Faceted Navigation in the Lattice of Resource Spaces

Hai Zhuge, *Senior Member, IEEE*, and Chao He

Abstract—In traditional navigation of multi-classification resources, users refine and generalize retrieved resources according to a fixed taxonomy. They often get frustrated when the refined result is empty. Faceted navigation solves this problem by a dynamic taxonomy which eliminates the categories leading to a dead-end. Such an interaction style makes it widely accepted in current e-commerce websits and digital libraries. The retrieved resources at each step during faceted navigation compose a *Resource Space*. Among all the category sets corresponding to the same space, the finest one is the *closed view* of the space. In this paper, we propose a series of novel operators to support more effective refinement and generalization, in order to make faceted navigation more preferable. To achieve fast response of these operators, all the closed views as well as their lattice structure should be mined in advance. Current algorithms for closed itemset mining can be tailored to output all the closed views, but few mines the lattice structure as well, no mention to manage the mining result for the sake of future navigation. Therefore, we devise a novel mining algorithm MIL which not only mines the lattice but also also incrementally builds up an indexing structure, by reusing the intermediate results during mining. Extensive experiments show MIL is significantly superior to current approaches for both mining and navigating the lattice of resource spaces.

Index Terms—Faceted navigation, Resource Space Model, closed itemset mining, interaction style, indexing methods.

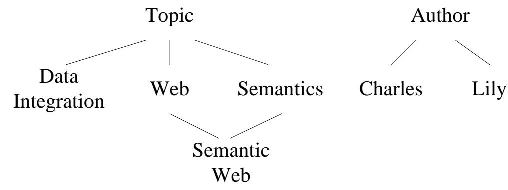
1 INTRODUCTION

CURRENT e-commerce websites and digital libraries often organize and retrieve resources by *facets*. Facet is a hierarchical structure of categories, embodying the categorization semantics of resources from a certain aspect. Faceted search retrieves the resources corresponding to a category set. Fig. 1 depicts a faceted database of six papers classified by the facets *Topic* and *Author*, where each paper is annotated by a set of categories from different facets. Given the query {Data Integration, Lily}, resources {#1, #3, #4, #5} can be retrieved. The result is a *Resource Space* if its cardinality exceeds a pre-defined threshold.

Given massive versatile resources, faceted navigation guides users through an ocean of resource spaces. A resource space can be accessed by a faceted query, or refining/generalizing another one. Modern faceted navigation visualizes only the categories refining current resource space to a non-empty one. Such a tiny innovation makes it widely accepted since the notorious dead-end is avoided during navigation, which rather frustrates users.

Current faceted navigation, however, can hardly mimic humans' awarness of the context of the categorization semantics of retrieved spaces. Typically, it includes:

1. the closed view of a retrieved space, since the given query often cover only partial semantics;
2. the closed view of the sub-, super- or similar spaces of a given space, with a certain ratio of common resources; and,
3. the closed view of the common sub- or super-spaces of multiple spaces,



(a) Facets

Res	Category Set
1	Data Integration, Web, Charles, Lily
2	Semantic Web, Lily
3	Data Integration, Web, Charles, Lily
4	Data Integration, Semantic Web, Lily
5	Data Integration, Semantic Web, Charles, Lily
6	Semantic Web, Charles

(b) Resource Categorization

Fig. 1. A faceted database with two facets and six papers.

where the *closed view* of a space is the finest one among all the category sets corresponding to the space （finest？）. Closed view represents the finest categorization semantics of a space. We propose eight navigation operators to embody these tasks: *redescribe*, *shrink*, *expand*, *maxsim*, *pred*, *succ*, *join* and *meet*.

These operators provide more options for further refinement and generalization. For example, after retrieving the space {#1, #3, #4, #5} by the query {Data Integration, Lily}, *redescribe* operator inform users of its closed view {Web, Data Integration, Lily}. According to the emerging category "Web", the space can be refined more effectively.

All the resource spaces compose a lattice. It can be

- *The authors are with the China Knowledge Grid Research Group, Key Lab of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, 100190, PO Box 2704-28, Beijing, China. E-mail: {zhuge, hc}@kg.ict.ac.cn.*
- *Chao He is also with the Graduate University of the Chinese Academy of Sciences.*

overwhelmingly large because of its nature of NP-completeness [1]. As a result, it is a big challenge to support fast response of the proposed context-aware operators. On one hand, the closed views of all the spaces should be pre-computed. Otherwise, even *redescribe* operator requires a full scan of all the resources in the retrieved space. On the other hand, the lattice structure should be pre-computed, too. Otherwise, a full scan of the whole lattice is required to answer *pred* (or *succ*) operator which retrieves minimal (or maximal) spaces containing (or contained by) a given space, no mention to other more complex operators.

Current approaches in closed itemset mining can be tailored to output all the closed views. To the best of our knowledge, however, only CHARM-L [3] mines their lattice structure simultaneously, which is superior to computing it after mining, as presumed normally. Besides, none provides a facility to support fast response in future navigation.

Accordingly, we devise an efficient algorithm *MIL* which not only mines all the closed views as well as their lattice structure, but also builds up an indexing structure to enable fast navigation.

The construction of *shrinking tree* and s*ubsumption check* are the essence of a top-down algorithm for closed itemset mining, like CHARM (-L) [3] and CLOSET+ [28]. Each node in a shrinking tree represents a space, denoted by its closed view and a shrinking sequence. Each category in shrinking sequence refines current node smaller, and if it is not *subsumed* (i.e. its refined space is not calculated yet), its refined space becomes a child of current node. Subsumption check guarantees the optimality of shrinking tree such that each node represents a unique space. MIL deploys the following techniques to generate the shrinking tree in top-down and depth-first way:

1. For each node in current path, a *categorization trie* is introduced to compress the categorization information of all its resources, which enables fast generation of the node's children;
2. Two specific orders are enforced. Firstly, shrinking sequence is both in the ascending order of space cardinality and in a linear extention. Secondly, the closed view of each node starts with its *reverse seed sequence*;
3. For each node in current path, MIL maintains its *maxsubs*, i.e. the maximal ones among all the visited nodes which are contained by the node; and,
4. For each category in shrinking sequence, MIL preserves whether it is subsumed, the identifier of its refined space, and whether its refined space is a maxsub, as the indexing structure.

Subsumption check can be fastly accomplished merely based on the indexing structure, the two specific orders and maxsubs. When the subtree rooted at a node is visited, its maxsubs become its succesors in the final lattice. In this way, the lattice structure is incrementally built up. After mining, the indexing structure can fulfill fast navigation.

Compared with previous approaches of mining the lattice of all the closed views, extensive experiments confirm that MIL runs faster while consumes less memory, both in orders of magnitude. The indexing structure of MIL is very small, less than 30% of the lattice size and even close to zero in some cases. Based merely on the lattice structure, tens of thousands of access to the preserved lattice in average are required to answer *redescribe* and *meet* operators. By contrast, two to three access in average is enough by utilizing the indexing structure of MIL.

2 Resource Space

Facet is a partially ordered set (poset) of categories such that for categories u and v, $u \le v$ holds if and only if u is equal to or a sub-class of v. Faceted database manages resources by their facets. It can be represented as a tuple $(\mathcal{F}, \mathcal{S}, \mathcal{R})$, where $\mathcal{F}$ is the union of the facets F_1, F_2, …, and F_m such that for any i and j, there is $F_i \cap F_j = \emptyset$, $\mathcal{S}$ is the set of all the resources, and $\mathcal{R}: 2^{\mathcal{F}} \to 2^{\mathcal{S}}$ defines the resource categorization such that $r \in \mathcal{R}(\{c\})$ if and only if resource r is annotated by some category $u \le c$. In default, let $\mathcal{R}(\{\ \}) = \mathcal{S}$.

We presume the faceted database is created in advance and focus on how to make the faceted navigation of its content more effective.

Denotation 1 The result of a faceted query C (a category set) is $\mathcal{R}(C) = \cap_{u \in C} \mathcal{R}(\{u\})$.

To restrict the number of resource spaces, we set a threshold ε for its minimum cardinality.

Definition 1 Resource set $S \subseteq \mathcal{S}$ is a *Resource Space* if and only if there exists a category set $C \subseteq \mathcal{F}$ such that both $S = \mathcal{R}(C)$ and $|S| \ge \varepsilon$ hold. Category set C is a *view* of resource space S if and only if $S = \mathcal{R}(C)$ and for any $u, v \in C$ $(u \ne v)$, u and v are incomparable.

Resource space S is *minimal* if and only if no other space is contained by it, and the ratio $|S|/|\mathcal{S}|$ is called its *support*. The *minimum support* of the above faceted database is $\varepsilon/|\mathcal{S}|$.

The number of all the resource spaces may be far less than that of all the combinations of resources or categories. Take the faceted database in Fig. 1 for example, where $\varepsilon = 2$. In total, there are nine resource spaces, as depicted in Fig. 2, while the number of resource combinations and category combinations are 2^6 (64) and 2^8 (256), respectively.

Definition 2 Let both C_1 and C_2 be a set of mutually-incomparable categories. $C_1 \le C_2$ holds if and only if for any $c_2 \in C_2$, there exists $c_1 \in C_1$ satisfying $c_1 \le c_2$.

The above defines a partial order between views. That view C_1 is less than view C_2 illustrates that C_1 can be got by refining the categories in C_2 and/or introducing new incomparable categories into C_2. Hence C_1 expresses a finer categorization semantics than C_2.

Denotation 2 Given category set C and category c, *floor*(C) $= \{\ u : u \in C$ and $\exists!\ v \in C\ (v < u)\}$.

Lemma 1 Let C_1 and C_2 be two category sets and c_1 and c_2

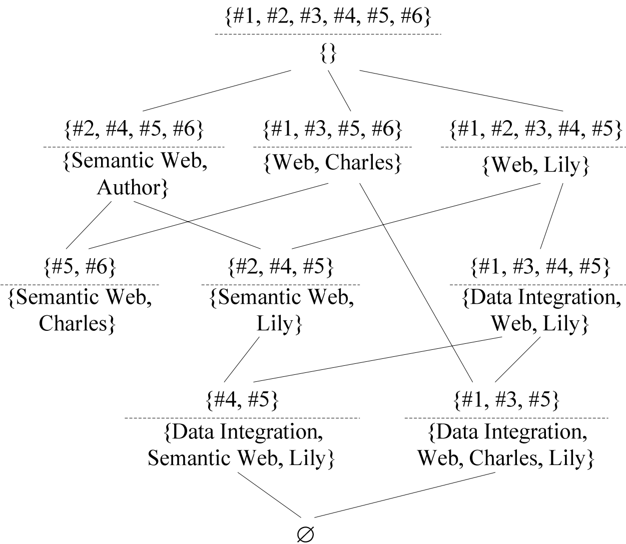


Fig. 2 The lattice of resource spaces in the faceted database in Fig. 1.

be two categories. Then (1) $\mathcal{R}(C_1 \cup C_2) = \mathcal{R}(C_1) \cap \mathcal{R}(C_2)$; (2) $C_1 \subseteq C_2 \Rightarrow \mathcal{R}(C_2) \subseteq \mathcal{R}(C_1)$; (3) $c_1 \leq c_2 \Rightarrow \mathcal{R}(\{c_1\}) \subseteq \mathcal{R}(\{c_2\})$; (4) If both C_1 and C_2 are mutually-incomparable category sets then $C_1 \leq C_2 \Rightarrow \mathcal{R}(C_1) \subseteq \mathcal{R}(C_2)$.

2.1 Closed View

The relation between resource spaces and views is one-to-many. For example, both {Data Integration, Lily} and {Data Integration, Web} are the views of resource space {#1, #3, #4, #5}. People often retrieve the same space by different views which express their own view points of the space's categorization semantics. It is helpful and desired to present current user with other people's views to uncover semantics. Since resource space may have many views which often overlap in semantics, a simple way is required to present the semantics in other views.

