AMSE JOURNALS-AMSE IIETA publication-2017-Series: Advances B; Vol. 60; N°1; pp 189-209
Submitted Jan. 2017; Revised March 15, 2017, Accepted April 15, 2017
https://doi.org/10.18280/ama_b.600112

An Incremental Updating Method for the K-anonymous Dataset using a Similar-B-tree

*Jinling Song, *Liming Huang, **Yan Gao, *Haibin Liu

* Hebei Normal University of Science & Technology
Qinhuangdao 066004, China (songjinling99@126.com)
** Liaoning Institute of Science and Technology
Benxi 117004, China

Abstract

K-anonymity is an effective method to prevent linking attacks and protect privacy. Although the *k*-anonymous dataset guarantees privacy, it must be constantly updated because the original dataset updates occasionally after a version of *k*-anonymous dataset has been exposed. So, how to update the *k*-anonymous dataset simultaneously when the original dataset has been updated becomes an urgent problem. To solve this problem, according to the mapping relation between tuples of the original dataset and *k*-anonymous dataset, a kind of tree structure similar to a B-tree is proposed, so that the update operations on the original dataset can be converted into the corresponding operations of leaf node in the similar-B-tree. Based on this, an incremental updating method for the *k*-anonymous dataset using a similar-B-tree is presented. This method can reflect the changes in the original dataset to the *k*-anonymous dataset in time, and it can greatly improve the update efficiency of the *k*-anonymous dataset.

Key words

Incremental update; *k*-anonymous dataset; similar-B-tree; spatial point

1. Introduction

Data publishing becomes an effective means of data exchange that is convenient for resource sharing. At the same time, the problem of data security becomes a new concern and attracts wide attention. The *K*-anonymity model [1] is an important method to prevent breaches of

privacy in data publishing. It defines the attributes of the publishing dataset (the basic form is a table) which can connect with other published datasets as a quasi-identifier, and makes each tuple(entity) repeat at least k ($k \geq 2$) times on the quasi-identifier through generalization operation. When the publishing dataset connects with another dataset on some attributes in the quasi-identifier, each resulting tuple will not be distinguished from other k-1 tuples, so the privacy of the entity is protected. However, the original dataset changes dynamically. For example, new tuples are inserted, the outdated tuples are deleted and some erroneous tuples are modified after a version of k-anonymous dataset has been released. If the published version cannot be updated in time, the validity and availability of k-anonymous dataset will be greatly reduced, and the practicability of k-anonymity model is restricted. Therefore, how to update the k-anonymous dataset urgently needs to be solved.

The naive update method for k-anonymous dataset is generating a new k-anonymous version for the updated original dataset that is more practical for updating large data. But for small updating data, the method will not only increase the system overhead greatly, but also produce a number of various k-anonymous versions, which may lead to new security breaches [2]. Generally, the size of updating data is relatively small, and if we can update the published k-anonymous dataset directly according to the update operations (incremental update), it cannot only effectively reduce the burden of the system but also avoid the emergence of multiple versions of k-anonymous datasets.

In this paper, we propose an incremental updating method for k-anonymous dataset based on a similar-B-tree. The mapping relation between the original dataset and the k-anonymous dataset is represented by the similar-B-tree, and the update operations such as add, delete, modify are then translated to the corresponding operations on the similar-B-tree.

2. Related Work

Previous research about the k-anonymity model mainly focused on the k-anonymization method and an improved model. In order to improve the accuracy of the k-anonymous tables, the Mingen algorithm [3] was proposed. Literature [4] presented a global domain Incognito algorithm which generalizes all values of the attribute. Literature [5] proposed the multi-dimensional k-anonymization algorithm which can generalize multiple attributes. A. Machanavajjhala et al. [6] presented an improved ℓ-diversity model. An anonymization algorithm suitable for high dimensional sensitive transaction data is presented in [7]. Xiaokui Xiao [8] pointed out that ℓ-diverse optimization is also NP-hard problem when there are only three

different sensitive values, and proposed a (ℓ,d)-approximate algorithm. Junqiang Liu [9] proposed a ℓ^+-diversity model building on the ℓ-diversity model and presented an anonymization algorithm building on complete sub-tree generalization. Ke Wang [10] pointed out that the sensitive information slopes in the temporary data could not meet the ℓ-diversity, and presented a tuple arranging strategy for constructing ℓ-diversity. Literature [11] presented a quasi-sensitive-attribute (QS) concept, and proposed the ℓ-diversity and *t*-closeness models for QS. Paper [12] presented a multi cooperative anonymization algorithm under the half-honest model. Bo WANG et al. [13] proposed a personalized (*a*, *k*)-anonymity algorithm based on entropy clustering. Gong Qiyuan et al. [14] proposed an anonymization method under the absence of data. In order to resist the approximate attack, Zhong Zheyun et al [15] proposed a (*k*, *l*, *e*)-anonymity model.

