International Journal of Database Theory and Application
Vol.7, No.5 (2014), pp.51-84
http://dx.doi.org/10.14257/ijdta.2014.6.5.05

An Optimized Splitting Attribute Algorithm for Inconsistent Conflict in Context Lattice

Zhou Zhong and Junzhong Gu*

Institute of Computer Application (ICA), East China Normal University
3663 Zhongshan Road N., 20062 Shanghai, China
zhzhong@ica.stc.sh.cn, jzgu@ica.stc.sh.cn

Abstract

With the emergence of Cloud computing and Internet of Things, Context-aware applications face new challenges. One of them is big data from huge context application and sources. The main stream of applications have used not only real-time versions but also history versions of context data. This paper concerned about optimization techniques of storage and reasoning in the CMS (context management system). For our storage of context data from different sources, FCA Lattice has been employed as a kind of storage schema to support modeling and fusion of these different context data. Further, context conditions about data are essential to logical reasoning. Under different context conditions, context data can be promoted to be knowledge, which makes context reasoning readily. In the dynamic environment, to get reasonable results, reasoning services require their input to keep consistent in the changeable conditions. The changeable conditions can be represented as context attributes, intervals and relations etc. To make consistent knowledge available in the conditions, our pervious works have analyzed incremental cache and check of consistent intervals, and proposed a context lattice-based distributed optimized update algorithm. In this paper, based on the algorithm, our problem is to optimize the split function. The split is needed when current lattice has no condition making knowledge consistent. The main aim of this paper is to improve time performance of splitting attributes or intervals or fuzzy relations that could be detailed. We propose a new parallel split algorithm. This algorithm computes the priorities of candidates. To reduce time cost, it decreases the split scope by choosing the split candidate with the highest priority value. To decrease the full lattice update time in the split process, it generates the sub lattices split by the candidates concurrently and merges them after. On the theory, we analyze the feasibility of the algorithm. On the test, as a new part of the whole update algorithm, it is compared with the na ïve one, and it shows the better time performance. What's more, it makes multi-threads execute on the same lattice to avoid producing more memory cost caused by copying the lattice for an independent thread.

Keywords: *FCA, Parallel algorithm, Lattice, Time cost saving, Context knowledge management, Attribute splitting*

---

* corresponding author

1. Introduction

With the Internet of Things and Big data emerging, context computing has more requirements of context data process. One of them is context storage. In the early sensor network period, context data is collected in real time, its history is usually used for analysis later. When society networks and location based services are prevalent, context data presents at least four features as following: mass, inconsistent, semantic, and dynamic. First, the context data increases and accumulates rapidly from the output by large users each day. Second, the different sources make same data with different context, which might leads to a large amount of inconsistent cases. In fact, the two latter features, semantic and dynamic, also result in inconsistency of the environment where semantic and context are changing. Semantic shows up in the context knowledge extraction, merging and reasoning. It is useful to assure context sharing in the multiple applications. Each kind of context-aware applications has often its own data structure and semantic definition. It causes the fusion and generic classify of structure and semantic, which makes a static structure little flexible for storing context data aggregated in the sharing process. Dynamic is a classic and important feature of context from the beginning of context-aware computing. Context is not static from the perspective of its value facet or its relationship facet, etc. Meanwhile, for recommendation based on user behavior, as input of context reasoning, context data has to combine both its real time and history version dynamically at any time. Thus, the persistency structure of context data should be adapted for these features while the storage module is reading and writing context according to it.

In our research, FCA lattice can be such a valid storage schema that implements this adaptive goal to some extent. Many previous works have used FCA lattice as a data mining tool. In the other words, this means that it supports mass data process well potentially. FCA could take data as its objects and their context as its attributes and the edges of the FCA lattice nodes satisfies the generic/concrete/combined relationship that makes semantic fusion and classify possible. And if context is modeled as a context lattice, the step and navigation function of this lattice could make dynamic access easier than the plan dimension tables. The dynamic context query by increasing or decreasing context query scope also could be supported by this function. As a context lattice is still small, the joint between real time version and history versions could be seen as different time context points or intervals on a timestamp or data time dimension. It could be also implemented by the mapping of the real time lattice and different history lattices as a context set becomes to be larger.

Meanwhile, FCA is useful to context reasoning. It shows that context reasoning is indispensible in user-based context applications. The important characteristic is that reasoning output is only valid in the special context scope. The knowledge derived from reasoning might also be consistent or inconsistent dependent on context scope. So inconsistency generates if the context scope changes. We could use context intervals as FCA attributes. These intervals have different scales by designing. These intervals could distinguish consistent or inconsistent knowledge from different context dimension. If these intervals are cached or stored, when certain knowledge in them is queried or reasoned again, the computing time of ascertaining the knowledge’s scopes will be saved.

We have been using distributed context lattices to store knowledge and their consistent information (consistent or inconsistent intervals). With the context lattice changing, we check and update these information.

In this paper, the context interval split is analyzed. As the context data is changing in the dynamic environment, the knowledge could be inconsistent soon in the previous consistent intervals. When inconsistency happens, the previous information cannot be used for context query and reasoning at the real time version. One solution is to split intervals into more small intervals to keep its consistency information in new intervals. This guarantees the later query and reasoning could search results in new consistency scopes timely such that the service makes the response more quickly and some cached results could be returned in advance.

However, in processing the large lattices with more intervals as attributes, using the current naïve split method is inefficient and delayed. Therefore, we propose two optimization strategies:

1. Ordering the lattice intervals by their priorities. In a lattice, the influence of each attribute scope is different. If splitting a big-scope attribute is equal to splitting a small-scope attribute on the functionality of querying and the small one is chosen, the splitting time could decrease obviously. In this paper, we analyze and argue that choosing small one cannot reduce the performance of query or reasoning.

2. Parallel splitting attributes. One attribute should often be split into more layered sub attributes such that certain assertion is consistent. In some cases, multiple attributes should be split together. We discover that the split process could be executed in a parallel way. If the lattice is split into more lattices in a parallel way, the merging of lattices should be executed after that. We found that the merging is also suitable to the parallel manner.

The two strategies aim at lowering the time cost of splitting. We design a splitting algorithm combined with these two strategies. The algorithm is designed to execute on the same lattice and does not need to clone the lattice for parallel running. The experiment shows that its performance is better than the naïve one from both time cost and memory cost.

In the second section, the related works are introduced. The third section describes the preliminary part which includes FCA theory, the definition of consistency about our context lattice and related proofs of this paper's algorithm. The fourth section presents algorithms. The fifth section compared the optimized with the naïve from the test log. The conclusion is discussed in the sixth section.

2. Related Works

As [1] described, Formal Concept Analysis has been developed as a field of applied mathematics based on a mathematization of concept and concept hierarchy. It thereby allows us to mathematically represent, analyze and construct conceptual structures. That has been proven useful in a wide range of application areas such as medicine and psychology, sociology and linguistics, archaeology and anthropology, biology and chemistry, civil and electrical engineering, information and library sciences, information technology and software engineering, computer science and even mathematics itself.

A lot of woks of the conceptual knowledge and text retrieval processing have focused on using FCA. [2-5] explains the important processes of organizing knowledge management: identification, acquisition, development, distribution, sharing, using and persisting of knowledge. [6] An Open-Source Toscanaj is implemented for developing the conceptual information system.

For its outstanding ability of the conceptual knowledge processing, the context-aware application area should also employ it for modeling, mapping and merging common context knowledge. In the previous works, we use it as a modeling and merging tool to analyze the context in the dynamic environment [7].

And related to the split operation, the previous works have been involved the attribute scaling algorithm, which is essential to our algorithm, for splitting can be decomposed into multiple attributes' add operation.

The paper [8] proposes an ICG, FCA-based methodology to extract generic parts out of the software models described as UML class diagrams. It summarizes the key elements of relation Context family. [1] introduces that Conceptual scaling [9] is a FCA technique that transforms a many-valued context K=(O,A,V,J) into binary one Kd=(O,Ad,Id) by replacing non-binary attributes from A by a set of binary ones, called scale attributes. Both normal attributes and object-inter-relations can be seen as the scale attributes. It implements object inter-relation extension and adds the inter-relations as attributes into the original lattice.

Similarly, the split of attributes in our research can be seen as the scaling of the binary context, in which attributes can be detailed as the interval-typed attributes. So they could be split for keeping the objects in the new consistent intervals.

It is obvious that ICG has to invoke the process of adding attributes. What we focus on is the process of adding attributes, because the split can be transformed to adding attributes. The adding attribute algorithm used by ICG is a common incremental attribute algorithm. It will be introduced in the part of the naïve algorithm. The related source code can be found at [10].

In our work, a kind of specific context lattice is designed and implemented for the context data distributed storage, consistent information update and splitting attributes.

By the additional Concepts as objects, it ensures the steady Concept Bottom Nodes B are located in the lattice. For the distributed cases, the transition nodes *T* are defined. The two kind nodes could make less iteration. And we proved the candidate nodes to be split should locate between *B* and *T*. On these definitions, a distributed optimized update lattice algorithm was proposed and implemented. It improved the time performance of lattice iteration part when stepping the distributed lattices with consistent check. It used the add attribute algorithm of [8] in the splitting part. In this paper, we continue with the previous work and improve the splitting part, and implement the two optimized strategies.

3. Preliminary

In the subsection 3.1, FCA theory related with this paper is introduced. Then the definition of previous works involved with this algorithm is given. Section 3.3 describes the concept of the attribute priority. After that, in the section 3.4, the proofs about this algorithm are presented.

3.1. FCA Theory

Formal Concept Analysis (FCA) was introduced by [11] and is completely developed in [12]. FCA is the process of abstracting conceptual descriptions from a set of objects described by attributes. The FCA has been used in works related to symbolic data analysis and knowledge representation [13]. We shall begin by introducing the basic notions defined by Wille.

Definition 3.1. Formal Context.

A formal context K is defined as a triple of sets, (G, M, I), where G is a set of objects, M is a set of attributes, and I is a binary relation between G and M (i.e. $I \subseteq G \times M$). (g, m) ∈ I is read “object g has attribute m”.

A possible confusion might arise from the double use of the word ‘context’ in FCA and in context model of context-aware applications. This comes from the fact that FCA and context model are two models for the concept of ‘context’ which arose independently. In this paper, we weaken the concept of context in context model, for we use FCA to model and store the data from context model.

Definition 3.2. Extent and Intent.

For $A \subseteq G$, we define $A' or\ f(A) := \{m \in M \mid \forall g \in A : (g,m) \in I\}$ and, for $B \subseteq M$, we define B' or $\mathrm{g}(B) := \{g \in G \mid \forall m \in B : (g,m) \in I\}$.

A formal concept of a formal context $L(K_T, \leq)$ is defined as a pair (A, B) with $A \subseteq G, B \subseteq M$, $A' = B$ and $B' = A$. The sets A and B are called the *extent* and the *intent* of the formal concept (A, B). On *K* a partial order relation ≤ can be defined through the following formula where (A, B), $(A', B') \in K$: $(A, B) \leq (A', B') \Leftrightarrow A \subseteq A' (\Leftrightarrow B \supseteq B')$. This relation is a generalization/specialization hierarchy relationship.

Definition 3.3. Concept Lattice $((G, M, I), \leq)$.

The set of all formal concepts of context K with the partial order ≤ is always a complete lattice, call the concept lattice (or Galois lattice).

Definition 3.4. closure system and closure operator.