Definition 3 View V is closed if and only if there does not exist another view U such that $\mathcal{R}(U) = \mathcal{R}(V)$ and $U < V$.

Closed view expresses the finest categorization semantics of its resource space, which is similar to closed frequent itemset in data mining [26]. Theorem 1 illustrates each space has exactly one finest categorization semantics. For example, space {#1, #3, #4, #5} has exactly one closed view {Data Integration, Web, Lily}. Thus it is enough to inform users of closed views during faceted navigation.

Lemma 2 Let S and S' be two resource spaces such that $S \subseteq S'$. For any view V' of S', $V \leq V'$ holds, where V is the closed view of S.

Theorem 1 Resource space has exactly one closed view.

Proof Let $\mathcal{V}$ be the set of all the views of resource space S. Assume S has no closed view. Hence for any view $V \in \mathcal{V}$, there exists $V' \in \mathcal{V}$ such that $V' < V$. Thus we can get a linear order of $\mathcal{V}$ such that $V_k < V_{k-1} < V_{k-2} < \ldots < V_1$ where $k = |\mathcal{V}|$. Since V_k is not closed, there exists $V_i < V_k$ where $V_i \in \mathcal{V}$. It contradicts that $V_k < V_i$. Therefore, S has at least one closed view.

Let V_1 and V_2 be two closed views of S. By Lemma 2, both $V_1 \leq V_2$ and $V_2 \leq V_1$ hold. Hence $V_1 = V_2$. Therefore, S has exactly one closed view.

□

2.2 Space Lattice

All the resource spaces in a faceted database compose a lattice with respect to the partial order of set containment. As a result, given any number of spaces, there is exactly one minimal space as their common superset, and exactly one maximal space as their common subset.

Theorem 2 Let $\mathcal{Q}$ be the set of all the resource spaces in a faceted database. Then $(\mathcal{Q} \cup \{\emptyset\}, \subseteq)$ is a bounded complete lattice.

Proof Let m be the cardinality threshold for a resource space, and $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ be the set of any number of resource spaces.

Denote $L = \cap_{i=1..k} S_i$. By Lemma 1, $L = \mathcal{R}(floor(\cup_{i=1..k} V_i))$ holds where V_i is the closed view of S_i. If $|L| < m$, then $\emptyset$ is the only lower bound of $\mathcal{S}$. Otherwise, L is a resource space. For any lower bound L' of $\mathcal{S}$, there is $L' \subseteq L$. Therefore, the greatest lower bound of $\mathcal{S}$ exists.

The resource space of all the resources is an upper bound of $\mathcal{S}$. Let $\{U_1, U_2, \ldots, U_t\}$ be all the minimal upper bounds of $\mathcal{S}$. Denote $U = \cap_{i=1..t} U_i$. There is $U = \mathcal{R}(floor(\cup_{i=1..t} U_i))$. Since $|U| \geq |\cup_{i=1..k} S_i| \geq m$, U is a resource space. Therefore, the lowest upper bound of $\mathcal{S}$ exists.

Accordingly, $(\mathcal{Q} \cup \{\emptyset\}, \subseteq)$ is a complete lattice. Since all the resource space of all the resources is the largest and $\emptyset$ is the smallest, the lattice is bounded as well.

□

We call the above lattice a *space lattice*. Fig. 2 depicts the space lattice of the faceted database in Fig. 1, which illustrates the resources and the closed view of each space.

3 Context-Aware Faceted Navigation

3.1 Navigation Operators

Space lattice can be very huge in real-world applications with massive versatile resources. Traditional faceted navigation guides users through it by the following operators in primary: (1) *search*(C), to retrieve the space $\mathcal{R}(C)$ given the query of category set C; (2) *generalize*(S, c), to generalize current space S by category c; and (3) *refine*(S, c), to refine current resource space S by category c. There are two shortcomings. First and foremost, the refinement often leads to an empty result, which frustrates user to a large extent. Secondly, users are oblivious of that a space remains the same after refinement or generalization. Correspondingly, current systems provide a dynamic taxonomy to avoid dead-end during navigation [7] [9]. They also illustrate the number of resources corresponding to further refinement or generalization.

This paper complements current faceted navigation by making users aware of the context of browsed spaces. Such context semantics are embodied in eight novel operators as follows:

1. *redescribe*(C) — to find out the closed view of the resource space corresponding to category set C;
2. *shrink*(S,δ) — to retrieve the closed views of all the minimal spaces among { S' : S' is a space such that $S' \subseteq S$ and $dist(S, S') < \delta$};

3. *expand*(S,δ) — to retrieve the closed views of all the maximal spaces among { S' : S' is a space such that $S' \supseteq S$ and $dist(S, S') < \delta$};
4. *maxsim*(S,δ) — to retrieve the closed views of all the maximal spaces among { S' : S' is a space similar to S such that $dist(S, S') < \delta$ } (Two spaces are *similar* if and only neither of them contains the other and their intersection is a space as well);
5. *pred*(S) — to retrieve the closed views of all the minimal spaces after generalizing S;
6. *succ*(S) — to retrieve the closed views of all the maximal spaces after refining S;
7. *join*(S_1, S_2, …, S_k) — to retrieve the minimal space containing all S_i; and,
8. *meet*(S_1, S_2, …, S_k) — to retrieve the maximal space contained by each S_i.

Denotation 3 The jaccard distance between resource spaces S_1 and S_2 is $dist(S_1, S_2) = 1 - |S_1 \cap S_2| / |S_1 \cup S_2|$.

redescribe operator expose the finest categorization semantics so that users have more options to carry out refinement and generalization. Current faceted navigation's style of step-by-step refinement and generalization often results in an overshooting such that the result space is too small or big. *shrink* and *expand* operators encounters the problem by allowing users configuring the ratio of decreasing and increasing the size of current space, respectively. *maxsim* operator makes it possible of retrieving similar spaces in terms of common resources. *shrink*, *expand* and *maxsim* are all based on the jaccard distance, a metric distance satisfying the triangle inequality: $dist(S_1, S_2) + dist(S_2, S_3) \geq dist(S_1, S_3)$, given spaces S_1, S_2 and S_3. *pred* (*succ*) retrieves the predecessors (successors) of a space in space lattice. It is the first time for users to generalize and refine multiple spaces simultaneously by *join* and *meet* operators, respectively. Based on lattice property, *join* (*meet*) retrieves the space as the lowest upper bound (greatest lower bound) of multiple spaces, which, for the first time, allows users to generalize (refine) multiple spaces simultaneously.

Based on the proposed context-aware navigators, a user interface with a novel interaction style can be designed to make users' navigation more effective and even be a process of learning.

3.2 Navigation Algorithms

Fast response is critical to a navigation system. The closed views of all the resource spaces should be computed beforehand. Otherwise, whenever a space is accessed, all its resources need visited to determined its closed view, which is rather time-consuming in the setting of massive resources. Besides, the lattice structure among all the spaces should be pre-computed, too. According to [1], it is NP-complete to decide the size of a space lattice, which can easily exceed tens of thousands even in high threshold of space cardinality, as shown in our experimental evaluation. Without the lattice structure, even *pred*/*succ* operators require a scan of the whole lattice, no mention to the others.

Accordingly, we need to pre-compute **the lattice structure of all the resource space as well as their closed views and cardinality**, as discussed in Section 4.

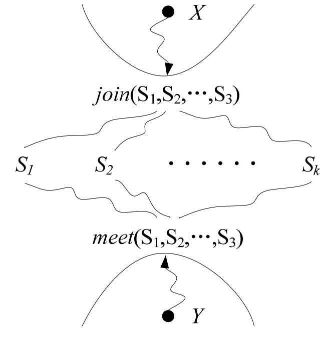


Fig. 3 Bootstrapping algorithms for join and meet operator.

Now we discuss the navigation algorithms merely based on the mined space lattice.

pred and *succ* can be directly ascertained.

shrink(S,δ) (or *expand*(S,δ)) can be realized by flooding downward (upward) from the node S in the lattice, until the nodes whose successors' (or predecessors') distances to S are all larger than δ, which are then returned as the final result.

Lemma 3 Let V_1 and V_2 be the closed views of spaces S_1 and S_2 respectively. Then $V_1 \leq V_2 \Leftrightarrow S_1 \subseteq S_2$.

join(S_1, S_2, …, S_k), i.e. the lowest upper bound, can be found out by the following bootstrapping algorithm. Start from the root node which is an upper bound of all S_i. If no predecessor of current node is an upper bound of all S_i (which can be easily decide by Lemma 3), then it is the join result; otherwise, move up into any of its predecessors which are an upper bound of all S_i. Fig. 3 illustrates the process.

Lemma 4 If $S' \in maxsim(S,\delta)$, then both $dist(X, S) < \delta$ and $|S'| \leq (2-\delta)/(1-\delta)|X| - |S|$ hold, where $X = S \cap S'$.

maxsim(S,δ) can be partitioned by their intersection with S. By Lemma 4, it is equal to max $(\cup_{X \in I} [X])$, where $I = \{X : X \subset S \text{ and } dist(X, S) < \delta\}$ and $[X] = \max\{S' : X = S \cap S' \text{ and } |S'| \leq (2-\delta)/(1-\delta)|X| - |S|\}$. Thus we devise the following efficient algorithm. It floods down from S until the nodes whose distance to S is no less than δ; for each encountered node X in I, flood upward until no predecessor of current node Y meets with S at X, which can be decided by Lemma 5.

Lemma 5 Given the spaces S, X and Y such that $X \subset S$ and $X \subset Y$. X is the meet of S and Y if and only if for any predecessor P of X such that $P \subset S$, $P \subset Y$ does not hold.

meet(S_1, S_2, …, S_k) can be realized in a symmetric way as *join*, as depicted in Fig. 3. However, the starting node, a lower bound, is hard to ascertain. We devise two different *meet* algorithms: *meetMinSpace* and *meetFlood*. Since the meet result is not null if and only if there exists a minimal space as their lower bound, *meetMinSpace* scans the preserved minimal spaces to determine a starting node, while *meetFlood* floods down from any S_i to find out one.

redescribe(C) is not empty if and only if there is a space

contained by or equal to $\mathcal{R}(C)$. Start from such a space and move up. If no predecessor of current space is contained by or equal to $\mathcal{R}(C)$ (decided by Lemma 2), return current space as the result; Otherwise, move up into any of its predecessors which are contained by or equal to $\mathcal{R}(C)$. We devise two algorithms *rdscMinSpace* and *rdscFlood* for *redescribe* operator which find out the starting node via scanning the minimal spaces and flooding from the root node, respectively.

The proposed algorithms for *pred*, *succ*, *shrink*, *expand*, *maxsim* and *join* scales well with the lattice size, while those for *meet* and *redescribe* are not. In worst case that the result is empty, the complexity of both *meetMinSpace* and *rdscMinSpace* is $O(M)$ where M is the number of all the minimal spaces, the complexity of *meetFlood* is $O(D)$ where D is the number of the descendants of the starting node, and the complexity of *rdscFlood* is $O(N)$ where N is the lattice size.

Lemma 6 $meet(S_1, S_2, \ldots, S_k) = redescribe(floor(\cup_{i=1..k} V_i))$ where V_i is the closed view of space S_i.

Therefore, we devise an indexing structure to support fast *redescribe*, as discussed in section 4. Experimental study shows the accessed number of nodes is only 2 in average. By Lemma 6, it supports fast *meet* operator as well.

4 MIL: Mining and Indexing Space Lattice

Although closed itemset mining algorithms can be tailored to output all the closed views, few of them generate the lattice structure at the same time, no mention to provide such a facility that supports fast navigation. We propose an efficient algorithm *MIL* which not only mine the lattice of all the closed views but also reuse the intermediate result during mining as an indexing structure for future navigation.