Although the above algorithms and improved models have some strengths, most of them assumed the dataset to be static dataset. Literature [16-19] considered the *k*-anonymization method under the case of the original tuples increasing monotonically. Literature [20] proposed the *m*-invariance updating method of the *k*-anonymous dataset for insert and delete operations, but it drew into a pseudo generalization technique, which is contrary to the original intention of *k*-anonymity. Literature [21] presented the update method of *k*-anonymous dataset for insert, delete, and modify operations, which is consistent with the discussion in this paper. However, it locates, splits, and merges *QI* groups based on the information loss, which can guarantee the data quality of *k*-anonymous dataset but lead to high time complexity and low update efficiency. Guo Kun et al. [22] proposed a *k*-anonymization method for data stream based on the clustering method.

3. Basic Definition

The dataset in this paper is a relational table modeled as R (A^{QI}, A^S), where A^{QI}= $\{A_1^{QI},A_2^{QI},\ldots, A_n^{QI}\}$ is quasi-identifier attributes, A^S is sensitive attribute. For simplification, the dataset is denoted as R in below. To any attribute set $A\subseteq A^{QI}\cup A^S$, $R[A]$ denotes the projection with duplication of table R on attribute set A, $t[A]$ denotes the values of tuple t on attribute set A.

Definition 1: *k*-anonymity constraint. For dataset R (A^{QI}, A^S), if each tuple in R [A^{QI}] repeats at least k-1($k\geq 2$), then the dataset R satisfies *k*-anonymity constraint.

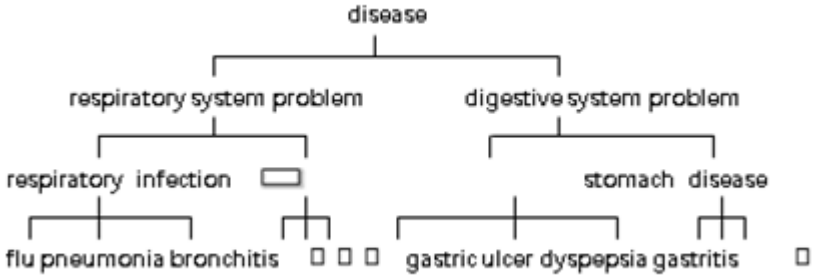

Fig.1. GTree of Disease

Definition 2: Generalizing Tree (*GTree*). Let *D* be a finite attribute domain, *GTree* is a tree which indicates the relations between attribute values of *D* and their generalized values. The leaves are attribute values. Each father node is a general value generalizing all children, and the root is a generalized value of all leaves. Fig.1 is a *GTree* of *disease* attribute.

Definition 3: Generalization. For an attribute, A_i, it's generalizing tree is *Gtree*, the generalization of A_i is a process of mapping a value *v* on A_i to an ancestor of *v* in the *Gtree*.

Definition 4: *k*-anonymous Dataset. For dataset *R* (A^{QI},A^S), generalize the values of A^{QI} and get the dataset *R** in which each tuple satisfies *k*-anonymity constraints on A^{QI}, then the dataset *R** is a *k*-anonymous dataset of *R*. The generalization process from *R* to *R** is called *k*-anonymization.

Definition 5 (*QI* Group) For a *k*-anonymous dataset *R** (A^{QI}, A^S), a group of tuples in *R**[A^{QI}] with same values are called a *QI* group, i.e. *QG*.

Tab.2 is a 2-anonymous table of Tab.1, where the quasi-identifier A^{QI}= {Sex, Age, Zipcode}, t_1, t_2 belong to one *QI* group, t_3, t_4, t_5 belong to another *QI* group.

Tab.1. Table *R*

Sex	Age	Zipcode	Disease
Female	11	101	gastric ulcer
Female	20	105	gastritis
Male	21	106	pneumonia
Male	26	107	bronchitis
Male	30	110	flu

Tab.2. A 2-anonymity table R^* of table R

Sex	Age	Zipcode	Disease
Female	[11-20]	[101-105]	gastric ulcer
Female	[11-20]	[101-105]	gastritis
Male	[21-30]	[106-110]	pneumonia
Male	[21-30]	[106-110]	bronchitis
Male	[21-30]	[106-110]	flu

Definition 6: Tuple-Generalized_Tuple Mapping. Let the original dataset and its *k*-anonymous dataset are R and R^* respectively. To any tuple $t \in R$, if there is $t^* \in R^*$ which makes $t[A_i^{QI}] \in t^*[A_i^{QI}]$ $(1 \le i \le n)$ and $t[A^S] = t^*[A^S]$, then t^* is the generalized tuple of t. Because the count of tuples in R and R^* is invariant, it is one-to-one mapping from tuple to generalized tuple.

