (used for descriptions),*
  - *L_q is the smallest set that contains L_1, and such that for every $q_1, q_2 \in L_q$, the formulas $q_1 \wedge q_2, q_1 \vee q_2, \neg q_1$ also belong to L_q (used for queries),*
- *the subsumption $\sqsubseteq$ is characterized by the following inference rules, for every $f_1, \dots, f_p \in L_1$, $d \in L_d$, $q_1, q_2 \in L_q$:*
  1. *if $f_1 \sqsubseteq_1 f_2$ then $f_1 \sqsubseteq f_2$,*
  2. *if $\exists f_i \in d : f_i \sqsubseteq_1 f_2$ then $d \sqsubseteq f_2$,*
  3. *if $d \not\sqsubseteq q_1$ then $d \sqsubseteq \neg q_1$,*
  4. *if $d \sqsubseteq q_1$ and $d \sqsubseteq q_2$ then $d \sqsubseteq q_1 \wedge q_2$,*
  5. *if $d \sqsubseteq q_1$ or $d \sqsubseteq q_2$ then $d \sqsubseteq q_1 \vee q_2$.*

The last three inference rules are justified by the fact that descriptions are understood under the CWA [FR04]. Subsumption can also be defined between queries, but we can save it because it is useful neither for querying, nor for navigation. Indeed, the navigation vocabulary of the argument context of *Root* is sufficient because boolean connectives can be introduced through the navigation process [Fer09].

Definition 11 (root context). *Let $K_1 = (\mathcal{O}_1, \mathcal{L}_1, X_1, d_1)$ be a logical context. The* root context *$Root(K_1)$ is defined as a logical context $K = (\mathcal{O}, \mathcal{L}, X, d)$, where $\mathcal{O}$ is a partition of $\mathcal{O}_1$ ($\bigcup \mathcal{O} = \mathcal{O}_1$, and $\forall o, o' \in \mathcal{O} : o \cap o' = \emptyset$), $\mathcal{L} = Root(\mathcal{L}_1)$, $X = X_1$, and $d(o) = \{d_1(o_1) \mid o_1 \in o\}$.*

An object of the root context is represented by a set of objects in the argument context, each holding a property of the root object. In the following, $\overline{o_1}$ denotes the root object that contains $o_1 \in \mathcal{O}_1$; and by extension, $\overline{O_1}$ denotes the set of all root objects that contain any element of $O_1 \subseteq \mathcal{O}_1$. The proper data structures of *Root* are limited to a table defining each root object as a subset of $\mathcal{O}_1$, and another table from each object $o_1 \in \mathcal{O}_1$ to its root object $\overline{o_1}$.

The operation $K.add(o, \{f_1, ..., f_p\})$ consists in, for each f_i, creating a new object o_i to be added to $\mathcal{O}_1$, calling the operation $K_1.add(o_i, f_i)$, and defining the root object o as the set $\{o_i \mid 1 \leq i \leq p\}$. Its worst case time complexity is therefore p times the worst case time complexity of the operation $K_1.add$. The operations $K.show$ and $K.axiom$ apply only to formulas of the argument context K_1, and so can be directly transmitted to K_1; their complexities are unchanged.

The computation of the extent of a query follows the definition of subsumption between descriptions and queries. For every $f_1 \in \mathcal{L}_1$, and $q_1, q_2 \in L_q$:

$$\begin{aligned} K.ext(f_1) &= \overline{K_1.ext(f_1)}, & K.ext(q_1 \wedge q_2) &= K.ext(q_1) \cap K.ext(q_2), \\ K.ext(\neg q_1) &= \mathcal{O} \setminus K.ext(q_1), & K.ext(q_1 \vee q_2) &= K.ext(q_1) \cup K.ext(q_2). \end{aligned}$$