4.1 The Shrinking Tree

The process of recent algorithms for closed itemset mining, like CLOSET+ [28] and CHARM (-L) [3], can been seen as a depth-first traversal of a *shrinking tree*. Let ($\mathcal{F}$, $\mathcal{S}$, $\mathcal{R}$) be the faceted database. Each node in a shrinking tree represents a unique resource space, denoted by its closed view and a shrinking sequence, a sequence of categories each of which refines the node into a smaller space. The root node represents the space $\mathcal{S}$, whose shrinking sequence is the list of all the categories each of which corresponds to a space smaller than $\mathcal{S}$. For current node X with closed view V and shrinking sequence L, if category c in L is not *subsumed* (i.e. the refinement of X by c is not visited before), then c corresponds to a child Y of X. The shrinking sequence of Y consists of those category after c in L which refines Y into a smaller space. In this way, all the spaces are generated exactly once.

The following algorithm describes the above process which calculates all the closed views correctly.

Algorithm 1 calcShrTree ($\mathcal{F}$, $\mathcal{S}$, $\mathcal{R}$).
1) Let V be an empty set and L be an empty sequence;
2) **foreach** (category c in $\mathcal{F}$) {
3) **if** ($|\mathcal{R}(\{c\})| == |\mathcal{S}|$) add c in V;
4) **else if** ($\varepsilon \le |\mathcal{R}(\{c\})| < |\mathcal{S}|$) append c at the end of L;
5) }
6) $V = floor(V)$; permute L in any linear order;
7) depthFirst(V, L);

Subroutine depthFirst(V, L)
1) **foreach** (category c in L) {
2) Let the set $V_1 = V \cup \{c\}$ and L_1 be an empty sequence;
3) **if** (the closed view of $\mathcal{R}(V_1)$ is outputted) **continue**;
4) **foreach** (category u after c in L) {
5) **Let** $S = \mathcal{R}(V \cup \{u\})$;
6) **if** ($\mathcal{R}(V_1) \subseteq S$) {
7) $V_1 = V_1 \cup \{u\}$;
8) **if** ($\mathcal{R}(V_1) == S$) remove u from L;
9) }
10) **else if** ($|\mathcal{R}(V_1) \cap S| \ge \varepsilon$) {
11) append u at the end of L_1;
12) **if** ($\mathcal{R}(V_1) \supset S$) remove u from L;
13) }
14) }
15) $V_1 = floor(V_1)$; permute L_1 in any linear order;
16) depthFirst(V_1, L_1);
17) }
18) Output V;

Different permutation of shrinking sequences lead to different shrinking trees. MIL permutes shrinking sequence, say $c_1\ c_2 \ldots c_k$, as follows: (1) for any $i < j$, the cardinality of the shrunk space by c_i is no larger than by c_j; and, (2) $c_1\ c_2 \ldots c_k$ is a *linear extension* of the poset $\{c_1, c_2, \ldots, c_k\}$ such that $c_i < c_j$ implies $i < j$.

Besides, MIL enforces a specific permutation of closed view. Let Y be the child of node X whose closed view and shrinking sequence is V and L respectively. Let $c_1c_2 \ldots c_k$ be all the categories in L which shrinks X into Y and in the same order as in L. The the closed view of Y is $floor(c_1 \bullet V \bullet c_2 \bullet c_3 \bullet \ldots \bullet c_k)$, where *floor* is order-preserving.

Fig. 4 depicts the shrinking tree of the faceted database of Fig. 1 generated by MIL. For each node, the categories in circle in the left side compose its closed view, while the right side is its shrinking sequence. Take category e in the

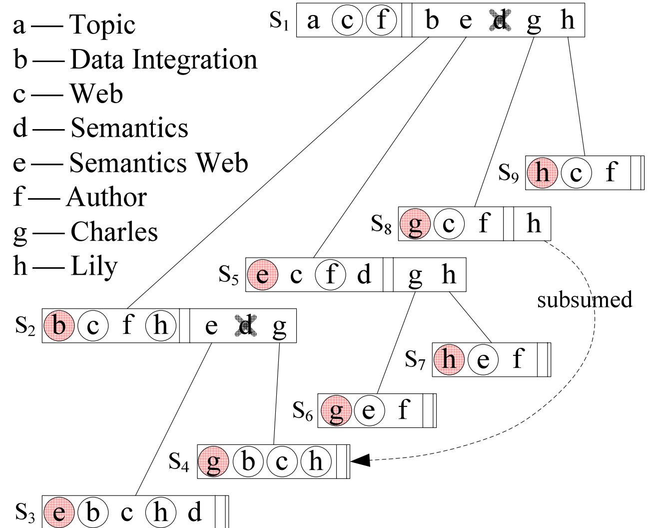


Fig. 4 The shrinking tree generated by MIL.

shrinking sequence of the root node S_1 for example. Since it is not subsumed, it corresponds to a child S_5 of S_1. The closed view of S_5 is determined in three steps. First, initialize category sequence V be *ecf*, the concatenation of e and the closed view *cf* of S_1. Then, append at the end of V with all the following categories $\{d\}$ of e whose shrunk spaces are equal to or contain S_5. Finally, the closed view of S_5 consists of the sub-sequence of the minimal categories in V.

CHARM (-L) and CLOSET+ adopts different approaches in generating shrinking sequences and doing subsumption check. Both maintain an additional index of outputted closed views to assist subsumption check, which becomes useless for post-mining navigation. By contrast, MIL reuses the intermediate results during mining as an indexing structure. As a result, lattice structure can be incrementally built up. Moreover, the indexing structure of MIL can support fast *redescribe* and *meet* operators for context-aware faceted navigation.

4.2 Data Structure and Indexing Structure

In the traversed part of the shrinking tree, we call all the outputted nodes as *index nodes*, and the other as *engaged nodes* which compose current path in depth-first traversal. An engaged node resides in memory, represented by (*id*, *pid*, *depth*, *card*, *cview*, L^+, L^-, *M*, *T*, *preds*). *id* is its identifier which is equal to the order of its birth. *pid* is the identifier of its parent. *depth* is its depth in the shrinking tree, where the root's depth is 0. *card* and *cview* are the cardinality and the closed view of the resource space it represents, respectively. L^+ is the visited part of its shrinking sequence, while L^- is the remaining part. The first element in L is under examination. M is the set of all its non-shrunk maxsubs. T is a patricia trie of the categorization information of its resources, called *categorization trie*. *preds* keeps its predecessors in the lattice.

Definition 4 In the traversed part of the shrinking tree, node Y is a *maxsub* of node X if and only if $Y \subset X$ and there does not exist another node Z satisfying $Y \subset Z \subset X$, where $Y \subset X$ is an abbreviation of that the space of Y is contained by that of X. Moreover, if there exists a visited category c in $X.L^+$ such that its shrunk space is equal to Y, then Y is called a *non-shrunk maxsub* of X; otherwise, Y is a *shrunk maxsub* of X.

The visited categories in shrinking sequence can be partitioned according to their shrunk spaces. Each class of categories under the partition, the identifier of their shrunk space, and a flag indicating whether the shrunk space is current node's maxsub, are called a *segment*, represented by (*cats*, *nid*, *bMax*). L^+ is a list of such segments.

L^- is denoted as a sequence of tuples (*cat*, *card*, *tnodes*), where *cat* is a remaining category in current node's shrinking sequence, *card* is the cardinality of its shrunk space, and *tnodes* is a set of its associated nodes in current node's categorization trie. Categorization trie can generate the closed view and the shrinking sequence of a node in the shrinking tree efficiently, as discussed in section 4.4.

Whether a visited category in shrinking sequence is subsumed or not can be deduced from node identifier.

Lemma 7 Let X be a shrinking tree node. For any segment $Q \in X.L^+$, $Q.cats$ were subsumed if and only if $Q.nid < X.id$.

After all the categories in the shrinking sequence of an engaged node are visited, it turns into an index node (*id*, *pid*, *depth*, *card*, *cview*, L^+, *M*, *preds*), supporting the following mining process as well as context-aware faceted navigation after mining.

4.3 Subsumption Check and Lattice Construction

Now we briefly introduce MIL depicted in Algorithm 2, where $\bullet$ denotes the concatenation of two sequences. For each category c in the shrinking sequence of current node X, check whether c is subsumed firstly (line 5~29); if subsumed, remove its following categories from the shrinking sequence of X whose shrunk spaces are the same as that of c (line 30~39); otherwise, c would correspond to a new child of X. Line 40~56 initializes the non-shrunk maxsubs of the new child. Then, remove the following categories of c from the shrinking sequence of X whose shrunk spaces are the same as c (line 56). Line 57, 59 and 60 finally determines the new child.

Algorithm 2 MIL

```
1) Calculate the root node X (1, -1, 0, card, V, ∅, L, ∅, T, ∅);
2) depthFirst(X);
```

Subroutine depthFirst(**EngagedNode** *X*)

```
1)  int count = 1;
2)  while (X.L- is not empty) {
3)      get and remove the first tuple P from X.L-;
4)      Set mxs = { }; Segment seg = (P.cat, -1, true);
5)      foreach (id in X.M) { // subsumption check
6)          IndexNode Y = getIndexNode(id);
7)          if ({P.cat} ≥ Y.cview AND P.card == Y.card) {
8)              remove id from X.M;
9)              seg.nid = id;
10)             break;
11)         }
12)         else if ({P.cat} ≥ Y.cview AND P.card >Y.card) {
13)             remove id from X.M;
14)             add id into mxs;
15)             break;
16)         }
17)         else {
18)             IndexNode Z = refineSpaceUp(Y, P.cat, X);
19)             if (Z is not null) {
20)                 if (P.card == Z.card) {
21)                     seg.nid = Z.id;
22)                     seg.bMax = false;
23)                     break;
24)                 }
25)                 else
26)                     add Z.id into mxs;
27)             }
28)         }
29)     }
30)     if (seg.nid != -1) { // is subsumed
31)         IndexNode Y = getIndexNode(seg.nid);
32)         foreach (P' in X.L-)
33)             if ({P'.cat} ≥ Y.cview AND P'.card == Y.card) {
34)                 seg.cats = seg.cats • P'.cat;
35)                 remove P' from X.L-;
36)             }
```

```
37)         append seg to the end of X.L+;
38)         continue;
39)     }
40)     foreach (Q in X.L+) { // initialize maxsub
41)         if (! Q.bMax)   continue;
42)         IndexNode Y = getIndexNode(Q.nid);
43)         if ({P.cat} ≥ Y.cview) {
44)             Q.bMax = false;
45)             add Y.id into mxs;
46)         }
47)         else {
48)             if (Y.id < X.id) {
49)                 IndexNode Z=refineSpaceUp(Y, P.cat, X);
50)                 if (Z is not null)   add Z.id into mxs;
51)             }
52)             else if (∃ Q′ ∈ Y.L+ such that P.cat ∈ Q′.cats)
53)                 add Q′.nid into mxs;
54)         }
55)     }
56)     mxs = max(mxs);
57)     (T, E, C, L) = projectCatTrie(X, P);
58)     remove all the categories in E from X.L-;
59)     Sequence V = floor (P.cat • X.cview • E•C);
60)     EngagedNode child = (X.id+count, X.id, X.depth+1,
          P.card, V, ∅, L, mxs, T, ∅);
61)     seg.cats = seg.cats • E; seg.nid = child.id;
62)     append seg to the end of X.L+;
63)     count += depthFirst(child);
64) }
65) foreach (id ∈ X.M∪{seg.nid : seg ∈ X.L+ and seg.bMax=true })
66)     set X be a predecessor of the node whose identifier is id;
67) output X as an index node;
68) return count;
```

Let X be current node and P be the tuple in $X.L^-$ currently under subsumption check.

Theorem 3 P is subsumed if and only if its shrunk space is contained by or equal to some node in $X.M$.