4. Description of Similar-B-Tree

A B-tree [23] has some characteristics of a balanced sorting tree, such as being able to quickly locate and easily add, delete, and modify leaves, as well as being able to split the nodes. If we can construct a B-tree between the generalized tuples in *k*-anonymous dataset and their original tuples, then the insert, delete and modify operations on the *k*-anonymous dataset can be transformed into insert, delete and modify leaves on the B-tree.

A B-tree with m order is an empty tree or an m order tree satisfying the following characteristics:

(1) Each node in the tree has at most m sub-trees, and m>2;

(2) If the root is not a leaf, then it has at least two sub-trees;

(3) Other nodes, except root and leaves, have at least $\lceil m/2 \rceil$ sub-trees;

(4) Each non leaf node contains the information (n, $P0$, $K1$, $P1$, $K2$, $P2$, …, Kn, Pn), where:

① n is the count of keys in the node, except the root, the keys in each other nodes satisfy $\lceil m/2 \rceil \le n \le m-1$

② Ki is a key and $Ki<K(i+1)$;

③ Pi is a pointer pointing to the sub-tree root, where $P0$ points to the sub-tree whose keys are less than $K1$, Pn points to the sub-tree whose keys are greater than Kn, each other pointer Pi points to the sub-tree whose keys are between (Ki, $K(i+1)$);

(5) All leaves are at the same level and with no information.

In *k*-anonymity model, each *QI* group in the *k*-anonymous dataset correspond to a certain number of tuples in the original table. The number of tuples in *QI* group is more than *k* and less than 2*k*, which is consistent with the number of keys in the B-tree with 2*k* order, so the B-tree can represent the mapping relationship between *QI* group and the original tuples.

When a *QI* group (i.e., the generalized tuples) is denoted by a sub-tree root and original tuples are denoted by leaves, the mapping of *QI* group to the original tuples is expressed by a sub-tree. All sub-trees can be connected to a root and then a three-layer tree is formed. In order to facilitate the following updating operations, each leaf in the tree should express an original tuple. Additionally, the number of *QI* groups in *k*-anonymous dataset is uncertain and *QI* groups cannot be sorted; namely, sub-trees are unordered, so the leaves and root in the tree do not possess the characteristics of a B-tree. To solve these problems, we improve the traditional B-tree to a new tree where leaves in each sub-tree are arranged in order only, but the sub-trees maybe disordered. We call the new tree similar-B-tree (SBT). The characteristics of the similar-B-tree are as follows:

(1) The similar-B-tree consists of three layers of nodes: root, middle nodes (sub-tree root), and leaves;

(2) The information in each leaf is $(SP_i(A_{i1}^{QI},A_{i2}^{QI},\ldots,A_{id}^{QI}),A_i^s)$, where $SP_i(A_{i1}^{QI},A_{i2}^{QI},\ldots,A_{id}^{QI})$ is the spatial point of the original tuple, A_i^s is the sensitive attribute value of the original tuple;

(3) The middle node has at least *k* and at most 2k-1 children. Each middle node contains the information: $(n, SP_e(A_{e1}^{QI}, A_{e2}^{QI},\ldots,A_{ed}^{QI}), SP_g(A_{g1}^{QI}, A_{g2}^{QI},\ldots,A_{gd}^{QI}), K1, P1, K2, P2, \ldots, Kn, Pn)$.

① *n* is the number of keys (that is child nodes) in the node, which satisfies $k \le n \le 2k-1$;

② $SP_e(A_{e1}^{QI}, A_{e2}^{QI},\ldots,A_{ed}^{QI})$ is the center point of the space rounded by all the children of the middle node, where $A_{ei}^{QI} = \frac{1}{n}\sum_{j=1}^{n} A_{ji}^{QI}$, A_{ji}^{QI} is the *i*th dimension value of the *j*th child of the node;

③ $SP_g(A_{g1}^{QI},A_{g2}^{QI},\ldots,A_{gd}^{QI})$ is the generalized coordinate of all children of the middle node, A_{gi}^{QI} is the generalized value of the *i*th dimension of all children of the middle node;

④ $Ki(1 \le i \le n)$ is the *i*th key, and $Ki < K(i+1)$;

⑤ $Pi(1 \le i \le n)$ is the pointer of one child, *Pi* points to the child whose key is *Ki*;

(4) The children of the root is not limited, and the root contains the following information:$(n, SP_{e1}(A_{e1}, A_{e2},\ldots,A_{ed}), K1, P1,\ldots, SP_{en}(A_{e1}, A_{e2},\ldots,A_{ed}), Kn, Pn)$, where *n* is the number of sub-trees of the root, $SP_{ei}(A_{e1}, A_{e2},\ldots,A_{ed})$ $(1 \le i \le n)$ is the center coordinate of *i*th middle node(root of *i*th sub-tree), $Ki(1 \le i \le n)$ is the max key of the *i*th middle node, $Pi(1 \le i \le n)$ is the pointer of *i*th

middle node.

