Universität Bern
Philosophisch-naturwissenschaftliche Fakultät
Institut für Informatik

Bachelorarbeit

Inferring schemata from semi-structured data with Formal Concept Analysis

Jan Luca Liechti
aus Landiswil BE
05-115-134

Betreuer:
Prof. Dr. Oscar Nierstrasz

Bern, 27.5.2017

Abstract

Semi-structured data do not conform to the schematic rigor of relational databases, but still present their content in a structured way. They are described as self-describing data, because they provide a schema for every record, for example as XML elements or JSON keywords. We are interested in inferring a relational schema from such semi-structured data that both preserves semantic information, i.e. keeps similar records together, and requires as little extra space as possible. We are operating under the assumption that we do not know records' true types. We employ well-established notions of Formal Concept Analysis and use them in ways similar to basic operations on graphs and automata. Specifically, we create a formal context from the data where records are objects and tags are attributes and compute its concept lattice. Based on the assumption that semantically similar records are also structurally similar, i.e. have a similar set of attributes, we designed and implemented an algorithm that iteratively performs updates on the lattice in order to obtain a partition of the data fulfilling the above mentioned criteria. In tests with real-life data, we obtain good results for datasets that are already highly structured—that contain few outliers and many structurally equivalent records—and mixed results at best for very diverse datasets.

Contents

1 Relational and semi-structured data

As gathering, storing, retrieving and manipulating data become more and more important in our world, so does the question of keeping that data clean. For example, as far back as 2002, it was estimated that dirty data cost the U.S. economy $600 billion ([8], [16]). How can anyone dealing with massive amounts of data—be it science, administration, the industry, or whoever else—work towards keeping their data clean?

For the longer part of the digital age, most data has been stored in relational databases, with earliest versions going back as far as the late 1970s. Relational databases store information in tables, with each row representing a record and each column representing a field, or attribute, of a record. In turn, each table represents records of one type or entity. Semi-structured data do not conform to the rigor of one schema for a whole table, but are still structured on the record level [1]. Think for example of XML elements or JSON keywords providing information about the values they hold.

Recently, so-called NoSQL databases—where 'No' stands for 'not only'—have begun to emerge and gain rather large market shares, especially with the increased need for scalability, e.g. in distributed or cloud applications. At the time of writing, MongoDB, a document-oriented (NoSQL) database, was sitting comfortably in the top 5 most widely used database systems[1]. Most NoSQL databases—certainly document-oriented ones and key-value stores—are operating on semi-structured data.

Both database types have their distinct advantages and disadvantages. NoSQL databases support a multitude of data models, are easily scalable and generally faster, more efficient and flexible. On the other hand, the community has had much more time to research and develop relational databases, which have a standard query language, described in ISO/IEC 9075-11:2016. Also, not all NoSQL databases are ACID compliant and not all support a wide range of joins and other operations [9]. Therefore, it is safe to say that the two paradigms cater to quite different needs.

This work is concerned with the question of migrating/transforming a NoSQL database to a relational one in such a way that the data fit in a relational database with as much of their structure preserved while occupying a minimal amount of extra storage. In other words, we want to 'wrap' our tables 'around' the data as tightly as possible while generating the minimal number of tables. Consider the following XML document:

```
<library>
	<book>
		<author>Bernhard Ganter</author>
		<author>Sergei Obiedkov</author>
		<title>Conceptual Exploration</title>
		<pages>315</pages>
		<year>2016</year>
		<isbn>978-3-662-49290-1</isbn>
	</book>
	<book>
		<author>Bernhard Ganter</author>
		<author>Rudolf Wille</author>
		<title>Formal Concept Analysis</title>
		<pages>284</pages>
		<year>1999</year>
		<publisher>Springer</publisher>
		<isbn>978-3-540-62771-5</isbn>
	</book>
```

[1] http://db-engines.com/en/ranking/

```
        <article>
                <author>Sergei Kuznetsov</author>
                <author>Sergei Obiedkov</author>
                <title>Comparing performance of algorithms...</title>
                <pages>28</pages>
                <year>2002</year>
                <journal>J. Exp. Theor. Artif. Intell</journal>
                <issn>1362-3079</issn>
                <volume>14</volume>
        </article>
</library>
```

Listing 1: Example XML document

If we were tasked with transforming this document to a relational database, what would that target database look like? We could of course just naively write everything in one library table:

	author	title	pages	year	journal	issn	volume	publisher	isbn
book1	Ganter	...	...	...	NULL	NULL	NULL	NULL	...
book2	Wille	...	...	...	NULL	NULL	NULL	...	...
article1	Obiedkov	...	...	...	...	...	...	NULL	...

Table 1: Example XML document naively transformed to a relational schema

Indeed, such an approach has considerable advantages, not just because of a lower number of tables cluttering up the target database, but also e.g. because data coming from one and the same source stays together. However, we would not say that this approach is feasible for any given dataset. For example, we might want to group different types of objects together, i.e. have a table for books, one for articles, and so on. If we did that in our example, we could drop columns 'journal', 'issn', and 'volume' for all books, and save a lot of space—this transformation would yield 75% less NULL values than the naive one. However, types might not be given, and other information might be missing as well—it is quite obvious that books do not appear within a journal, but what if we didn't have the real column names, for example?

In this work, we apply a few restrictions on this broad spectrum of questions. We restrict our semi-structured datasets to be of XML, JSON, and BibTeX format, even though there are many others. These three formats do not have the exact same expressive power—they do have in common, though, that their respective expressive power allows far more than just pairing keys and values. However, we will not need all that added functionality such as namespaces, types etc. In this context, a semi-structured dataset is just a collection of records with tags, or objects with attributes, where there is no explicit underlying schema for these attributes (as there always is in a relational database).

We will also ignore whether an attribute appears once or more than once in a record. In the example above, all records have two authors, but we do not make that distinction—we find the information whether a record *has* an author or *not* much more valuable than whether there is one or many. It is trivial to find out whether an attribute is given once or more than once (in at least one record), and it is equally trivial to generate 1:N relation tables in the latter case.

There is a third property of—in this case XML—files that we completely ignore. Consider the author elements of the above document. Here, these are identified by a single string representing the name. However, elements could be nested much deeper and also contain various information about authors and other objects. We are looking only at one nesting level (and its children) throughout this work—in this case, the library level.

We will examine two main questions here: 1. How can we best transform semi-structured data into a relational schema with a minimum number of tables and a minimum number of NULL values (the two quite obviously being a trade-off for each other), and 2. How accurately can we gauge the type of a record based only on its attributes? In the example context above, the article object has three article-specific attributes that the books do not have, but the book objects do not have identical attributes either. Some kind of type information was present in all datasets that we analyzed, but it is not hard to imagine a scenario where that information is not available.
Our approach is based on the methods of Formal Concept Analysis, which we will introduce next (chapter 2). After that, we give a short overview of existing work, both in schema inference as well as in Formal Concept Analysis (chapter 3). Since our approach is quite novel, we describe it and its underlying assumptions in detail (chapter 4). We then give some details of our implementation (chapter 5), share our results (chapter 6) and discuss future work in the field (chapter 7).

2 Introducing Formal Concept Analysis

The beginnings of Formal Concept Analysis go back to an article by Rudolf Wille [10], published in 1982 and reprinted in 2007 [11]. The most comprehensive account of its mathematical foundations, especially in English, is by Wille and Bernhard Ganter [12], which we quote extensively here and whose scope goes far beyond what we need for this work. The next quotes and definitions can all be found in the first chapters there.
"Formal Concept Analysis is a field of applied mathematics based on the mathematization of *concept* and *conceptual hierarchy",* write the authors in the preface (p. VII). As such, it relies on quite a few concepts from mathematics—especially from lattice theory, order theory, and set theory. We introduce only notions specific to Formal Concept Analysis here; Ganter and Wille's book is a good reference for going into more detail.
One of the paramount concepts of Formal Concept Analysis is that of a formal context:

Definition 1. A **formal context** $\mathbb{K} := (G, M, I)$ consists of two sets G and M and a relation I between G and M. The elements of G are called the **objects** and the elements of M are called the **attributes** of the context. In order to express that an object g is in a relation I with an attribute m, we write gIm or $(g, m) \in I$ and read it as "the object g **has** the attribute m".

There are no restrictions to the domains of the objects, attributes, and the relation. We can conceptualize any set of objects with Formal Concept Analysis. These contexts can be represented by a cross table, where the rows represent objects, the columns represent attributes, and a cross in row a, column b means that object a has attribute b.
We are giving two examples of contexts, choosing them from deliberately different domains.

	composite	even	odd	prime	square
1			×		×
2		×		×	
3			×	×	
4	×	×			×
5			×	×	
6	×	×			
7			×	×	
8	×	×			
9	×		×		×
10	×	×			

Table 2: Example formal context nr. 1

The first is a very simple mathematical example. The objects are the numbers from 1 to 10, and the attributes are "composite", "even", "odd", "prime", and "square". It is obvious that for each pair of one number and one attribute, we can say exactly whether this number has the given attribute or not, i.e. the relation is binary. Note how the attributes "even" and "odd" are mutually exclusive, just as we would expect. "Composite" and "prime" are also mutually exclusive, with the exception of the number 1, which is neither composite nor prime.

	Forst	Wald	Gehölz	Hain	Holz	Trödelkram	Gerümpel	Nutzholz	Bauholz	Balken
forest	×	×								
wood		×	×		×					
grove			×	×						
lumber					×	×	×	×		
timber								×	×	×

Table 3: Example formal context nr. 2

The second example is taken from quite a specialized application, namely an article on FCA-based linguistics [15]. Here, objects and attributes are words from different languages, and the binary relation is "is a translation of". We find that words such as the English "forest" or "wood" can have multiple, even overlapping translations to German—for instance, "Wald" translates to both of them. It is with such overlappings and their absence that we will be preoccupied for the large part of this work. Note that here, we don't technically have objects and attributes, but rather objects that can be related to other objects in a binary relation of our choosing. This is certainly not the most traditional form of Formal Concept Analysis, but it is one of its applications nonetheless.
Both these examples show that the cross table visualizes content of nontrivial complexity rather neatly. In order to further clean up the visualization, we introduce another key idea of Formal Concept Analysis.

Definition 2. For a set $A \subseteq G$ of objects we define

$$A' := \{m \in M | gIm \text{ for all } g \in A\}$$

(the set of attributes common to the objects in A). Correspondingly, for a set B of attributes we define

$$B' := \{g \in G | gIm \text{ for all } m \in B\}$$

(the set of objects which have all attributes in B).

Definition 3. A **formal concept** of the context (G, M, I) is a pair (A, B) with $A \subseteq G$, $B \subseteq M$, $A' = B$ and $B' = A$. We call A the **extent** and B the **intent** of the concept (A, B). $\mathfrak{B}(G, M, I)$ denotes the set of all concepts of the context (B, M, I).

Trivially, each attribute is a concept and all objects that have it are that concept's extent. But consider for example the attribute pair {odd, prime} in our first example context. If this is our intent, then the extent are the objects {3, 5, 7}. Similarly, if we consider the intent {even, composite}, then the extent is {4, 6, 8, 10}.
In the second example, one concept would be the one with intent {Wald}, which would have {forest, wood} as its extent, or the one with intent {Hain} and extent {grove}. We can visually interpret concepts as 'blocks' in the cross table after rearranging any number of rows and/or columns, which we can do without loss or distortion of information, since these are both unordered. Consider the first example. With just reordering the rows such that we first have the odd numbers in ascending order and then the even numbers in ascending order, we end up at the following cross table:

	composite	even	odd	prime	square
1			×		×
3			×	×	
5			×	×	
7			×	×	
9	×		×		×
2		×		×	
4	×	×			×
6	×	×			
8	×	×			
10	×	×			

Table 4: Formal concepts in example formal context nr. 1

It should be easy to see that we cannot make either of these 'blocks' larger by adding another column with all concerned rows crossed, and vice versa. Since in both blocks, all objects have all attributes and all attributes are present in all objects, these really are concepts of the context. Let us now introduce the last of the chief concepts of Formal Concept Analysis that we will be using in this work.

Definition 4. If (A_1, B_1) and (A_2, B_2) are concepts of a context, (A_1, B_1) is called a **subconcept** of (A_2, B_2), provided that $A_1 \subseteq A_2$ (which is equivalent to $B_2 \subseteq B_1$). In this case, (A_2, B_2) is a **superconcept** of (A_1, B_1), and we write $(A_1, B_1) \leq (A_2, B_2)$. The relation $\leq$ is called the **hierarchical order** (or simply **order**) of the concepts. The set of all concepts of (G, M, I) ordered in this way is denoted by $\underline{\mathfrak{B}}(G, M, I)$ and is called the **concept lattice** of the context (G, M, I).

The concepts, under the $\leq$ relation, are partially ordered sets. We assume the reader is familiar with this, as well as with lattices and the closely related concepts of join and meet, and do not introduce these definitions here, as they are not specific to Formal Concept Analysis and easily found in the literature. We will however give examples throughout this work.
The basic theorem on concept lattices states that concept lattices are *complete* lattices, i.e. a set (of concepts), in which *all* subsets have a supremum (= join) and an infimum (= meet). This means that for every subset of objects, there is a minimal set of attributes that all these objects have, and for every subset of attributes, there is a minimal set of objects in which all these attributes appear. Note that both this set of attributes as well as the set of objects may

be the empty set.
Lattices have the great property of being even more visually intuitive than cross tables while not losing any information. Let us have a look at the concept lattices of our examples above.

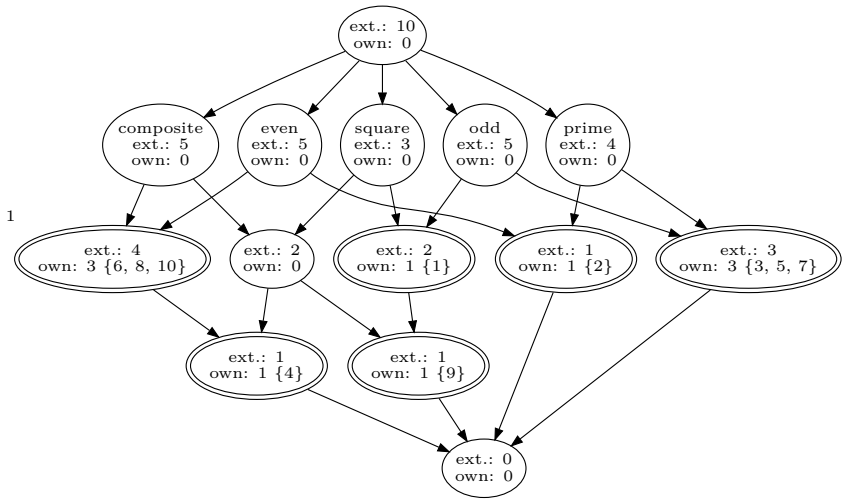

Figure 1: Example context nr. 1

Figure 1 shows the lattice for our first sample context. Each node represents a subset of all attributes and displays some information regarding that subset in the context.
In the first, optional, line (here only used in the children of the top node), we write all attributes that this particular node adds to the context. We use the term "own attribute" for these, following the terminology in the popular lattice visualization tool ConExp[2]. We see that here, not all nodes add attributes—in fact, only a minority of them do. Each node represents all attributes of its ancestor nodes and all own attributes, if there are any.
Looking at node 1, we find that this node represents the concept with intent {even, composite}. (Indices are placed in the top left corner of an imaginary rectangular box around the node). We remember from above that the extent of this concept is {4, 6, 8, 10}. Indeed, in the second line, the extent ("ext.") size is specified as 4. But these four objects do not have the same attributes; namely, 4 is also square. The other three objects have exactly the intent represented by this node, making them its "own objects", again in ConExp terminology. The number of own objects is written in the third line—in this very simple example alongside a list of the concrete objects. Object 4, however, is an own object of the only child node of node 1, and we find that this node indeed also has {square} as an ancestor node. We draw nodes that have own objects with a double border, because in this work, these nodes are of special interest.
Some pointers on how to read these lattices: The top node contains all attributes that are common to all objects in the context; in this case, this is the empty set. However, if we also had an attribute "≤ 10", all numbers would have that attribute, and it would be an own attribute of the top node. Similarly, the bottom node represents the concept with intent $= M$ which in this case has no own objects.
Similarly to a concept's *intent* being all own attributes of ancestor nodes as well as any own attributes, a concept's *extent* is equal to all own objects plus all own objects of descendant nodes. The join of two nodes—their unique closest common ancestor node—represents the intersection of their attributes, and the meet—their unique closest common descendant node—represents the minimal attribute set that contains the union of their attributes.

[2] `http://conexp.sourceforge.net/`, also cf. appendix A.4

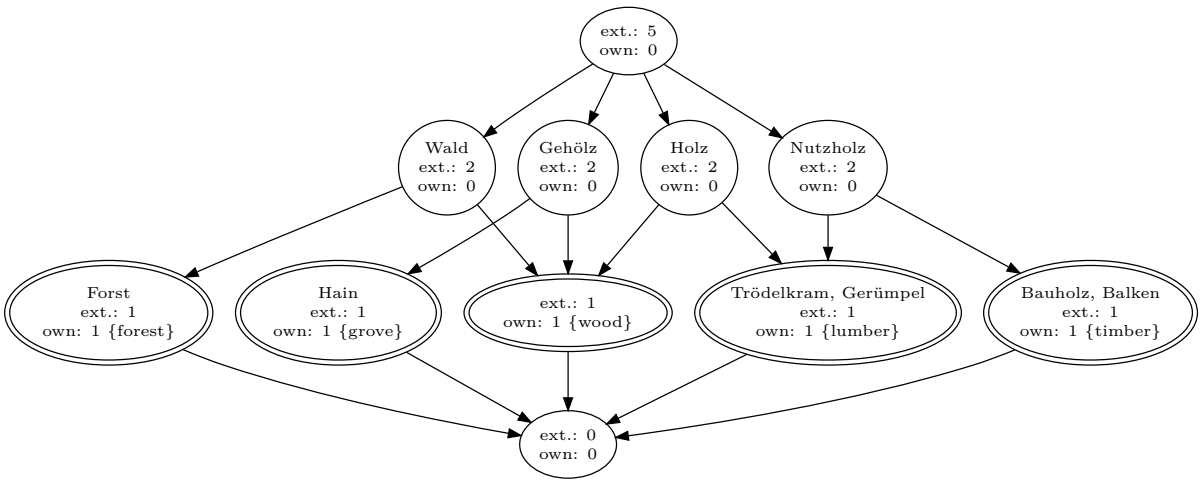

Figure 2: Example context nr. 2

In the second example, shown in figure 2, following a node downward and listing all objects we encounter gives us all English words that can be translated as the set of German words represented by the initial node. If we follow the "Wald" node downward, we find that both "forest" and "wood" are translations, which we already know from the cross table. In this context, there are two other interesting things to consider. First, there are two nodes that introduce more than one attribute, namely {Trödelkram, Gerümpel} and {Bauholz, Balken}. This means, that as far as the objects in this context are concerned, the presence of one of these attributes implies the presence of the other and vice versa, i.e. these attributes are either not found in an object or they are found together. Second, both these nodes as well as the node that adds {Forst} have ancestors that add attributes to the context, which we didn't have in the first example. For the {Forst} node, that means that whenever an object has attribute "Forst", it *must* also have "Wald", but *not* the other way around. This property will be very important when we look at attributes of semi-structured data.
Even though these are just two examples, they exemplify quite different uses to which Formal Concept Analysis can be put.
In the first example, we would probably be interested in finding concepts with as many attributes and/or objects as possible, which allow us to make statements about which attributes are linked to each other in which way, etc. This gives us some structural information about the underlying context. Of course, the bigger part of these attributes are opposites or very close to opposites by definition, but it should not be hard to imagine how this plays out in a richer context.
In the second example, we would be less interested in finding frequently occurring combinations of German translations, but in finding an English word that expresses exactly that combination of German word that we would like to express (say, to find the most nuanced translation). Here, we would be less interested in structural information but more in unique examples.
We will see in section 4.1 that our use of Formal Concept Analysis differs very much again from both these approaches.