Proof Let N be all the current non-shrunk maxsubs of X. Obviously, P is subsumed if and only if its shrunk space is contained by or equal to some maxsub of X. Since shrinking sequence is in ascending order of space cardinality, P is not contained by any of N. No matter a visited category is subsumed or not, all its following categories with the same shrunk space are removed, as done in line 35 and 58. As a result, P is not equal to any of N. Therefore, P is subsumed if and only if its shrunk space is contained by or equal to some node in $X.M$. □

Line 5~29 accomepletes the subsumption check according to the above theorem. Let S be the shrunk space of P and Y be any node in $X.M$. It can be easily verified that $S \supseteq Y$ holds if and only if $\{P.cat\} \geq Y.cview$. In case of $S = Y$ (line 7), P is subsumed and its shrunk node is Y, so Y becomes a shrunk maxsub of X. In case of $S \supset Y$ (line 12), P is not subsumed and would correspond to a new child of X, so Y is removed from $X.M$. Otherwise, P is subsumed if and only if there is $|Z|=|S|$ (line 20), where $Z = Y \cap S$ which can be retrieved by *refineSpaceUp*(Y, $P.cat$, X).

Suppose P be not subsumed. It corresponds to a new child of X, denoted by N. The initial non-shrunk maxsubs of N is determined by the following lemma.

Lemma 8 Let M_b be all the maxsubs of X right before N is generated. Right after the birth of N, $N.M = \max\{ S \cap N : S \in M_b\}$, $X.M = M_b - \{S : S \in M_b \text{ and } S \subset N\}$, and the maxsubs of any other visited node remain the same.

During the subsumption check of P, *mxs* keeps the intersection of the shrunk space S of P and each non-shrunk maxsub of X. After that, the intersection of S and each shrunk maxsub Y of X is calculated (line 40~55). The case of $S \supset Y$ can be easily decided (line 43). If Y is a child of X, $S \cap Y$ is a resource space if and only if $P.cat$ is in the shrinking sequence of Y (line 52). Otherwise, $S \cap Y$ can be found out by *refineSpaceUp*(Y, $P.cat$, X) (line 49). Finally, the maximal ones in *mxs* compose the initial non-shrunk maxsubs of the new child N (line 56). When the shrinking tree is traversed, the maxsubs of each node are exactly its successors in the space lattice, which are preserved in the indexing structure.

projectCatTrie(X, P) do projection on the categorization trie of X to get the categorization trie T of the new child P (line 57). During the process, the following categories of P in the shrinking sequence of X, whose shrunk spaces are equal to and contain that of P, are returned as E and C, respectively. Categories in both E and C are in the same order as in the shrinking sequence of X. Besides, the shrinking sequence L of the new child is acertained. Then, E is removed from $X.L^-$ (line 58) and the closed view of the new child is calculated (line 59). At last, the new child

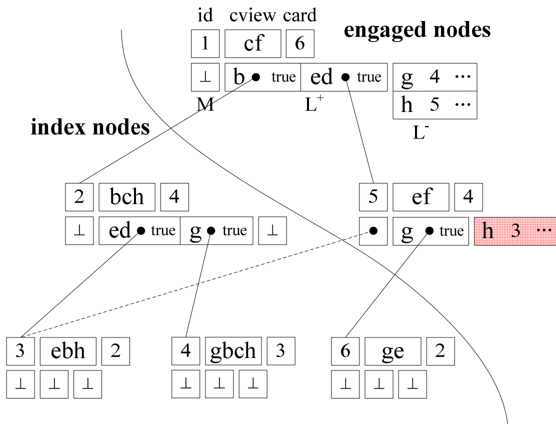


Fig. 5 A running example of MIL when current node is S_5 and h is the category in its shrinking sequence currently under examination.

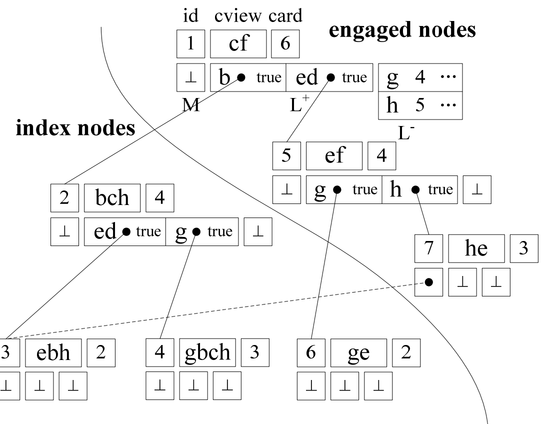


Fig. 6 After the generation of the new child S_7.

id	max. category set
1	a, b, c, f, g, h
2	a, c, d, e, f, h
3	a, b, c, f, g, h
4	a, b, c, d, e, f, h
5	a, b, c, d, e, f, g, h
6	a, c, d, e, f, g

(a)

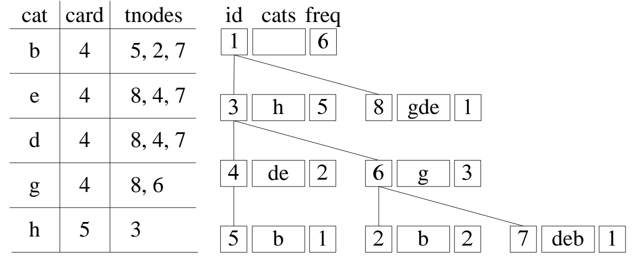
id	card
a	6
b	4
c	6
d	4
e	4
f	6
g	4
h	5

(b)

id	cond. Seq.
1	hgb
2	hde
3	hgb
4	hdeb
5	hgdeb
6	gde

(c)

Fig. 8 The first scan generates (a) the maximal category set of each resource, and (b) the frequency of each category. The second scan generates the conditional sequence of each resource.



cat	card	tnodes
b	4	5, 2, 7
e	4	8, 4, 7
d	4	8, 4, 7
g	4	8, 6
h	5	3

(a) shrinking sequence (b) categorization tree

Fig. 7 The second scan generates the shrinking sequence and categorization trie of the root node.

is given birth to (line 60).

Then, the visited categories in the above process, i.e. the concatenation of *P.cat* and *E*, become a new segment of $X.L^+$, whose shrunk space is the new child and *bMax* flag is already determined during subsumption check and maxsub initialization.

Finally, when the subtree rooted at *X* is visited, its maxsubs are exactly its successors in the final lattice. Hence add *X* as a predecessor of each maxsub (line 65~66). After that, output *X* as an index node (line 67).

Fig. 5 depicts a running example of MIL when current node is S_5 and *h* is the category in its shrinking sequence currently under examination. Index nodes are in the bottom-left while engaged nodes are in the upper-right. The non-shrunk maxsubs of S_5 are $\{S_3\}$. The shrunk maxsubs of S_5 are $\{S_6\}$. Since $\{h\}$ is greater than the closed view of S_3 and the cardinalityof the shrunk space by *h* is 3 which is larger than that 2 of S_3, *h* is not subsumed and corresponds to a new child S_7 with identifier 7. Since *h* is not in the shrinking sequence of S_5, the intersection between S_7 and S_6 is not a resource space. Therefore, the initial non-shrunk maxsubs of S_7 are $\{S_3\}$. The non-shrunk maxsubs of S_5 become empty. The maxsub nodes of any other node remain the same. Fig. 6 depicts the result.

4.4 Function *projectCatTrie*

Let *X* be a node in the shrinking tree whose resource space is *S* and shrinking sequence is *L*. For each resource *r* in *S*, its *conditional* (*categorization*) *sequence with respect to X* is a permutation of $\{ c : c \in L \text{ and } r \in \mathcal{R}(\{c\}) \}$ in the reverse order as in *L*. The *categorization trie* of *X* organizes the conditional sequences of all its resources in a patricia trie. For each trie node, it counts the number of resources whose conditional sequences start with the category sequence of the path from the root to the node. Unlike patricia trie, it does not keep resources in leaves, because the categorization trie of *X* aims at counting the cardinality of any refinement of *X* by one or more categories in *L*.

First of all, we explain how to construct the root node $(1, -1, 0, card, V, \varnothing, L, \varnothing, T)$ in two scans of a faceted database. Take Fig. 1 for example. The first scan generates the maximal set of categories each resource can be categorized into, as described in Fig. 8 (a). The number of all the resources is calculate so that *card* = 6. It also counts the frequency of each category, as described in Fig. 8 (b). Hence the closed view of the root node is *floor*(*acf*) = *cf* and its shrinking sequence is *bedgh*. After that, the second scan generates the conditional sequence of each resource and adds it into the categorization trie *T*. The final *L* and *T* are described in Fig. 7 (a) and (b) respectively.

Algorithm 3 depicts the function *projectCatTrie*, which projects the categorization trie of the enaged node *X* by the tuple *P* in its shrinking sequence currently under examination. It returns the categorization trie *T* and the shrinking sequence *L* of the new child of *X* corresponding to *P*, the following categories *E* of *P* whose shrunk spaces are the same as that of *P*, and the following categories *C* of *P* whose shrunk spaces contain that of *P*. Both *E* and *C* are in the same order as in the shrinking sequence of *X*.

Algorithm 3 projectCatTrie(**EngagedNode** *X*, **Tuple** *P*)

1) **Sequence** $E = \varnothing, C = \varnothing, L = \varnothing$;
2) **foreach** (category *c* in $X.L^-$) initialize $f(c) = 0$;
3) **foreach** (**CatTrieNode** *n* in *P.tnodes*) {
4) **Set** *prefix* = (the categories before *P.cat* in *n.cats*) $\cup$ ($\cup_{p \text{ is an ancestor of } n}$ *p.cats*);
5) **foreach** ($c \in prefix$ and $c \in X.L^-$) $f(c)$ += *n.freq*;
6) }
7) **foreach** (category *c* in $X.L^-$)
8) **if** ($f(c)$ == *P.card*) $E = E \bullet c$;
9) **else if** ($f(c) > P.card$) $C = C \bullet c$;
10) **else if** ($f(c) \geq \varepsilon$) append $(c, f(c), \varnothing)$ to the end of *L*;
11) permute *L* both in ascending order of *f* value and in linear extension;
12) **CatTrie** $T = \varnothing$;
13) **foreach** (**CatTrieNode** *n* in *P.tnodes*) {
14) Let *prefix* = (the categories before *P.cat* in *n.cats*) $\cup$ ($\cup_{p \text{ is an ancestor of } n}$ *p.cats*);
15) **foreach** ($c \in prefix$ and $c \notin L$) remove *c* from *prefix*;
16) permute *prefix* in reverse order as in *L*;
17) insert *prefix* in *T*;
18) }
19) **foreach**(tuple *Q* in *L*) $Q.tnodes = \{m : m \in T \text{ and } Q.cat \in m.cats\}$;
20) **return** (*T*, *E*, *C*, *L*);

First of all, for each following tuple of *P* in the shrinking sequence of *X*, the cardinality of the intersection of their shrunk spaces is calculated (line 3~6), which is the sum of the frequency of the paths containing both tuples' categories. As a result, *E*, *C* and *L* can be decided (line 7~11). After that, the categorization trie of the new child is built up (line 12~18), the conditional sequences of whose resources are determined according to *L*. Finally, associate each category in the new child's shrinking se-

quence with all the nodes in the new child's categorization trie which contain the category (line 19). Only two scans of the categorization trie is required.

4.5 Function *refineSpaceUp* and *refineSpaceDown*

Now we illustrate how to utilize the indexing structure and the specific orders of both shrinking sequence and closed view to accomplish *refineSpaceUp* and *refineSpaceDown*. No additional facility is required.

Definition 5 Let Y be the child of X_k and u be the first category in the shrinking sequence of X_k which refines X_k into Y. Then u is the *seed* of Y. Root node has no seed. Let the path from root node X_0 to Y be $X_0X_1 \ldots X_kY$ whose seeds are $\varnothing$, c_1, …, c_k and u respectively. Then $c_1c_2\ldots c_ku$ is called the *seed sequence* of Y, and $c_{i+1}c_{i+2}\ldots c_ku$ is called the *seed sequence* from X_i to Y.

As shown in Fig. 4, the seed of a node is always at the first place in its closed view, depicted in red circles. Besides, the reverse seed sequence of a node is always in the front of its closed view.

Denotation 4 The refined result of resource space S by category set C is $\mathcal{R}(C\,|\,S)$.