In [14], closure system and closure operators of FCA are introduced. Let 2^M denote the power set of a set *M*. By a closure operator we mean a mapping $C : 2^M \to 2^M$, which is extensive, monotone and idempotent, i.e. which satisfies for all *A*, $B \subseteq M$:

$a)\ A \subseteq C(A)$

$\mathrm{b})\ A \subseteq B \Rightarrow C(A) \subseteq C(B)$

$c)\ C(C(A)) = C(A)$

A set X is closed iff X=C(X). The collection of all closed sets of some closure operator is called a closure System. A' and B' are the two mappings $A' : 2^G \to 2^M$ and $B' : 2^M \to 2^G$ from a Galois-connection. As an immediate consequence one obtains that the family of all extents and the family of all intents of (G, M, I) both are closure systems, the corresponding closure operators are the mapping $Y \to Y''$ on *M* and *G*, resp.

3.2 Definition of Consistency Check About Lattices

The following definitions are about context consistency check by FCA lattice. We only list the part related to this paper.

Definition 3.5. Context Interval.

Let *c* be a concept, given one of its context scale: *λ* which is a unit to partition *c*'s context intervals. And its context interval ε is an interval partitioned by *λ*, it exits a triple<*λ*, ε >, called one *c*'s context interval. Context interval set can be labeled with Ψ .

Definition 3.6. GABOX Context.

$K_{GA} =< C \cup A,\ M \cup \Psi_G, I_{GA} >$ where Ψ_G is the intervals of the full ABOX Context, called GABOX Context, is the whole Formal Context for the given domain. Its objects contain both *C* and *A*, and attributes contain both *M* and Ψ_G .

Definition 3.7. PABOX Context and Lattice.

$K_{PA} = < C_P \cup A_P,\ M \cup \Psi_P, I_{PA} >$ called PABOX Context, is a partial Context of the GABOX Context, where $A_P \subseteq A$, $C_P \subseteq C$, $\Psi_P \cap \Psi_G \neq \varnothing$. Let Ψ_{Pinit} be $\Psi_P \cap \Psi_G$. Ψ_{Pinit} is the initial assigned interval sets derived from the GABOX Context. For each $< \lambda_{init\_split}, \varepsilon_{init} >$ in Ψ_{Pinit} , there is a one-to-one correspondence between it and one ContextConcept $m_i \in M$, where λ_{init_split} belongs to Δ_{init_split} that is an initial scale set for splitting GABOX Context into PABOX Contexts.

Definition 3.8. Concept Bottom Node.

Given a GABOX Lattice or PABOX Lattice *L* and a Concept *c*, a node in the lattice can be called concept bottom node B_C , if its own extent contains Concept *c* from Θ , which is a meta-ontology that specifies the constraint relations between entities and contexts in context model such that entities and contexts are easier to be transformed as the objects and attributes in FCA context.

Definition 3.9. Transition Node.

Given a PABOX Lattice *L* and a Concept *c*, let $M(C)$ be *c*'s ContextConcepts from Θ in this lattice and Ψ_C be *c*'s context intervals in *L*'s initiation version, which has only the intervals derived from GABOX Lattice but not any own detailed interval. A node in the lattice is called transition node T_C , if its intent equals to $\Psi_T \cup M(C)$ where $\Psi_T = \{<\lambda,\varepsilon>| \exists \Psi_T \subseteq \Psi_C . \forall <\lambda,\varepsilon> \in \Psi_T . \neg \exists <\lambda',\varepsilon'> \in \Psi_T, \lambda' = \lambda, \varepsilon' \neq \varepsilon \}$.

Definition 3.10. Identifiable object.

Given a PABOX Lattice *L* and a Concept *c*, an object *o* is an identifiable object which belongs to *L* definitely, if $f(o) \supseteq \text{intent}(T_C)$.

It implies that an identifiable object o belongs to the extent of successors of T_C .

Once the Concepts and ContextConcepts about an entity's assertions are confirmed from context model to FCA model by Θ , if the entity is as an object in FCA, its related B_C set and T_C set will be useful to the step-by-step update with its consistency check in distributed Partial ABOX lattices.

Definition 3.11. Consistent Node and Inconsistent Node.

Given a Lattice *L* and an object *o*, for each node $n \in nodes(L)$, if $\exists o \in g(n)\ \forall e \in (g(n) - o)$, $e \neq !o$, where !*o* is a negative instance inconsistent with *o*, then n is called *o*'s consistent node, else n is called *o*'s inconsistent node.

Definition 3.12. Consistent object and Inconsistent object.

Given a GABOX Lattice *L* and let $\cup Lp_i$ be its PABOX Lattice set of *L*. An *o* is called a consistent object if there is at least one PABOX Lattice Lp_i in which *o* is an identifiable object and at least one node n of $\text{nodes}(Lp_i)$ is *o*'s consistent node, or else *o* is called an inconsistent object.

Definition 3.13. Divisible attribute.

Let *o* be an identifiable but inconsistent object of a PABOX Lattice *L*, a context interval $<\lambda,\varepsilon>$ of *o*, defined by definition 3.5, is called a divisible attribute d of *o* if ε has not been split completely by λ or there exists a finer λ' to split ε in more detail intervals. This definition assures candidate attributes are valid.

Definition 3.14. Divisible node.

Let *o* be an object and *n* be an inconsistent node where its intent equals to $f(o)$ and contains a divisible attribute *d*, it is called a divisible node n_d of *o*.

Theorem 3.15. Given an *o* and a node *n*, if *n* is one of *o*'s consistent nodes, the successor nodes of *n* must be *o*'s consistent node.

3.3. Attribute Priority

If an object o has no consistent node, the split process will be executed to create at least one consistent node for query later. The split process will choose one divisible node as the candidate from all the inconsistent nodes where their extent contains o. Generally, we choose the node having most attributes. This node should have many divisible attributes in its intent. To each divisible attribute, a consistent node would be produced possibly if it would be split. But the influence by splitting each divisible attribute on its lattice is different. In this paper, we consider the update influence that is caused by attribute scope. Every attribute has its different scope we mean the amount of lattice nodes in which the intent contains this attribute. This is one of the main factors which influenced the scope of the lattice update by splitting. If one attribute's scope is larger, then the size of related nodes is more. So the time cost of a split is more. If we choose a candidate attribute owning a smaller scope currently, then the time will be less. Thus, we could sort all divisible attributes by their priorities dynamically and choose a most suitable one if the split function is invoked.

Here, two problems are inevitable. One is feasibility and the other is weight. The feasibility problem is whether it is only a little different for query on two result lattices if the process splits the most optimal attribute instead of others. The weight is how to set the priority for each attribute.

3.3.1. The Analysis of the Feasibility

Firstly, we present the feasibility analysis of the sorting by priorities.

There are two cases invoking split. The first case is that there is no consistent context interval that satisfies the context query with specific context scope. The second case is that there is no consistent context interval for one object when it is added into the related lattice. The split at the first case facilitates subsequent queries with similar context scopes. The second case is a split in advance relative to the first case. Though each context query cannot have its context scope equal to the assigned intervals exactly, every time it only need compute the intersection and difference of the split intervals' bounds and avoid iterating the split intervals' objects which have been computed. So the first case is mostly equal to the second case on the query performance after split.

For both the two cases, the attributes of the split are not assured. All the candidate attributes can make subsequent queries optimized later. So the priority strategy is feasible.

We take an example to make the feasibility intuitive. Given are a context lattice *L* with consistent information and an inconsistent node n which is about an object o1 and contains any an inconsistent object o2 with o1. Assume that there exists a divisible attributes set DA belonging to n's intent. Let A and B be any two divisible attributes in DA. We assume that an attribute forest Ft contains two trees *TA* and *TB* corresponding to A and B respectively. *TA* has three nodes (A, A1, A2), where A is the root and has two children (A1, A2). *TB* has three nodes (B, B1, B2), where B is the root and has two children (B1, B2). The tree node and its children have "part of" relation which means that A1's value interval and A2's value interval are bounded by A's value interval. The tree brother nodes have "no intersection of" relation which means that A1's value interval and A2's value interval has no intersection region.

Assume that both A1 and B1 can distinguish o1 from o2. Thus, the split of (A1 A2) can place o1 within A1 and o2 within A2 separately, and the split of (B1, B2) can place o1 within B1 and place o2 within B2 separately. After splitting, o1 is only included in the extent of the lattice nodes where their intent could contain A1 or B1 but not contain A2 or B2. This ensures that o1 belongs to the lattice nodes where their extent does not contain o2.

Assume that $L_{\text{priorityA}}$ and $L_{\text{priorityB}}$ are sub lattices after splitting A first and splitting B first respectively. $L_{\text{priorityA}}$ and $L_{\text{priorityB}}$ is showed in Figure 1, where each node's extent only shows (o1,o2) and its' intent only shows *TA* and *TB*'s attributes. The solid lines show the sub lattices after first split and the dotted lines show the sub lattices after second split. From Figure 1, we can find the lattices linked by the dotted lines are same. It means the order of split does not influence the lattice generating after the second split.

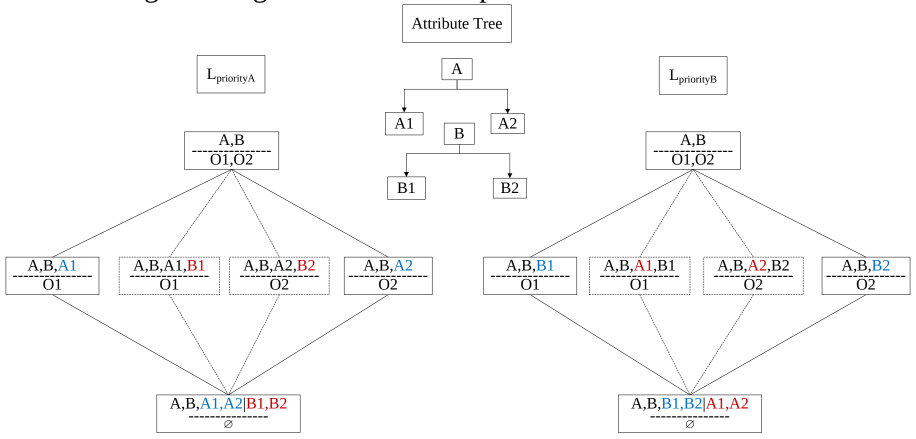

Figure 1. The Sample of the Feasibility Analysis

Then we need consider the difference of lattices after the first split. It is obvious that the two sub lattices by the solid lines are different in the example of Figure 1. What is important for us is the difference of the function and performance when queries are on these two lattices after the first split and before the second split.

One difference may be the influence on query scope. As mentioned above, the context scopes of the queries are changeable. If some conditions of the query scope falls into the range of (A, B), it is impossible that the scope's related condition is just right equal to A1, A2, B1 or B2 each time. As a random query condition, whichever it is about A or B, it must be split into two or more intervals usually. And for a query about multi-dimensions has more conditions in its scope, it is very likely to contain the conditions about both A and B. So whatever A or B is to split, the prior choice of A or B facilitates avoiding splitting query scope weakly. For optimization, it only need ensure that o1 is located into a smaller consistent interval of (A, B), then all the later queries within (A, B) can be benefited from avoiding rechecking this consistent interval with its successors according to Theorem 3.15, and it only need compute the other intervals that don't contain any object like o2 but have intersection with the difference between the query scope and this interval.