3 Existing work

The process of extracting a schema from schemaless data is referred to as 'inference'. We are not the first to try to infer schemata from semi-structured data, and shall mention some other works that have helped our understanding of the problem. And we are especially not the first

to apply the methods of Formal Concept Analysis to real life data—we will point out some interesting approaches here as well.

3.1 In schema inference

In a rather new survey article [2], the authors give current overviews of such diverse aspects of relational data profiling as finding different kinds of dependencies in the data (e.g. inclusion dependencies, functional dependencies), value distribution within columns, and finding clusters and outliers. Especially where functional dependencies are concerned, we are talking about schema inference as well, since the schema of a relational database not only contains all the tables of that database with their respective fields, but also some semantics between these tables, e.g. primary/foreign key relations. In order to check whether a database conforms to some normalization standards (which guarantees a certain quality of the data), we are also interested in functional dependencies between two sets of columns in a table.
While the authors go into much detail and provide a rich bibliography (that does not need to be duplicated here) concerning the above mentioned points, the process of extracting the physical schema—the fields and tables—is touched upon only briefly [2, p. 577]. However, significant work has been done in that area at least since the early 2000s.
A technical report [3] lists the most relevant of the contributions dealing with the inference of XML schemata, of which some are quite widely cited. These approaches fall into two basic categories: First, inferring a grammar, and second, based on heuristics. A heuristic might be that if an element has three child elements of a certain type, it is highly probable that it can have any number of them. In methods that infer a grammar, all analysed XML documents are seen as words of a language, and the underlying schema as the grammar of that language. The inference methods usually focus on some subclasses of regular languages. One example is that of k-contextual languages, in which two states p_i, p_j are seen as identical if there exist distinct states p_{i0}, p_{j0} and a sequence a of input symbols $(a_1\ a_2\ \ldots\ a_k)$ of length k so that if we are in p_{i0} and read a, we end up in p_i, and if we read a while we are in p_{j0}, we end up in p_j. Thus, approaches that exploit the theory of formal languages are able to guarantee a certain quality of the inferred result in terms of a class of language its output belongs to.
The general approach in both heuristics-based as well as grammar-inferring methods is to first construct an automaton that accepts all input test XML documents. The XML elements, then, are symbols of the language, and the states are positions in an XML document that you can 'reach', e.g. first reading two authors and then one publisher. Initially, as XML documents are analyzed, transitions are simply added as read, normally leading to a much too complicated automaton. After that, either heuristics-based or grammar-inferring rules are applied in order to simplify the automaton, normally by merging its states. The algorithm in this work is clearly influenced by these approaches. After that, some easier steps such as type inference are applied, and finally all kinds of semantic rules are inferred.
Results obtained through this method are usually quite good. As far as we could discern, Bex, Neven, and Vansummeren [5] obtain the best results by focussing on k-local single occurrence regular expressions (SOREs). However, all those approaches, though they are obviously inferring schemata, are hardly comparable to our approach, since the schema they infer is either a DTD (Document Type Definition) or an XSD (XML Schema Definition) type schema. These are specifically designed for XML and are much more expressive than a relational database schema. Relational schemata can express only three types of relations: 1:1, 1:N, and M:N. These more flexible definition languages built for semi-structured data, however, can for example specify ranges in regard to how many times an element should occur. For example, a book might have between one and three authors. XSD is again more expressive than DTD. For example, in

DTD, every element is identified only by its name, while in XSDs, it is possible to use the same element name in different contexts and have it mean different entities.
Irena Mlýnková's group at Charles University in Prague has, among other things, focussed on these approaches. One notable contribution is *jInfer*, an open source NetBeans platform framework for implementing new or existing inference approaches with defined interfaces between the different steps of the inference, e.g. generation of the initial grammar and the various cleansing/merging steps. *jInfer* is documented in [4].
However, this has nothing to do with relational schema inference from semi-structured data, nor with Formal Concept Analysis. Let us turn to some research that in this regard is closer to our own method.

3.2 In Formal Concept Analysis

In a two-part survey ([6], [7]), the authors give a relatively in-depth overview of the field of Formal Concept Analysis. A more recent publication is Ganter's new book [13], which we will use again in section 7. Instead of diving into FCA-related research in all its subtleties, we want to give two examples of work that is thematically or methodically close to ours.
Renée Miller—who has already worked in the field of schema inference before, applying information-theoretic tools to the domain [17]—and collaborators use Formal Concept Analysis in order to find a set of minimal conditional functional dependencies in data [16].
Functional dependencies are tuples (X, Y) of attribute sets X, Y in a relation R. Each value in X is mapped to exactly one value in Y. A trivial example is that of primary keys (as X), because they uniquely identify all other values in the relation. But that is not the only kind of functional dependency we find in relational databases. Very often, for example, a city name is determined by a ZIP code (but not the other way around). *Conditional* functional dependencies are a special case of functional dependencies, holding over a fraction of (specific) values in a relation, which is often desired to be large. For example, in Switzerland the area code of a telephone number is uniquely determined by the city a number is first registered in, *except for Zürich,* which has 43 and 44. So—discounting for the fact that numbers can be moved around—there could be a table where the area code column is functionally dependent on the city column, except if city = Zürich. In order to find such conditional functional dependencies, the authors build the attribute lattice (although attribute *value* lattice might be more precise) from the power set of attribute value combinations and traverse it breadth-first. This means that they are first considering all functional dependencies where $|X| = 1$, then 2, and so on. They use this method in order to be able to prune a large portion of possible candidates early, e.g. if a candidate is a superset of an already discovered dependency.
This approach still only takes into account attribute *values* and not the attributes themselves. In the example above, it is assumed that there is always a city and a ZIP code. It is our exact goal to group records together by their attribute *composition*, not by their attribute *values*.
The work that has influenced the way we apply the methods of Formal Concept Analysis to our problem the most is Lindig [14]. In it, in order to mine patterns and violations in software, the author uses the methods of Formal Concept Analysis in a way that very similar to ours.
The author tries to mine patterns and violations in much the same way we try to infer schemata. Prior knowledge about the specification of the software is explicitly absent, just as we are working without knowing the actual types of records. Instead, the paper focusses on finding patterns in the data and instances that *almost* match these patterns. The domains of these patterns are sets of function calls by other functions. In FCA terms, the calling function is the object, and the called functions are the attributes. Very much in the same way as table 4 shows, we are interested in finding 'blocks' of attributes that often occur together (in fact, we

borrowed the term from that very paper). For example, in the Ruby 1.8.4 source code, 17 different methods are calling first `va_start` and then `va_end`, which seems very natural. However, there are functions *violating* that pattern by not calling `va_end` after calling `va_start`.
The question here is of course: How can we know if some object violates a pattern or simply belongs to another pattern? The authors answer this question quantitatively, just as we will. The confidence that some instance violates a pattern defined by similar instances is simply defined as the number of 'good' objects (that constitute the pattern) divided by the number of violators whose pattern is ever so slightly 'off'. This way of working quantitatively with Formal Concept Analysis is indeed a cornerstone of our approach to schema inference, even if we have refined the confidence measuring to also include the *absolute* size of both the pattern and the violation. This should be no surprise, since even though the method is very similar, the domain it is applied to is fundamentally different. For example, if you repeatedly allocate space in memory and never deallocate it (and don't have garbage collection), then your memory will just fill up. As a programmer, you can test your code heavily for exactly this kind of problem. On the other hand, if you enter some new records into a semi-structured dataset and leave some fields blank, your data may not look as regular as if you had entered all values, but no *errors* will occur. In fact, it is exactly the *purpose* of semi-structured data stores to accommodate this kind of record. Therefore, if the potentially violating set of instances is large enough, it is highly probable that it is a type in its own right rather than a deviation from some other, more prevalent type. Thus, if we have 10 objects and one violator on the one hand and 100 objects and 10 violators on the other hand, we assume that the first instance is an actual violation with greater probability than the second. Another concrete difference between the two types of data is that while the patterns analyzed by Lindig [14] are rarely more than 2 or 3 attributes 'wide', while the data we are working with often has upward of 10 attributes.
Despite substantial bodies of work both with Formal Concept Analysis as well as in schema inference, to the best of our knowledge, we are the first to try an approach similar to what is described in the rest of this work.

4 Inferring schemata with Formal Concept Analysis

We shall now describe our approach to inferring a schema from semi-structured data with Formal Concept Analysis.
First, we describe our approach in a general way. We introduce some key concepts, such as merging lattice nodes; clusters; and connected nodes. Then we describe our understanding of semi-structured data and what it means for an object to have a type. Lastly, we describe the algorithm—or, rather, the family of algorithms—we have derived from these considerations, and the many ways in which such an algorithm can be parametrized.

4.1 Key concepts and goal

Equipped with the definitions of semi-structured data (cf. section 1) and the various concepts from Formal Concept Analysis (cf. section 2), we can get right to describing what exactly we are trying to achieve.
In its most basic form, the idea is to create a formal context out of our semi-structured dataset. The records will be the objects, and the keys will be the attributes. In other words, our binary relation is defined as "Record r has at least one key k". In order to generate schemata for relational database tables and populate them with objects that ideally are all of the same type, we are trying to find patterns in the lattice that help us separate different types from each other. That is the general idea. We are not interested in the values any of the keys have, we are only

interested in which keys are present in which records.
Each different combination of keys in the data, i.e. each different kind of record, corresponds to a node *with own objects* in the lattice. Each intersection of any number of such combinations that doesn't also exist exactly in this combination in the data corresponds to a lattice node without own objects. Note that we always have a top node—with all attributes that all objects in the context share, which in some cases is the empty set—and a bottom node with all attributes in the context.
Consider again the XML dataset from listing 1:

```
<library>
	<book>
		<author>Bernhard Ganter</author>
		<author>Sergei Obiedkov</author>
		<title>Conceptual Exploration</title>
		<pages>315</pages>
		<year>2016</year>
		<isbn>978-3-662-49290-1</isbn>
	</book>
	<book>
		<author>Bernhard Ganter</author>
		<author>Rudolf Wille</author>
		<title>Formal Concept Analysis</title>
		<pages>284</pages>
		<year>1999</year>
		<publisher>Springer</publisher>
		<isbn>978-3-540-62771-5</isbn>
	</book>
	<article>
		<author>Sergei Kuznetsov</author>
		<author>Sergei Obiedkov</author>
		<title>Comparing performance of algorithms...</title>
		<pages>28</pages>
		<year>2002</year>
		<journal>J. Exp. Theor. Artif. Intell</journal>
		<issn>1362-3079</issn>
		<volume>14</volume>
	</article>
</library>
```

Listing 2: Example XML document (again)

This very small dataset would correspond to the following cross table, which naturally looks very similar to table 1:

	author	title	pages	year	journal	issn	volume	publisher	isbn
book 1	×	×	×	×					×
book 2	×	×	×	×				×	×
article 1	×	×	×	×	×	×	×		

Table 5: Cross table of example bibliographic context

As expected from the XML data, all records have (at least) an *author*, a *title*, a *pages* count and a *year*. That means that we are not dealing with completely unrelated objects—say, books and fruit. Of course, a journal article does not have an ISBN and thus has no such attribute. (Technically, it just might, and probably there are these cases in real life data, but that would

clearly constitute an error). As is well known, only standalone publication have ISBN numbers. The equivalent for periodical publications is the ISSN ("International Standard Serial Number"), which our sample record also has as an attribute. Only the first book has a *publisher* here—the second one might just as well, and of course does so in reality, as does the journal. This would be an example where not all records of the same type have the same set of compulsory fields. If we add a few more attributes and objects to this library context, we might end up with a cross table like the one in table 6. Figure 3 shows the lattice generated from the context in table 6.

	author	title	pages	year	journal	issn	volume	publisher	isbn	abstract	booktitle
book 1	×	×	×	×					×		
book 2	×	×	×	×				×	×		
book 3	×	×	×	×				×	×		
book 4	×	×	×	×				×	×		×
article 1	×	×	×	×	×	×	×				
article 2	×	×	×	×	×	×	×				
article 3	×	×	×	×	×	×	×				
article 4	×	×	×	×	×	×	×	×			
article 5	×	×	×	×	×	×	×			×	
article 6	×	×	×	×	×	×	×	×		×	

Table 6: Cross table of extended example bibliographic context

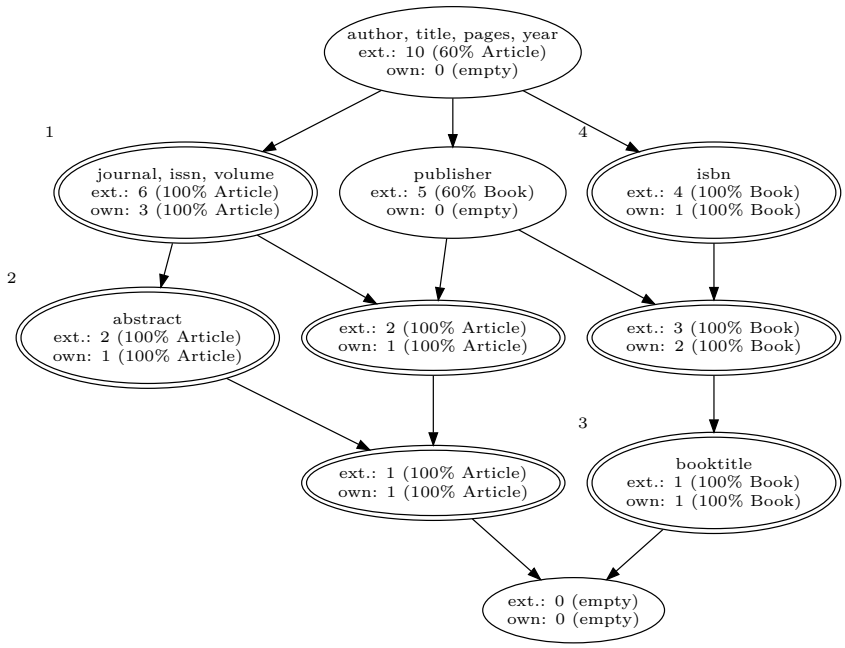

Figure 3: Example bibliographic context

As we can see, in this example there are four attributes in the top node, meaning that these attributes are present in all objects. There are two nodes with $|\text{ext}| = 6$ and 4, respectively—these are the big blocks on the left of the cross table, separating books and articles in a reliable fashion by three attributes. Most attributes clearly belong to one of the two types, except *publisher*, which appears in two articles and two books; the corresponding node is in the middle of the lattice and has children on both the article as well as the book "side".

Some observations here: Every object that has an *abstract* also has a *journal*, an *ISSN* and a *volume*, since the corresponding node 1 is node 2's parent. In the same way, every object with a *booktitle* has an *ISBN*. There is no object that has all attributes present in the context, since the bottom node has no own objects.
This visualization, which we will use for all further lattices, is slightly different than the (even simpler) examples from section 2. Instead of providing a list of all objects for each node, we just give some summary information about the most frequent object types among objects that belong to a node. Especially for large contexts, it would be quite impossible to reasonably visualize this information in more detail. This summary is both given for the own objects of a node as well as for its extent. When running our algorithm and testing different settings, this would always give us a lot of information at first glance. However, in a real life setting, the difficulty is of course that all type-related information is exactly what is missing and what we are trying to algorithmically gauge. So imagine the exact same context from above, but without all the information that is in brackets. How can we know that node 1 is the highest node to only contain article type records, etc.? Of course we can't. But we can try to create an algorithm that predicts this information based on our understanding of some properties of these contexts gained from semi-structured datasets. This is the goal of this work.

4.2 Assumptions about semi-structured data and what they represent

What we know from a dataset are how attributes are distributed in objects, and nothing more. So what exactly can we infer from this information?
One very general assumption we make is that the more similar the attribute composition of two objects, the higher the chance that both objects have the same type. One intuitive measure for similarity in attribute composition is the number of attributes that are present or absent in both objects; the higher this number, the more similar two objects are.
However, we would like to make an important distinction here. Consider two objects with identical attributes except for two. These two attributes are either both present in one object and absent in the other, or one of them is present in one and absent in the other, and the other way around for the second attribute. In an intuitive understanding of objects and attributes, we would say that in the first case, the objects are decidedly more similar than in the second case. Here, we have a proper subset-superset relation, whereas in the other case, the objects are really just similar to a certain degree. If an object has all attributes another object also has and just adds two attributes on top, we assume that this object is probably some specialized case of the other; in the other case, the two objects are merely similar, but we have no idea about their relation.
The second assumption we are building our algorithm on is that each object type has some kind of archetypical attribute composition, from which individual objects may differ in varying degrees. For any given dataset, we assume the archetype of a specific type to contain all attributes that are present in 50% or more of all objects of that type. For example, if 99% of all article objects have an *author*, then this attribute is certainly a part of the article archetype. If, on the other hand, only 10% of all books have an *editor*, then this attribute wouldn't belong to the book archetype. We will see that we cannot always guarantee that at the end of our algorithm, we return tables with only the archetypical attributes—but in most real life uses, we nevertheless will.
At this point, we define two notions that we will use throughout this work.

Definition 5. We call two lattice nodes A and B **connected**, if both A and B have at least one own object and A is a parent node of B or vice versa.

Definition 6. We call the transitive closure of lattice nodes under the connectedness relation a **cluster**.