Theorem 4 Let $c_1c_2...c_n$ be the seed sequence of node X. Then (1) $\{c_1, c_2, \ldots, c_n\}$ is a view of X, and (2) $c_nc_{n-2}...c_1$ is the prefix of $X.cview$.

Proof There are two observations: (1) a node's seed is always in its closed view; and, (2) a node's seed is contained in the shrinking sequence of any of its ancestors.

Let $Y_0Y_1 \ldots Y_n$ be the path from the root (Y_0) to X (Y_n), where the shrinking sequence of Y_i is L_i.

For any $i, j\in [1, n]$ where $i < j$, c_i and c_j are incomparable. It can be proved as follows. Since c_j refines Y_{j-1} smaller and there is $Y_{j-1}\subseteq Y_i$, c_j refines Y_i smaller. Based one the first observation, $c_j \geq c_i$ does not hold. Assume $c_j < c_i$. Since both of them are in L_{i-1} and L_{i-1} is in linear order, c_j is before c_i in L_{i-1}. As a result, c_j is not in L_i. This contradicts the second observation. $c_j < c_i$ does not hold. Therefore, c_i and c_j are incomparable.

Since $\mathcal{R}(\{c_1, c_2, \ldots, c_n\}) = X$, $\{c_1, c_2, \ldots, c_n\}$ is a view of X.

Assume $c_i \notin X.cview$. Then there exists $u \in X.cview$ such that $u < c_i$. Based on the first observation, $u \notin Y_i.cview$ holds. Hence u is in the shrinking sequence of Y_i. So both u and c_i are in L_{i-1}. Since $u < c_i$, u is before c_i in L_{i-1}. Hence u is not in L_i. This is a contradiction. Therefore, $\{c_1, c_2, \ldots, c_n\} \subseteq X.cview$.

As a result, $c_nc_{n-2}...c_1$ is the prefix of $X.cview$.

□

Both *refineSpaceUp*(Y, C, X) and *refineSpaceDown*(Y, C, X) intend to find out the index node $\mathcal{R}(C\,|\,Y)$, presuming:

1. X is an engaged or index node in the shrinking tree;
2. The category set C is contained by the shrinking sequence of X; and,
3. Y is an index node such that $Y \subset X$; If c is the first category in the shrinking sequence L of X which refines X into Y, then all the categories of C are behind c in L.

In case that Y is a child of X, *refineSpaceDown* is called. Otherwise, *refineSpaceUp* is called. They refine Y by moving upward and downward in the shrinking tree, respectively.

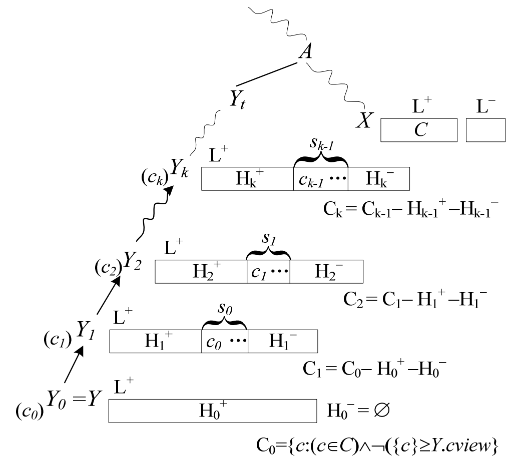


Fig. 9 refineSpaceUp(Y, C, X)

4.5.1 *refineSpaceUp*(Y, C, X)

As described in Fig. 10, A is the least common ancestor of X and Y in the shrinking tree, $Y_0Y_1 \cdots Y_t A$ be the upward path from Y (=Y_0) to A. For any Y_i, c_i is its seed, and s_i is the segment in $Y_{i+1}.L^+$ which contains c_i. C_0 is the reduction of C by removing the categories which can not refine Y smaller, such that $C_0 = \{ c : c \in C$ and ($\{c\} \geq Y.cview$ does not hold) $\}$.

Theorem 5 The result of *refineSpaceUp*(Y, C, X) is a resource space only if C_0 is contained by the shrinking sequence of B.

Proof Assume there exist a category c in C_0 which is not in the shrinking sequence of B. Since c is in the shrinking sequence of X, it must be in the shrinking sequence of A and after b which is the seed of B. There are only two reasons for the absence of c in the shrinking sequence of B: $B = \mathcal{R}(\{c\}\,|\,B)$ or $|\mathcal{R}(\{c\}\,|\,B)| < \varepsilon$. If $B = \mathcal{R}(\{c\}\,|\,B)$, then $\{c\} \geq B.cview$. Since $B \supseteq Y$, $B.cview \geq Y.cview$ holds. Hence $\{c\} \geq Y.cview$. This contradicts the definition of C_0. If $|\mathcal{R}(\{c\}\,|\,B)| < \varepsilon$, then $|\mathcal{R}(C\,|\,Y)| \leq |\mathcal{R}(\{c\}\,|\,Y)| \leq |\mathcal{R}(\{c\}\,|\,B)| < \varepsilon$. Hence $\mathcal{R}(C\,|\,Y)$ is not a resource space, leading to a contradiction. Therefore, the assumption does not hold. Each category in C_0 is in the shrinking sequence of B.

□

Denote H_i^+ (or H_i^-) as the intersection of C_i and the categories before (or after) s_{i-1} in $Y_i.L^+$. Categories in both H_i^+ and H_i^- are in the same order as in $Y_i.L^+$. Initially, H_0^- is empty and H_0^+ is the intersection of C_0 and $Y_0.L^+$. The following equation decides other values:

$$C_{i+1} = C_i - H_i^+ - H_i^- \qquad \text{(Equation 1)}$$

Lemma 9 $Y_i.L^+$ contains C_0 if and only if C_i is empty.

Proof For any $m < n$, there is $Y_m.L^+ \subset Y_n.L^+$. Hence $\cup_{m=0..i-1} (H_m^+ \cup H_m^-)$ is contained by $Y_i.L^+$. For any category u in any segment s_j ($0 \leq j \leq t$), $\mathcal{R}(\{u\}\,|\,Y) = Y$ holds. Hence $u \notin C_0$.

$C_i = \varnothing \Leftrightarrow C_{i-1} - (H_{i-1}^+ - H_{i-1}^-) = \varnothing \Leftrightarrow C_{i-2} - (H_{i-2}^+ - H_{i-2}^-) - (H_{i-1}^+ - H_{i-1}^-) = \varnothing \Leftrightarrow \cdots \Leftrightarrow C_0 - \cup_{m=0..i-1} (H_m^+ \cup H_m^-) = \varnothing \Leftrightarrow C_0 = \cup_{m=0..i-1} (H_m^+ \cup H_m^-)$. Therefore, $Y_i.L^+$ contains C_0 if and only if C_i is empty.

□

Suppose Y_k be the first node in the upward path from Y to B such that $C_k = \varnothing$.

Theorem 6 The result of *refineSpaceUp*(Y, C, X) is a re-

source space only if for any $m \le k$, $H_m^- = \varnothing$ holds.

Proof Assume there exists $m \le k$ such that $H_m^- \ne \varnothing$. $m > 0$ holds since H_0^- is initialized as $\varnothing$. Let u be any category in H_m^-.

First of all, we demonstrate that $u \notin Y_0.L^+$. Since $H_m^- \subseteq C_m$, there is $u \in C_m$. Since $C_m = C_{m-1} - H_{m-1}^+ - H_{m-1}^-$, $u \notin H_{m-1}^+ \cup H_{m-1}^-$ holds. In case that $m > 2$, let *seg* be the segment in $Y_{m-1}.L^+$ such that $seg.nid = Y_{m-2}.id$. Assume $u \in seg.cats$. There is $\mathcal{R}(\{u\} \mid Y_{m-1}) = Y_{m-2} \supseteq Y$. $\{u\} \ge Y.cview$ holds, which contradicts that $u \in C_0$. Thus $u \notin seg.cats$. Therefore, $u \notin H_{m-1}^+ \cup seg.cats \cup H_{m-1}^- = Y_{m-1}.L^+$. In case that $m = 2$, $u \notin Y_0.L^+$.

There are only two reasons for the absence of u in the shrinking sequence of Y_{m-1}: $Y_{m-1} = \mathcal{R}(\{u\} \mid Y_{m-1})$ or $|\mathcal{R}(\{u\} \mid Y_{m-1})| < \varepsilon$. If $Y_{m-1} = \mathcal{R}(\{u\} \mid Y_{m-1})$ holds, then $\{u\} \ge Y.cview$ since $Y_{m-1} \supseteq Y$. This contradicts with the definition of C_0. If $|\mathcal{R}(\{u\} \mid Y_{m-1})| < \varepsilon$ holds, then $|\mathcal{R}(C \mid Y)| \le |\mathcal{R}(\{u\} \mid Y)| \le |\mathcal{R}(\{u\} \mid Y_{m-1})| < \varepsilon$. Hence $\mathcal{R}(C \mid Y)$ is not a resource space, leading to a contradiction.

Therefore, the assumption does not hold. $\mathcal{R}(C \mid Y)$ is a resource space only if for any $m \le k$, $H_m^- = \varnothing$ holds.

□

refineSpaceUp(Y, C, X) is realized in Algorithm 4, which utilizes the above observations for discovering that the result is not a space as early as possible. First of all, reduces C into C_0 (line 1). Then move upward from Y till Y_t. If some H_i^- is not empty or no C_i is empty, $\mathcal{R}(C \mid Y)$ is not a resource space (line 9 and 17 respectively). Otherwise, get the shrunk node Z of Y_k by the first category u in H_k^+ and then return $\mathcal{R}(C_0 \cup \{c_0, c_1, \ldots, c_{k-1}\} - \{u\} \mid Z)$ as the final result, because $\mathcal{R}(C \mid Y) = \mathcal{R}(C_0 \mid Y_0) = \mathcal{R}(C_0 \cup \{c_0\} \mid Y_1) = \mathcal{R}(C_0 \cup \{c_0, c_1\} \mid Y_2) = \ldots = \mathcal{R}(C_0 \cup \{c_0, c_1, \ldots, c_{k-1}\} \mid Y_k) = \mathcal{R}(C_0 \cup \{c_0, c_1, \ldots, c_{k-1}\} - \{u\} \mid \mathcal{R}(\{u\} \mid Y_k)) = \mathcal{R}(C_0 \cup \{c_0, c_1, \ldots, c_{k-1}\} - \{u\} \mid Z)$. If Z is a child of Y_k, refine it downward (line 15). Otherwise, refine it upward (line 14). The subroutine *getIntersectCats* determines all H_i^+ and H_i^-.

Algorithm 4 refineSpaceUp(**IndexNode** Y, **Set** C, **ShrTreeNode** X)

1) remove any category c in C such that $\{c\} \ge Y.cview$;
2) **if** (C is empty) **return** Y;
3) **ShrTreeNode** *lca* = leastCommonAncestor(Y, X);
4) **int** *upTimes* = $Y.depth - lca.depth - 1$;
5) **Category** *prevSeed* = $\varnothing$;
6) **Sequence** *upSeq* = $\varnothing$;
7) **while** (true) {
8) (*leftHit*, *rightHit*) = getIntersectCats(Y, C, *prevSeed*);
9) **if** (*rightHit* is not empty) **return** null;
10) **if** (*leftHit* == C) {
11) Let *seg* be the segment in $Y.L^+$ such that *leftHit*[0] $\in$ *seg.cats*;
12) **IndexNode** Z = getIndexNode(*seg.nid*);
13) $C = C \cup upSeq - leftHit[0]$;
14) **if** ($Z.id < Y.id$) **return** refineSpaceUp(Z, C, Y);
15) **else** **return** refineSpaceDown(Z, C, Y);
16) }
17) **if** (*upTimes* == 0) **return** null;
18) *prevSeed* = $Y.cview[0]$;
19) *upSeq* = *upSeq* • *prevSeed*;
20) $C = C - leftHit$;
21) Y = the parent of Y;
22) *uptimes* – –;
23) }

Subroutine getIntersectCats(**IndexNode** Y, **Set** C, **Category** *seed*)

index nodes

engaged nodes

root : #1

#508 #1008

#532 #577 #1048

#533 #537 #539 #578

#534

L | 94,⋯ | 34,⋯ | 86,⋯

M #578,#1014,#1032,#1039,#1047

① up 94
② up 110
③ down 58
④ up ∅

id	#539
cview	58,102,13,85, 86,94,36,34
M	EMPTY
L^+	[24], #534, t

id	#578
cview	110,58,13,85, 36,24,90
M	#534
L^+	EMPTY

id	#537
cview	110,102,13,85,86,34
M	EMPTY
L^+	[58], #534, f
	[36], #538, t
	[24,94], #536, t

id	#532
cview	102,13,85,86
M	EMPTY
L^+	[53], #192, f
	[1], #533, f
	[38,90], #536, f
	[110], #537, f
	[58], #539, f
	[24], #540, t
	[94], #544, f
	[36], #545, t
	[34], #547, t

id	#577
cview	58,13,85,36
M	EMPTY
L^+	[53], #192, f
	[34,86,94], #539, t
	[110], #578, f
	[24,90], #579, t

Fig. 10 The process of *refineSpaceUp*(#578, {94}, #1048).