In the similar-B-tree, the keys in middle nodes are arranged in order to represent the order of leaves, so keys are important information in the middle nodes. Considering each middle node (i.e., *QI* group) represents a spatial region, the distance from each spatial point (a leaf) to the region center is different, which indicates the degree of proximity or compactness of the point with the center. So, we use the distance as the key of a leaf, which cannot only contribute to sort leaves but also facilitate the later split operation. Considering different attributes play distinct roles in practical application, we add a weight W_j to each attribute (i.e., each dimension in the space) where $0 \leq W_j \leq 1$. The greater the weight, the more important the attribute is, so the generalized value on the attribute will be as close to the original data as possible. The distance from any spatial point $SP_i(A_{i1}^{QI}, A_{i2}^{QI}, \ldots, A_{id}^{QI})$ to a center can be expressed as $DSP_i = \sqrt{\sum_{j=1}^{d} W_j (A_{ij}^{QI} - A_{ej}^{QI})^2}$.

5. Conversion of tuple to multidimensional spatial point

To construct a similar-B-tree, the tuples in the original dataset *R* need to be mapped into spatial points at first. Since the *k*-anonymity model protects privacy by generalizing the quasi-identifier attributes, each attribute in the quasi-identifier can be regarded as one dimension of space, and the values of each tuple on the quasi-identifier can be taken as the coordinates of all dimensions and form a spatial point. Based on this principle, any tuple t_i can be transformed into a spatial point $SP_i = (t_i[A_1^{QI}], t_i[A_2^{QI}], \ldots, t_i[A_d^{QI}])$. Since each coordinate value is numerical, every attribute should be converted to numeric before the tuple is transformed into a spatial point:

(1) Numeric: no need to process because it accords with the coordinate type of a spatial point.

(2) Non-numeric: must be converted to numeric by combining the relevant mathematical knowledge and semantic mapping techniques. But it should ensure:

① Uniqueness. The different values of an attribute should be converted to different numericals value as well, while ensuring the mapping is unique.

② Reversibility. The non-numerical value and the converted value is one-to-one mapping, which can guarantee the reverse conversion.

③ Similarity. In the conversion process, the attribute value with a similar semantic should be converted to a close numeric value.

Note: The conversion from tuple to a spatial point can be converted back too according to the above principles.

6. Creating a Similar-B-Tree

With the method mapping the tuple to a multidimensional spatial point, we can create the SBT based on the original dataset *R* and its *k*-anonymous dataset *R**. According to the definition of the similar-B-tree in Section 4, the creating method of SBT is as follows:

(1) Create some sub-trees according to each *QI* group (taken as middle node) in a *k*-anonymous dataset and their original tuples (taken as leaves). One sub-tree SubTree_i is created as follow: Create the middle node M_i, where the tuple count (number of leaves) in *QI* group QG_i is taken as *n* value, the generalized value ($QG_i[A_1^{QI}]$, $QG_i[A_2^{QI}]$,…, $QG_i[A_d^{QI}]$) of QG_i on the quasi-identifier is taken as the generalized spatial coordinate $SP_g(A_{g1}^{QI}, A_{g2}^{QI},\ldots,A_{gd}^{QI})$; calculate the average coordinates of all points corresponding to the original tuples in QG_i and thus obtaining the spatial center coordinates $SP_e(A_{e1}^{QI}, A_{e2}^{QI},\ldots,A_{ed}^{QI})$; create leaf r_l according to the *j*th ($1 \le j \le n$) original tuple in QG_i: take the distance from the spatial point SP_j converted by the *j*th tuple to the center SP_e of middle node M_i as the key of r_l, and insert the key into M_i by order denoted as *Kl*, then points *Pl* to leaf r_l. Repeat the above steps until all leaves have been created.

(2) Create root *RN*, where the sub-tree count is taken as *n* value, the center coordinates $SP_{ei}(A_{e1}^{QI}, A_{e2}^{QI},\ldots,A_{ed}^{QI})$ of the SubTree_i root M_i is taken as SP_{ei}, the max key *Kn* in SubTree_i root M_i is taken as a key *Ki* in *RN*, and points *Pi* to M_i.