Another possible difference of choosing A or B is the number of their generating sub intervals. If the sub intervals are little or many, the optimized purpose to avoid checking more objects doesn't meet obviously. The query scope will be matched with the sub intervals and the unmatched intervals can skip the objects iteration when we determine whether a query scope is a consistent scope. In the ultimate cases, except the consistent interval, the number of the new intervals is equal to the amount of the inconsistent objects or only one. These two cases make the later queries inefficient. However, if we make every candidate for this kind check by the precomputing to avoid ultimate cases before split, the update time will increase. Fortunately, these two ultimate cases happen rarely. The first case could be solved by the extra new uneven intervals containing these new intervals. And more split operation will make the second ultimate case disappear. In the practice, we check this in running time and not use it as a priority weight.

Otherwise, in fact, both A and B will be split many times after these splits. Thus, splitting A or B is same for o1's consistent information maintaining in related sub lattices of *L* basically.

Let *LA* and *LB* be the whole lattices of *L* by splitting A and B. In fact, *LA* is the merging of *L* and $L_{\text{priorityA}}$, and *LB* is the merging of *L* and $L_{\text{priorityB}}$. The difference between *LA* and $L_{\text{priorityA}}$ (or *LB* and $L_{\text{priorityB}}$) is that *LA* (or *LB*) will generate more nodes through the splits. In *LA* and *LB*, the split functionality for o1's update is equivalent. But the updated scopes of *LA* and *LB* are different. Therefore, we consider it feasible that the process computes the priorities of A and B by determining A's and B's scopes and then chooses a candidate having the higher priority.

3.3.2. The Weight of the Sort

Scale attribute splitting in FCA can be seen as a special attribute increment, which will cause recomputation of the nodes of the related objects. When no node can keep one target object o be consistent in their context scope, the candidate nodes containing o are chosen, and they will be split with their scales until o falls into a new consistent node.

Table 1. The Weight Levels of the Sort

level	**The attribute with the node**
level 1	The candidate node's own intent
level 2	The target object's own attributes
level 3	The other attributes

The main levels can divide the attributes into three scopes. The first level's scope influences the lattice minimally. The candidate node's own intent can only influence itself and its own successors. The second level's scope influences the closure where the nodes' intent belongs to the power set of the intent of the target object's concept bottom node. The third level represents the other attributes. And the sort of attributes within each level is according to its influence scope in ascending order.

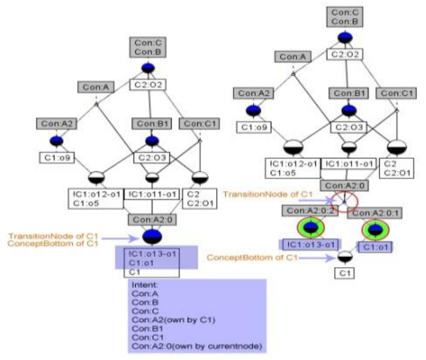

Figure 2. The Initial State and the State by Splitting Own Intent of the Example

From Figure 2 and Figure 3, the priority strategy can be illustrated obviously. The left of Figure 2 shows the initial state of splitting about a partial ABOXlattice defined by definition 3.7, at which the zero node is just both T_C and B_C of C1. The node's intent is {Con:A,Con:B,Con:C,Con:A2(owned by C1), Con:B1,Con:C1,Con:A2:0 (owned by current node)}. If 'Con: A2:0' is chosen for splitting, two new nodes are created. The updated lattice is showed in the right of Figure 2. No more nodes will be changed or created because the choice of the node's own intent won't influence the other nodes such that only the node is split. The red circles denote that the original node and the new nodes. And the light green shaded area indicates the different nodes from the initial.

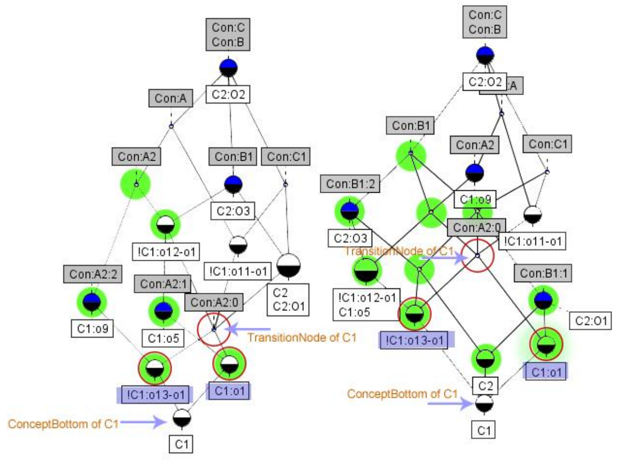

Figure 3. The States After the Splitting by the Con:A2 and the Con:B1.

If we use 'Con: A2' owned by C1 independently, the situation of the lattice after splitting is displayed in the left of Figure 3. From Figure 3, the number of updated nodes (light green shadow) increases, compared with Figure 2. But only the boundary about 'Con: A2' is influenced, the C2's objects distribution is no changed. Instead, if 'Con: B1', a common scale attribute of C1 and C2, is selected, the number of difference is much more than using 'Con: A2'. All related objects in C1 and C2 are influenced so that the updated boundary of the lattice is expanded (showed in the right of Figure 3).

3.4. Theory of the Optimized Parallel Split

After analysis of feasibility, the parallel theory of split is defined. In the following description, the function "closure (A)" is to compute the lattice node set which contains all nodes owning certain intent A in a lattice.

Theorem 3.16.

Given a lattice *L*, suppose that an attribute *A* should be split into *A1…An* and if the respective objects corresponding to *A1…An* has no intersection with each other except the Concept-typed objects of these objects, then the updates of other nodes respectively by updating *A1…An* can be paralleled after the update of the nodes corresponding to the intent closure in which each intent contains any two attributes of *A1…An* at least.

Proof. Let any two attributes from *A1…An* be *Ai* and *Aj*, about which the object sets are *Oseti* and *Osetj*. Assume that the intersection of *Oseti* and *Osetj* only contains the Concept-typed objects C. For *Oseti* and *Osetj* should own *Ai*, *Aj* respectively and both *A*, *C* should have *A*, *Ai* and *Aj*. C's objects belong to the extents of the related Concept bottom nodes Bs of *L*. So, besides *A*, Bs' intersection intent *i_C*(*A*) should contain *Ai* and *Aj* after splitting. The closure (*A*), in which any node's intent including *i_C*(*A*) at least, will generate or update the nodes containing both *Ai* and *Aj*. Except this closure, the nodes of *L*(*Ai*)- closure (*A*) and *L*(*Aj*)-closure(*A*) contains *Oseti*-C and *Osetj*-C, which have no intersection. The merging of *L*(*Ai*)-closure(*A*) and *L*(*Aj*)-closure (*A*) cannot generate a new node that contains *Ai* and *Aj* in its intent or the intersection set of *Oseti* and *Osetj* in its extent. The update of *L*(*Ai*)- closure (*A*) and *L*(*Aj*)- closure (*A*) can be paralleled. So in the same way, *A1…An* attributes can be also updated in a paralleled way.

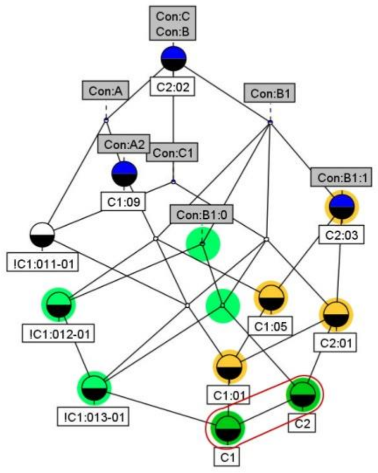

Figure 4. A Simple Example About Theorem 3.16

For example in Figure 4, Attribute B1 is split into two attributes: B1:0 and B1:1. From the example, the green part *g* is the sub lattice *L* (B1:0) after B1:0 update, the orange part *o* is the sub lattice *L* (B1:1) after B1:1 update, and the shared dark green part *d* is the intersection of *L*(B1:0) and *L*(B1:1). The *d* part's nodes are composed of the Bottom Nodes $B_{(C1)}$ and $B_{(C2)}$, in which the intersection of their intent contains both B1:0 and B1:1. In this example, the nodes owning this intersection are only $B_{(C1)}$ and $B_{(C2)}$ exactly in the closure. The closure could be updated by B1:0 and B1:1 firstly. Then the original nodes to the *g-d* part and the *o-d* part are updated as the two new parts concurrently, by B1:0 and B1:1 respectively. Then they are merged together. In fact, the merging only considers the link update at most for there is no intersection between the *g-d* part and the *o-d* part.

Theorem 3.17

Given a Lattice *L*, let o be an object to be checked, n be a candidate node of it and *k* be a candidate attribute of n. Assume that *k* is divided into multiple layers to keep o fall into a new candidate node, and let *Tk* be a pre-divided attribute tree of *k* that consists of *k, k0…kn*, where *k* is the root of *Tk*, and *k0…kn* are the successors of *k*. Let *ki* be any one nonleaf node of *Tk* and *ki1…kin* be its children and *kp* be its parent. Suppose that update of *L*(*kp*) have been completed. Assume that *f* (*nset, a*) is the function of adding a new attribute into a lattice, where *nset* is the nodes of an input lattice and *a* is a given attribute. The split on *L(kp)* by *ki*, *ki1…kin* can be expressed as $f(f(\ldots f(f(f(f(L(kp),ki),ki1),ki2),ki3),\ldots),kin-1),kin)$. Assume that *u* (*l*,*a*) is the function of updating a new attribute for each node's intent in a lattice where *l* is an input lattice and *a* is a given attribute. And *Lki1…Lkin* are the sub lattices by updating *ki1...kin* respectively, according to Theorem 3.16. Merging of *Lki1…Lkin* can be expressed as $\sum_{j=1}^{n} Lkij$. Then the following expression is true:

$$f(f(\ldots f(f(f(f(L(kp),ki),ki1),ki2),ki3),\ldots),kin-1),kin) \quad (1)$$

$$= f(f(\ldots f(f(f(f(L(kp),ki1),ki2),ki3),\ldots),kin-1),kin),ki) \quad (2)$$

$$= f(\sum_{j=1}^{n}(f(Lkp,kij)),ki) \quad (3)$$

$$= \sum_{j=1}^{n}(u(Lkij,ki)) + f(\sum_{j=1}^{n}(f(Lkp,kij)) - \sum_{j=1}^{n}(Lkij - TopNode(Lkij)),ki) \quad (4)$$

Proof1. (1)-(2)

From the point of the functionality, the sequential adding an attribute first and its sub attributes second is equal to the execution in reverse. So the expression 1 is equal to the expression 2.

Proof2. (2)-(3)

According to Theorem 3.16, an immediate result is:

$f(f(\ldots f(f(f(L(kp),ki1),ki2),ki3),ki4),\ldots),kin-1),kin) = \sum_{j=1}^{n}(f(Lkp,kij))$. In the other words, the split of ki1…kin can be paralleled. Thus the expression 2 is equal to the expression 3.