In figure 3, for example, we have two clusters. One consists of node 1 and its descendants, and one of node 4 and its descendants (both without the bottom node however).
Consider a context with two types of objects in it. Since we assume these types to exhibit some semantic differences, we also assume them to have different attribute archetypes. However, as we are working with non-relational data, no strict schema is enforced, and individual objects may differ from the archetype. As long as the dataset is not too messy, we would imagine such a context to have two clusters—or maybe even one, if the types are not too different from each other—that are organized around a few nodes. These nodes in turn are very close to the archetype in terms of attribute composition. We naturally expect to find fewer objects of the same type with a large attribute difference, as these would be considered outliers.
We designed the context from figure 3 to be such an example. For each type of objects—book and article—, there is a node with the most own objects. In fact, for both types, these nodes contain exactly 50% of the objects. That means that all their attributes would belong to the respective archetypes.
It is quite nice to see that here, nodes belonging to the same types are organized in clusters. Indeed, we observe that this is quite often true in real data and we have in fact designed our algorithm such as to make use of this property.
This would change immediately if even one object with the four ubiquitous attributes as well as {publisher} and no other attribute were present in the context. In this case, we would wind up with just one large cluster where all objects are connected in the sense of our definition. If we are of the mind that frequently, clusters are closely related to types, how would we prevent this context from translating to one single type? We will see in the following section that we have indeed some answers to this question.

4.3 Formalizing assumptions into an FCA-based algorithm

We will now describe in detail an algorithm we designed in order to partition a context that resembles the real types as closely as possible. In order to achieve our goal of inferring optimal relational schemata for semi-structured datasets with the help of Formal Concept Analysis, our main activity is to make repeated small changes to the objects in our context. For example, in figure 3, we could just erase the *booktitle* attribute of node 3, making it identical to its parent node, which is the node with most own objects containing book objects. The information lost—we will see that it is not really lost—is minimal, yet the context quite significantly cleaner after this operation. The closest thing to this operation we found in the literature would be the case where a developer adds a missing `va_end` after only calling `va_start` in a method, after an analyst using Lindig's method [14] makes them aware of that fact. In order to formally describe this process, which really is the cornerstone of our approach, we borrow some terminology from graph theory:

Definition 7. We call the operation of setting the intent of a node B to that of a node A **merging B into A**.

Our algorithm is based on the idea of always finding the best such merge and performing it, until we run out of possible merges. As we will see, there are different conditions that make certain merges impossible. We will only ever merge connected nodes.
When merging nodes, we set the intent of one node to that of the other (by convention always B into A, never the other way around). Concretely, that means we remove all attributes that aren't in the larger node from all objects in the smaller node, and fill up all attribute columns

that appear only in the larger node with NULL in the objects of the smaller node. Since these nodes are by definition not equal, at least one of these operations will happen in every merge step.

Definition 8. If a node is merged into one of its parent nodes, the child node has at least one more attribute than the parent. In order to not lose that information, we write the key-value pair into a "legacy data" field. We call this pair a **legacy value**.

Definition 9. If a node is merged into one of its child nodes, it will have at least one attribute less than the child node. Just like it would be in a relational database table, we call these new non-existing values **null values**.

Note that every merge will either raise the total legacy count of a context, or the total null count, or both (if a node with legacy values is merged into a child node, or if a node with null values is merged into a parent node). We are keeping track of the percentage of null and legacy values during all our tests. Note that these percentages as we use them do not mean the exact same thing: Null means the share of all cells in all tables that is NULL; legacy means the share of all values in all tables that are not written in a column with that value's name. There can be more than one legacy value in any given legacy cell post-transformation.
This is our algorithm in a nutshell: Build the initial lattice. Check for possible merge node pairs. Perform the best merge. If there are still candidates, merge the next best candidate pair, and so on until there are no more possible merges to perform.
A key question here is: How can we assess which pairs of connected nodes are good merges, and which are not? After all, the order of merges and with it to some extent the quality of the result is determined by that measure. Stemming from our understanding how objects are represented in semi-structured data—that is, with an implicit archetype to which the individual objects conform to a variable degree—, we have measured the merge score of two connected nodes in the following, straightforward way:

$$mergeScore = \frac{|A|}{|B|^2}$$

Here, we define $|\cdot|$ to be the number of own objects of a node. Other than in [14, p. 27], where the denominator is not raised to the power of 2, we place extra weight on how many own objects our smaller node has. Reformulated, we compute the ratio between own objects of both nodes, divided by the number of own objects of the smaller node. That way, we are making sure to merge outliers, of which only very few exist, first, and larger nodes—even though the ratio of own objects may be the same—later. This is important for one crucial feature that we will discuss after providing an example.
In the following, we describe our algorithm in pseudo-code. Note how we rebuild the complete lattice based on the changed intents of nodes. This is ultimately much simpler—and given the moderate size of our contexts, not considerably slower—than just updating the lattice locally.

Algorithm 1 LatticeMerge

```
function LatticeMerge(L)                                   ▷ L is our lattice
    score ← FindAndMergeNodes(L)
    while score > 0 do
        L.rebuild()
        score ← FindAndMergeNodes(L)
    end while
    return L
end function

function FindAndMergeNodes(L)
    A, B ← NIL                                             ▷ A, B are nodes
    score, scoreTmp ← 0
    for all connected pairs of nodes (C, D) with |C| ≥ |D| do
        scoreTmp ← MergeScore(C, D)                        ▷ |·| = number of own objects
        if scoreTmp ≥ score then                           ▷ NB: C, D connected → D ≠ ∅
            score ← scoreTmp
            A ← C
            B ← D
        end if
    end for
    if score > 0 then
        for all objects O in Node B do
            O.intent ← A.getNodeIntent()                   ▷ Merges objects from B into A
        end for
    end if
    return score
end function

function MergeScore(A, B)                                  ▷ A, B are nodes in the lattice
    return |A|/|B|²
end function
```

4.4 Example

In this section, we shall give a thorough example of what our algorithm does. We also introduce the concepts of *legacy* and *null* values, used again later in section 6.4.
Recall our example context from section 4.1 (cf. figure 4). Let's have a look at how our algorithm transforms this context.

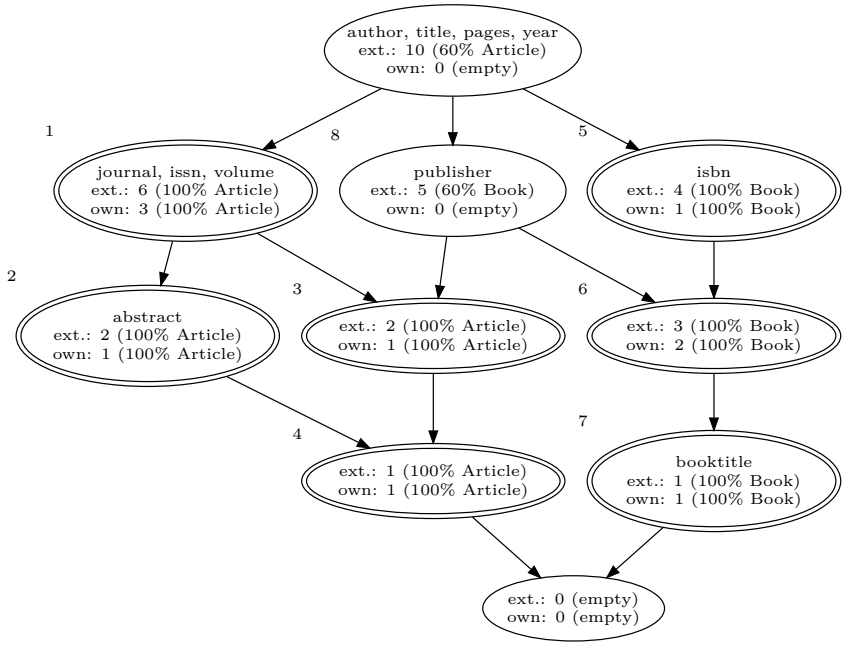

Figure 4: Example bibliographic context (again)

In the first step, the highest *mergeScore* will be 3, and both the node pairs $(1, 2)$ and $(1, 3)$ will have it. In this case, we visit $(1, 2)$ later than $(1, 3)$ and the two pairs have equal merge scores, thus, we merge 2 into 1. This produces the lattice shown in figure 5.

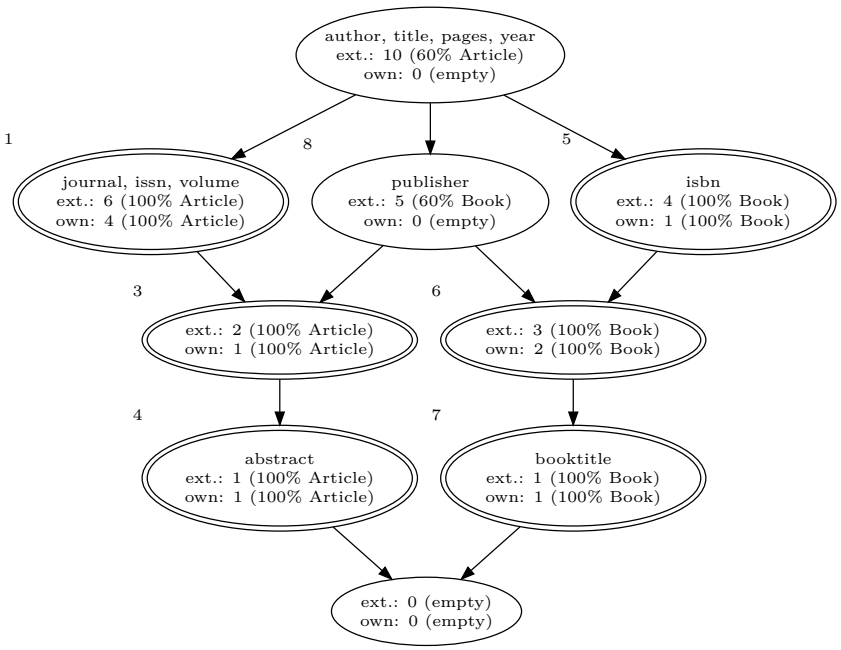

Figure 5: Example lattice after one merge operation

Note how node 1 now has 4 own objects instead of 3 as before, and how node 2 has disappeared entirely since it has been merged into node 1 (in this example, we keep the numbering consistent across all figures). We also note that the clustering of the context is still the same, with an easy mapping from clusters to object types (article, book). Looking at the statistics, we see that we now have a *legacy* value of 1.429. That means that in our relational target database, 1.429% of all attributes are not stored neatly in a column that stores exactly this attribute, but in a "legacy data" column, possibly in a field with other such "legacy" attributes. Checking the numbers, we see that we have:

1 object with 5 attributes,
2 objects with with 6 attributes each,
5 objects with 7 attributes each,
1 objects with 8 attributes each, and
1 object with 9 attributes

for a total of 70 attributes. Our first merge operation had exactly one attribute "stripped off" its object, namely *abstract* from the object in node 2. Now, $\frac{1}{70} = 1.429$—this is how the legacy value is computed.
Looking at figure 5, it is easy to see that the next merge will be node 3 into 1, and then 4 into 1. After these steps, all nodes corresponding to article objects have been consolidated and the publisher attribute has been stripped off all article objects, as can be seen in figure 6.

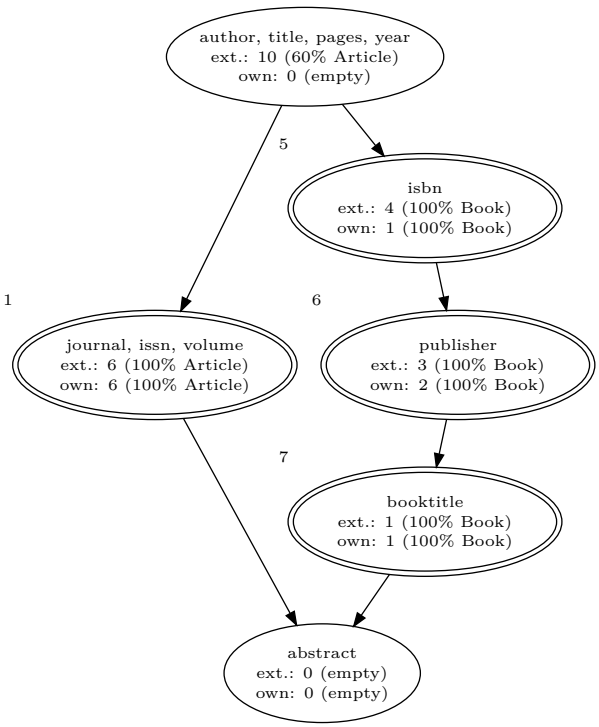

Figure 6: Our lattice after three merge operations

At this point, our legacy value is 5.714, and it is no accident that this is 4x the value from above. We consolidated into node 1 from figure 4: two nodes that each had one attribute more, and one node that had two attributes more, yielding a total of 4 out of 70 attributes that are now stored in a legacy field.
Looking at figure 6, the next merge can be either node 5 into 6 or 7 into 6—both have a *mergeScore* of 2. Since we traverse the lattice top-down, the last pair with the highest *mergeScore* we find is $(6, 7)$. We perform this merge, raising the legacy score to 7.143, and lastly our only downward merge in this context with $(5, 6)$. We end up with the final context that doesn't have any pairs of connected nodes anymore, shown in figure 7.

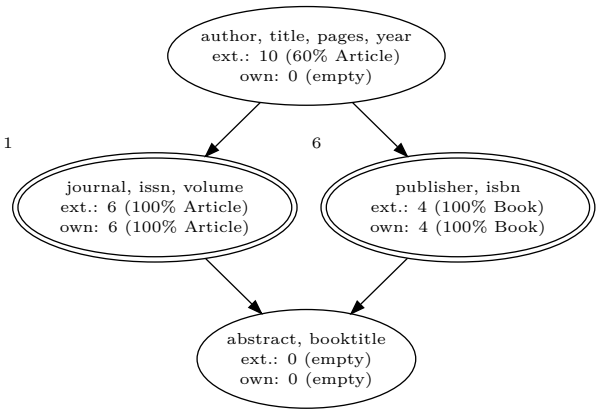

Figure 7: Example context after the last merge step

After this last merge step, we end up with a null value of 1.515 and the legacy value of 7.143 from above. Since this is our only downward merge, our relational tables 'wrap' tightly 'around'

the other values, and the missing *publisher* in node 1 is the first null value. Thus, we have:

6 objects in a 7 column table, and
4 objects in a 6 column table

making for a total of 66 cells, of which one is empty. This is how we compute null values.
Some general remarks. When we say that we only merge connected nodes and don't specify our break conditions, why can't we just look at all clusters and define them to be our target types? The answer is that clusters can actually "fall apart" in our algorithm. For example, there may be only one node with own objects connecting two otherwise unconnected clusters. As soon as this node and one of its parent or child nodes have the highest merge score, we perform the corresponding merge, after which the two clusters are no longer connected, because this one node has been merged into the node with the most own objects it was connected to—which of course lies in only one of the clusters. In this case, we had two different concepts with a few objects sharing many of their respective attributes. More concretely, imagine node 8 in figure 4 having exactly one own object. We would first perform the three merges on the article side, effectively severing the connection between the article and book clusters when we merge node 3 into node 1. Node 8 would eventually get merged into node 6, like both other book type nodes.
During testing, we experimented with not just considering connected parent and child nodes for merging, but also "horizontal" neighbors—that is, nodes with own objects that share a parent or a child node with a given node. However, when also considering these nodes, we almost always ended up with just one node in the end or another number that was far smaller than the number of types in the context. Extending the definition of connectedness to also encompass this relation, the transitive connectedness closure of almost any object was the whole lattice. Note that two such "horizontal" neighbors can still end up merged together after termination of the algorithm—they must have, or in the process of merging, obtain, a shared parent or child node with own objects, however.
Of course, this has been a constructed example that by design works very well with our algorithm. We will see in section 6 that there are indeed some semi-structured datasets that conform to our assumptions about semi-structured data quite well—as well as some that are more complex.

4.5 Parameters

However simple our algorithm is, there are many things about it that we can customize.

1. We can calculate the *mergeScore* in whichever way we want; it doesn't change anything fundamental about the algorithm. It will still repeatedly compute the best pair of nodes and merge them.

2. The algorithm doesn't have to terminate only when the score is 0; any value can be the threshold. Note that scores < 1 are in fact quite weak candidates the way we compute the score. (If two nodes both have exactly one own object, the score will be 1, but if two nodes both have exactly 2 objects, the score will be $\frac{1}{4}$ and so on). However, we have found that we hardly make too many merges with the threshold set to 0, but that of course depends on the data.

3. In this configuration, the *mergeScore* of two connected nodes is always > 0. However, we can define some cases in which we return 0 and thus will certainly not perform a certain merge. We have implemented this behavior in two of our three merge configurations (cf. section 6). One such condition would be if the child node has three or more attributes

more than the parent node, marking a significant semantic distance. The other is that if the child node has an own attribute, we do not perform the merge in either direction.

4. We could also configure the *mergeScore* function to look at quite some other features of the lattice other than just the two nodes being merged. We have however not done that in this work and will only briefly discuss it in section 7.

5 Implementation

In order to test our algorithm with various real life datasets, we developed a Java program with various functionalities. Here, we describe its main functionality. The program is licensed under the WTFPL[3] and accessible on GitHub[4]. All information regarding its use can be found in the readme file on GitHub.
Our tool performs the following steps:

1. Read semi-structured data from one or more repositories at a time. Supported file formats are JSON, XML, and BibTeX.

2. Generate a formal context from the input data where the records are the objects and their attributes are the attributes. Output this context to the .cxt Burmeister format (cf. appendix A.4, "Anleitung zum Wissenschaftlichen Arbeiten").

3. Perform the manipulations to the context defined by our algorithm and output detailed results for every step (e.g. total number of nodes after each step, null and legacy values, and all other values defined in section 6.4). The algorithm can be parametrized.

4. Output, also for every step, a .dot file that can be converted to a number of image formats, including .png and .svg, with GraphViz. This file represents the lattice at a certain point in the algorithm. All lattice visualisations used in this work—except for the screenshots in appendix A.4 have been created this way.

5. We also output some GraphViz commands that can simply be copy-pasted to the terminal in order to obtain image files.

There are a few noteworthy things about the program.

1. For JSON and XML files, two parameters have to be defined by the user. One is the name of the elements that will be the objects, the other is the attribute name where the type is stored.

2. In general, there are a number of things we allow the user to parametrize. As we will see in the next section, we have three options for the merging process, each of which can be enabled or disabled independently. One option also has a suboption that has no effect on the other two options. In total, there are 12 possible configurations the algorithm can run in. Also, as we have seen, we defined a threshold for the *mergeScore*, and if the actual value is below the threshold, we will not perform any merges anymore. That is also parametrizable, along with the maximum number of records considered for the lattice. (In the .cxt file, we always have the full context, but this is much cheaper computationally).

3. Files can be processed individually or folder-wise. There may be files of different formats in the same folder, we choose the right parser at runtime.