The complexity of this operation depends on the number k of atoms in the query: there are k calls to $K_1.ext$, and $(k-1)$ set intersections in $O(n)$ on resulting extents. The computation of increments is based on the fact that, if an increment is frequent in K, it is also frequent in K_1 because for every object $o \in \mathcal{O}$ and $x \in X$: if $o \in K.ext(x)$ then $\exists o_1 \in o : o_1 \in K_1.ext(x)$. Because the reverse is not true, the increments computed in K_1 must be checked to be increments in K.

$$\begin{aligned} &K.incrs(x, O, m) = \\ &\quad \{(y, O') \mid (y, O'_1) \in K_1.incrs(x, K_1.ext(x) \cap \bigcup O, m), O' = \overline{O'_1}, |O'| \geq m\} \end{aligned}$$

The additional cost for computing those increments is limited to computing O' for each increment coming from the argument context: $O(npi)$.

5 Applications

We now present how the context examples presented in Section 3 can be defined with logical context functors. The first context K_1 is a collection of documents described by various properties such as title, authors, publisher. Because each property can be represented by a string-valued attribute, that context is defined as $K_1 = Root(Prod(Attr, String))$, where *Attr* is an instance of the functor *Taxo*. The use of *Taxo* for representing attribute names adds the ability to abstract similar attributes into a more general attribute: e.g., "author" and "editor" can be grouped under "person"; "title", "subtitle", and "keywords" can be grouped under "subject". The use of *String* enables the automatic extraction of keywords from titles. The second context K_2 is a collection of documents described by pairs (user, tag), meaning that some user put some tag on it, where user and tag terms can be organized into two taxonomies. That context is defined as $K_2 = Root(Prod(User, Tag))$, where *User* and *Tag* are two instances of *Taxo*. In order to describe and retrieve documents by both valued attributes and (user,tag) pairs, we can defined a context $K_3 = Root(DisjointUnion(Prod(Attr, String), Prod(User, Tag)))$.

Table 1 presents the complexities of the five operations on the contexts K_1 and K_2, depending on the use of a logic cache or logical context functors. Those complexities are expressed in function of the number n of objects, the number p of properties per object in *Root*, the maximum height h of taxonomies, and the maximum length s of strings. In practice, p, h, s are often bounded, so that complexities can be expressed in term of the number n of objects only. We note that the complexity for adding an object, or showing a formula, is in $O(1)$ or $O(\ln(n))$ with functors instead of $O(n)$ with a logic cache. This achieves our objective to make the filling of a context in $O(n)$ or $O(n \ln(n))$ instead of $O(n^2)$. About the browsing operations, the complexity $O(n)$ for computing the extent and each increment is obtained, like with logic caches.

Even if detailed experiments remain to be done, first experiments are very conclusive w.r.t. the efficiency of functors compared to logic caches. We used K_1 for representing BibTEX files, where entries are objects, and fields are valued attributes. For a file with 1500 entries, it took 46s on a standard laptop to build the context with the logic cache, while it took only 26s with functors, comprising the additional computation of all maximal substrings. Without this additional computation, the time is about 10 times less, hence a 20 fold speed up. We

Table 1. Complexities of LIS operations on contexts K_1 and K_2, depending on the use of a logic cache or logical context functors

	K_1		K_2	
operation	logic cache	functors	logic cache	functors
axiom		phn		phn
add	p^2s^5n	$ps\ln(psn)$	p^2h^5n	ph
show	ps^3n	s	ph^3n	1
ext	n	$s+pn$	n	pn
incrs	ni	pni	ni	pni

managed to build the context of a file with 30,000 entries in about 1,000s, while this is not possible with a logic cache, even without generating any navigation feature.

6 Conclusion

Logical context functors are introduced as reusable components for composing logical contexts according to the structure of its formulas. They allow for efficient implementations of LIS because each functor can use specific data structures and algorithms. The genericity of the LIS framework is retained by the ability to freely compose functors. Examples of composed contexts are given for string-valued attributes, taxonomies, and pairs of taxonomic terms. The filling of a context is shown to be in $O(n)$ or $O(n\ln(n))$ instead of $O(n^2)$, comprising the computation of a rich navigation vocabulary on strings. Several taxonomies can be updated incrementally and efficiently. Functors form an open collection so that new functors can be designed and added to the collection, indepently of existing functors. Moreover, if a better data structure or algorithm is found, it can be integrated in an existing functor without impacting other functors or applications using it. We are developping functors for numbers, dates, intervals, and functors can be composed to represent vectors, lists and trees.