Proof3. (3)-(4)

Then, $f(\sum_{j=1}^{n}(f(Lkp,kij)),ki)$ should be computed. For any one *Lkij* belongs to $\sum_{j=1}^{n}(Lkij)$, if *ki* will be updated into *Lkij*, all objects having *ki* is obtained firstly, called Extent(*ki*,*Lkij*). The Extents of each node in *Lkij* should be compared by Extent(*ki*,*Lkij*). Assume that any one node *n* has its extent *e* and let *E* be all nodes' extents. If *e*∩Extent(*ki*,*Lkij*) !⊆*E*, a new node should be generated, or else the node's extent equal to *e*∩Extent(*ki*,*Lkij*) should be updated. Because any object containing *kij* must contain *ki*, it is true that Extent(*kij*,*Lkij*)⊆Extent(*ki*,*Lkij*). It is obvious that for any $e \in E$ it is true that $e \subseteq$ Extent(*kij*,*Lkij*)⊆Extent(*ki*,*Lkij*). So, *e*∩Extent(*ki*,*Lkij*)=*e*. It does not exist any new node if $f(\sum_{j=1}^{n}(Lkij),ki)$ is executed. It only need update *Lkij* by *ki* ,i.e., *u*(*Lkij*, *ki*). To every *Lkij* in $\sum_{j=1}^{n} Lkij$, *u*(*Lkij*,*ki*) can be paralleled. And the generating of *Lkij* and *u*(*Lkij*, *ki*) can be executed sequentially in a single thread. So all the generating of *Lkij* can be expressed as $\sum_{j=1}^{n}(u(Lkij,ki))$. Because *ki* is a child of *kp* and has the children: *ki1…kin*. Let *g*(*a*) be the function that returns all objects having an attribute a. It is true that *g*(*ki*) ⊆ *g*(*kp*) and *g*(*ki1*)∪…*g*(*kin*)⊆*g*(*ki*). The nodes' intent containing *ki* must contain *kp*. For each node *n* in *Lki*, *n*'s intent must contains both *ki* and *kp*. It is true that *n* belongs to *Lkp*. For each node *n*' in *Lki1…Lkin*, *n'*'s intent must contains *ki*. It is true that *n'* belongs to *Lki*. So, according to definition 3.2, the nodes where the intent contains both *ki* and *kp* must be the successors of the nodes where the intent only contains *kp* in (*ki, kp,ki1…kin*). The nodes where the intent contains only *ki* and *kp* in (*ki, kp, ki1…kin*) must be the predecessors of the nodes of *Lki1…Lkin*. So if $\sum_{j=1}^{n}(u(Lkij,ki))$ is executed firstly, the scope of $f(\sum_{j=1}^{n}(f(Lkp,kij)),ki)$ can be reduced to:

$$\sum_{j=1}^{n}(f(Lkp,kij)) - L(ki1) + TopNode(L(ki1)) - \ldots - L(kin) + TopNode(L(kin))$$

,where the TopNodes of *L*(*ki1*)...*L*(*kin*) are retained to update links with the new nodes created by the function *f*. Thus, the expression 3 is equal to the expression 4, where the plus sign "+" means the parallel execution of its left equation and right equation.
Therefore,

$$f(f(\ldots f(f(f(f(L(kp),ki),ki1),ki2),ki3),\ldots),kin-1),kin)$$

$$= \sum_{j=1}^{n}(u(Lkij,ki)) + f(\sum_{j=1}^{n}(f(Lkp,kij)) - \sum_{j=1}^{n}(Lkij - TopNode(Lkij)), ki)$$

Theorem 3.18.

Given a lattice *L*, assume that there are two pre-split attribute Trees *TA* and *TB*. Let *Ak* be any nonleaf attribute of *TA* and *Bk* be any nonleaf attribute of *TB*. For any sub brother attribute *Ai*, *Aj* of *Ak* and any sub attribute *Bi* of *Bk*, let *L*(*Ai*), *L*(*Aj*) and *L*(*Bi*) be the sub lattices after updating *Ai*, *Aj* and *Bi*. After the update of common nodes where their intent should contain both *Ai*, *Aj* and *Bi*, the merging *L*(*Ai*,*Bi*) of *L*(*Ai*) and *L*(*Bi*) can be paralleled with the merging *L*(*Aj*,*Bi*) of *L*(*Aj*) and *L*(*Bi*), and the merging of *L*(*Ai*,*Bi*) and *L*(*Aj*,*Bi*) only need be executed by overlay simply.

According to Theorem 3.16, *L*(*Ai*) and *L*(*Aj*) can be paralleled to be generated and simply overlaid after the computation of the closure nodes closure(*Ai*,*Aj*) where the intent contains *Ai* and *Aj*. *L*(*Ai*) and *L*(*Aj*) only have the intersection nodes in the closure(*Ai*,*Aj*). To all sub attributes of *Ak*, the closure(sub(*Ak*)) should be computed firstly. In the same way, *L*(*Bi*) has its related first computed closure(sub(*Bk*)). Let *common* be {sub(*Ak*,*Bk*)}^{2}-{*Am*| *Am* ∈sub(*Ak*)}-{*Bn*| *Bn* ∈sub(*Bk*)}-{*Am*,*Bn*|*Am* ∈sub(*Ak*), *Bn* ∈sub(*Bk*)}-∅. If the computation of the closure(*common*) is executed firstly, *L*(*Ai*)-closure(*common*) and *L*(*Aj*)-closure(*common*) have no intersection node with each other. *L*(*Bi*)- closure(*common*) has no intersection node with other *L*(sub(*Bk*)-*Bi*). Therefore, the merging *L*(*Ai*,*Bi*) between *L*(*Ai*)-closure(*common*) and *L*(*Bi*)- closure(*common*) is singly executed relative to the merging *L*(*Aj*,*Bi*) between *L*(*Aj*)- closure(*common*) and *L*(*Bi*)- closure(*common*), and the merging *L*(*Ai*,*Aj*,*Bi*) of *L*(*Ai*, *Bi*) and *L*(*Aj*,*Bi*) is only the overlay operation.

One problem is that if the closure(*common*) is executed firstly, it is whether the parallel execution parts will result in updating the closure(*common*) again.

The problem has two sub problems. One is whether it exists that any original node is updated in the closure(*common*) when the parallel execution is in progress. The other is whether it exist that any new node is added in the closure(*common*). For the first problem, it is obvious that the original nodes in the closure have been updated with all attribute set in *common* before the beginning of the parallel parts. And any parallel execution is to adding or updating some attribute set contained in any element of *common*, so the original nodes do not need to be updated again.

For the second problem, after the closure(*common*) is computed, all the objects owning the element of *common* is included in the closure(*common*). For any parallel execution later, there does not exist a extra extent that contains some new object o and has a new intersection extent e with the extents already in the closure(*common*), where o and e are not already in the closure(*common*). Thus, no new node is added in the closure(*common*) for the parallel execution later.

We have given the optimized theorems about the parallel execution of the update of single attribute tree and the merging of multiple trees. One vital problem is that the performance of the parallel update by attributes and the merging later should be better than the one of sequential update by the attribute trees successively. The next theorem is defined for solving this problem.

Theorem 3.19

Given a lattice *L* and two pre-split attribute Trees *TA* and *TB*, assume that *LA* and *LB* is the sub lattices after *L* is paralleled split by any two attributes, *A* and *B*, which belong to *TA* and *TB* respectively. The merging is only for the two closures, one in *LA* and the other in *LB*, where the object set is equal to the intersection of the objects of *LA* and the objects of *LB*. The merging of the closures is equal to the operation that the closure of *LA* and the closure of *LB* add the complement attributes respectively. Except the closures, *LA* and *LB* only needs the overlay simply.

Proof.

To the original lattice *L*, *L*(*A*) and *L*(*B*) increase the related nodes of *A* and the related nodes of *B* respectively. After merging, let *L*(*A*,*B*) be the merging lattice of *L*(*A*) and *L*(*B*). Let *e* be the set of objects having *A* and *B*. The extents of all nodes about both *A* and *B* are contained in the power set of *e*. Let *Le*(*A*) and *Le*(*B*) be the two node sets where the extent is sub set of *e* in *L*(*A*) and *L*(*B*). Let *Lr*(*A*) be the rest of *L*(*A*) except *Le*(*A*) and *Lr*(*B*) be the rest of *L*(*B*) except *Le*(*B*). Except *Le*(*A*) and *Le*(*B*), the merging of *Lr*(*A*) and *Lr*(*B*) cannot generate any node that has the intent containing both *A* and *B*. It means that *Lr*(*A*)'s original nodes containing *A* will not be updated and it is same to *Lr*(*B*)'s original nodes containing *B*. Therefore the merging is executed between *Le*(*A*) and *Le*(*B*), and the overlay is executed for *Lr*(*A*) and *Lr*(*B*).

The complement attribute adding is that adding *B* with *B*'s sub attributes into *L*(*A*) and adding *A* with *A*'s sub attributes into *L*(*B*). According to the naïve algorithm, adding *A* into *Le*(*B*) is equal to adding a new concept (*e*, *A*) into *Le*(*B*). Adding *B* into *Le*(*A*) is equal to adding a new concept (*e*, *B*) into *Le*(*A*). The naïve algorithm will be introduced in the next section. The new nodes after adding must contain both *A* and *B*. If *L* is added by the concept (*e*,(*A*,*B*)) to generate *Le*(*A*,*B*) firstly and then *Le*(*A*,*B*) added by *A* or *B* won't generate new node. The context of *Le*(*A*,*B*) is equal to the context of *Le*(*A*) adding *B* or the context of *Le*(*B*) adding *A*. For the same contexts, the lattices of them are same. Therefore the nodes generated by *Le*(*A*) adding *B* is equal to the nodes generated by *Le*(*B*) adding *A*, i.e. *Le*(*A*,*B*).

If *B* is added into *L*(*A*) to generate *L*(*A*,*B*) and let *EB* be all the objects about *B* in *L*(*A*), it is equal to a new concept (*EB*,*B*) added into *L*(*A*). The adding new node is a process to get new intersections in the naïve algorithm. Because *e* is contained in *EB* and for any extent of *Le*(*A*) $e' \subseteq e$, the intersection $e' \cap EB = e' \cap e$, using (*EB*, *B*) is equal to using (*e*, *B*) when updating *Le*(*A*) in *L*(*A*). So *Le*(*A*,*B*) is equal to all the nodes where the intent contains both *A* and *B* in *L*(*A*,*B*). Thus, the merging of the closures is equal to the operation that the closure of *LA* and the closure of *LB* add the complement attributes respectively.

Assume that the update time of the sequential execution of adding *A* and *B* into *L* are *OA* and *OB*, where *A* is added first and *B* is added second. The update time of the parallel execution of adding *A* and *B* into *L* are *OAp* and *OBp*. The merging time of *L*(*A*) and *L*(*B*) is *OmAB*. The merging time of *Le*(*A*) and *Le*(*B*) is *OmeAB*. The merging time of *Lr*(*A*) and *Lr*(*B*) is *OmrAB*. *OmAB*=*OmeAB*+*OmrAB*. The sum time of the parallel execution is *OmAB*+Max(*OAp*,*OBp*).

It should assure that it is true that *OA*+*OB*>*OmAB*+Max(*OAp*,*OBp*). For *A* and *B* is any two given attributes, assume that *OAp*>=*OBp*. For *A* is added first, *OAp* is equal to *OA*. Then the expression can be presented as *OA*+*OB*>*OmAB*+*OA*, which can be deduced as *OB*>*OmAB*. Assume that *OAp*<*OBp*, then *OA*+*OB*>*OmAB*+*OBp*. Because *B* is added second, it means that *B* is added into *L*(*A*). The amount of *L*(*A*)'s nodes is equal to or larger than the one of *L*. Compared with *B* added into *L*, *L*(*A*) has the nodes to be iterated more than *L*. As *OB* is the time of adding *B* into *L*(*A*) and *OBp* is the time of adding *B* into *L*, *OB*>=*OBp*. The expression is deduced as *OA*>*OmAB*.