The storage structures of nodes in the similar-B-tree are described as following:

```
Typedef struct rootnode  /* Structure of root */
{ int n;  /*count of sub-trees*/
   single SPe[n][d]; /*spatial center coordinates array SPe[1,2,…,n] [1,2,…,d]/
   single K[n] ;  /*K[1,2,…,n] key array K[1,2,…,n]*/
   SBTMNode * P[n]; /*child node pointer array P[1,2,…,n]*/
 } SBTRNode;
 Typedef struct middlenode  /*Structure of middle node */
 {int n;  /*count of keys*/
  single SPe[d]; /*center coordinates array SPe[1,2,…,d]*/
  string SPg[d] ;  /*generalized point coordinates array SPg [1,2,…,d]*/
  SBTRNode * parent; /*pointer of father node*/
  single K[2k] ;  /* key array K[1,2,…,n]*/
  struct SBTLNode * P[2k]; /*child node pointer array K[1,2,…,n]*/
  } SBTMNode;
  Typedef struct leafnode  /* Structure of leaf */
```

{ SBTMNode * *parent*; /*pointer of father node */
single *SP*[*d*]; /**SP* [1,2,…,*d*] point coordinates array *SP* [1,2,…,*d*] */
string *S* ; /* sensitive attribute value*/
} SBTLNode;

7. Incremental updating method basing on Similar-B-Tree for k-anonymous dataset

7.1 Insert Operation

When a tuple *t* is inserted into the original table *R*, the generalized tuple *t** of tuple *t* must be inserted into the *k*-anonymous dataset *R** of table *R*. If a similar-B-tree SBT is created based on *R* and *R**, then the tuple *t* is converted into a leaf *L* (here the spatial point corresponding to *t* is *SP*) of SBT, and the operation inserting *t** into *R** can be transformed to inserting leaf *L* into SBT.

In inserting a new leaf into a similar-B-tree, we comply with the rules that the tree is still a similar-B-tree after inserting, which indicates that all nodes should satisfy their characteristics in the similar-B-tree. Since the leaves are orderly in each sub-tree, the leaf *L* must be inserted into one sub-tree by order. The steps of inserting a new leaf *L* into a similar-B-tree are: ① Choose the sub-tree should be inserted into; ② Determine the insertion position; ③ Insert leaf *L*.

Algorithm 1. Insert algorithm (InsertLeafNode).

Input: the inserted tuple *t*, the root pointer *RN* of similar-B-tree SBT;

Output： the root pointer *RN* of SBT after inserting tuple *t.*

InsertLeafNode(SBTRNode **RN*, *t*)

(1) SBTLNode *L*; Result1 *RS*1; int *w*;

(2) Create leaf *L*;

(3) FOR *i*=1 TO *d* /*Create the spatial point *SP* corresponding to tuple *t* */

(4) *L*->*SP* [*i*]=*t*[A_i^{QI}]; *L*->*S*= *t*[A^s];

(5) *RS*1=SearchSubTree(*RN*, *L*); /*Pick up the sub-tree to be inserted and the key of leaf *L* */

(6) *w*=SearchPosition(*RS*1->*M*, *RS*1->*K*); /*Fix the inserting key position of the leaf *L**/

(7) *RS*1->*M*=InsertNode(*RS*1->*M*, *RS*1->*K*,*w*,*L*) ; /*Insert leaf *L* to the sub-tree root */

(8) return(*RN*);

The method of choosing sub-tree is as follows: Check each spatial center coordinate $SP_{ei}(A_{ei1}, A_{ei2},\ldots,A_{eid})$ in root *RN*, and calculate the distance DSP_i (which is also the key of leaf *L*) from *SP* to spatial center SP_{ei} $(A_{e1}, A_{e2},\ldots,A_{ed})$; if DSP_i is no more than *Ki*, indicating that the

spatial point *SP* belongs to the region represented by the center node, then the sub-tree pointed by *Pi* (root is *Mi*) is the objective sub-tree; otherwise find the minimum DSP_i indicating that the region represented by the center node is nearest to the spatial point *SP*, and the sub-tree pointed by *Pi* (root is *Mi*) is the objective sub-tree.

Typedef struct /*The returning result structure of algorithm picking up the sub-tree to be inserted and the key of leaf *L* */

{struct SBTMNode **M*; /*The root pointer of sub-tree which leaf *L* to be inserted */

single *K*; /*The key of leaf *L* */

} Result1;

Procedure 1: The SearchSubTree algorithm of picking up the inserted sub-tree.

Input: the root pointer *RN* of similar-B-tree SBT, the leaf *L* to be inserted;

Output: The root pointer of the sub-tree, the key corresponding to leaf *L*.