[3] `http://www.wtfpl.net/`
[4] `https://github.com/lucaliechti/FCAInference/`

In the remarks above, we have mentioned the computational complexity of the algorithm. In fact, building the lattice is extremely expensive, as is detailed in a widely cited paper by Kuznetsov and Obiedkov [18], where different algorithms are compared and benchmarked. Because of its above-average performance on different test sets and its relative ease of implementation, we opted for Norris' algorithm [19]. The worst-case time complexity of this algorithm is $O(|G|^2|M||L|)$, where $|G|$ is the number of objects, $|M|$ the number of attributes and $|L|$ the size of the lattice, i.e. the number of nodes.
In our tests on a dual core Intel i7 with 2.5 GHz and 16 GB RAM, the high computational costs for building the lattice became painfully clear. While one run of our algorithm hardly took more than a millisecond per iteration even for our largest contexts, building the lattice took anywhere from that amount of time to two, sometimes almost three orders of magnitude longer.

6 Results

In this section, we present the results of our experiments with different NoSQL datasets. It is organized as follows: First, we introduce our datasets. Then, we present some standout results, i.e. cases where our algorithm works especially well or poorly. This should help familiarize the reader with the whole process. After that, we are turning to a more general analysis, where we measure a few key statistics of the context before and after merging, and with the different options on and off.

6.1 Datasets

We obtained datasets from three main sources, all containing bibliographic data. In addition to these, we processed a few unrelated datasets as well.

zbMath The first source is zbMath[5], a search engine for mathematical literature. It is possible to export obtained search results to various data formats, including XML and BibTeX, but only 100 search results at a time. We generated 12 BibTeX datasets out of those; 6 having 100 objects and 6 having 500 objects, pasting 5 exported sets together manually. The objects are always the first ones that the search for a given term yields. We are naming these datasets zb_size_xyz, where size is either 100 or 500 and xyz is the term that produced these search results, e.g. zb_100_algorithm.

dblp The second source is dblp[6], a similar service to zbMath but with a focus on computer science. Dblp allows exports of up to 1000 search results to XML or JSON, and up to 100 results to BibTeX. We generated 6 XML datasets that contain the first 1'000 search results for some search terms. These datasets are named dblp_xyz in a similar fashion to the zbMath datasets.

IESL The third source is a collection of over 4'000 BibTeX libraries, with anywhere from less than ten to several thousand records each, found on the website of the Information Extraction and Synthesis Laboratory (IESL)[7]. We analyzed 10 datasets with at least 100 entries, but for feasibility reasons considering only the first 100.

[5] `https://zbmath.org/`
[6] `http://dblp.uni-trier.de/`
[7] `http://www.iesl.cs.umass.edu/data/bibtex`

That means in total we have 16 datasets with exactly 100, 6 with exactly 500 and 6 with exactly 1000 objects. We are naming the IESL datasets iesl_x_y, where x and y are two keywords from the original filename. See appendix A.1 for exact references. Unrelated to those sources, from which it was possible to obtain sets of datasets, we have analyzed the bibliography of the SCG at Bern University (BibTeX, 8662 objects), a list of computer science professors and staff working at Swiss universities (JSON, 293 objects), and Saharon Shelah's mathematical bibliography[8] (BibTeX, 1137 objects).
The quality and cleanliness of datasets varies considerably between sources, and so do the results we achieve. Generally, the data obtained from zbMath and dblp are much more regular and have far fewer outliers than the ones from IESL and the unrelated datasets.

6.2 Tested configurations

As described in section 4.3, there are many parameters in our algorithm that determine which nodes to merge and which not, when to stop, etc. We have chosen 3 of those that we either enable or disable, yielding 8 different results per dataset. These are the parameters:

1. **Limiting the difference in number of attributes** to 2 when merging nodes: If this option is enabled, and if we are calculating the merge score of two nodes that are connected, but the child node contains 3 or more attributes more than the parent node, the merge score is set to zero. We use the term *attribute difference* for this option.

2. **Removing unique nodes:** If this option is enabled, before the merging takes place, we remove all objects from the context whose intent exists only once in the context, i.e. in this very object, and which don't have parent or child nodes with own objects. The latter are excluded to keep the context "dense" where nodes are not necessarily outliers, in order to allow for more merging possibilities. After the merging, we retrofit these removed objects back into the context, assigning them to the node with the most similar intent. We keep statistics from before and after that process of retrofitting, effectively yielding 12 total results per dataset (4 without this option, 4 with this option before retrofitting, 4 with this option after retrofitting). We call this option the *retrofit* option.

3. **Forbidding merging into nodes with own attributes:** If this option is enabled, whenever we encounter a node with an own attribute in the lattice, this node's and all parent nodes' merge scores are set to zero. The scores with this node's child nodes are calculated as usual, because these child nodes have the node's attribute. We interpret this rule as giving more weight to attributes that are only present in a node and its child nodes and therefore appear only 'locally' in the lattice. We are referring to this option as the *own attributes* option.

Whenever we run the algorithm, all of these options will either be enabled or not. We will write these configurations as three-digit codes, each digit taking values 0 or 1, depending on whether the corresponding option is enabled or not. 000 would be our algorithm with no attribute difference limit to the merging, without removing unique nodes, and with merging any two nodes together, regardless whether the child node has own attributes, and so on. We have a special column for the retrofit value. If this column is empty, nothing was removed in the first place (i.e. the code is $*0*$). If the value is 0, the results describe the lattice before merging the outliers back in, and if the value is 1, the lattice after the retrofitting is considered.

[8] `http://shelah.logic.at/listb.bib`

6.3 Exemplary results

We have a few combinations of merge configurations and datasets where the results are *perfect* in the sense that all objects are located in nodes that only contain objects of *exactly one* type after the merging, and no two nodes with objects of the same type. I.e. if we have books, articles, and theses in a context, we expect to end up with 3 nodes, each of them containing only objects of one of these types.
We have 12 such combinations, as shown in table 7.

config	retrofit	dataset
000		zb_100_Number
100		zb_100_Number
000		zb_100_Algorithm
010	0	zb_100_Algorithm
100		zb_100_Algorithm
110	0	zb_100_Algorithm
000		zb_100_Integer
100		zb_100_Integer
000		zb_100_Compute
010	0	zb_100_Compute
100		zb_100_Compute
110	0	zb_100_Compute

Table 7: Contexts and Configurations with perfect results

We notice a few things right away. All datasets in this list are from the zb_100 exports, which are by far the cleanest contexts with 3 different types at most. (The zb_500 datasets all have exactly 3 types). Since regardless of the exact configuration, our method is rather susceptible to minimal changes in the data, this is no surprise. Also, we find that the *own attributes* option—the restriction to never merge into nodes that contain own attributes—never achieves a perfect result in our tests. This is no surprise either, because all three options were designed to help deal with contexts that violate some of our assumptions detailed in section 4.2. In these small contexts, however, the data are already so well structured as to render any changes to the most basic form of our algorithm not only useless, but even counterproductive. We will see later, however, some examples of the third option achieving very good results in terms of context cleanliness.
For zb_100_Number and zb_100_Integer, applying the *retrofit* option and thus removing all unique unconnected nodes with no neighbors with own objects prior to running the algorithm makes a result non-perfect that was perfect without that option. In both those contexts, there is a single incollection object with a unique combination of attributes that gets removed and merged back with this option enabled, but which should just remain in the lattice as is. In the other datasets, switching the *attribute difference* and *retrofit* options on and off did not have an effect on the perfectness of the result.
Let us look at one such perfect result now, namely zb_100_Number with configuration 000 (figure 8).

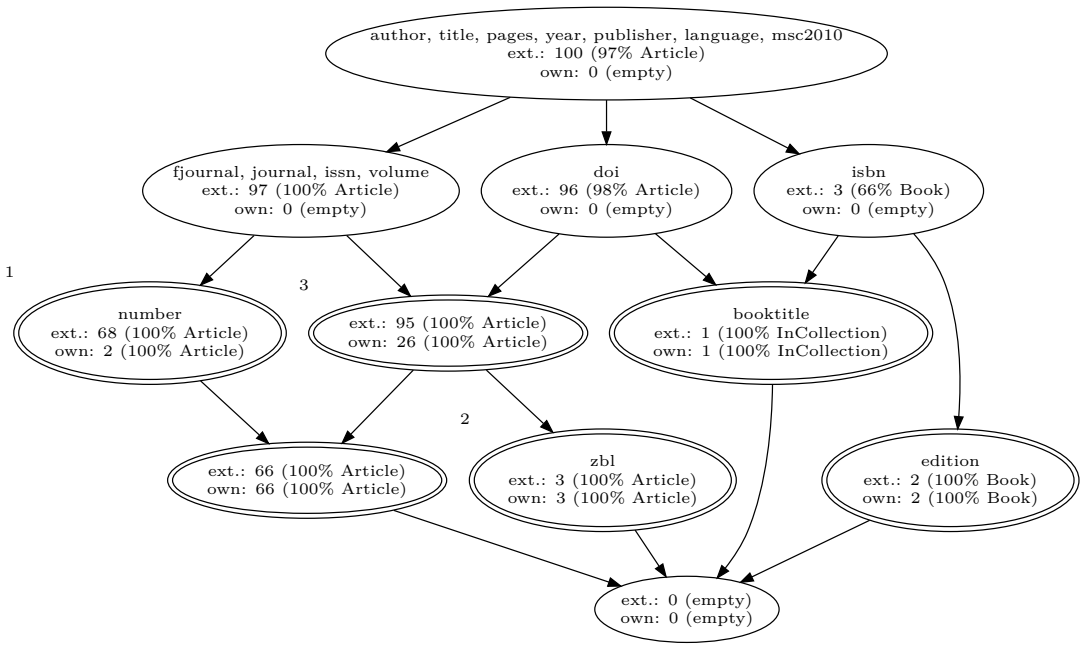

Figure 8: zb_100_Number before merging, configuration = 000

Clearly, this context conforms perfectly to the assumptions we have made about contexts. First of all, in the original context, all nodes only contain objects of one type. (We have in fact some contexts in our test data where this is not given.) Second, almost all objects have the same (article) type, but are distributed over four nodes. The few objects that are not articles are sufficiently different in terms of attribute composition, and are even unconnected in the lattice. This ensures the nodes that contain them are not merged into any other node and vice versa. Third, all article nodes are connected in such a way that the cluster does not get split up during merging. The first merge that happens is node 1 into its child node, then node 2 into its parent node, then node 3 into its only remaining child node. After that, we have arrived at the state depicted in figure 9 where there are no further merges to perform.

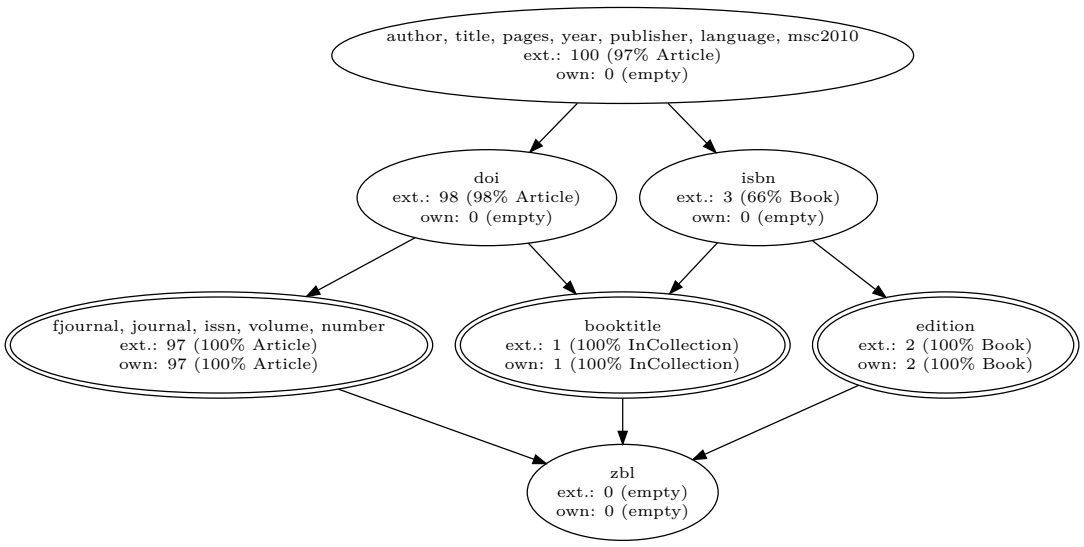

Figure 9: zb_100_Number after merging, configuration = 000

Note how in figure 9, attribute *zbl* is associated with the bottom node of the lattice, which itself contains no own objects. We find that in the second merge, this attribute gets "stripped off", because it is much more economical to write it 3 times into a legacy data field as opposed to adding a "zbl" column to the article table which will be NULL *all but* 3 times.
This is indeed a context for which our algorithm produces the ideal result. However, when experimenting with the exact same search results from zbMath, but considering the first 500 instead of the first 100 results, the results are not perfect anymore. Yet despite having one node with own objects in excess, we still consider this result very good.
Adding another 400 objects—as noted above: from the very same set of search results—to the context adds some variety in the form of new combinations of attributes. However, this is still a context that is very well suited for our approach. In figure 10, we see in nodes 1 and 2 the book and incollection objects from zb_100. In this context, there are 2 more book objects with a different intent. Since these also have no parent or child nodes with own objects in the lattice, they will also remain the same throughout the merges, leaving us with 2 book nodes in the end. The rest of the nodes contain article objects and merge together perfectly, since they are all connected and those nodes with many own objects tend to be in the center of the article cluster and the outliers away from the center, as shown in figure 11.

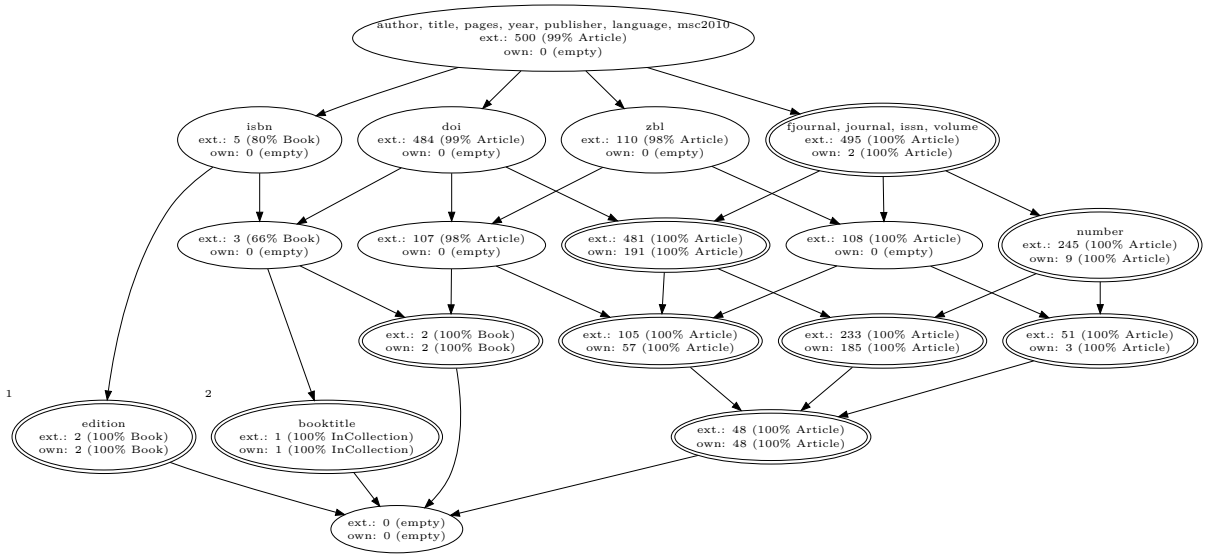

Figure 10: zb_500_Number before merging

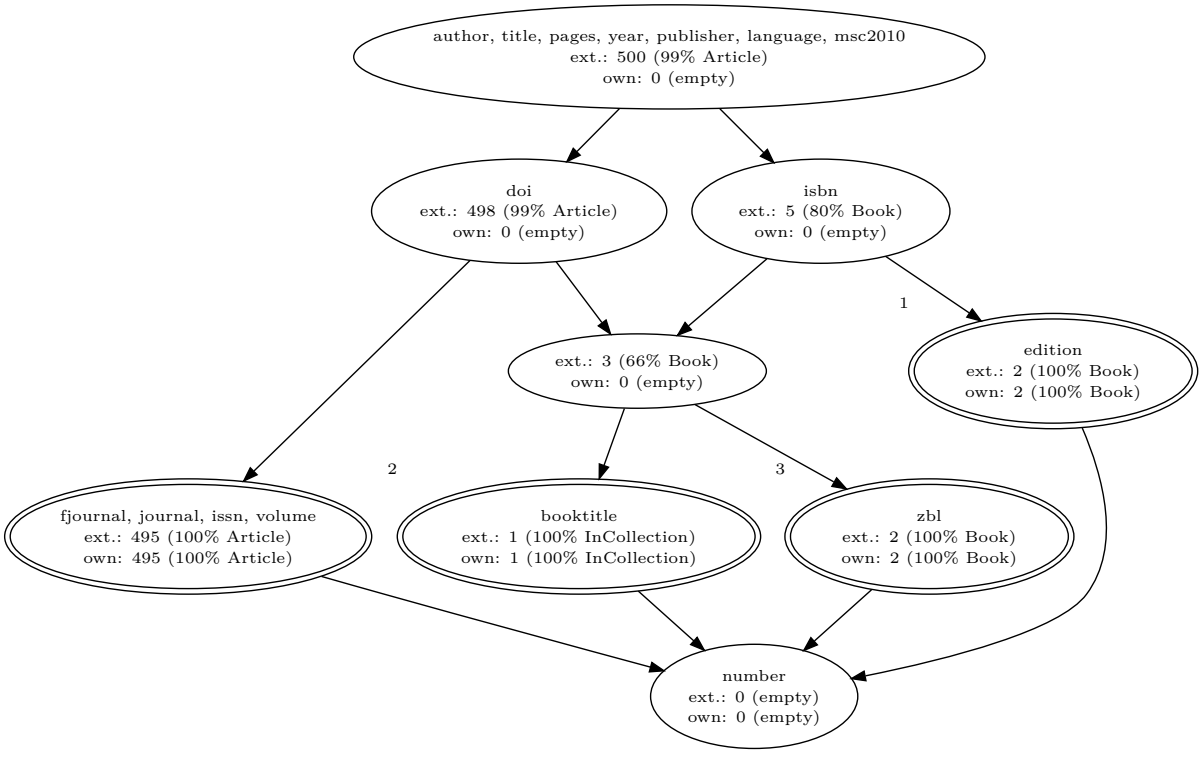

Figure 11: zb_500_Number after merging, configuration = 000

One recurring theme of this work presents itself here. Note how in figure 11, nodes 2 and 3 are more alike than nodes 1 and 3, if we only go by the attributes they have and don't have (the node numbering is consistent with this lattice's earlier state, cf. fig. 10). Node 2 has no *zbl*, and node 3 has no *booktitle*, but apart from that they are identical. Node 1 has no *zbl* either, and node 3 has no *edition*, but on top of that, node 1 also doesn't have a *doi* like node 3. However, nodes 1 and 3 have the same type, and node 2 has a different type. This is one of many examples where our data simply doesn't correspond to our assumptions about similar objects having similar attributes. In this case, not merging any of these nodes is clearly the best solution, because simply based on attribute similarity, we would have to perform the merge of nodes 2 and 3, and that would be wrong.
The *own attributes* option forbids merging into nodes with own attributes. It turns out that for the zbMath and dblp datasets, this option prevented quite a few wrong merges, as the following example illustrates (the output is the same for all configurations and differs only concerning the *own attributes* option).
In the zb_500_Groups dataset, we perform one wrong merge if the *own attributes* option is disabled, and we don't perform a good merge when it's enabled. In figure 12, we see that the incollection node contains one non-incollection object, whereas in figure 13, two nodes with many objects are not merged, even though they both contain only article objects. Apart from that, the fully merged contexts do not differ.