1) **Sequence** *leftHit* = $\varnothing$, *rightHit* = $\varnothing$;
2) **bool** *bleft* = true;
3) **foreach** (segment *seg* in $Y.L^+$) {
4) **if** (*seg.cats*[0] == *seed*) {
5) *bleft* = false;
6) **continue**;
7) }
8) **foreach** (category c in *seg.cats*)
9) **if** ($c \in C$)
10) **if** (*bleft*) *leftHit* = *leftHit* • c;
11) **else** *rightHit* = *rightHit* • c;
12) **return** (*leftHit*, *rightHit*);

4.5.2 *refineSpaceDown*(*Y*, *C*, *X*)

refineSpaceDown(Y, C, X) is realized in Algorithm 5. As presumed, Y is a child of X and all the categories of C_0 are behind the seed of Y in the shrinking sequence of X. If $\mathcal{R}(C \mid Y)$ is a resource space, C_0 must be contained by the shrinking sequence of Y (line 3~4). Then find out the first segment in $Y.L^+$ which contains some category of C_0 (line 5). If the segment corresponds to a child Z of Y, refine Z downward (line 7); otherwise, refine Z upward (line 8).

Algorithm 5 refineSpaceDown(**IndexNode** Y, **Set** C, **ShrTreeNode** X)

1) remove any category c in C such that $\{c\} \ge Y.cview$;
2) **if** (C is empty) **return** Y;
3) (*leftHit*, *rightHit*) = getIntersectCats(Y, C, $\varnothing$);
4) **if** (*leftHit* != C) **return** null;
5) Let *seg* be the segment in $Y.L^+$ such that *leftHit*[0] $\in$ *seg.cats*;
6) **IndexNode** Z = getIndexNode(*seg.nid*);
7) **if** ($Z.id < Y.id$) **return** refineSpaceUp(Z, C, Y);
8) **else** **return** refineSpaceDown(Z, C, Y);

4.5.3 *An Example of Thrashing*

As shown in experimental evulation, the average times of up and down is often less than two, during which process

two to three nodes are accessed in average. In worst case, however, there can be many times of up and down, i.e. a thrashing.

Mushroom is a benchmark dataset for closed itemset mining which has 8124 resources and 119 mutually-incomparable categories. Fig. 10 depicts the process of *refineSpaceUp*(#578, {94}, #1048) during the mining of mushroom, i.e. refining the space #578 which a non-shrunk maxsub of the space #1048, by the category 94. The dashed arrows depict all the accessed nodes. The first time of moving is upward which terminates at node #577, because its shrinking sequence contains the category 94. Category 110 is the seed of #578, and the category 94 refines #577 into #539. Hence $\mathcal{R}(\{94\}\,|\,\#578) = \mathcal{R}(\{94, 110\}\,|\,\#577) = \mathcal{R}(\{110\}\,|\,\mathcal{R}(\{94\}\,|\,\#577)) = \mathcal{R}(\{110\}\,|\,\#539)$. Since #539 is generated before #577, the second time of moving is still upward, calling *refineSpaceUp*(#539, {110}, #577). It terminates at #532 whose shrunk space by 110 is its child #537. Since $\mathcal{R}(\{110\}\,|\,\#539) = \mathcal{R}(\{110, 58\}\,|\,\#532) = \mathcal{R}(\{58\}\,|\,\mathcal{R}(\{110\}\,|\,\#532)) = \mathcal{R}(\{58\}\,|\,\#537)$, the third time of moving is downward, calling *refineSpaceDown*(#537, {58}, #532). Since 58 is in the shrinking sequence of #537, its shrunk space #534 is the final answer.

4.5 Navigation based on MIL's Indexing Structure

The algorithm *rdscMIL* for *redescribe*(*C*) is realized by calling *refineSpaceDown*(S_r, C, $\varnothing$), where S_r is the root of the shrinking tree. The algorithm *meetMIL* for *meet*(S_1, S_2, …, S_k) is realized by calling *rdscMIL*(*floor*($\cup_{\,i = 1..k}\, V_i$)), where V_i is the closed view of space S_i. Experimental study shows only two nodes in average would be accessed for both *rdscMIL* and *meetMIL*.

5 Experimental Evaluation

In this section, we evaluate the running time of MIL with respect to lattice mining and context-aware navigation operators. We also evaluate the storage overhead of its indexing structure.

5.1 Test Environment, Datasets and Constrasting Algorithms

All the experiments are conducted in Dell OPTILEX GX620 with 3.4GHZ CPU, 512M memory, running on Linux (Ubuntu 8.04). All codes are written and compiled using GNU C++.

We adopt the following benchmark datasets of frequent itemset mining: *connect*, *mushroom*, *pumsb*, *pumsb** [2]. These datasets come from real world. *connect* is about game state, *mushroom* is about the characteristics of various mushroom species, and *pumsb*(*) are census data. Table 1 describes their characteristics, including the number of all the resources, the number of all the categories, the average length and the maximal length of each resource's tagged categories.

Table 1 Dataset Characteristics

Dataset	Res. Num.	Cat. Num.	Ave. Len.	Max. Len.
connect	67557	129	43	43
mushroom	8124	119	23	23
pumsb	49046	2113	74	74
pumsb*	49046	2088	50.4821	63

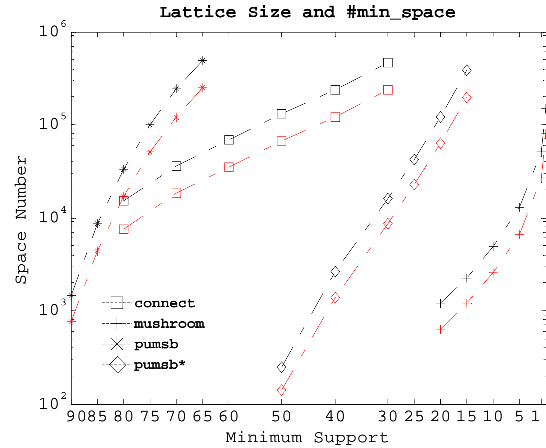


Fig. 11 The lattice size (in black curve) and the number of minimal spaces (in red curve).

Fig. 11 depicts the number of all the resource spaces and the minimal spaces of each dataset under different values of minimum support (min_supp). The lattice can be huge even given a large min_supp, e.g. the lattice generated from pumsb with min_supp 0.65 has 496,070 spaces. The number of minimal spaces is about half the lattice size. For different datasets under the same min_supp, the size of generated lattices as well as the number of minimum spaces can be quite different.

We compare MIL with CHARM-L [3] with respect to lattice mining. Although closed itemset mining algorithms can be tailored to output all the closed views, most of them do not consider the lattice structure. They presume it could be calculated after mining by pair-wise comparison of all the mined closed views. This is the problem of sorting a partially ordered set rather than the linearly ordered set, which seldomly exists in the large literature of sorting algorithms [4]. Intuitively, to calculate a space's successors, we need to scan the whole lattice to get all its descendants and then pick up the maximal ones. Thus the complexity of such post-mining lattice computation POST-M is $O(N^t)$, where N is the number of all the closed views and $t > 2$. To the best of our knowledge, CHARM-L is the first and the only one which calculates the lattice structure during mining. It keeps an inverted list of the closed views already mined by treating category as keyword. Hence the descendants of a new closed view can be fastly retrieved. No specific algorithm is given to determine the maximal ones, however. Compared with POST-M, CHARM-L has the same complexity, but runs faster because of the inverted list. Therefore, we compare the performance and memory sumption of MIL with CHARM-L.

Moreover, we show MIL is also competitive for the task of only mining closed views, by comparing it with CHARM-diff, one of the most superior algorithms in closed itemset mining.

Then we evalute the navigation performance based on MIL's indexing structure, and compare it with those algo-

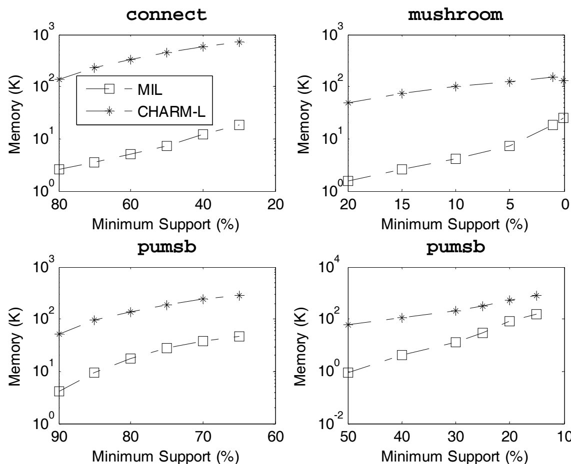


Fig. 12 Memory Consumption.

rithms only based on the lattice structure.

MIL's indexing structure includes two parts: the lattice structure and some intermediate results during mining. The later one is the storage overhead of MIL, which is evaluated at last.

5.2 Memory Consumption

The primitive CHARM algorithm CHARM-prim associates each category in shrinking sequence with a *ridset*, i.e. all the resources of its shrunk space. CHARM-diff replaces *ridset* by *diffset* which is the difference between *ridset* and all the resources in current engaged node. Both mine only closed views. They do subsumption check by a hashtable of the closed views already mined. CHARM-L is based on CHARM-prim or CHARM-diff, but it also mines the lattice structure. CHARM-L does subsumption check by an inverted list of the closed views already mined, which also help incremental lattice construction. As claimed in [3], CHARM-diff is often more memory-saving than CHARM-prim so as to be more efficient. Thus we compare MIL with CHARM-L based on CHARM-diff.

Since the root node of the shrinking tree prepares data for mining, its memory and calculation time is not taken into account for both CHARM-L(or -diff) and MIL. Since both the indexing structure in MIL and the inverted list in CHARM-L can reside in disk, we do not take them into account of the memory consumption.

On one hand, the memory consumption of MIL is about an order of magnitude less than CHAMR-L, as depicted in Fig. 12. The major memory consumption for MIL is the categorization trie while for CHARM-L(or -diff) is *diffset*. *diffset* is a kind of inverted list which keeps the categorization information of all the resources in each engaged node, while categorization trie compresses such information according to the common categories of resources. On the other hand, Fig. 12 shows that the compression becomes less effective with the decreasing of minimum support. The lower the minimum support is, the more categories appear in the front of shrinking sequence. However, these categories are at the bottom levels in categorization tries, so they are less likely to be compressed.

5.3 Running Time

Apart from the generation of the shrinking tree, subsumption check and lattice construction takes major responsibility of the efficiency of mining closed itemsets and their lattice structure. They are often accomplished by visiting the closed itemsets already mined. Hence their efficiency depends on how many ones are accessed.