Result1 SearchSubTree(SBTRNode **RN*, SBTLNode **L*)

(1) Result1 *RS*1; single *DSP*=0; int *min*;

(2) FOR *i*=1 TO *n*

(3) { DSP_i= *DSP*(*RN*->SP_e[i], *L*-> *SP*); /*Calculate the distance from the spatial point *SP* of leaf *L* to SP_{ei} in *RN* */

(4) IF(DSP_i<= *RN*->*K*[*i*]) THEN

(5) { *RS*1->*M*= *RN*-> *P*[*i*];

(6) *RS*1->*K*= DSP_i;

(7) return(*RS*1);

(8) }

(9) ELSE

(10) { IF (*i*=1) THEN { *DSP*=DSP_1; *min*=1;}

(11) ELSEIF (*DSP*>DSP_i) THEN {*DSP*=DSP_i; *min*=*i*;}

(12) }

(13) }

(14) *RS*1->*M*= *RN*-> *P*[*min*];

(15) *RS*1->*K*= DSP_{min};

(16) return(*RS*1);

The worst time complexity of the algorithm is *O*(*n*), where *n* is the *QI* group count in *k*-anonymous dataset.

To a sub-tree rooted as *Mi*, the essence of determining the insertion position of leaf *L* is to

find the insertion position of key *K* of leaf *L* in the root *Mi*: If *K*<=*Kn*. then compare *K* with each keyword *Kj* in the root *Mi* from the beginning of *K*1. If the key *Kj* is more than *K*, then the position of *Kj* is the insertion position; otherwise, the insertion position of key *K* is behind *Kn*.

Procedure 2: The SearchPosition algorithm to determine the insertion position of the key of leaf *L* in the sub-tree root *M*.

Input: the root pointer *M* of sub-tree which leaf *L* to be inserted, the key *K* of leaf *L*;

Output: The inserting position *w* of the key of leaf *L* to be inserted in the sub-tree root.

int SearchPosition(SBTMNode **M*, single *K*)

(1) int *w* ;

(2) IF (*K*<=*K*[*n*]) THEN

(3) FOR *j*=1 TO *M*->*n*

(4) IF (*K*[*j*]>*K*) THEN *w*=*j*;

(5) ELSE

(6) *w*= *M*->*n*+1;

(7) return(*w*);

The worst time complexity of the algorithm is *O*(*k*).

Once the insertion position *w* of leaf *L* has been determined in the sub-tree root, according to the key count (i.e., the value of *n*) in sub-tree root (middle node) *Mi*, the operation of inserting leaf *L* into sub-tree can be divided into two cases:

(1) The key count in the node *Mi* is less than 2*k*-1.

If key count *n*<2*k*-1 in node *Mi*, then the children of *Mi* are still less than or equal to 2k-1 after the leaf *L* is added. This does not violate the requirement of the middle node of similar-B-tree, so the key *K* can be added at the position of *Kw* in node *Mi* directly. Then update the key count, the spatial center coordinates, the generalizing spatial point coordinates in node *Mi* and *Mi* information in root *RN*. The inserting operation is shown in Fig.2.

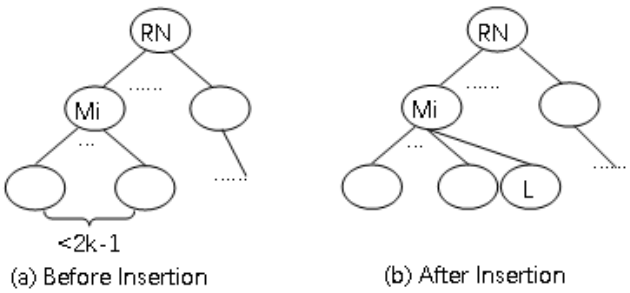

Fig. 2 The children of *L*'s parent are less than 2*k*-1 before *L* is inserted

(2) The key count in node *Mi* is equal to 2*k*-1.

If key count *n*=2*k*-1in node *Mi*, then the children of *Mi* will reach 2*k* after leaf *L* is added. This violates the requirement of the middle node of similar-B-tree, so *Mi* must be split into two middle nodes including *k* children respectively after leaf *L* is inserted.

The inserting method is: First add key *K* at the position of *Kw* in node *Mi* firstly, then split the node *Mi*. Because the keys of *Mi* are in order, we can retain front *k* keys (which have a more intense connection) in *Mi* to be a middle node directly, and split the last *k* keys to be a new middle nodes *Mj*; finally, update the node information in *Mi* and *Mj*, update the sub-tree count and corresponding information of *Mi* in the root, add the corresponding information of the *Mj* in the root. The inserting operation is shown in Fig.3.