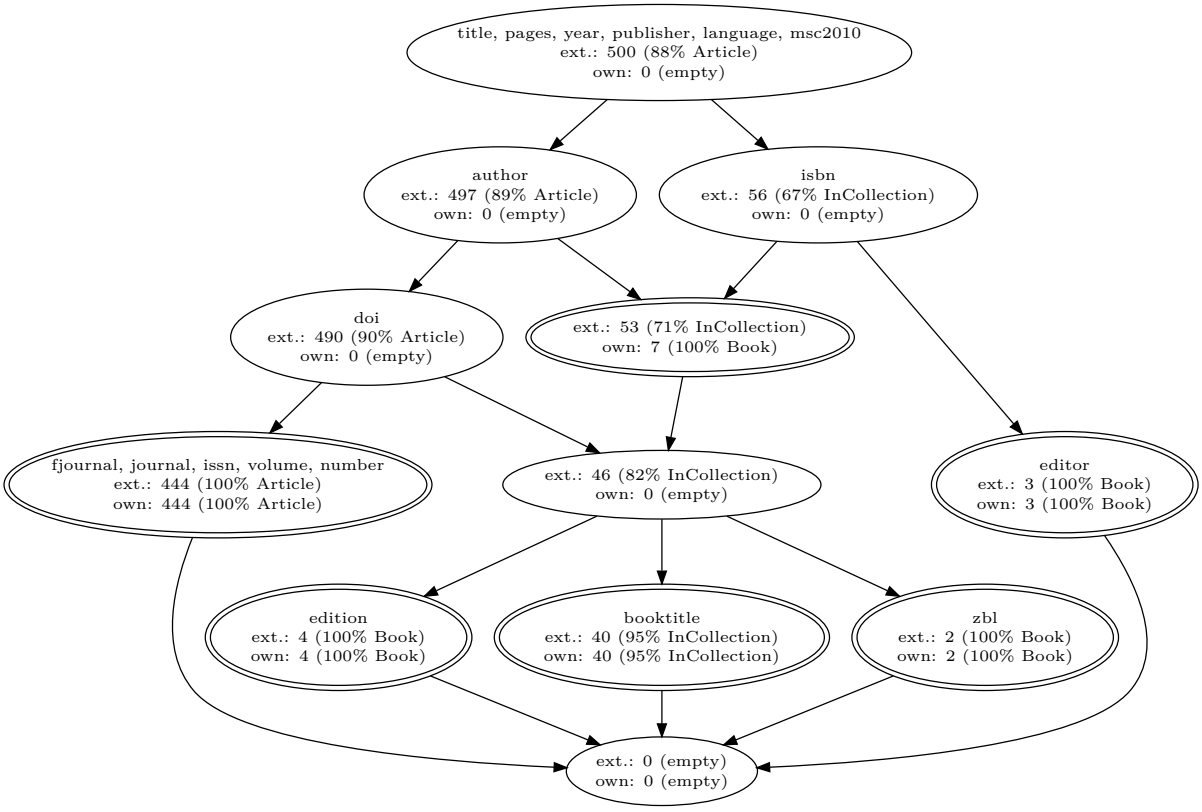

Figure 12: zb_500_Groups after merging, configuration 000

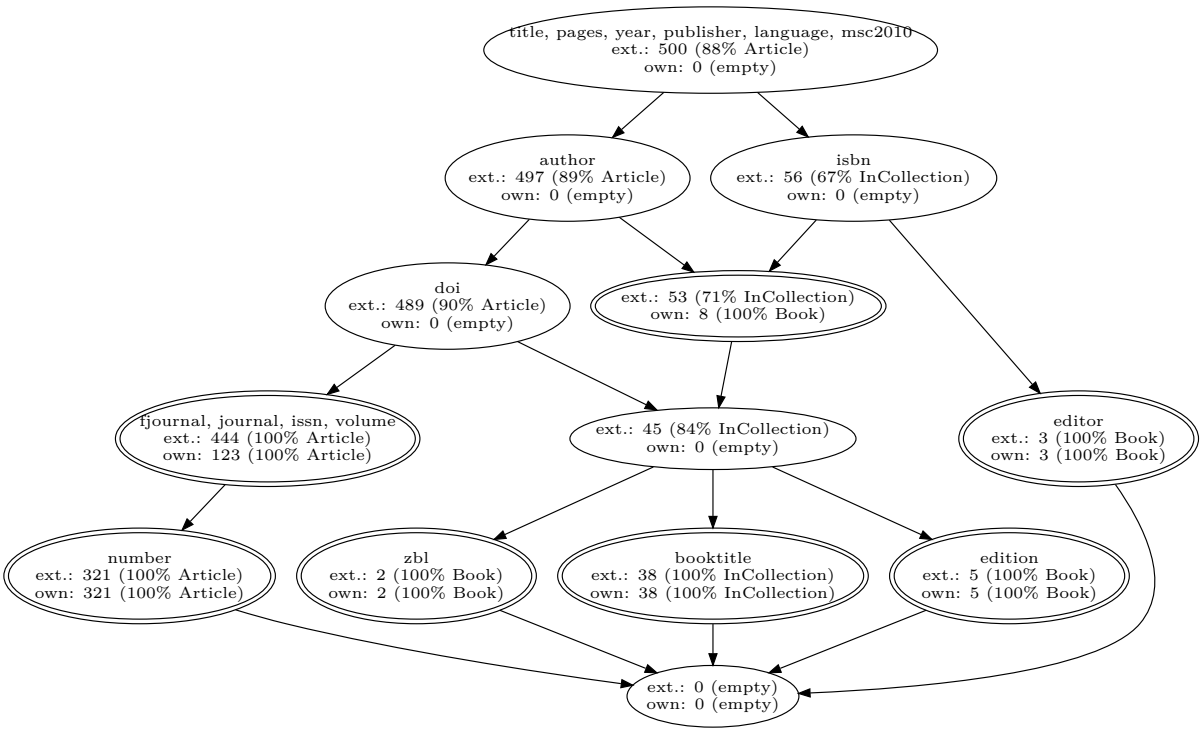

Figure 13: zb_500_Groups after merging, configuration 001

There is no general answer to the question which of these results we prefer, but in this case, the two article nodes in the second lattice can be confirmed to be the same very easily by a domain expert, while the wrong book object in the first lattice would have to be searched for more painfully. In section 6.5, we will analyze the *own attributes* option's cleaning potential in more detail. Let us now look at one of the two main problems that we can have with our data: That the similarity of the objects is too high. (We will look at the second problem, that the similarity is too low, after that.)
By just looking at the numbers, the dblp datasets are great candidates for our algorithm. We get a very small number of nodes after merging everything. However, this number is too small, i.e. smaller than the number of types in the dataset. We illustrate this problem with the dblp_Inference dataset in figures 14 and 15.

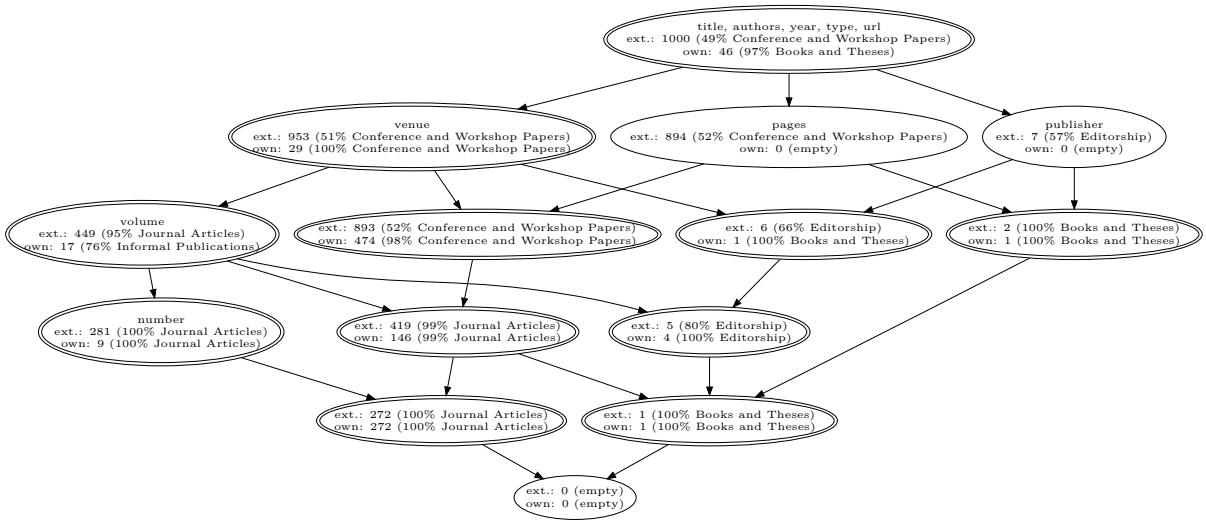

Figure 14: dblp_Inference before merging

We can see right away why we get such a small number of nodes after merging: In spite of containing 7 object types, all nodes with own objects are connected and, depending on which nodes have how many objects, can be merged together into one single node, in the most extreme case. Almost all nodes have a large majority type, so a favorable merge result is theoretically possible.

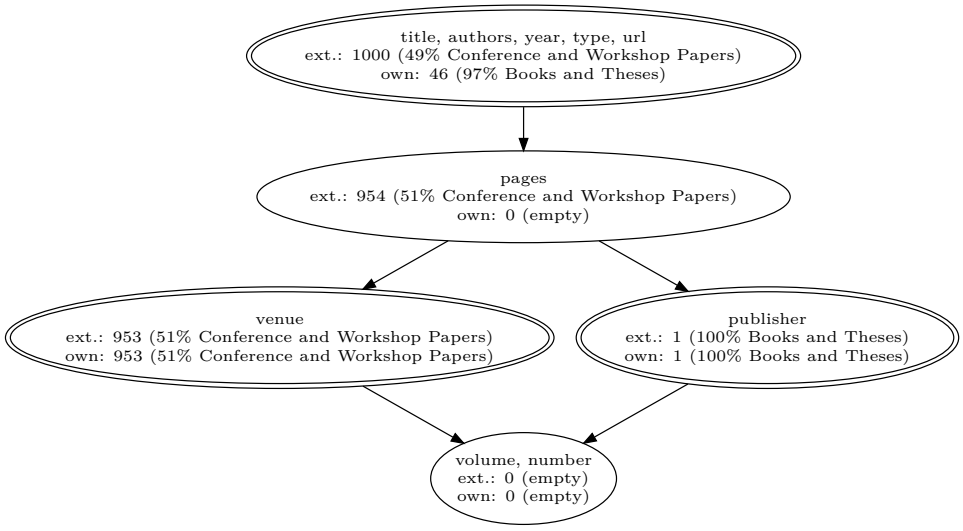

Figure 15: dblp_Inference after merging, configuration 000

Indeed, with the *own attributes* option disabled (and regardless of the exact configuration of the other options) we get a lattice with only three nodes with own objects left, one of them only barely having a majority type that is an actual majority of objects, and one being a single object in a context of 1'000 objects. Clearly, this result is not what we were looking for. If we enable the *own attributes* option, the result is much more friendly, but still nowhere near ideal (figure 16).

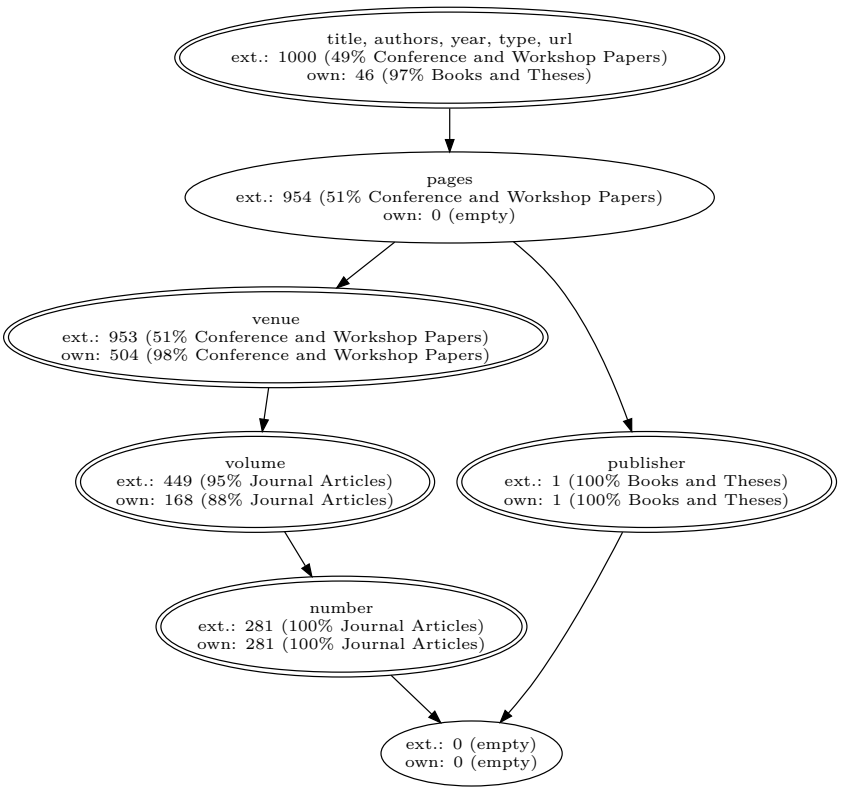

Figure 16: dblp_Inference after merging, configuration 001

We can see that enabling the *own attributes* option (not merging into nodes with own attributes) prevents the attributes *volume* and *number* from being "shaved off" the lattice, instead defining an own node each, and in both cases with a rather large type majority. This is what the *own attributes* option was introduced for, and here, it works well. But that is not enough to keep the lattice clean.

The big question arising from analyzing the results of the dblp datasets is: How can we determine if we have merged enough nodes and can stop the algorithm before it terminates naturally?

Since we perform only one merge at a time and choose the merge based on a score with a rather intuitive meaning, we tried setting thresholds for the score and only performing the merge if that threshold was reached, i.e. the merge had a certain significance. Outputting every merge step's value, we tried to find out at which point for which dataset the algorithm would ideally terminate. Unfortunately, even across datasets gathered from within the same source, the score at this ideal stopping point varied considerably, often by an order of magnitude or more. We also couldn't find a suitable ratio of the current *mergeScore* to the one of the last step vs. the last to the one before that etc.

We have to leave the question of stopping the algorithm open, as we suspect it might not have an easy solution.

The problem that our objects are too similar to each other (i.e. connected in the lattice) is probably less prevalent in big contexts. When looking at the IESL datasets, even though only consisting of 100 objects, we find such a large number of distinct nodes that are often the only one with a specific attribute that we run out of options to merge long before we get close to the number of types in the context. On average, the IESL datasets contain 43.8 attributes, which is decidedly more than the zbMath (16.8) or dblp (10) datasets. While these contexts are too large to be sensibly visualized, we still give a simplified illustration of the cleaned up

context uwaterloo_reports—being one of the tidiest ones—in appendix A.2. But mainly, to understand these large contexts, we are providing a list of all attributes that appear only once in at least one of the IESL contexts in appendix A.3, and we indicate in how many IESL contexts they appear (except where they appear more than once). From looking at that table, it becomes obvious that the data stored in these contexts is quite simply very diverse. For most of these attributes, it is immediately clear what they mean in a bibliography context (e.g. "abstract", "comments", "description", "glossary", "keyword" and most other terms). We have two cases where attributes both appear in singular as well as in plural ("descriptor" – "descriptors"; "keyword" – "keywords"), something that could be checked for during the creation of the context; we have a few pre- ("xxisbn", "xxpages"', "xxpublisher") and postfixed attributes ("address-l"; "bdsk-file-2", "publisher-l", "url-2"), but the vast majority of these attributes denotes something of its own and cannot simply be mapped to some other value.
Unfortunately, we are of the mind that cleansing such extremely diverse contexts is a very delicate and difficult matter.

6.4 Success measures

In the following section, we present the aggregated results of our processing in tabular form, with the following key measures:

obj The size of the dataset. As written above, we mostly have datasets with exactly 100, 500, or 1000 objects.

types The number of types in the dataset. This value acts as our type ground truth with which we compare the number of nodes after the merging. This and the following three measures are compared pre- and post-processing for all datasets.

nodes Number of lattice nodes *with own objects*, i.e. the number of different combinations of attributes in objects in the dataset after the merging.

major Percentage of objects that belong to a lattice node where this object's type is the most common among object types. Note that in some cases, for nodes with very diverse object types, this value can be less than 50%.

clean Percentage of objects that belong to a node that only contains objects of exactly one, i.e. this object's, type.

null The percentage of all fields in all tables that would be NULL after importing our processed dataset into a relational database. This number is proportional to the extra amount of storage space we need for that database.

legacy The percentage of all values in all objects of the original dataset which after the processing has to be stored in some kind of "legacy values" field in a relational database.

If row in a table represents more than one processed dataset, we always show the average over all considered datasets.