OmeAB is the update time of adding (*e*,*B*) into *Le*(*A*) or adding (*e*,*A*) into *Le*(*B*). *OmrAB* is the update time of overlaying *Lr*(*A*) and *Lr*(*B*). If *OA* can be seen as the time *OeA* and the time *OrA* of adding (EA, A) into *Le*(*B*) and *Lr*(*B*),i.e. *OA*=*OeA*+*OrA*, then *OeA* is equal to or larger than *OmeAB* for EA$\supseteq$*e*. *OrA* is the time of adding (EA,A) into *Lr*(*B*), which contains the iteration of all nodes of *Lr*(*B*) to compute new nodes. *OrA* is larger than *OmrAB*, because the overlaying only need compare the common original nodes from *Lr*(*A*) and *Lr*(*B*) to make them consistent. This overlay operation doesn't need to compute the common nodes by iterating again for they have been kept when updating *A* into *L* and *B* into *L* is executed concurrently. So *OeA*+*OrA*>*OmeAB*+*OmrAB*. The same case is to *OB*. So, on the theory, *OB*>*OmAB* and *OA*>*OmAB*.

Therefore, it could ensure that *OA*+*OB*>*OmAB*+Max(*OAp*,*OBp*) on the theory.

For *A* or *B* is any attribute in *TA* and *TB*, the theorem is true to any other attribute. So if using the optimized parallel execution and merging later, the whole performance should be better than using sequential adding by all attributes of *TA* and *TB*.

4. Algorithms

In this section, we describe the naïve algorithm of adding attributes firstly and the optimized secondly. The split of an attribute can be seen as the operation of adding more sub attributes. The algorithms are described in the java oriented-object style. In practice, the coding is a little more complex in java, but this description is enough to present the main functions.

4.1. Naïve Algorithm

The naïve split algorithm is not presented here. It is implemented by using the adding attribute algorithm in this subsection more time in special scopes. The function addattributeToLattice is the main entry. The input variable {attnode} is about the target attribute, the type of which is a TreeNode structure, called AttributeTreeNode defined by us. AttributeTree {T} is the related Tree of attnode. Adding the target attribute is equal to add a new concept in which its extent equals to all the objects and its intent is equal to this attribute. The variable {StartSplitNode} is an extra variable in our version. It is assigned by the first node related to T's root attribute from top to bottom. Its function is recording the first node containing the root attribute to reduce the scope about adding {newConcept}, for only the nodes containing the pre-split attribute need be updated. The function has the other variables as following: {newConcepts, modified, minimal, vSort, and lower}. {newConcepts} records each possible new concept derived from adding {newConcept}. {modified} records all the original nodes modified. {minimal} is a temp variable that keeps the value of the minimal node in each iteration. {vSort} is a TreeMap that lists all the new extents sorted by extent size in ascending order. {lower} is a temp variable recording the minimal node's sub nodes.

There are two parts of which the main function consists. The first part is about generating all new extents and finding their minimal locations in the original lattice. The second part is about adding all new concepts corresponding to these new extents and updating new links between these concepts and the original lattice.

Algorithm 1. AddattributeToLattice

```
addattributeToLattice(AttributeTreeNode attnode,AttributTree T)
main function variable:
Vector newConcepts = new Vector();
Hashtable modified = new Hashtable();
ConceptNode minimal;
TreeMap vSort = new TreeMap();
Vector lower = new Vector();
Concept newConcept = attnode.c;
Concept StartSplitNode= T.root.original;
    {part1}
    {part2}
```

In the first part, the line 1 shows the preprocess (StartSplitNode). It sorts the nodes in the splitting scope by intent size in ascending order. The lines 2-26 are an iteration process that each node is iterated. And the extents of it's concept with newConcept are compared. In the line 8, the intersection e of the extents is computed. vSort gets the hashtable ht that keeps all the extents having the same size of e and their current minimals in the line 11. In the line 12, the minimal of e is obtained. Here, we describe the process simply. In fact, if ht has no minimal about e, current node n will be set as minimal. If the size of A is equal to the one of e, it means that A is a concept to be updated. Then A's intent includes the newConcept's intent. A is included in the modified. ht removes e since e is the extent that has already been existed in the original lattice and won't generate a new node. If the condition of the line 13 is not true, then A is set as current minimal about e and ht records e and its minimal. vSort updates ht in the line 24. After this part, all the extents of new concepts are found.

Algorithm 1. AddattributeToLattice#part1

```
1.   Iterator iterNode = preProcess(StartSplitNode );
2.   While(iterNode.hasnext())
3.   {
4.     ConceptNode[] N=GetConceptsof_intent_size(iterNode);
5.     For(ConceptNode n:N)
6.    {
7.       Concept A= n.Concept;
8.       Extent e=Interection_of_Extent (newConcept,A);
9.       Integer eSize = new Integer(e.size());
10.      Hashtable ht;
11.      ht = (Hashtable) vSort.get(eSize);
12.      minimal = (ConceptNode) ht.get(e);
13.      if (A.getExtent().size() == e.size())//   A is a update_concept;
14.      {
15.        UnionIntent(A,newConcept);
16.        modified.add(A);
17.        ht.remove(e);
18.       }
19.      else
20.     {
21.       e.setminimal(A);
22.       ht.put(e, minimal);
23.      }
24.      vSort.update(ht);
25.     }
26.   }
```

Part2 is an iteration process for each new extent which need generate a new concept. The iteration is executed in the ascending order of extent size. It keeps the small size extent is computed firstly. The line 3 shows obtaining a Hashtable ht in which each extent has same size. In the line 4-18, it is a process that iterates each extent e in E and generates its concept And E is an array of ht’s all value. In the line 6, the minimal of e is got. The line 7 shows the union intent of the minimal’s intent and newConcept’s intent. The line 8 shows the generating of new concept of e. Then genC is added into the newConcepts in the line 9. Then the minimal’s low cover is computed. The minimal’s low cover Can is filtered. The directed successors min of genC in the Can is computed. The lines 13-16 present the updating link process where the nodes of the min drop links with minimal and add links with genC. In the line 17, a new Link is created between genC and minimal.

Algorithm 1. AddattributeToLattice#part2

```
1.   While(vSort.hasnext())
2.   {
3.     Hashtable ht =Gethashtableofextentsize(vSort);
4.      Foreach(e: ht.E)
5.     {
6.       ConceptNode minimal= ht.get(e);
7.       intent=Union(minimal.intent, newConcept.intent);
8.       ConceptNode genC=new ConceptNode(e, intent);
9.       newConcepts.add(genC);
10.      lower= getLowcover(minimal);
11.      Can=minCandidate(lower);
12.      Conceptnode[] min =minClosed(genC, Can);
13.      Foreach(Conceptnode n:min)
14.      {
15.        updatelink(genC,n,minimal);// drop(minimal， n);newlink(genC,n);
16.      }
17.      newlink(genC,minimal);
18.     }
19.  }
```

4.2. The Optimized Algorithm of Splitting Attributes

The optimized algorithm also uses adding attribute algorithm with the special scopes. The optimized algorithm is composed of the following functions.

The main entry is a fragment as following:

Main Entry of the Algorithm

```
If(Canbedivided(ConceptNode e,Object aunit, ArrayList<Attribute>attrs))
        IncrementSplitAttributeLattice (predividedforestgraph);
```

If the function Canbedivided returns true, the incremental attribute split is executed by the function IncrementSplitAttributeLattice with input predividedforestgraph. The predividedforestgraph contains the pre-split attribute Tree and related extra information. The function Canbedivided contains the optimized function of priority sorting. The function IncrementSplitAttributeLattice contains the optimized paralleled split function.

4.2.1. The First Optimized Functions

4.2.1.1. Canbedivided

The Canbedivided function is to judge whether a ConceptNode e is a divisible node that could create a consistent node of aunit after split. The variable {attrs} is aunit's Concept attributes. The variable {flag} is a temp variable used to record whether the pre-split succeeds. The variable {ownintent} records e's own intent. The variable {candidateintent} contains the candidates. The variable {conceptowns} contains the concept intent of aunit's concept in e. The variable {iterator} is the iterator of the parents of e. The variable {q} is a candidate list sorted by the priorities. The variable {iteratora} is the iterator of candidateintent. The variable {predividedforestgraph} contains the pre-split attribute Trees.

Algorithm3 Canbedivided

```
boolean Canbedivided(ConceptNode e,Object aunit,ArrayList<Attribute>attrs)
main function variable:
boolean flag;
Intent ownintent=(Intent)e.getConcept().getIntent().clone();
Intent candidateintent=(Intent)e.getConcept().getIntent().clone();
ArrayList conceptowns = new ArrayList();
Iterator iterator=e.getParents().iterator();
LinkedList<Attribute> q = new LinkedList<Attribute>();
Iterator iteratora=candidateintent.iterator();
Graph predividedforestgraph =new Graph();
1.  while(iterator.hasNext())
2.  {
3.    ConceptNode p=(ConceptNode)iterator.next();
4.    ownintent.removeAll(p.getConcept().getIntent());
5.  }
6.  while(iteratora.hasNext())
7.  {
8.    Attribute p=(Attribute)iteratora.next();
9.    if(attrs.contains(p))
10.   {
11.     conceptowns.add(p);
12.   }
13. }
14. q =computePriorityList(candidateintent,ownintent,conceptowns);
15. while(!q.isEmpty())
16. {
17.   Attribute candidate=q.removeFirst();
18.   flag=predivideintervals (candidate,aunit,e.extent-aunit);
19.   if(flag)
20.   {
21.     return true;
22.   }
23. }
```

The lines 1-5 show the iteration of e’s parents by which the own intent is computed by removing e’s parents’ intent. The lines 6-13 show computing the concept own intent of e. The line 14 shows the invoking of the function computePriorityList. The sorted result is assigned to q. The iteration of q is executed in the lines 15-23. In one loop, the first candidate of q is fetched. Then the function predevideintervals is executed to judge whether the current candidate can create a consistent interval after pre-dividing and the result is assigned to flag. If flag is true, the function is returned. If flag is false, the while clause is continued.

4.2.1.2. Predivideintervals

The function predivideintervals is to compute the sub intervals that ensure current target falling into a consistent interval. The input variable {candidateinterval} is an argument to pass current attribute from the function Canbedivided. The variable {target} passes the target object, which is often an assertion from context knowledge. The variable {objs} passes other objects. The variable {flag} is a boolean variable as a return parameter. The variable {n} is a TreeNode-type variable. The corresponding attribute Tree Node of the input candidateinterval is assigned to it. The variable {targetAttribute} records the pre-split sub attribute of target. The lines 3-17 show computing the pre-split sub attributes of other objects. If any attribute of an object is equal to targetAttribute and it is inconsistent with target, flag is set false; the targetAttribute’s priority is computed and written into q after sorting.

Algorithm4 Predivideintervals

```
boolean predivideintervals (Attribute candidateinterval, Object target,
ArrayList<Object> objs)
main function variable:
boolean flag=true;
TreeNode n= Forestgraph.getAttNode(candidateinterval);
Attribute targetAttribute=computePresplitAttribute(target, n) ;
 1.  if(targetAttribute==null)
 2.    return false;
 3.  For (int k=0;k<objs.size();k++)
 4.  {
 5.     Object obj= objs.get(k);
 6.     tempattr =computePresplitAttribute(obj,n);
 7.     if(tempattr== targetAttribute &&target.inconsistentwith(obj))
 8.     {
 9.       flag=false;
10.     }
11.  }
12.  if(!flag)
13.  {
14.    Priority p=computePriority(targetAttribute);
15.    Sort(q, targetAttribute,p);
16.  }
17.  return flag;
```

4.2.1.3. ComputePresplitAttribute

The function computePresplitAttribute is to check a sub interval owned by the input obj. If q has contained this sub interval, it returns null. If it is null, the sub interval will be created. The pre-divided forest graph adds it.