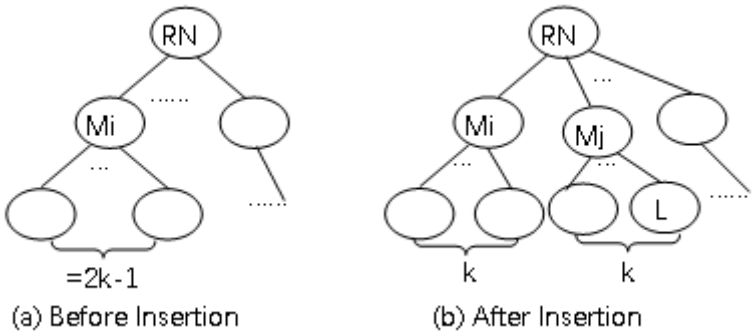

Fig.3. The children of *L*'s parent node are 2*k*-1 before *L* is inserted

Procedure 3: The InsertNode algorithm of inserting leaf *L* to a sub-tree root.

Input: the root pointer *M* of sub-tree to be inserted, the key *K* of leaf *L*, the insert position *w* of leaf *L*, the inserting leaf *L*;

Output: the root pointer *M* of sub-tree after inserting leaf *L*.

InsertNode (SBTMNode **M*, single *K*, int *w*, SBTLNode **L*)

(1) IF (*w*>*M*->*n*) THEN {*M*->*K*[*n*+1] = *K*; *M*->*P*[*n*+1]= *L*;}

(2) ELSE

(3) {Move *K*[*n*], *P*[*n*],…, *K*[*w*], *P*[*w*] back to next position in turn;

(4) *M*->*K*[*w*]= *K*; *M*->*P*[*w*]= *L*;

(5) }

(6) *M*->*n*= *M*->*n*+1 ;

(7) IF (*M*->*n* <=2*k*-1) THEN

(8) {Calculate and replace the information *M*-> SP_e[1,…,*d*], *M*-> SP_g[1,…,*d*] in *M*;

(9) *RN*= *M*->*Parent*;

(10) Replace the information *RN*->SP_e[1,…,*d*], *RN*->*K*[*i*] of node *M* in *RN*;

(11) }

(12) ELSE

(13) {Create a new middle node *Mj*;

(14) Split the behind *k* keys of *M* and put into node *Mj*;

(15 *M*->*n*=*k*; *Mj*->*n*=*k*;

(16) Calculate and replace *M*-> SP_e[1,…,*d*], *M*-> SP_g[1,…,*d*];

(17) Calculate and replace *Mj*->SP_e[1,…,*d*], Mj-> SP_g[1,…,*d*];

(18) *RN*= *M*->*Parent*;

(19) *RN*->*n*= *RN*->*n*+1;

(20) Replace the information *RN*->SP_e[1,…,*d*], *RN*->*K*[*i*] of node *M* in *RN*;

(21) Add the information *RN*->SP_{en}[1,…,*d*], *RN*->*K*[*n*], *RN*->*P*[*n*] of node *M*j in *RN*;

(22) }

(23) return(*M*);

The worst time complexity of InsertNode algorithm is $O(k)$. In summary, the worst time complexity of InsertLeafNode algorithm is $O(n+k)$, where n is *QI* groups in the *k*-anonymous dataset.

7.2 Delete Operation

When a tuple *t* is deleted from the original table *R*, the generalized tuple *t** of tuple *t* must be deleted from the anonymous dataset *R**. If a similar-B-tree SBT is created based on *R* and *R**, then the tuple *t* is converted into a leaf *L* of SBT, and the deleting operation transforms to delete leaf *L* from SBT.

To delete leaf *L* from SBT, first, locate leaf *L*, then delete the leaf. The method of locating the leaf *L* si: First, check each center coordinate in the root, calculate the distance DSP_i between spatial point *SP* of leaf *L* and each spatial center SP_{ei} (A_{e1}, A_{e2},…,A_{ed}) respectively. If DSP_i is no more than the maximum key K_i, then turn to the middle node *Mi* pointed by a P_i. Check each key in middle node *Mi* reversely and find the key K_j which is the same as DSP_i. The leaf corresponding to K_j is *L*.

When a leaf *L* is to be deleted, according to the key count in its father *Mi*, the deleting operation can be divided into two cases:

①The number of children of *Mi* is more than *k* before deletion.

Under this circumstance, deleting the leaf *L* will lead to the number of children of *Mi* no less than *k*, so the deleting operation is: delete the leaf *L* and the information of it in the *Mi* directly,

then replace the center coordinate and generalized coordinate in the *Mi* node, and replace the corresponding information in the root *M*.

②The number of children of *Mi* is equal to *k* before deletion.

Under this circumstance, deleting the leaf *L* will lead to the number of children of *Mi* to be fewer than *k*, so *Mi* should be merged to other middle nodes after deletion. Delete operation is as follows: first remove leaf *L*, then merge the leaves of *Mi* to another middle node and delete *Mi*.

Algorithm 2: The deleting algorithm DeleteLeafNode.

Input: the root pointer *RN* of similar-B-tree SBT, the tuple *t* to be deleted;

Output: the root pointer *RN* of similar-B-tree SBT after deleting tuple *t*.