Table 2 Accessed Closed Views in MIL (connect)

supp	a single refinement		accessed closed views		
(%)	#u_d	#cv	s-check	init_M	total
80	1.951	1.998	7.104	1.916	9.019
70	1.961	1.998	7.634	1.875	9.510
60	1.964	2.001	8.012	1.849	9.861
50	1.966	2.008	8.497	1.833	10.330
40	1.969	2.017	9.000	1.816	10.816
30	1.971	2.030	9.503	1.805	11.307

Table 3 Accessed Closed Views in MIL (mushroom)

supp	a single refinement		accessed closed views		
(%)	#u_d	#cv	s-check	init_M	total
20	1.823	2.645	6.226	3.124	9.350
15	1.789	2.584	7.215	3.023	10.238
10	1.842	2.649	8.239	3.014	11.252
5	1.855	2.695	9.925	3.023	12.948
1	1.877	2.756	12.493	3.033	15.527
0.1	1.890	2.738	12.945	2.905	15.850

Table 4 Accessed Closed Views in MIL (pumsb)

supp	a single refinement		accessed closed views		
(%)	#u_d	#cv	s-check	init_M	total
90	1.782	2.083	5.511	2.262	7.773
85	1.859	2.090	7.835	2.169	10.004
80	1.897	2.091	9.708	2.120	11.828
75	1.926	2.089	11.375	2.084	13.459
70	1.951	2.089	12.799	2.059	14.859
65	1.964	2.095	13.709	2.051	15.760

Table 5 Accessed Closed Views in MIL (pumsb*)

supp	a single refinement		accessed closed views		
(%)	#u_d	#cv	s-check	init_M	total
50	1.774	2.476	3.835	2.444	6.278
40	1.934	2.446	6.927	2.085	9.012
30	1.959	2.396	8.885	2.007	10.892
25	1.965	2.374	9.852	2.005	11.857
20	1.974	2.368	10.788	1.989	12.776
15	1.978	2.348	11.912	1.982	13.894

CHARM-L accesses all the descendants of the itemset to check subsumption, which, in worst case, are a majority of the closed itemsets already mined. By contrast, MIL calculates the refinement of each non-shrunk maxsub of current engaged node by the checked category. In this process, about 10 closed views in average need accessed, as illustrated in the *s_check* column of Table 2 ~ Table 5, even when the lattice size approaches half a million. If the checked category is not subsumed, MIL calculates its refinement of all the other maxsubs of current engaged node for the sake of lattice construction. In this process,

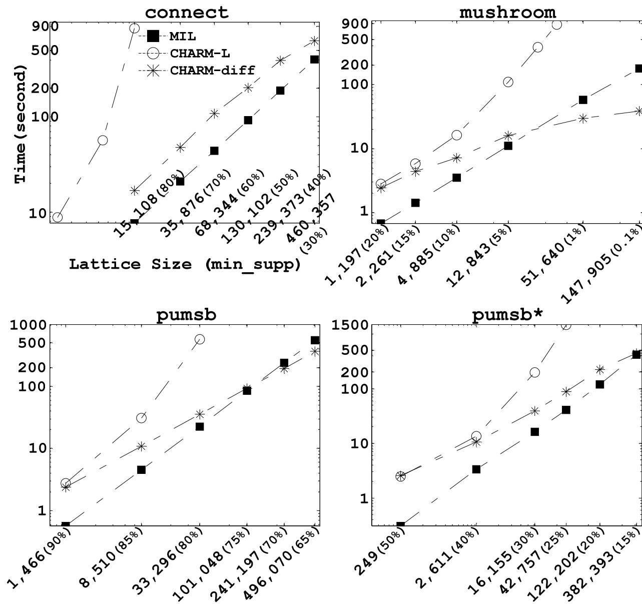


Fig. 14 Mining time.

about 2 nodes in average need accessed, as depicted in the *init_M* column. In total, MIL accesses around 10 to 15 closed views each time of subsumption check and incremental lattice construction, as shown in the *total* column, which slightly grows with the lattice size. Thus MIL is orders of magnitude faster than CHARM-L, as described in Fig. 14.

The above efficency of MIL can be explained as follows. MIL finds out the result of each refinement by moving up and down in the mined part of the whole lattice. Theorectically, the up-down process may iterate many times, and such a thrashing can lead to bad performance. Fig. 13 depicts the average percentage of all the subsumed categories in a shrinking sequence. For pumsb and connect, nearly none is subsumed, and for pumsb*, the subsumption pertage is below 5%. Even mushroom is just around 20%. This means that most categories in shrinking sequences are not subsumed. That is good news for MIL because most tricks it deploys take effect when the refinement result is not a resource space, i.e. the checked category is not subsumed. This accounts for why, in average, there are only 2 times of up-and-down and just two closed views need accessed for each refinement, as depicted in the *#u_d* and *#cv* columns of Table 2 ~ Table 5, respectively. Sometimes, a bad thrashing, accessing hundreds of mined closed views, can be observed.

Fig. 14 also indicates that MIL is comparable to and even faster than CHARM-diff which does not mine the lattice structure. CHARM-diff organizes the closed itemsets already mined in a hashtable such that the key of a closed itemset is the sum of the identifiers of all its corresponding resources. This is verified to be very effective such that most entries have only one closed view. Considering the low percentage of subsumed categories in shrinking sequences, the subsumption check in CHARM-diff requires nearly zero access to the closed views already mined. Hence CHARM-diff is more efficient in subsumption check than MIL. However, CHARM-diff is more costly than MIL in computting shrinking sequences.

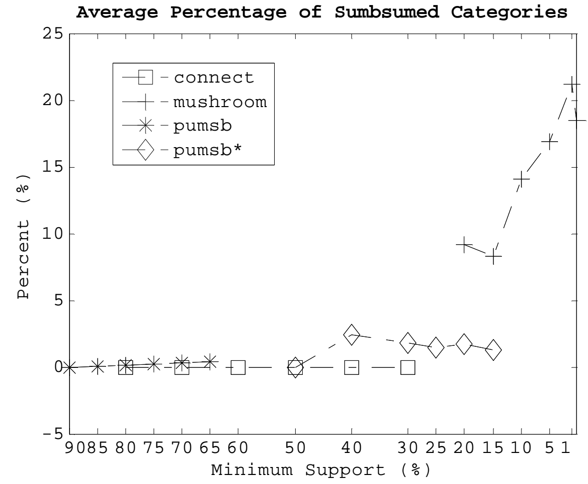


Fig. 13 The average percentage of all the subsumed categories in a shrinking sequence.

CHARM-diff scans many times the *diffsets* associated with each category in shrinking sequences, while MIL scans only two times of each categorization trie. Besides, as analyzed in the memory consumption of MIL and CHARM-L(-diff), categorization trie consumes much less memory than *diffsets*. On the whole, MIL performs better than CHARM-diff in many cases, especially when the lattice is relatively small. For mushroom, pumsb and pumsb* datasets, it performs worse than CHARM-diff when the lattice grows bigger, because more closed views need visited to finish subsumption check and MIL has the additional cost of computing the lattice structure. Fig. 14 shows that MIL curve grows faster than CHARM-diff curve.

5.4 Performance of Navigation Operators

We propose eigth operators for context-aware faceted navigation. Section 3.2 discusses their optimal algorithm based on the lattice structure. The algorithms *meetMinSpace* and *meetFlood* for *meet* and the algorithms *rdscMinSpace* and *rdscFlood* for *redescribe* are non-scalable to the lattice size. Based on MIL's indexing structure, we illustrate that the algorithm *meetMIL* for *meet* operator and the algorithm *rdscMIL* for *redescribe* operator are more efficient and scalable.

Table 6 Accessed Spaces of *meet* in mushroom.

supp(%)	*meetMIL*	*meetFlood*	*meetMinSpace*
20	3.070	6.831	528.983
15	3.068	6.012	1,107.700
10	3.180	7.835	2,347.120
5	3.244	9.866	6,342.350
1	3.505	19.120	25,323.500
0.1	3.259	40.903	77,377.200

We issue 1000 meet queries of two spaces randomly selected and calculate the number of accessed spaces in average by *meetMIL*, *meetFlood* and *meetMinSpaces*, respectively. Table 6 depicts the result. For MIL, only 3 spaces are visited. By contrast, *meetFlood* needs to access tens of spaces while *meetMinSpaces* needs to scan tens of thousands of minmum spaces. Moreover, the cost of *meetMIL*

keeps stable while *meetFlood* and *meetMinSpace* grows very fast. Hence *meetMIL* is also more scalable than the other two

We issue 100 redescribe queries for each length of 2, 3, 4, 5 and 6. The average number of accessed spaces by *rdscMIL*, *rdscFlood* and *rdscMinSpace* are depicted in Table 7. Based on MIL, only 2 spaces in average are accessed. By contrast, *rdscFlood* and *rdscMinSpace* need to visit tens of thousands of spaces. Similar to the result of *meet* operator, the cost of *rdscMIL* grows more stablely so that it is more scalable than the other two.

Table 7 Accessed Spaces of *redescribe* in mushroom.

supp(%)	*rdscMIL*	*rdscFlood*	*rdscMinSpace*
20	1.23	1,095.07	609.9
15	1.24	2,040.3	1,117.28
10	1.23	4,517.17	2,380.56
5	1.57	10,210.8	5,306.66
1	1.89	33,083.2	16,370.6
0.1	2.26	100,758	47,242.8

5.5 Storage Overhead

The storage overhead of the indexing structure consists of the non-maxsub categories in shrinking sequences, because the other categories correspond to successors so as to be part of the lattice structure. Fig. 15 depicts the average ratio of all the non-maxsub categories in shrinking sequences. For pumsb, the ratio is nearly 0. For both connect and pumsb*, the ratior is below 20%. For mushroom, it is around 25%. Besides, the storage overhead does not necessarily increase with the lattice size, like pumsb* and mushroom. Therefore, the storage overhead is quite small.

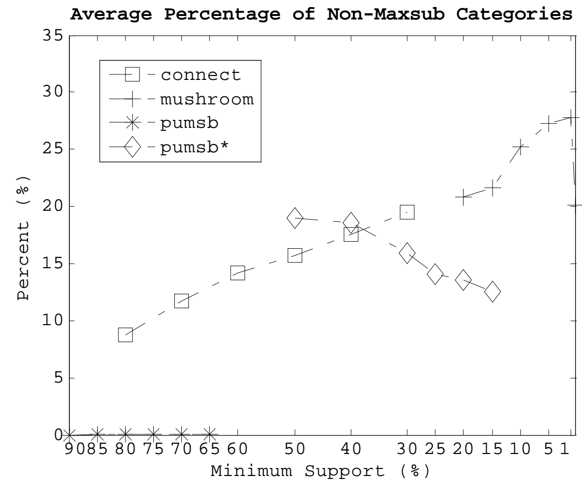


Fig. 15 The average percentage of all the non-maxsub categories in a shrinking sequence.

6 Related Work

6.1 Faceted Navigation

Faceted navigation adopts faceted classification as its conceptual model which is first proposed by librarian to flexibly classify books by a set of mutually-exclusive facets [5].

Flamenco project is one of the first efforts in innovating the interaction style [6][7][8]. It provides a dynamic taxonomy which list only non-dead-end categories for further refinement [9]. A visualization style scaling well with screen size and facet hierarchy is proposed in [10]. A personalized user interface is devised to cater user preference [11]. In [12], users are asked questions at each step during navigation, according to which the next most promising set of facets are recommended. In [13], for current resource set, users are informed with such a category set that has the minimum length. Faceted navigation is also found intuitive to represent semi-structured resources like RDF [14] [15]. Current trend is to combine the aggregation power of data cube with the easy-to-use power of faceted navigation together [16] [17] [18]. All the above works focus on recommending appropriate categories, while this paper treats resource sets as basic elements and focuses on their set relationships.