Algorithm5 ComputePresplitAttribute

```
computePresplitAttribute(Object obj, TreeNode n)
main function variable:
Attribute nextattribute
1.   nextattribute=checksubInterval(obj,n.childs);
2.   if(q.contains(nextattribute))
3.   {
4.     return null; //used.add(nextattribute);
5.   }
6.   if(nextattribute==null)
7.   {
8.     nextattribute =createNewAttributeTreeChild(n, objs);
9.   }
10.  predividedforestgraph.addandUpdate(n, nextattribute, objs);
11.  return nextattribute;
```

4.2.2. The Second Optimized Functions

4.2.2.1. The Non Clone Characteristic

The second functions are for parallel optimization, which aims at the less time cost. When splitting and merging is being run in the concurrent way, the simple way is to make multiple copies to ensure that any thread of splitting or merging does not conflict or disorder each other such that the final lattice is the right result. However, if the lattice in use is larger, the clone way is not available. Even if the big memory or disk space is supporting in backend, it cannot afford to the space cost produced by a mount of concurrent threads. So we chose to design the algorithm in the non clone way, which means it is executed on the same lattice. Figure 5 sketched the clone way and non-clone way. At the left of the figure, the clone way will create two clones of Original lattice in a simple concurrent execution of two splitting threads. And at the right of the figure, the dashed Split1, Split2 and Merged lattices are virtual and just the middle states of the updated Original lattice, so there is no extra full copy of the Original lattice.

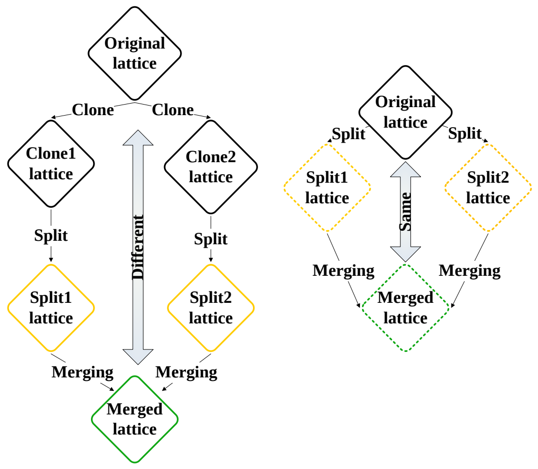

Figure 5 The Clone Way and Non-clone Way of the Algorithm

From Figure 5, it is obvious that the clone lattices occupy different space so that the two threads can read or write in an isolated lattice at once respectively and not be confused. To read or write in the original lattice, our non-clone way is a little more complicated than the optimized theories defined above. It is involved in the extra implement of synchronization, isolation and integrity, which we describe in the following algorithms further. Synchronization is for the threads concurrency don't influence the next step of the process. Isolation is for one thread can read or write their data in an isolation manner. Integrity is for one thread obtains its full and correct input data to compute such that the final result is correct after parallel running.

4.2.2.2. IncrementSplitAttributeLattice

The variable {aset} is an array that contains all new attributes to split. The variable {NBs} contains all the original Concept Bottom nodes having aset in their intents. The variable {commonintent} contains the intersection intent of all NBs. The variable {Conceptclosure} is an array that contains all nodes owning commonintent. In GetClosureByIntent, the power set of commonintent is computed. Then it computes all the parallel intent set for the parallel adding and merging. The power set removes all the parallel intent set and computes the corresponding nodes, i.e. Conceptclosure. According to Theorem 3.16 and Theorem 3.18, to avoid too redundant computing of the common nodes, the lines 1-6 show the update computing of the common Concept closure in advance. The lines 7-17 show starting each thread for the parallel execution function SplitLatticeBySingleTree of every attribute Tree. After waiting the Threads stopped, the merging of the new sub lattices is executed concurrently in the lines 19-20. The lines 18 and 21 are to wait all sub threads stopping for synchronization.

Algorithm6 IncrementSplitAttributeLattice

```
void IncrementSplitAttributeLattice (Graph predividedforestgraph)
Attribute[]aset= predividedforestgraph.GetT().allnewattributes();
ConceptNode[]NBs=GetConceptBottomNodeRelatedAttribute(aset);
Intent commonintent=intentIntersection(NBs);
Node[] Conceptclosure=GetClosureByIntent( commonintent);
ConceptNode startNode=computeStartNode(NBs);
1.    foreach(Attribute attribute : aset)
2.    {
3.      Extent extent =getExtent(attribute, Conceptclosure);
4.      ConceptNode newconcept=new ConceptNode(extent, attribute);
5.      addattributeToLattice (L,newconcept, startNode);
6.    }
7.    for(int k=0;k<predividedforestgraph.Tree.size();k+=2)
8.    {
9.      AttributeTree T1=predividedforestgraph.Tree.get(k);
10.     Thread n=new Thread(SplitLatticeBySingleTree(predividedforestgraph.Tree.get(k));
11.     n.start();
12.     If(k+1< predividedforestgraph.Tree.size())
13.     {
14.       AttributeTree T2=predividedforestgraph.Tree.get(k);
15.       Thread n2=new Thread(SplitLatticeBySingleTree(predividedforestgraph.Tree.get(k+1));
16.       n2.start();
17.   }
18.     waitAllsubThreadEnd();//for synchronization
19.    If(k+1< predividedforestgraph.Tree.size())
20.      mergingTwoSubLattices(T1,T2);
21.   waitAllsubThreadEnd();//for synchronization
22.  }
```

4.2.2.3. SplitLatticeBySingleTree

The function SplitLatticeBySingleTree is adding an attribute Tree concurrently into the lattice in a single thread. Figure 6 presents the sketch of the process. It computes the first Concept Node containing T's root attribute (to be split) from up to bottom. It uses a bottom-up iterating the Tree to start the Threads of each level of the Tree. For each level, the attributes of this level is assigned to the ArrayList level. Then the iteration for each element in level is done. In the second loop, for each attribute, the related objects are fetched from the predividedforestgraph. A new ConceptNode c is generated and assigned to the variable c of anode. Then the function splitConcept is executed in a new thread. After all threads of this level are completed, the Threads of the next level are started. The line 18 waitAllsubThreadEnd is for synchronization. For integrity, the line 19 is to drop all old links between the nodes in the original lattice after all new nodes are added and linked by threads. This drop of links at last ensures one thread can always read its original links (without needless new links and nodes).

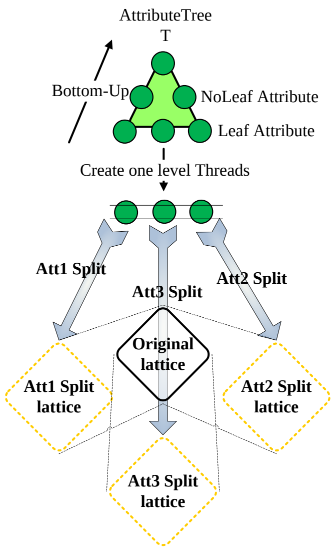

Figure 6 The Sketch of the Function SplitLatticeBySingleTree

Algorithm7 ComputePresplitAttribute

```
SplitLatticeBySingleTree(Tree T)
Attribute[] attrnodes;
T.root.original= ComputeFirstNodeOfRoot(T.root);
Iterator<Integer> bottomup=T.levels.descendingKeySet().iterator();
1.    for (; bottomup.hasNext(); )
2.    {
3.      ArrayList<AttributeTreeNode> level=levels.get(bottomup.next());
4.      ArrayList<Thread> ths=new ArrayList<Thread>();
5.      for(AttributeTreeNode anode:level)
6.     {
7.        ArrayList<Object> temos=predividedforestgraph.getobjects(attrnode);
8.        Extent extent=new SetExtent();
9.        extent.addAll(temos);
10.       Intent intent=new SetIntent();
11.       intent.add(anode.attribute);
12.       ConceptNode c=new ConceptNode(extent, intent);
13.       anode.c=c;
14.       Thread t= new Thread(splitConcept (anode, !anode.isLeaf(),T));
15.       ths.add(t);
16.       t.start();
17.   }
18.   waitAllsubThreadEnd(ths);// for synchronization
19.   droplinks(allTdroplist);// for integrity
20.   }
```

4.2.2.4. SplitConcept

The function splitConcept is according to Theorem 3.17. It calls the extra functions about split except for the na ïve version of adding attribute. It is presented in the line 1 that the current formal context adds the input attribute. The line 2 shows that att.c is assigned to the variable {newConcept}. If the input notleaf is true, the att's subNodeSetforOneAttribute includes all nodes within the scope of sub attributes and att's firstsubnodes adds TopNodes of all sub attributes according to the expression (4) of Theorem 3.17. Then the function addConceptfornotleaf is executed. Or else the attribute is added by invoking the addConceptforleaf. The addConceptfornotleaf and addConceptforleaf are two special versions of addattributeToLattice for split.

Algorithm8 SplitConcept

```
splitConcept (AttributeTreeNode att, boolean notleaf, AttributTree T)
main function variable
ConceptNode newConcept;
1.    currentContext.addAttribute(att.attribute);
2.    newConcept=att.c;
3.    if(notleaf)
4.    {
5.       for (AttributeTreeNode n:att.children())
6.       {
7.          att.subNodeSetforOneAttribute.addAll(n.subNodeSetforOneAttribute);
8.          att.firstsubnodes.add(n.subNodeSetforOneAttribute.getFirst());
9.       }
10.      addConceptfornotleaf(att,T);
11.   }
12.   else addConceptforleaf (att,T);
```

4.2.2.5. MergingTwoSubLattices

The function mergingTwoSubLattices is merging the two sub lattices (showed in Figure 7). The two Trees, Tree T1 and T2, record Lattice l1's and Lattice l2's different attribute Trees with each other. The first 'for' loop is the iteration of each level of T1. The second 'for' loop is the iteration of the nodes of the current level of T1 from the first loop. In the second loop, the sub lattice of each attribute {a} merges with l2. This process is presented in the lines 8- 19, which have two loops of T2 where the first loop iterates each level of T2. In each level of T2, each attribute {b} is iterated, and the merging {la-b} between l(b) and l(a) is executed in a single thread for {b} according to Theorem 3.18. The line 18 is for synchronization. From

Figure 7, we can find that la is a part of l1, lb is a part of l2 and la-b is a part of the final merged lattice.