References

[BCM$^+$03] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)

[CES03] Cole, R.J., Eklund, P.W., Stumme, G.: Document retrieval for email search and discovery using formal concept analysis. Journal of Applied Artificial Intelligence 17(3), 257–280 (2003)

[CM00] Chaudron, L., Maille, N.: Generalized formal concept analysis. In: Ganter, B., Mineau, G.W. (eds.) ICCS 2000. LNCS, vol. 1867. Springer, Heidelberg (2000)

[DVE06] Ducrou, J., Vormbrock, B., Eklund, P.W.: FCA-based browsing and searching of a collection of images. In: Schärfe, H., Hitzler, P., Øhrstrøm, P. (eds.) ICCS 2006. LNCS, vol. 4068, pp. 203–214. Springer, Heidelberg (2006)

[Fer07] Ferré, S.: The efficient computation of complete and concise substring scales with suffix trees. In: Kuznetsov, S.O., Schmidt, S. (eds.) ICFCA 2007. LNCS, vol. 4390, pp. 98–113. Springer, Heidelberg (2007)

[Fer09] Ferré, S.: Camelis: a logical information system to organize and browse a collection of documents. Int. J. General Systems 38(4) (2009)

[FR00] Ferré, S., Ridoux, O.: A logical generalization of formal concept analysis. In: Ganter, B., Mineau, G.W. (eds.) ICCS 2000. LNCS, vol. 1867, pp. 371–384. Springer, Heidelberg (2000)

[FR04] Ferré, S., Ridoux, O.: An introduction to logical information systems. Information Processing & Management 40(3), 383–419 (2004)

[FR06] Ferré, S., Ridoux, O.: Logic functors: A toolbox of components for building customized and embeddable logics. Research Report RR-5871, INRIA, 103 p. (March 2006)

[GK01] Ganter, B., Kuznetsov, S.: Pattern structures and their projections. In: Delugach, H.S., Stumme, G. (eds.) ICCS 2001. LNCS, vol. 2120, pp. 129–142. Springer, Heidelberg (2001)

[GMA93] Godin, R., Missaoui, R., April, A.: Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. International Journal of Man-Machine Studies 38(5), 747–767 (1993)

[GW89] Ganter, B., Wille, R.: Conceptual scaling. In: Roberts, F. (ed.) Applications of combinatorics and graph theory to the biological and social sciences, pp. 139–167. Springer, Heidelberg (1989)

[GW99] Ganter, B., Wille, R.: Formal Concept Analysis — Mathematical Foundations. Springer, Heidelberg (1999)

[Mog89] Moggi, E.: A category-theoretic account of program modules. In: Dybjer, P., Pitts, A.M., Pitt, D.H., Poigné, A., Rydeheard, D.E. (eds.) Category Theory and Computer Science. LNCS, vol. 389. Springer, Heidelberg (1989)

[PS99] Prediger, S., Stumme, G.: Theory-driven logical scaling: Conceptual information systems meet description logics. CEUR Workshop Proceedings, vol. 21, pp. 46–49. CEUR-WS.org (1999)

[Sac00] Sacco, G.M.: Dynamic taxonomies: A model for large information bases. IEEE Transactions Knowledge and Data Engineering 12(3), 468–479 (2000)

[TCH06] Tane, J., Cimiano, P., Hitzler, P.: Query-based multicontexts for knowledge base browsing: An evaluation. In: Schärfe, H., Hitzler, P., Øhrstrøm, P. (eds.) ICCS 2006. LNCS, vol. 4068, pp. 413–426. Springer, Heidelberg (2006)