6.5 Global results

In this section, we are giving an overview of how the different parameters affected the outcome in general, i.e. for a set of datasets or all datasets together.
Comparing all applied methods will answer some questions about how well these methods work and which combinations are most favorable. However, our data is just too diverse to extract any meaningful information about the datasets this way. We will look at this kind of aggregate result a little later.
For now, we are splitting our datasets into two halves: clean and dirty datasets, with those terms obviously describing a degree and not an absolute state. The clean half is defined as all datasets from the dblp and zbMath exports, and the dirty half is the rest.
Indeed, the results we obtain when comparing the averages of some measures between the clean and dirty sets reveal some interesting properties of both the datasets and our configurations.

config	retrofit	nodes	major	clean	null	legacy
000		3.39	86.74	59.09	2.94	3.17
001		5.11	93.72	67.64	1.11	1.10
010	0	2.94	86.52	58.48	2.98	3.17
010	1	2.94	86.32	28.53	3.02	3.25
011	0	4.83	93.72	67.64	1.11	0.98
011	1	4.83	93.52	50.10	1.13	1.06
100		3.61	86.76	63.98	2.78	3.04
101		5.27	93.73	69.59	1.10	0.96
110	0	3.16	86.53	63.37	2.82	3.04
110	1	3.16	86.33	33.42	2.86	3.12
111	0	5.00	93.73	69.59	1.10	0.84
111	1	5.00	93.53	52.01	1.12	0.92

Table 8: Obtained values from running all clean datasets (dblp and zbMath)

For the clean datasets, the ground truth pre-processing averages are: nodes = 10.83, types = 4, major = 97.96, clean = 80.53.

config	retrofit	nodes	major	clean	null	legacy
000		27.31	88.06	69.54	4.84	5.27
001		31.08	89.20	73.03	3.85	3.80
010	0	14.54	86.92	66.35	5.48	5.63
010	1	14.54	81.29	40.92	10.89	8.74
011	0	19.54	89.43	72.82	3.60	3.85
011	1	19.54	84.20	48.88	9.24	7.26
100		51.23	94.32	85.33	0.38	0.55
101		51.00	94.25	85.02	0.42	0.52
110	0	38.08	94.18	84.66	0.38	0.62
110	1	38.08	87.17	67.68	7.05	4.80
111	0	37.00	94.04	84.47	0.49	0.69
111	1	37.00	87.09	66.30	7.04	4.81

Table 9: Obtained values from running the IESL and unrelated datasets

For dirty datasets, the according pre-processing average values are: nodes = 58.92, types = 8.62, major = 95.2, clean = 88.88.
Interestingly, we find that the diversity of a context has not much influence on the major measure, being less than 1 percentage point off post-processing between these two sets of datasets.

However, on average the clean measure is around 14 percentage points higher in the diverse contexts. This is no surprise. Diverse contexts, at least in our data, are not primarily diverse because they contain a huge number of different types and clusters centering around these types, but because they have many objects whose attribute combination is rare. Wrongly merging into one of these nodes lowers the clean index by much less than if we are wrongly merging into one of the nodes with many objects from the less diverse contexts. It should be pointed out that especially the zbMath datasets contain a few nodes that make up for most (sometimes 90%+ in 2-3 nodes) of all objects.
A finding that is no surprise at all either is that in those runs with the retrofit configuration—i.e. where we first remove unique values and fit them back into the data after the merges—, the null and unique measures are the highest. This makes sense: The nodes removed and fit back into the data are these that otherwise would stay untouched by the merging, since they do not have parent or child nodes with own objects, as per our criteria. It is quite intuitive that merging these nodes with the 'closest' available node (in terms of attribute composition) creates potentially many null and legacy values, especially (for nulls) if we consider that if two nodes have the same attribute distance to such a node, we merge the previously removed node into the bigger one, i.e. the one with more own objects, which will lead to all those objects having one more NULL value.
Note, however, that this disparity is much bigger in the less clean datasets. This has two reasons: First, we have many more outliers in these datasets, while in the clean datasets, there are not many at all. Second, it can be assumed that the attribute distance of the outliers in the clean datasets to the closest available node is probably smaller than in the other datasets, further contributing to that effect.
Lastly, from these tables we can see pretty clearly the different overall effects our algorithm has on the data. In the cleaner datasets, despite starting at more than two and a half times the actual value, post-processing the number of nodes with own objects ends up quite close to the real number of types in the dataset and is sometimes even smaller. That would mean that the algorithm works almost too well on this kind of data, merging nodes even where that is not necessary. This is due to the datasets extracted from dblp. These are clustered tighter than what our assumptions suggest. When we analyze these datasets in detail, we will see how it is very difficult to find out when to stop merging nodes.
On the other hand, for the not so clean datasets we do not come close to the real number of types in the data, and often these numbers are quite frankly far off. Furthermore, the configurations of our algorithm producing the best results in terms of the clean and major measures are quite clearly those that are reluctant to merge nodes, thereby leaving a more cluttered context for the domain expert to clean up. For dirty datasets, correlation between *nodes* and *major* is 0.78 and 0.76 between *nodes* and *clean*. This is a sobering result: For this kind of context, either we have not found the right way to merge nodes, or there simply is none, and we are left with a trade-off between correctness and completeness of mergings.
Returning to tables 8 and 9, we can now analyze some properties of the configurations that are context-independent, i.e. are the average from a wide variety of contexts.
Not surprisingly, enabling the *attribute difference* option (i.e. merging only nodes where one has at most 2 more attributes than the other one; configurations 1**) generates larger, cleaner contexts. All this option does is prevent some merges from happening, and the numbers we see are proof that there are actually merges that would have happened with that option off. In fact, when we look at the following table we can see that contexts that have been created with that option on have some 80% more nodes with own objects compared to contexts created without this option:

config	nodes	major	clean	null	legacy
0**	11.17	88.59	58.04	3.84	3.65
1**	20.08	90.83	67.15	2.24	1.99

Table 10: Comparison of values with the *attribute difference* option on and off

On the upshot, contexts with this option have some 2 percentage points more objects in majority nodes and are around 9 percentage points cleaner.
Since we are now looking at all datasets, we have to establish a baseline for these values before the processing. These values are: nodes = 31, types = 5.95, major = 96.8, clean = 84.03.
We observe the same effect for the *own attributes* option, which is not surprising either, because all we do here is again prevent some merges from ever happening. The exact numbers are shown in table 11.

config	nodes	major	clean	null	legacy
**0	14.70	87.43	58.67	3.71	3.61
**1	16.55	92.00	66.53	2.37	2.03

Table 11: Comparison of values with the *own attributes* option on and off

These look puzzling at first glance, and indeed we find them to be quite interesting. Can it be that this option seems to perform decidedly better on average than the first option, leaving just above 10% more nodes in the context, but producing more than 4 percentage points more objects in majority nodes? And it wouldn't be just that: not just relatively, but also absolutely speaking, the values we obtained with this option are clearly more desirable than those obtained with the first option, shrinking the context by some 3.5 objects on average, yet maintaining a major measure that is more than one percentage point higher.
We have to apply just a little more scrutiny to this result to find out what's really going on. If we compare the results that these methods yield in clean and dirty contexts separately, we gain a better understanding of the factors at play here.
These are the values we obtain from the clean datasets only:

config	nodes	major	clean	null	legacy
1**	4.20	90.10	58.66	1.96	1.99
**1	5.01	93.66	62.76	1.11	0.98

Table 12: Effects of the *attribute difference* and *own attributes* option on clean datasets

And these are the results from the dirty datasets:

config	nodes	major	clean	null	legacy
1**	42.06	91.84	78.91	2.63	2.00
**1	32.53	89.70	71.75	4.11	3.49

Table 13: Effects of the *attribute difference* and *own attributes* option on dirty datasets

Apparently, this effect is created by two opposing effects overlapping, namely, the *attribute difference* option making more merges in clean datasets and the *own attributes* option more in dirty ones. This isn't quite as exciting as the combined effect, but it should be pointed out that in the clean datasets, the ground truth average of types is 5.95, so the *own attributes* option is actually closer to that value and has better outcomes in terms of how clean the context is as well.
We are turning quickly to the option that probably yields the least unexpected results—the *retrofit* option—before comparing all configurations.

config	retrofit	nodes	major	clean	null	legacy
0		19.36	90.75	70.59	2.14	2.26
1	0	13.76	90.55	69.93	2.21	2.30
1	1	13.76	87.84	47.27	4.77	3.90

Table 14: Comparison of values with the *retrofit* option on and off, before and after retrofitting

Naturally, we have fewer nodes with own objects left after merging if we remove uniques before merging. It is apparent that over all datasets, we remove some 6-7 nodes on average. This lowers the major and clean values in quite an insignificant way (since each unique object always constitutes a clean and a majority node). Fitting the removed nodes back into the cleaned up context of course doesn't create any new nodes, but has quite a large impact on nulls and legacies. This is no surprise, because for any given unique node, the closest node has either stayed the same through the mergings or become even more different.
After having examined the different datasets and configurations in some detail, we now analyze all combinations of configurations on the full range of contexts before looking at some specific examples.

config	retrofit	nodes	major	clean	null	legacy
000		13.42	87.30	63.47	3.74	4.05
001		16.00	91.83	69.90	2.25	2.23
010	0	7.81	86.69	61.78	4.03	4.21
010	1	7.81	84.21	33.77	6.32	5.55
011	0	11.00	91.92	69.82	2.15	2.18
011	1	11.00	89.61	49.57	4.53	3.66
100		23.58	89.93	72.94	1.77	1.99
101		24.45	93.95	76.06	0.81	0.77
110	0	17.81	89.74	72.30	1.80	2.03
110	1	17.81	86.68	47.78	4.62	3.82
111	0	18.42	93.86	75.83	0.85	0.78
111	1	18.42	90.83	58.00	3.60	2.55

Table 15: Obtained values from running all datasets in all configurations

Since the *attribute difference* and *own attributes* options both simply prevent some specific

merges from happening, they generally increase the clean and major measures of the resulting context at the expense of leaving more nodes unmerged. Thus, it is quite expected that applying both those rules just increases this effect, which it indeed does (configurations with 1*1). It is noteworthy that stacking the *own attributes* option on top of the *attribute difference* one seems to yield quite nice results as compared to enabling only the *attribute difference* option. Regardless of the *retrofit* option, the number of remaining nodes is only slightly larger, while the clean and major measures are clearly higher. Comparing these results with tables 8 and 9, we can see that this effect stems mostly from the clean contexts, whereas in the dirty contexts, stacking these options on top of each other changed the outcome only marginally. Generally, in these contexts, the effects of the *retrofit* option were quite weak, which can be explained from the data. Apparently, in bigger and more diverse contexts you will find much fewer neighboring nodes with a bigger attribute difference than 2—there is probably often a node with 2 or fewer attributes between two such nodes.
Very generally, the *attribute difference* option seems to have a bigger impact on dirty contexts, and the *own attributes* one on clean ones. But only when combined, and only in clean contexts, do the two options complement each other in a desirable fashion.
Overall, the *retrofit* option generates the fewest remaining nodes. We observe that the *own attributes* option, if stacked on top of the *retrofit* one, has a much bigger impact than the *attribute difference* option. There is no obvious explanation for this phenomenon, but the *own attributes* option seems to prepare a more fitting context for the unique nodes to be fit back into.

7 Conclusion and further work

We have obtained mixed results in our experiments. Clearly, our assumptions on how exactly semi-structured data behave with respect to schema rigor have been either too optimistic, or not complete as far as the datasets labeled 'dirty' in the last chapter are concerned. Appendix A.2, where we have already simplified the visualisation, is the perfect example for that point. Often, data would simply be all over the place in terms of schema diversity, and even though we do not think our assumptions are flat out false, it is evident that these alone are not enough to describe and manage the diversity we found in our test data.
On the other hand, when working with data that is already pretty clean, we find that our method yields good results. This becomes clear when we look at table 8. Starting from a value two and a half times as big, we end up with a number of concepts that is reasonably close to the ground truth. Data does not get dirtied in a significant manner, as the major index does not change much at all and in all configurations, we end up with less than 4% of either legacy or null values, sometimes clearly less. These are cases in which our algorithm performs almost at its optimal level, since by design, at least one of these two values is always > 0. Since as we noted before, we are the first to try this approach (or an approach similar to it), this should inspire confidence in using FCA to infer schemata this way, for example in order to be able to also handle very diverse, messy datasets.
In their recent work on Conceptual Exploration [13]—a method completely based on the concepts of FCA—, the authors, among them Bernhard Ganter, list—in an overview chapter aptly named "Exploration galore!"—some of the promising new directions taken by authors in the field. As we see it, two of these directions could potentially be used to improve our results on complex and dirty datasets.
The first of the subchapters is simply named "Faulty data", which we can interpret as "semi-structured data diverging from an implicit schema", even though it is not technically faulty. In it, the work of Daniel Borchmann is summarised. Borchmann analyses non-perfect implications

in formal contexts. Normally, an implication is a pair of attribute sets ($A \rightarrow B$) where in the given context, if an object has all the attributes from A, it also has all attributes from B. In visual terms, this can be described as there being an upward path from the lattice node corresponding to B to the node corresponding to A. What is interesting about this are two things. First, when he traverses the lattice to collect implications, Borchmann maintains two lists: One for perfect implications and one for implications above a certain confidence threshold. This evens out the all-or-nothing nature of FCA that we have encountered in our algorithm: Either two nodes are connected, or they are not. Secondly, and more interestingly, implications may hold over *any* two nodes of the lattice, which is in stark contrast to the locality that is pervasive in our algorithm. We only ever analyze neighboring lattice nodes, and while during the merging process, nodes that were not neighboring at the beginning may become so, it is a very different approach to always look at patterns in the lattice that may consist of any two nodes. It would certainly be desirable to incorporate this aspect in future research. As we have seen, in many cases lattice nodes with many own attributes have a high probability to be a different type than all ancestor nodes. It is quite probable that other such patterns can be found.
The second subchapter is concerned with fuzzy settings. This is a version of Formal Concept Analysis in its own rights, where we still analyze a set of objects and a set of attributes and a relation between the two—just that this relation is not binary, but continuous. Fuzzy Formal Concept Analysis was established early on and is well researched. For our kind of problem, it does not seem like a good fit intuitively. Quite obviously, an object either has or does not have a certain attribute. However, applications are still worth considering. For example, it would be interesting to see in which way a context is changed if an object could have an attribute only up to a certain degree, and if that degree were depending on how frequently the given attribute occurs in the context. This has some similarities to our second merge option, where clear outliers, which often have unique attributes, are removed before the process. However, the results might not be identical, because making all attributes fuzzy would not only leave the very frequent ones more or less unchanged and the very rare ones unimportant, but it would also affect those attributes with an average occurrence.
Very generally, we are struggling the most with the rigor of our method, and its sensitivity to very small changes in the data. Broadening the scope of each merge operation (as with implications that can hold anywhere in the context) or making the relations themselves fuzzy are possible ways to help keeping that rigor in check. There is no lack of groundwork that can be used here.

8 References

[1] Peter Buneman: Semistructured data. In: Proc. ACM Symposium on Principles of Database Systems, Tucson, AZ, (1997), p. 117-121.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann: Profiling Relational Data: A survey. In: VLDB, Vol. 24, 2015, p. 557-581.

[3] Irena Mlýnková: XML Schema Inference: A Study. Technical Report. 2008. Available at: http://www.ksi.mff.cuni.cz/∼holubova/doc/tr2008-6.pdf/

[4] Irena Holubova (Mlýnková) et al.: *jInfer:* A framework for XML Schema Inference. In: The Computer Journal, Vol. 58/1, 2015, p. 134-156.

[5] Geert Jan Bex, Frank Neven, and Stijn Vansummeren: Inferring XML Schema Definitions from XML Data. In: VLDB, Vol. 07, 2007, p. 998-1009.

[6] Jonas Poelmans, Sergei Kuznetsov, Dmitry Ignatov, and Guido Dedene: Formal Concept Analysis in knowledge processing: A survey on models and techniques. In: Expert Systems with Applications, Vol. 40/16, 2013, p. 6601-6623.

[7] Jonas Poelmans, Sergei Kuznetsov, Dmitry Ignatov, and Guido Dedene: Formal Concept Analysis in knowledge processing: A survey on applications. In: Expert Systems with Applications, Vol. 40/16, 2013, p. 6538-6560.

[8] Wayne Eckerson: Achieving Business Success through a Commitment to High Quality Data. Technical Report, Data Warehousing Institute, 2002.

[9] Amena Nayak, Anil Poriya, and Dikshay Poojary: Type [sic] of NoSQL Databases and its [sic] Comparison with Relational Databases. International Journal of Applied Information Systems. Vol. 5/4, March 2013, p. 16-19.

[10] Rudolf Wille: Restructuring Lattice Theory. An Approach Based on Hierarchies of Concepts. In: Ivan Rival (Ed.): Ordered Sets. NATO ASI Series, Vol. 83, 1982, p. 445-470.

[11] Rudolf Wille: Restructuring Lattice Theory. An Approach Based on Hierarchies of Concepts. In: Sébastien Ferré, Sebastian Rudolph (Eds.): Formal Concept Analysis. 7th International Conference, ICFCA 2009. Darmstadt, Germany, May 21-24, 2009. Proceedings, p. 314-339.

[12] Bernhard Ganter and Rudolf Wille: Formal Concept Analysis: Mathematical Foundations. Springer Verlag, 1999.

[13] Bernhard Ganter and Sergei Obiedkov: Conceptual Exploration. Springer 2016.

[14] Christian Lindig: Mining Patterns and Violations Using Concept Analysis. In: Christian Bird et al (Eds.): The Art and Science of Analyzing Software Data. Elsevier, 2015, p. 17-38.

[15] Uta Priss and John L. Old: Conceptual Exploration in Semantic Mirrors. In: Bernhard Ganter and Robert Godin (Eds.): Formal Concept Analysis. Third Internatinal Conference, ICFCA 2005. Lens, France, February 14-18, 2005. Proceedings, p. 21-32.

[16] Renée J. Miller and Fei Chiang: Discovering Data Quality Rules. In: VLDB '08, August 24-30, 2008, Auckland, New Zealand.

[17] Periklis Andritsos, Renée J. Miller, and Panayiotis Tsaparas: Information-Theoretic Tools for Mining Database Structure from Large Data Sets. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of Data, June 13-18 2004, Paris, France, p. 731-748.

[18] Sergei O. Kuznetsov, Sergei Obiedkov: Comparing Performance of Algorithms for Generating Concept Lattices. In: Journal of Experimental and Theoretical Artificial Intelligence, Vol. 14, 2002, p. 189-216.

[19] E. M. Norris: An algorithm for computing the maximal rectangles in a binary relation. Revue Roumaine de Mathématiques Pures et Appliquées, Vol. 23/2, 1978, p. 243–250.