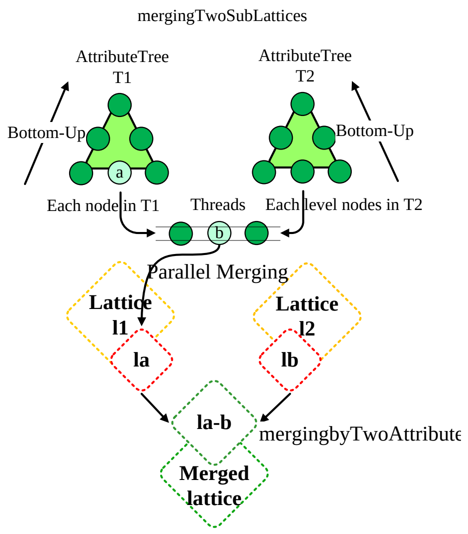

Figure 7 The Process of the Function MergingTwoSubLattices

Algorithm9: MergingTwoSubLattices

```
mergingTwoSubLattices(ConceptNode[] l1, ConceptNode[] l2)
main function variable
Tree T1=relatedTree(l1);
Tree T2=relatedTree(l2);
1.For(int i=T1.level;i<T1.level;i--)
2.{
3.   For(int j=0;j<T1.level[i].nodes.size;j++)
4.  {
5.     AttributeTreeNode a=T1.level[i].nodes.get(j);
6.     If(a.isRoot)
7.        break;
8.      For(int m=T2.level;m<T2.level;m--)
9.      {
10.        For(int k=0;k<T2.level[m].nodes.size;k++)
11.        {
12.          AttributeTreeNode b=T2.level[m].nodes.get(k);
13.          If(b.isRoot)
14.             break;
15.          Thread t=new Thread(mergingbyTwoAttributes(a,b,t1,t2));
16.             t.start();
17.         }
18.         waitAllsubThreadEnd();
19.     }
20.   }
21. }
```

4.2.2.6. MergingbyTwoAttributes

According to Theorem 3.19, the function mergingbyTwoAttributes completes the merging two sub lattices. Each lattice has one attribute to be considered as the optimized factor.

Algorithm10 MergingbyTwoAttributes

```
mergingbyTwoAttributes(AttributeTreeNode a, AttributeTreeNode b, AttributTree T, AttributTree T2)
main function variable
Extent IntersectionExtent;
ConceptNode[] lawithb;
ConceptNode[] lbwitha;
ConceptNode[] lawithoutb;
ConceptNode[] lbwithouta;
List la=a.subNodeSetforOneAttribute;
List lb=b.subNodeSetforOneAttribute;
{part1}
{part2}
{part3}
```

In part1, the line 1 shows the intersection IntersectionExtent of the extent of a and the one of b. Lines 2-23 shows the part1 computing of the nodes {lawithb} owning the extent contained by IntersectionExtent, the nodes {lawithoutb} not owning the extent contained by IntersectionExtent and the {lawithb} nodes' minimal parents in la. Because sometimes the lattice does not always have {lawithb} or the links between lawithb, lbwithouta and lawithb's minimal parents should be updated, the line 24 presents that {lawithb} adds all minimal parents for overlay function being executed correctly.

Algorithm10 MergingbyTwoAttributes#part1

```
1.   IntersectionExtent=Intersection (a. Extent, b.Extent);
2.   foreach(ConceptNode e:la)
3.   {
4.     If(e.Extent⊆IntersectionExtent)
5.     {
6.      lawithb.add(e);
7.     }
8.     else
9.     {
10.     lawithoutb.add(e);
11.    }
12.    If(IntersectionExtent⊆ e.Extent)
13.    {
14.      if(minparents.size()==0)
15.          minparents.add(c);
16.      else
17.      {
18.        boolean flag=   checkIsminparents(c,minparents);
19.        if(flag)
20.          minparents.add(c);
21.      }
22.    }
23.  }
24.  lawithb.addAll(minparents);
```

In part2, the lines 1-11 show the part2 computing of the nodes {lbwitha} owning the extent contained by IntersectionExtent and the nodes {lbwithouta} not owning the extent contained by IntersectionExtent in lb.

Algorithm10 MergingbyTwoAttributes#part2

```
foreach(ConceptNode e:lb)
{
  If(e.Extent⊆IntersectionExtent)
  {
   lbwitha.add(e);
  }
 else
 {
   lbwithouta.add(e);
  }
}
```

In part3, if lawithb's size>0, the adding new concept (IntersectionExtent,b) into lawithb (lawithb =>la-b) are executed by the function {addConceptforMergeforNotclone}, and it executes the overlay of {la-b} and {lbwithouta}, which is different from the clone version. In the clone version, {lawithoutb} and {lbwithouta} in two different copies are overlaid into one of them according to Theorem 3.19. Then {lbwitha}-{lawithb}is removed from current lattice by the lines 6 and 8 in the non-clone version. This operation does not exist in the clone version for la and lb are in different copies. If lawithb's size =0, the overlay of {la} with {lb} is executed. Figure 8 shows the process of mergingbyTwoAttributes.

Algorithm10 MergingbyTwoAttributes#part3

```
1.    Intent bintent=b.c.getIntent();
2.    ConceptImp  newConcept=new ConceptImp(b.c.getExtent(),bintent);
3.    if(lawithb.size()>0)
4.    {
5.      addConceptforMergeforNotclone(newConcept,lawithb.getFirst(),a, b, T,T2,lawithb);
6.      lbwitha.removeAll(lawithb);//for nonclone
7.      overlay(lawithb,lbwithouta,lbwitha);
8.      remove(lbwitha); //for nonclone
9.    }
10.   else{overlay(la,lbwithouta,lbwitha);}
```

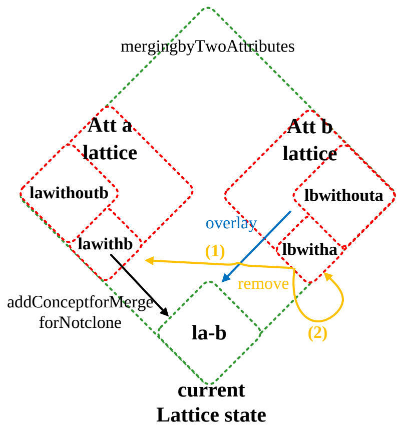

Figure 8 The Process of MergingbyTwoAttributes

4.2.2.7. Overlay

Overlay function in non-clone way only needs updating links for all new nodes are created before. The first and second loops are to iterate each node {c} in {lbwithouta} by descending order. The third loop is to check every node {c'} in {lawithb}. The lines 6-7 and lines 11-12 execute the continue operation if c or c' is removed in the parallel way. The line 13 shows {CheckandUpdateLinks} part, which is the main process of checking links between {c} and {c'}.

Algorithm11 Overlay#loops

```
overlay (lawithb,lbwithouta,lbwitha)
main function variable ∅
1.  for(Integer k:lbwithouta.descendingKeySet())
2.  {
3.     List<ConceptNode> values=( List<ConceptNode>)lbwithouta.get(k);
4.     for(ConceptNode c:values)
5.     {
6.        if(c.isremoved)
7.          continue;
8.        List<ConceptNode> Childinla=new List<ConceptNode>();
9.        for(ConceptNode c':lawithb)
10.       {
11.         if(c'.isremoved)
12.            continue;
13.         {CheckandUpdateLinks}
14.       }
15.    }
16. }
```

From the line 2 to the line 6 in the {CheckandUpdateLinks} part, if c.extent ⊇c'.extent, then update of c'.intent union with c.intent is executed. From the line 7 to the line 11, if c'.extent⊇c.extent, then update of c.intent union with c'.intent is executed. The line 12 shows the condition whether the third loop needs to be continued by the temp variable needcontinued. If the intents of c and c' have the inclusion relation, the temp variable needcontinued is set as true and the loop continues the rest part. If not, the loop is continuing to the next iteration of it. It is presented in the lines 14-18 that the links of common children and parents of c and c' are checked and removed. Figure 9 shows the cases of common children and parents.

Algorithm11 Overlay# CheckandUpdateLinks

```
1.  boolean needcontinued=false;
2.  if(c. Extent⊇c'.Extent))
3.  {
4.     {part1}
5.     {part2}
6.  }
7.  else if(c'.Extent⊇c.Extent)
8.  {
9.     needcontinued=true;
10.    checkandupdateIntent(c, c'.Intent);
```

```
11.   }
12.   if(! needcontinued)
13.     continue;
14.   if(haslink(c, c'))
15.   {
16.     checkandRemoveCommonParentLink(c, c',lbwitha);
17.     checkandRemoveCommonChildrenLink(c, c',lbwitha);
18.   }
```

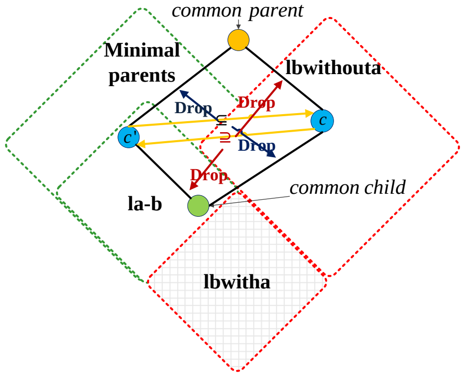

Figure 9 The Cases of Common Children and Parents

In Figure 9, If c.Extent⊇c'.Extent, then it drops the links pointed by the deep blue vector lines. If c'.Extent⊇c.Extent, then it drops the links pointed by the red vector lines. When it checks the common children and parents, it does not need to consider lbwitha, which will be removed after merging if its size>0.

In the {CheckandUpdateLinks#part1}, it first checks whether the intent of c' contains the intent of c. If not, the intent of c' should include the intent of c. Then the conditions for continuing the loops are checked. Expression 1) and 2) are the continuing condition if there could exist any middle node between c and c' (showed in Figure 10).

Algorithm11 Overlay# CheckandUpdateLinks#part1

```
1.   checkandupdateIntent(c',c.Intent);
2.   if(∃parent ∈ c'.getParents(), c.Extent ⊇ parent.Extent)        1)
3.     continue;
4.   if(∃child ∈ c.getChildren() ∩ lbwithouta, child.Extent ⊇ c.Extent)  2)
5.     continue;
6.   needcontinued=true;
```

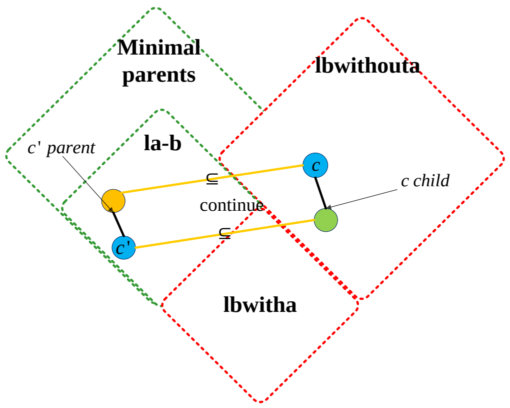

Figure 10 The Continuing Conditions of Expression1 and 2

In the {CheckandUpdateLinks} part2, Expression 3) and 6) are to add new link between c and c'. Expression 4) and 5) are the conditions to determine whether the current c and c" should be linked (showed in Figure 11).