Another research spectrum on faceted navigation is how to automatically build up facets. A supervised approach for extracting useful facets from a text database is proposed in [19], while the approach in [20] is unsupervised. In [21], a multi-faceted overview of the keywords provided by users is mined in an unsupervised way. Since the above clustering-based approaches often exhibit low-quality semantics, human annotation seems indispensable.

To the best of our knowledge, RSM is the first logical model for faceted navigation [22][23][24]. It organizes resources in a multi-dimensional space under the constraints of certain normal forms which guarantee integrity and consistency of faceted semantics. A series of operations for manipulating spaces are also provided so that users can easily manage a faceted database.

6.2 Closed Itemset Mining

Frequent itemset mining has been studied extensively and intensively in data mining, which is firstly proposed in [25] for discovering association rules. Closed itemsets are found more desirable than frequent itemsets, because they are lossless in information but orders of magnitude less in quantity. A large literature of algorithms for closed itemset mining have been proposed. With respect to dataset format, they can be classified into two categories: horizontal approaches such as A-close [26], CLOSET [27] and CLOSET+ [28], and vertical approaches such as MAFIA [29], CHARM (-L) [3] and LCM [30].

A-close is a breath-first approach which scans the database many times. Its performance deteriorates rapidly in the case of the dataset with long patterns. As shown in CLOSET and CLOSET+, this can be avoided by extending FP-growth algorithm [31]. In vertical approaches, all the frequent itemsets are traversed in a depth-first way and different strategies are deployed to prune non-closed frequent itemsets. MAFIA optimizes the task by a compressed vertical bitmap structure while CHARM by IT-tree. The basic idea of LCM is much similar to CHARM.

Closed frequent itemset, presuming no hierarchy among items, is a special case of closed view proposed in this paper. Hence all the closed frequent itemsets com-

pose a lattice, as illustrated in [26] as well. Besides, closed itemset mining algorithms can be tailored to output all the closed views. However, few of them study the issue of mining the lattice structure as well, since the management of the mined closed itemsets attracts little attention. By contrast, we focus on the lattice structure to support post-mining context-aware faceted navigation.

7 CONCLUSION

Faceted navigation guides users through a lattice of resource spaces by providing different interaction styles, where resource space is such a resource set that can be retrieved by a category set. We have studied the problem of **making users aware of the context of categorization semantics during faceted navigation**. To this end, eight novel operators are proposed: *redescribe*, which exposes the closed view, i.e. the category set with the finest categorization semantics, of a given space; *expand* (*shrink*), which finds out the maximal (minimal) spaces among those which contain (are contained by) a given space and have a certain ratio of common resources with the given one; *maxsim*, which returns the maximal ones among the spaces similar to a given space; *pred* (*succ*), which finds out the minimal (maximal) resource spaces which contain (are contained by) a given space; and *join* (*meet*), which finds out the smallest (largest) resource space containing (contained by) multiple given spaces.

The second challenge we encounter is how to achieve fast response of the proposed operators. We devise an efficient algorithm *MIL* for mining not only all the closed views but also their lattice structure. Moveover, MIL incrementally builds up an indexing structure in the mining process, by preserving some intermediate results of the mining. Compared with current approaches, MIL runs faster and consumes less memory, both in orders of magnitude. The storage overhead of the indexing structure of MIL is very small, less than 30% and even close to zero in our experimental evaluation. Moreover, MIL is even close to and sometimes faster than current approaches which mine only all the closed views. With merely the lattice structure, tens of thousands of access to the mined results are required for the operators *redescribe* and *meet*, while based on the indexing structure of MIL, only two to three access in average is enough.

With the popularity of faceted navigation and the increasing of faceted resources on Web, an interaction style based on our proposed context-aware operators can help users express their query intention clearly and make their surfing more effective. Besides, the proposed algorithm MIL can be applied anywhere the management of mined closed itemsets is required.

REFERENCES

[1] G. Yang, "The complexity of mining maximal frequent itemsets and maximal frequent patterns," *Proc. 10th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining* (*KDD '04*), pp. 344-353, 2004.

[2] B. Goethals and M. Zaki, "Advances in frequent itemset mining implementations," *Workshop on Frequent Itemset Mining Implementations*, 2003.

[3] M.J. Zaki and C. Hsiao, "Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure," *IEEE Trans. Knowledge and Data Engineering*, vol.17, no. 4, pp. 462-478, 2005.

[4] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997.

[5] S.R. Ranganathan, "The Colon Classification," *Rutgers Series on Systems for the Intellectual Organization of Information (ed. Susan Artandi).* New Jersey: Rutgers University Press, vol. 4, 1965.

[6] K.-P. Yee, et al, "Faceted metadata for image search and browsing," *Proc. 21th SIGCHI Conf. Human Factors in Computing Systems* (*SIGCHI '03*), pp. 401–408, 2003.

[7] M. Hearst, "Clustering versus Faceted Categories for Information Exploration," *Commun. of the ACM*, vol. 49, no. 4, pp. 59-61, 2006.

[8] M. Hearst, et al, "Finding the flow in web site search," *Commun. of the ACM*, vol. 45, no. 9, pp. 42–49, 2002.

[9] G.M. Sacco, "Dynamic Taxonomies: A Model for Large Information Bases," *IEEE Trans. Knowledge and Data Engineering*, vol. 12, no. 3, pp. 468-479, May 2000.

[10] R. Dachselt, M. Frisch and M. Weiland, "FacetZoom: a continuous multi-scale widget for navigating hierarchical metadata," *Proc. 26th SIGCHI Conf. Human Factors in Computing Systems* (*SIGCHI '08*), pp. 1353-1356, 2008.

[11] J. Koren, Y. Zhang and X. Liu, "Personalized interactive faceted search," *Proc. 17th Int'l Conf. World Wide Web* (*WWW '08*), pp. 477-486, 2008.

[12] S.B. Roy, et al, "Minimum Effort Driven Dynamic Faceted Search in Structured Databases," *Proc. 17th Int'l Conf. Information and Knowledge Management* (*CIKM '08*), pp. 13-22, 2008.

[13] D. Tunkelang, "Dynamic category sets: An approach for faceted search," *ACM SIGIR. 2006 Workshop on Faceted Search*, 2006.

[14] V. Sinha and D.R. Karger, "Magnet: Supporting navigation in semistructured data environments," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (*SIGMOD '05*), pp. 97–106, 2005.

[15] E. Oren, R. Delbru and S. Decker, "Extending faceted navigation for RDF data," *Proc.5th Int'l Semantic Web Conf.* (*ISWC '06*), pp. 559-572, 2006.

[16] D. Dash, et al, "Dynamic faceted search for discovery-driven analysis," *Proc. 17th Int'l Conf. Information and Knowledge Management* (*CIKM '08*), pp. 3-12, 2008.

[17] P. Wu, Y. Sismanis and B. Reinwald, "Towards keyword-driven analytical processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (*SIGMOD '07*), pp. 617-628, 2007.

[18] Ben-Yitzhak, et al, "Beyond basic faceted search," *Proc. Int'l Conf. Web Search and Web Data Mining* (*WSDM '08*), pp. 33-44, 2008.

[19] W. Dakka, P. Ipeirotis, and K. Wood, "Automatic construction of multifaceted browsing interfaces," *Proc. 14th Int'l Conf. Information and Knowledge Management* (*CIKM '05*), pp. 768-775, 2005.

[20] W. Dakka and P.G. Ipeirotis, "Automatic Extraction of Useful Facet Hierarchies from Text Databases," *Proc. Int'l Conf. Data Engineering* (*ICDE '08*), pp. 466-475, 2008.

[21] X. Ling, et al, "Mining multi-faceted overviews of arbitrary topics in a text collection," *Proc. 14th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining* (*KDD '08*), pp. 497-505, 2008.

[22] H. Zhuge, *The Knowledge Grid*, World Scientific, Singapore. 2007.

[23] H. Zhuge, *The Web Resource Space Model*, Springer, 2008.

[24] H.Zhuge, Y.Xing and P.Shi, "Resource Space Model, OWL and Database: Mapping and Integration," *ACM Transactions on Internet Technology*, 8/4, 2008.

[25] R. Agrawal, T. Imielinski and A. Swami, "Mining association rules between sets of items in large databases," *Proc. ACM*

SIGMOD Int'l Conf. Management of Data (*SIGMOD '93*), pp. 207-216, 1993.

[26] N. Pasquier, et al, "Discovering frequent closed itemsets for association rules," *Proc. 7th Int'l Conf. Database Theory* (*ICDT '99*), pp. 398-416, 1999.

[27] J. Pei, J. Han, and R. Mao, "CLOSET: An efficient algorithm for mining frequent closed itemsets," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (*SIGMOD '00*), pp.21-30, 2000.

[28] J. Wang, J. Han and J. Pei, "CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets," *Proc. 9th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining* (*KDD '03*), pp. 236-245, 2003.

[29] D. Burdick, M. Calimlim and J. Gehrke, "MAFIA: A maximal frequent itemset algorithm for transactional databases," *Proc. Int'l Conf. Data Engineering* (*ICDE '01*), pp. 443-452, 2001.

[30] T. Uno, M.Kiyomi and H. Arimura, "LCM ver.2: Efficient mining algorithms for frequent/closed/maximal itemsets," *Proc. IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2004.

[31] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (*SIGMOD '00*), pp. 1-12, 2000.

This paper is the extended from the following paper:

H.Zhuge and C.He, Faceted Exploration of Emerging Resource Spaces, KGRC-2009-02, 2009. www.knowledgegrid.net/TR. Available at Arxiv preprint arXiv:0903.1680, 2009.

The following are references on the Resource Space Model, which are available at:

http://www.knowledgegrid.net/~H.Zhuge/RSM.htm

[1] H.Zhuge and Y.Xing, Probabilistic Resource Space Model fro Managing Resources in Cyber-Physical Society, IEEE Transactions on Service Computing, http://doi.ieeecomputersociety.org/10.1109/TSC.2011.12.

[2] H.Zhuge, The Web Resource Space Model, Springer, 2007.

[3] J. Liu, X. Li, and L. Feng, Resource space view tour mechanism, Concurrency and Computation: Practice and Experience, 20(7)(2008) 863 - 883.

[4] H. Zhuge and X. Li, RSM-based Gossip on P2P Network, Keynote at ICA3PP2007.

[5] H.Zhuge, P.Shi, Y.Xing and C.He, Transformation from OWL Description to Resource Space Model, 1st Asian Semantic Web Conference, Beijing, China, Sept. 3-7, 2006. (Keynote), LNCS 4185, pp.4-23.

[6] H.Zhuge and E.Yao, Completeness of Query Operations on Resource Spaces, Proceedings of 2nd International Conference on Semantics, Knowledge and Grid, Guilin, China, November, 2006, IEEE Computer Society Press. (Keynote)

[7] H.Zhuge and Y.Xing, Integrity Theory for Resource Space Model and Its Application, The 6th International Conference on Web-Age Information Management (WAIM2005), Hangzhou, China, Oct.11-13, 2005. (Keynote), LNCS 3739, pp.8-24.

[8] H.Zhuge, E.Yao, Y.Xing, and J.Liu, Extended Normal Form Theory of Resource Space Model, Future Generation Computer Systems, 21 (1) (2005) 189-198.

[9] H.Zhuge, The Knowledge Grid, World Scientific, Singapore, 2004. (Chapter 2 discusses the normalization of semantic link network, chapter 3 discusses resource space model, and chapter 4 discusses the integration of the semantic link network and the resource space model to form a single semantic image.)

[10] H.Zhuge, Resource Space Grid: Model, Method and Platform, Concurrency and Computation: Practice and Experience, 16 (14) (2004) 1385 - 1413.

[11] H.Zhuge, Fuzzy resource space model and platform. Journal of Systems and Software, 73(3)(2004)389-396.

[12] H.Zhuge, Resource Space Model, Its Design Method and Applications. Journal of Systems and Software, 72(1)(2004)71-81.

[13] H. Zhuge, A knowledge grid model and platform for global knowledge sharing. Expert Systems with Applications, 22(4)(2002)313-320.