A Appendices

A.1 Files used from the IESL repo

Name in this work	Original file name
caltech_hp	thesis.library.caltech.edu#2213#1#hp.bib
ctan_texbook1	mirrors.ctan.org#info#biblio#texbook1.bib
gp_bibliography	gp-bibliography.bib
harvard_texbook3	mirrors.med.harvard.edu#ctan#info#biblio#texbook3.bib
hpjava_thesis	www.hpjava.org#theses#shko#thesis_bib.bib
netlib_einstein	netlib.cs.utk.edu#bibnet#authors#e#einstein.bib
netlib_lanczos	www.netlib.org#bibnet#authors#l#lanczos-cornelius.bib
utah_mac	ftp.math.utah.edu#pub#tex#bib#macsyma.bib
uwaterloo_bibliography	ftp.gwdg.de#pub#languages#rexx#uwaterloo#ADV#bibliography#adv.bib
uwaterloo_reports	orion.uwaterloo.ca#∼hwolkowi#henry#reports#publs.bib

Table 16: Mapping of IESL filenames

A.2 The cleaned up uwaterloo_reports context

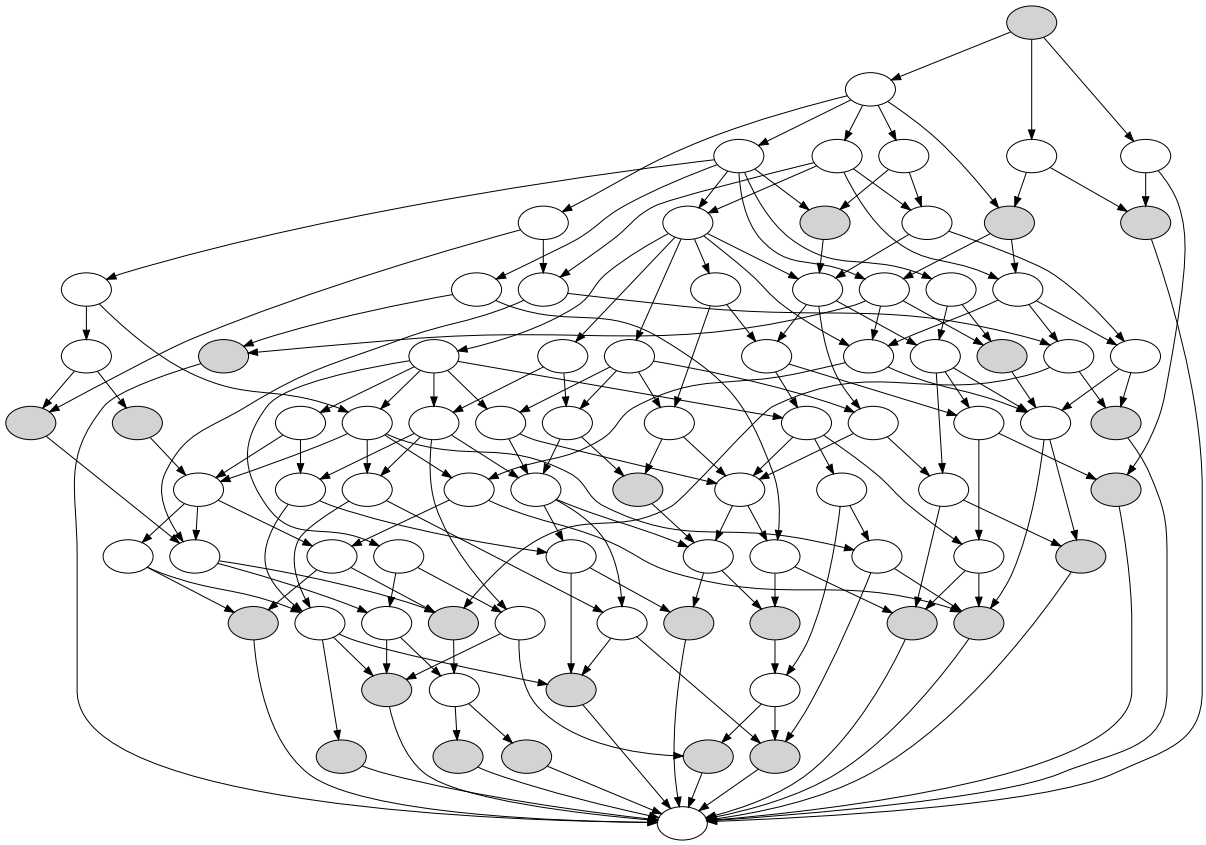

Figure 17: uwaterloo_reports after 41 merge steps, configuration 000

Note how we have applied a slightly different set of visualisation rules here. We omit the text content of the nodes—which would not be readable anyways—and emphasize nodes that have own objects by marking them gray.

A.3 Attributes appearing only once in IESL datasets

The number in column "occurrence" indicates in how many of the IESL contexts the corresponding attribute appears exactly once.

Attribute	Occurrence
abstract	1
address-l	1
advisor	1
affiliation	1
annote	1
author-dates	1
bdsk-file-2	1
bibsource	1
bookpages	1
bookreview	2
broken	1
citeseer-isreferencedby	1
citeseer-references	1
classcodes	1
co	1
collaboration	1
comments	1
corpsource	1
country	1
date	1
day	2
description	1
descriptor	1
descriptors	1
edition	2
editor	2
email	1
enum	1
file	1
glossary	1
howpublished	2
howpublished-l	1
identifier	1
isbn	2
isbn-13	1
iso-source-abbreviation	1
issn	1
issue	1
keyword	1
keywords	1
l3	1
libnote	1

location	1
organization	1
partnumber	2
printermarks	1
ps	1
publisher-l	2
references	1
referred	1
remark	1
review	1
reviewer	1
revision	1
rights	1
school	4
stand	1
subject	1
subject-dates	2
supplement	1
translator	2
treatment	1
type	2
url-2	1
xxisbn	1
xxpages	1
xxpublisher	1

Table 17: Unique attributes and their occurrence in IESL datasets

A.4 FOSS FCA tools (Anleitung zum wissenschaftlichen Arbeiten)

There are a couple of free and open source software tools around that help one to visualize, explore, and exchange formal contexts, and more. We will discuss some of them here, focussing on standalone applications (as opposed to IDE plugins, interfaces, frameworks, etc.) that have to do with core FCA applications, such as editing contexts, visualizing lattices, etc. At the time of writing, Uta Priss' website[1] is the premier online source on FCA tools.
We will use the following animal context to demonstrate different FCA tools:

	four-legged	hair-covered	thumbed	intelligent	marine
Cat	×	×			
Dog	×	×			
Gibbon		×	×	×	
Human			×	×	
Dolphin				×	×
Whale				×	×

Table 1: Example animal context

This is the concept lattice generated by the animal context, with double circles representing nodes with own objects:

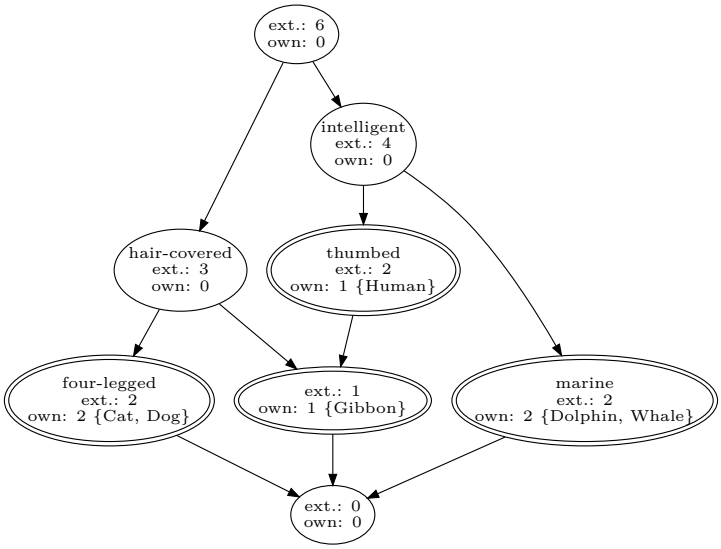

Figure 1: Example animal context lattice

ConExp

ConExp (as in ConceptExplorer) is one of the most widely used Formal Concept tools, with screenshots taken in it even appearing in scientific publications. It offers support for concept exploration, providing a guided dialogue for attribute exploration, and automatic implication and association rule calculation. The provided visual output is clear, informative, and no frills. There are a few forks of ConExp around, making it one of the

[1] `http://www.upriss.org.uk/fca/fcasoftware.html`

most influential FOSS FCA tools.
ConExp is hosted on SourceForge, where one can download an executable jar file[2].

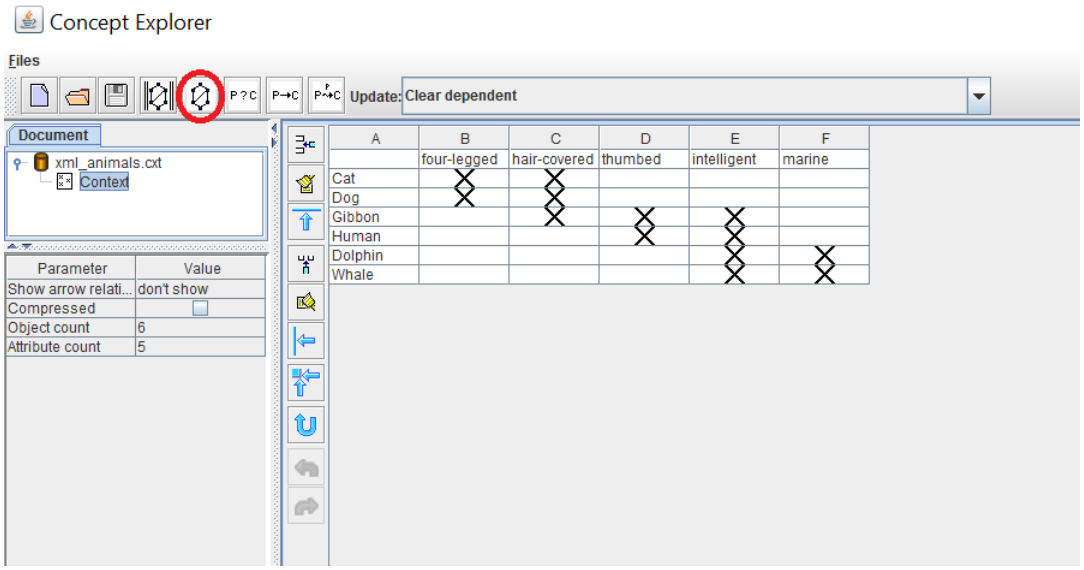

Figure 2: Cross table view in ConExp

ConExp has two different views: cross table view, and lattice view. Figure 2 depicts the cross table view, where we can verify that the context is indeed the same as described in table 1. By clicking on the button highlighted in red in the top ribbon, ConExp switches to the lattice view. For very large contexts, this can take several minutes or more, as computing lattices is very expensive.

[2] `http://conexp.sourceforge.net/`

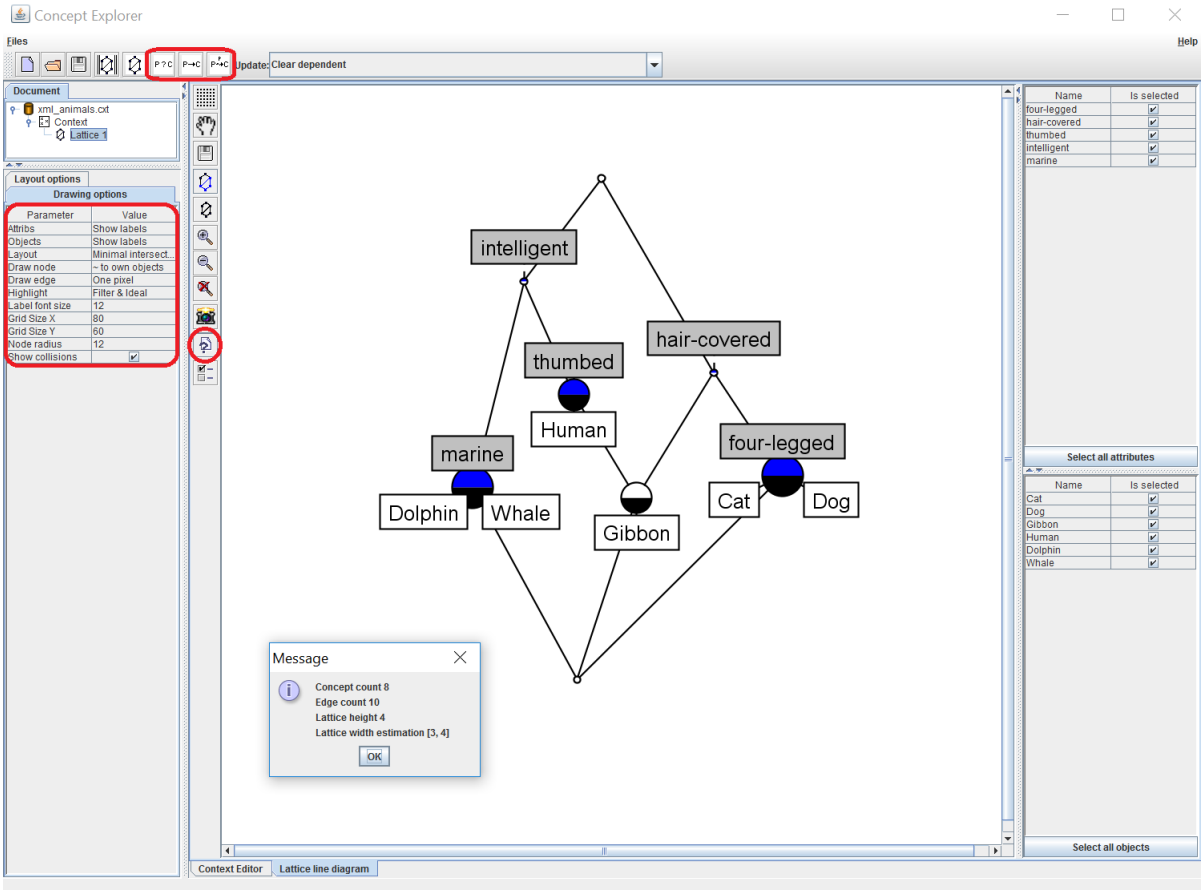

Figure 3: Lattice view with lattice statistics in ConExp

In the lattice view, ConExp displays a lot of information quite economically:
- Concepts are displayed as circles. If the upper half of such a circle is blue, the concept has own attributes. If the lower half is black, it has own objects.
- Attributes are shown in gray, and objects in white. This can be configured in the box highlighted in red all the way on the left.
- In this view, the node size is proportional to the number of own objects it has. This is customizable, along with a few other visualization parameters.
- In the screenshot, we are showing the lattice statistics. These can be obtained by clicking on the button highlighted in red in the vertical ribbon.

Note how on the right, ConExp displays a checkbox for each attribute and each object. By checking or unchecking an object, this specific object is added to/removed from the context. By checking or unchecking an attribute, this attribute is considered/ignored when computing the lattice.
There are three buttons in the horizontal top ribbon that are highlighted in red. These are for context exploration (attribute exploration, implications, and association rules).

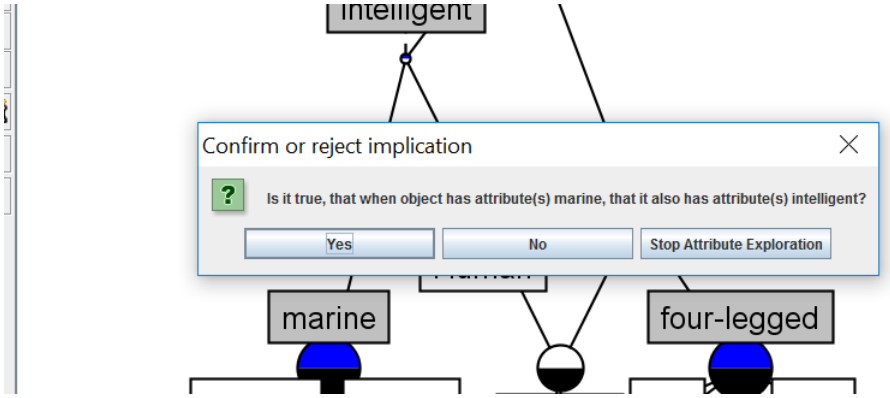

Figure 4: Attribute exploration in ConExp

The attribute exploration is a guided dialogue where the user either confirms rules extracted from the context, or enrichens the context by providing a counterexample, in accordance with the basics of FCA use.

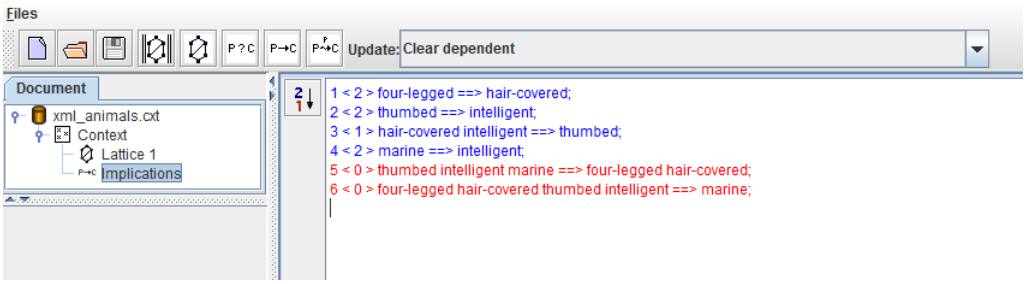

Figure 5: Implications in ConExp

Figure 5 shows the calculated implications for our example animal context. The leftmost number simply enumerates the implications. The number in angle brackets denotes the support of a given implication, i.e. the number of objects for which it is true.

Concept Explorer FX

Concept Explorer FX is a fork of ConExp, focussing on rich visual output. For example, the program offers a "polar" lattice view—something none of the other discussed tools have.
Concept Explorer FX is hosted on Francesco Kriegel's page at TU Dresden, where an installer for Windows and install instructions for Linux are available[3].