Algorithm11 Overlay# CheckandUpdateLinks#part2

```
if(Childinla.size()==0)
 {
   Childinla.add(c');
   newLink(c, c');                                          3)
  }
  else
{
   boolean flag=false;
   Vector<ConceptNode> cr=new Vector<ConceptNode>();[*]
   if(∃c"∈ Childinla,c".Extent ⊇ c.Extent)                  4)
     flag=true;
   if(∃c"∈ Childinla,c.Extent ⊇ c".Extent)                  5)
     cr.add(c");
   if(!flag)
   {
      Childinla.add(c');
      newLink(c, c');                                       6)
      Childinla.removeAll(cr);
   }
}
```

[*] //for removing smaller children

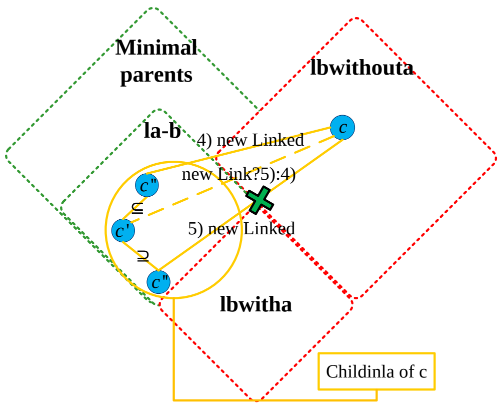

Figure 11 The Conditions of Expression 4 and 5

Expression 4) shows the condition that if there has existed any new link between c and c", where c" is a parent of c', then the dotted line is not added between c and c'. Expression 5) shows the condition that if there has existed any or no new link between c and c", where c" is c' child, then the dotted line is added between c and c'. If Expression 5) is true, new link is operated by the lines 14-19, in which the links between c and all c"s should be dropped in the line 18 (denoted by the green cross in Figure 11).

The following other functions are also designed to implement the non-clone parallel way (in the Table 2). We use the oriented-object characteristic to override the ConceptNodeImp type from the Galicia source code. We add the fields into it: f1) addrootlabels, f2) updatelabels, f3) lock, and f4) isremoved.

Table 2 The List of the Other Functions About the Non-clone Parallel Way

Function	Related field in use
1.addConceptforleaf	f1),f2), f3)
2.addConceptfornotleaf	f1),f2), f3)
3.addConceptforMergeforNotclone	f1),f2), f3)
4.minClosedforparallel(in 1,2,3)	
5.minCandidateformerging (in 3)	f1) [*]
6.preProcessForParallel(in 1)	f1),
7.preProcessForParallelforaddnotleaf (in 2)	f1),f2),f3)
8.preProcessForParallelforMergingNotClone(in 3)	f1)

For integrity, we use addrootlabels and updaterootlabels.

When a new node is created, the addrootlabels records all the AttributeTrees' root nodes of the attributes added into the node's intent and it ensures that the node is created and iterated in the isolated state by one single thread of one attribute adding.

When an original node is updated by multiple threads of attribute adding, the updatelabels records all the attributes that are updated into the node. It avoids the single thread of adding attribute to update the intent of its new node by using the intent created by the threads of other AttributeTrees.

The lock field is for isolation when a node is updated. When updating Node A's intent, we use lock as following:

```
while(A.lock)
```

[*] uses f1 to remove not related low cover by transferring parameters implicitly

```
{ }
A.lock=true;
{A update process}
A.lock=false;
```

And in the overlay process, the isremoved field is for avoiding needless iteration of nodes removed by the remove function of merging. The isremoved field is referenced by the functions: overlay and remove. In addition, the function minClosedforparallel uses extent to determine the minClose instead of intent.

5. Performance Evaluation

For first optimization strategy has obvious advantage compared with the naïve, we tested and reported the second parallel strategy in this paper. The three tests are run: 1. The objects increasing. 2. The attributes increasing. 3. Both the objects and attributes increasing. We take the random contexts by the max size {50×50}. The coding is implemented by java and used the Galicia source code "MagaliceAGen.java", on which we overwrite its "addConcept" function and implements the parallel algorithms. The coding has a little difference with the expression of algorithms in the paper, for the easier running and debugging programs with oriented-object coding style. The current tests are executed on the single Notebook, with Intel core Duo CPU, @{1.86GHZ,1.87GHZ}, and java virtual machine with max memory 512 mb. And the Tables 3-5 and Figures 5-7 show the time cost of the naïve and the optimized parallel algorithm. The three tests show the time cost of the optimized is obviously less than the one of the normal. In non-clone way, the optimized does not result in the out of memory heap space meanwhile.

Table 3 The Test 1 With Attributes Increasing

Objects:Attributes	50:10	50:20	50:30	50:40	50:50
normal(msec)	391	7840	91733	767347	3096120
optimizal(msec)	1310	2263	4805	10172	22116

Table 4 The Test 2 With Objects Increasing

Objects:Attributes	10:50	20:50	30:50	40:50	50:50
normal(msec)	157	5855	79166	548719	3027889
optimizal(msec)	499	1140	1156	7708	45252

Table 5 The Test 3 With Both Attributes and Objects Increasing

Objects:Attributes	5:5	10:10	15:15	20:20	25:25	30:30	35:35	40:40	45:45	50:50
Normal(msec)	24	31	78	359	1482	6630	30810	172979	1174575	2692189
optimizal(msec)	226	250	234	343	718	1458	1606	4437	13621	32714

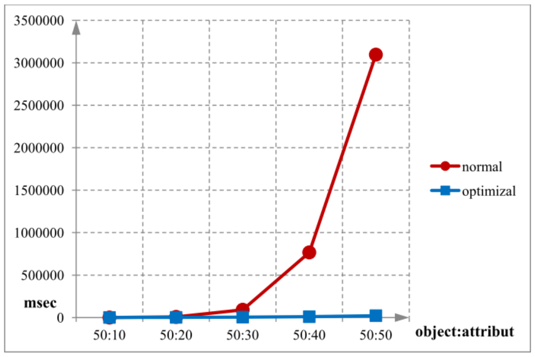

Figure 12 The Time Comparison in Test1

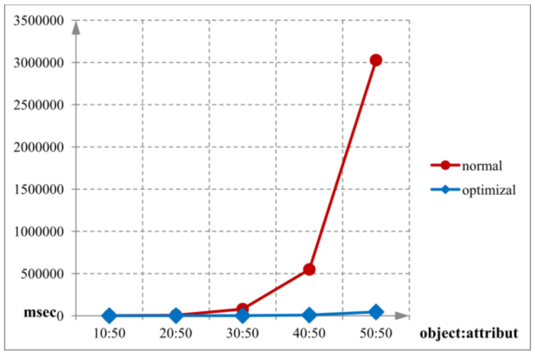

Figure 13 The Time Comparison in Test2

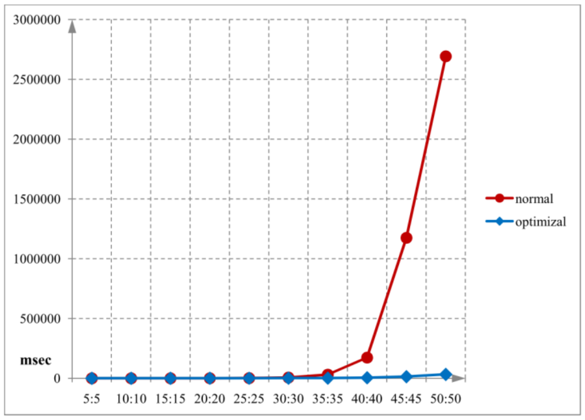

Figure 14 The Time Comparison in Test3

6. Conclusion

In this paper, we analyze the lattice-based storage method of inconsistent knowledge in the context-aware application. Further, to keep provide the newer context knowledge timely, the maintaining of the consistent information about the context knowledge is concerned. The aim is that based on context lattice storage, it implements the update of lattice dynamically to keep knowledge in the consistent intervals. For the consistent result is changeable in the dynamic environment, the split of the intervals is proposed such that the consistent knowledge fall into the split intervals to provide the query and reasoning faster. Then the paper focuses on the priority sort of the pre-split attributes and the parallel execution of splitting. Based on FCA theory, the feasibility of the priority sort is analyzed and the optimized theories of the parallel execution are proved. Using two strategies, we propose an optimized split algorithm, which is based on the naïve one. It is tested and shows better performance compared with the naïve. And the parallel algorithm is in a non-clone way to assure that the memory does not overflow owing to the copies of the lattice by multiple threads.

The future work: 1. Testing on the bigger data set. It is necessary to testing on the real big data set (TB level) such that the algorithms could be modified to adapt to the real big data environment. 2. Improving the query performance on current lattice storage and implementing the reasoning techniques on the context lattices. A better query method is needed, and a series of the reasoning techniques about consistency should be specialized on the context lattices.

Acknowledgements

This work is supported by Shanghai Scientific Development Foundation under Grant No. 11530700300.

References

[1] B. Ganter, G. Stumme, and R. Wille, Editor, “Formal Concept Analysis Foundations and Applications, Lecture Notes in Computer Science, Springer, Volume 3626”, (**2005**).

[2] R. Wille and M. Zickwolff, Begriffliche Wissensverarbeitung. Grundfragen und Aufgaben [Broschiert].B.I.-Wissenschaftsverlag, **(1994).**

[3] G. Stumme and R. Wille, Editor, Begriffliche Wissenverarbeitung –methoden und anwendungen. Springer Heidelberg, **(2000).**

[4] C. Carpineto and G. Romano, “Concept Data Analysis: Theory and Applications”, John Wiley \& Sons**(2004).**

[5] B. Ganter, G. Stumme, and R. Wille, “Formal Concept Analysis”, Springer Berlin Heidelberg, **(2005).**

[6] B. Ganter, G. Stumme, and R. Wille, “The ToscanaJ Suite for Implementing Conceptual Information Systems, Springer Berlin Heidelberg”, **(2005).**

[7] Z. Zhong, X. Lin, and J. Gu, Dynamic Context Modeling based FCA in Context-aware Middleware. “The proceedings of 13th International Conference on Enterprise Information Systems (ICEIS)”, **(2011)** June 6-8; Beijing, China, pp.103-110.

[8] M. Dao, M. Huchard, M.R. Hac ène, C. Roume, et.al, “Improving Generalization Level in UML Models Iterative Cross Generalization in Practice, in Conceptual Structures at Work”, **(2004)**, pp. 346-360.

[9] B.Ganter and R. Wille, Conceptual Scaling, in Applications of combinatorics and graph theory to the biological and social sciences, “The IMA volumes in Mathematics and its applications”, New York **(1989)**, Vol. 17, pp.139–167.

[10] P. Valtchev, D. Grosser, C. Roume, and M.R Hacene, “GALICIA: an open platform for lattices, in Using Conceptual Structures: Contributions to 11th Intl. Conference on Conceptual Structures (ICCS’03), Aachen (DE), Shaker Verlag”, **(2003)**, pp.241–254.

[11] B. Ganter, G. Stumme, R. Wille, T. Tilley, R. Cole, P. Becker, and P. Eklund, A Survey of Formal Concept Analysis Support for Software Engineering Activities. “In Formal Concept Analysis: Springer Berlin Heidelberg”, **(2005).**

[12] R. Wille, “Restructuring lattice theory: An approach based on hierarchies of concepts, in Ordered Sets, Reidel, Dordrecht-Boston”, **(1982)**, pp. 225-470.

[13] B. Ganter and R.Wille, “Formal concept analysis: mathematical foundations, Springer”, **(1999).**

[14] R. Godin, G. Mineau, R. Missaoui and H. Mili, “M éthodes declasification conceptuelle bas ées sur les treillis de galois et applications, Revued’Intelligence Artificielle”, **(1995)**, Vol. 9(2) pp.105-137.

Authors

Zhou Zhong, Ph.D. Student (2009-2014) in Institute of Computer Application, East China Normal University, China. His research interests now include context aware computing, database management, Formal concept analysis.

Junzhong GU, Professor of Computer Science, Head of Institute of Computer Application, East China Normal University, China. He received the M.S. degree in Computer Science from East China Normal University in 1982. He works at East China Normal University since 1982. He worked as visit professor at GMD, and University Mannheim, Germany (1987-1989, and 1991-1993). His research interests now include context aware computing, distributed data management, Web searching and multimedia information processing.