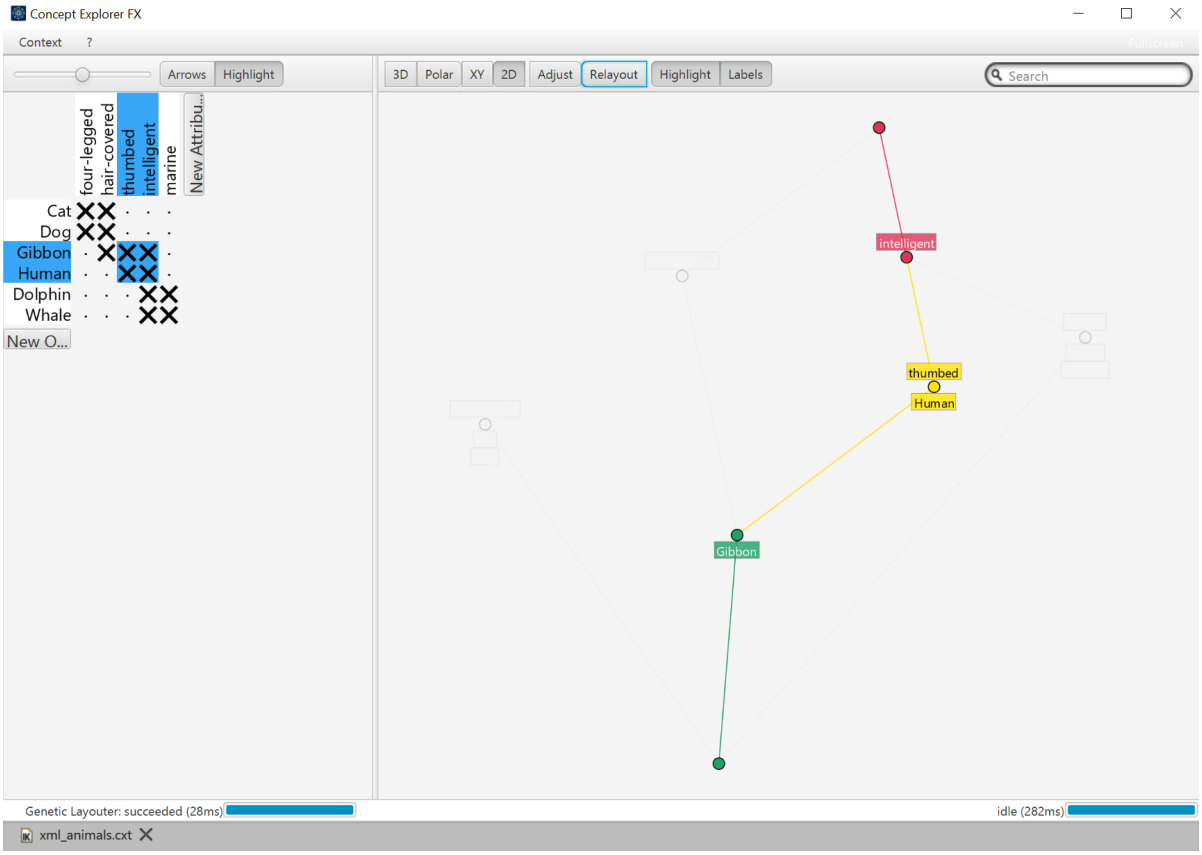

Figure 6: Interface and highlighting a node in Concept Explorer FX

Figure 6 shows the interaction with a lattice in concept Explorer FX. Here, we hover over the yellow node with intent {thumbed}. All adjacent edges are colored yellow, all lower nodes and edges green, and all upper nodes and edges red. On the left, we find the cross table view of our context. All objects that have the hovered-over attribute are highlighted, as well as the attribute itself and all ancestor attributes.
Concept Explorer FX offers some very nice graphic tricks—node size, edge thickness, label content and a lot of other parameters can be set by the user—, but ultimately offers less exploration functionality than ConExp.

[3] `https://lat.inf.tu-dresden.de/~francesco/conexp-fx/download.html`

ConExp-NG

ConExp-NG (NG stands for 'next generation') is another fork of ConExp. It has a nearly identical feature set and slightly polished graphics. At the time of writing, it didn't contain any major features that ConExp doesn't. However, compared to ConExp, ConExp-NG has been worked on much more recently. An executable jar file is available on GitHub[4].

conexp-clj

conexp-clj, a ConExp reimplementation written in Clojure, contains some additional features such as working with fuzzy contexts. conexp-clj is supervised by Daniel Borchmann, whose PhD thesis at TU Dresden was supervised by Bernhard Ganter. At the time of writing, the latest commit was less than two months old, with many more in the last half year. conexp-clj is in alpha state. In the documentation, an "experimental GUI" is mentioned, and unfortunately, the link to the pre-compiled version[5] is broken. It is to be hoped that the tool matures into a stable version. conexp-clj is hosted on GitHub[6].

[4] `https://github.com/fcatools/conexp-ng`
[5] `http://www.math.tu-dresden.de/~borch/conexp-clj/`
[6] `https://github.com/exot/conexp-clj`

LatticeMiner

LatticeMiner is another general-purpose FCA tool. Similarly to ConExp, it allows one to create, edit, and save contexts, and to visualize the concept lattice. LatticeMiner also supports implication and association rule exploration. It is a downloadable jar file, and hosted on SourceForge[7]. At the time of writing, the latest version was only a couple of days old (from April 14, 2017).

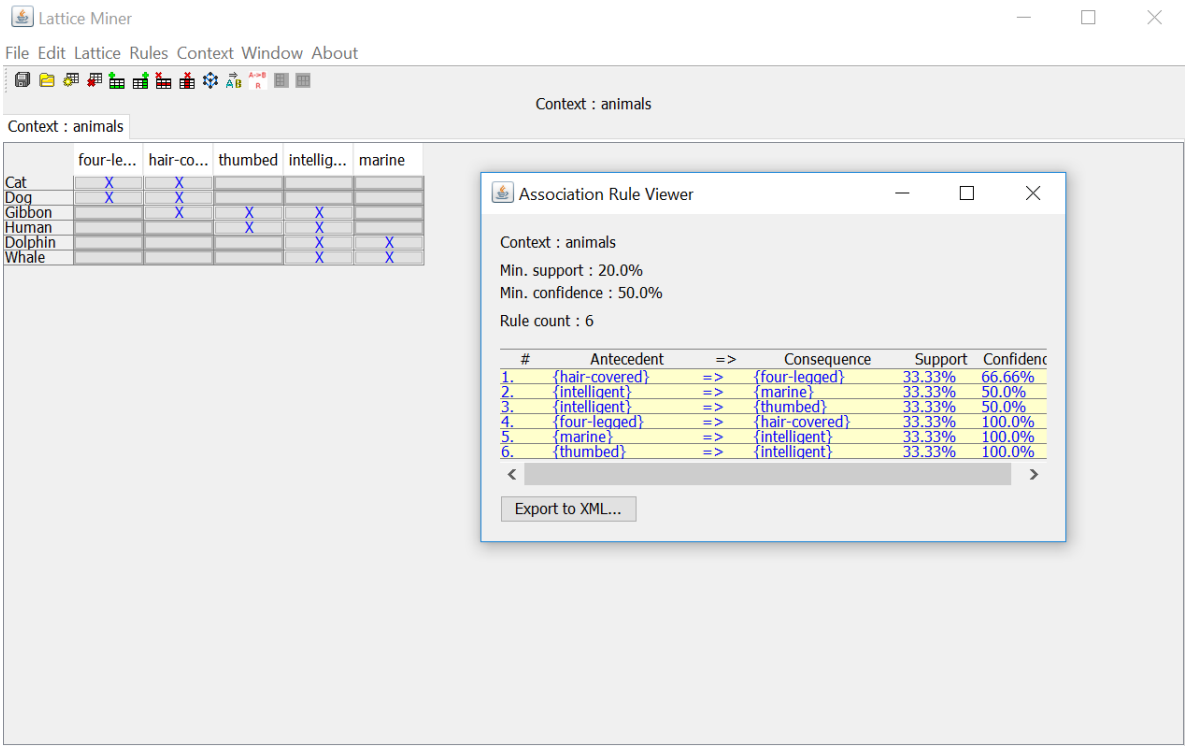

Figure 7: Cross table view and association rule viewer in LatticeMiner

Note that for the results in figure 7, we could specify the minimum support and the minimum confidence in the rules ourselves.

[7] `https://sourceforge.net/projects/lattice-miner/`

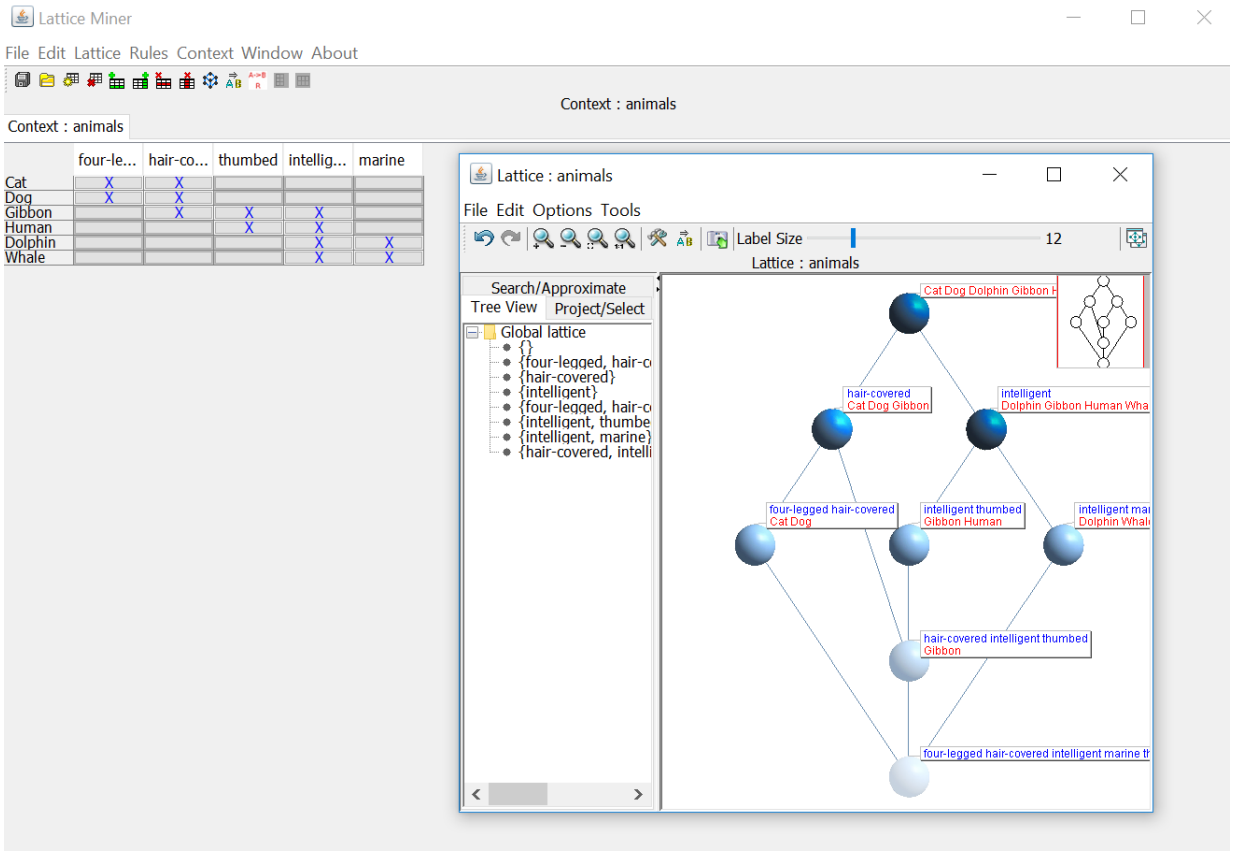

Figure 8: Lattice view in LatticeMiner

LatticeMiner also supports visualisation of a context lattice. Many things are parameterizable, e.g. the color intensity of the lattice nodes (in figure 8, it is proportional to the size of the extent), the size of the nodes (here, it is the same for all—'small'), and many other things. There seems to be a bug that whenever the size of the lattice visualisation window is changed, the lattice zooms forever and seems to be unable to recover.

Galicia

Galicia is similar in scope to the tools discussed before, but has some very interesting properties that set it apart. It is hosted on SourceForge, where an executable jar file is available[8].
One of the very interesting things about Galicia is its ability to compute a lattice with different algorithms, all of which are well documented in the literature, thereby allowing to compare them in regard to different datasets:

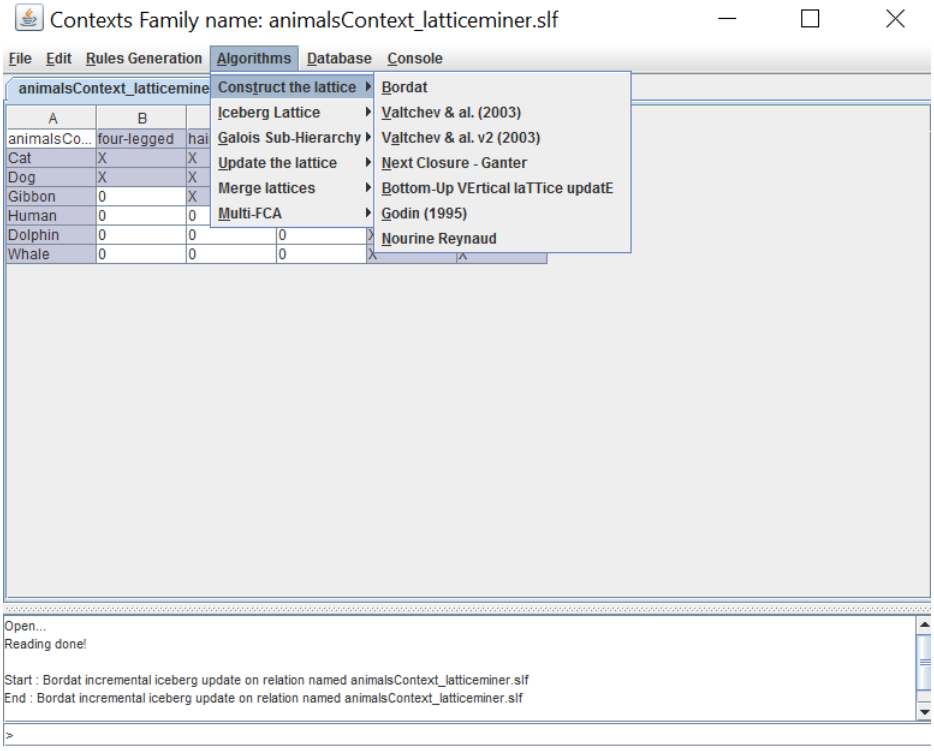

Figure 9: Computing the lattice with different algorithms in Galicia

[8] https://sourceforge.net/projects/galicia/files/

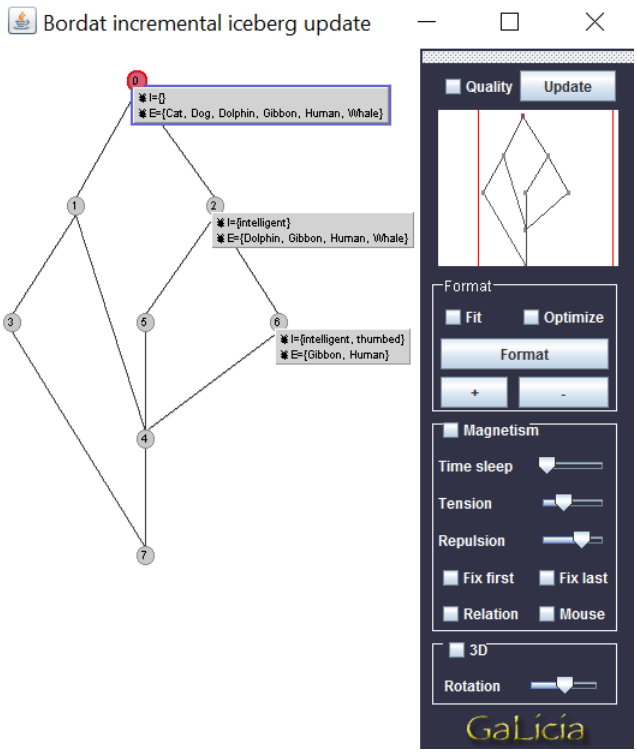

Figure 10: Lattice visualisation in Galicia

Figure 10 shows our by now well-known animal context visualized by Galicia. The information that is shown for some nodes was obtained by right-clicking the node. Some parametrization options are available on the right side, but these are less diverse than in the other tools. Galicia also supports merging of two different contexts—a feature none of the other discussed tools have.

Storing and exchanging formal contexts

When collaborating on projects, or even just in order to compare and assess different tools, it is necessary to exchange formal context information between tools. We are dedicating a very short section to lattice file formats.
Generally, there are two kinds of file formats: XML style markup formats, and (easier) human-readable ones that imitate the topology of the cross table. Table 2 lists some formats that can be read by at least two of the tools discussed here.

	.cxt	.cex	.slf
ConExp	×	×	
ConExp-NG	×	×	
ConExp FG	×		
LatticeMiner		×	×
Galicia			×

Table 2: Formal context file formats

.cxt (Burmeister's format) and .slf are human-readable formats and indeed very similar. (The biggest difference is that in .cxt, . and X are used for the binary relation, and 0 and 1 in slf). This is the animal context in .cxt:

```
B

6
5

Cat
Dog
Gibbon
Human
Dolphin
Whale
four-legged
hair-covered
thumbed
intelligent
marine
XX...
XX...
.XXX.
..XX.
...XX
...XX
```

Listing 1: animals.cxt

The first line in a Burmeister formal context file by definition is a single 'B'. The second line may contain the name of the context, but is optional. The next lines hold the number of objects and the number of attributes, after which follow some lines that describe the actual relation—X meaning that the object whose row it is has the attribute whose column it is, and . that it does not.

On the other hand, this is the animal context exported to .cex (ConExp XML format):

```
<?xml version="1.0" encoding="UTF-8"?>
<ConceptualSystem>
  <Version MajorNumber="1" MinorNumber="0" />
  <Contexts>
    <Context Identifier="0" Type="Binary">
      <Attributes>
        <Attribute Identifier="0">
          <Name>four-legged</Name>
        </Attribute>
        <Attribute Identifier="1">
          <Name>hair-covered</Name>
        </Attribute>
        <Attribute Identifier="2">
          <Name>thumbed</Name>
        </Attribute>
        <Attribute Identifier="3">
          <Name>intelligent</Name>
        </Attribute>
        <Attribute Identifier="4">
          <Name>marine</Name>
        </Attribute>
      </Attributes>
      <Objects>
        <Object>
          <Name>Cat</Name>
          <Intent>
            <HasAttribute AttributeIdentifier="0" />
            <HasAttribute AttributeIdentifier="1" />
          </Intent>
        </Object>
        <Object>
          <Name>Dog</Name>
          <Intent>
            <HasAttribute AttributeIdentifier="0" />
            <HasAttribute AttributeIdentifier="1" />
          </Intent>
        </Object>
        <Object>
          <Name>Gibbon</Name>
          <Intent>
            <HasAttribute AttributeIdentifier="1" />
            <HasAttribute AttributeIdentifier="3" />
            <HasAttribute AttributeIdentifier="2" />
          </Intent>
        </Object>
        <Object>
          <Name>Human</Name>
          <Intent>
            <HasAttribute AttributeIdentifier="3" />
            <HasAttribute AttributeIdentifier="2" />
          </Intent>
        </Object>
        <Object>
```

```
        <Name>Dolphin</Name>
        <Intent>
          <HasAttribute AttributeIdentifier="3" />
          <HasAttribute AttributeIdentifier="4" />
        </Intent>
      </Object>
      <Object>
        <Name>Whale</Name>
        <Intent>
          <HasAttribute AttributeIdentifier="3" />
          <HasAttribute AttributeIdentifier="4" />
        </Intent>
      </Object>
    </Objects>
  </Context>
 </Contexts>
</ConceptualSystem>
```

Listing 2: animals.cex

This format has the obvious disadvantages of being less easily readable and taking up more space. However, XML files are easier to automatically generate than custom formats because of the large number of libraries with that exact purpose.
The debate of different context files is practically moot, since there exists a tool[9] by the aforementioned Uta Priss that can convert between all major context formats, including those mentioned here and many more.

Jan Luca Liechti, Bern, 27.5.2017

[9] `http://fcastone.sourceforge.net/`