DeleteLeafNode(SBTRNode **RN*, *t*)

(1) SBTLNode *L*; Result1 *RS*1; SBTMNode **M* ; int *w*;

(2) Create a leaf *L*;

(3) FOR *i*=1 TO *d*

(4) *L*->*SP* [*i*]=*t*[A^{QI}_i];

(5) *L*->*S*= *t*[A^s];

(6) *RS*1=SearchSubTree(*RN*, *L*); /*Pick up the sub-tree containing leaf *L* */

(7) DeleteNode(*RS*1->*M*, *RS*1->*K*) ; /*Delete leaf *L*/

(8) return(*RN*);

Procedure 4. The DeleteNode algorithm of deleting leaf *L*.

DeleteNode(SBTMNode **M*, single *K*)

(1) FOR *j*=1 TO *M*->*n*

(2) IF (*M*->*K*[*j*]= *K*) THEN *w*=*j*;

(3) Move *K*[*w*+1], *P*[*w*+1],…, *K*[*n*], *P*[*n*] ahead a position in turn;

(4) *M*->*n*= *M*->*n*-1 ;

/*While the children of *M* are less than *k*, seek a middle node with *k* children and is nearest to it */

(5) IF (*M*->*n*<*k*) THEN

(6) {*RN*= *M*->*parent*; *j*=0;

(7) FOR *i*=1 TO *RN*->*n*

(8) {*N*=*RN*->*P*[*i*];

(9) IF (*N*->*n*=*k*) THEN {*j*=*j*+1; $DSP_j = DSP(RN\text{->}SP_{ei}, M\text{->}SP_e)$;}

(10) }

(11) *N*=the middle node with least *DSP*;

/*Insert each child of M into N in turn*/

(12) FOR EACH child node $L \in M$;

(13) {K= DSP(N->SP_e, L-> SP);

(14) w=SearchPosition (N, K); /*Ascertain the inserting position of the key of leaf L */

(15) InsertNode (N,K,w,L) ; /*Insert leaf L into N */

(16) }

(17) Replace the information of middle node N in root RN, and delete the information and pointer of M;

(18) }

(19) else

(20) {Calculate and replace M-> SP_e[1,…,d], M-> SP_g[1,…,d];

(21) RN= M->$parent$;

(22) Replace the information RN->SP_e[1,…,d], RN->K[i] of node M in root RN;

(23) }

The worst time complexity of this algorithm is $O(n+k^2)$, where n is QI group count in the k-anonymous dataset. In summary, the worst time complexity is $O(n+k^2)$.

7.3 Update Operation

When a tuple t in the original table R is updated to t', the corresponding generalizing tuple t* in k-anonymous dataset R* need to be updated to t*′. If a similar-B-tree SBT is created based on R and R*, then tuple t and t' can be converted into leaf L and L' of SBT, and the updating operation transforms to delete leaf L from SBT and insert leaf L' into SBT.

The method of deleting leaf L from SBT and inserting leaf L' into SBT is: Locate L at first, then calculate the distance DSP from leaf L' to the center coordinates of parent node M of L. If DSP is less than the maximum key of M, which indicates that L and L' belong to one middle node, then delete L from M directly and insert new key and pointer of L' into M based on DSP. Otherwise delete leaf L firstly, and insert the modified L' into similar-B-tree again. The algorithm is omitted.

7.4 Conversion from Similar-B-tree to the K-anonymous Dataset after Update

After updating the similar-B-tree SBT it must be converted to a k-anonymous dataset. The following method: each leaf in SBT is mapped to a generalizing tuple, which is obtained by attribute values of the quasi-identifier. These values are obtained from each coordinate of the

generalizing spatial point in the corresponding middle node, and the sensitive value is obtained from the sensitive value of each leaf. According to the discussion in preceding sections, the *k*-anonymous dataset only needs to be updated incrementally according to the changed nodes in similar-B-tree, so the update efficiency is greatly enhanced.

8. Experimental Analysis

In this section, we test our algorithms by experiments. We mainly compare the execution time and data quality between InsertLeafNode, DeleteLeafNode algorithms, naive update algorithm (denoted as Static) and I3M^{+}, I3M^{-} algorithm [23] when the original dataset inserts, or deletes tuples. With the naive update method we selected the classic multidimensional *k*-anonymity algorithm [5]. The data quality is measured by the information loss formula [21].

8.1 Experimental Setup

We use Windows 7 operating system, 2.4GHz dual core processor, 2G memory. The experimental data is selected from the Adult dataset [24], quasi-identifiers are {Age, Work-class, Education, Marital-status, Native-country, Race, Sex, Occupation}, the sensitive attribute is Capital-gain.

8.2 Experimental Result Analysis

(1) Let k=5; when the tuples in the dataset vary from 5000 to 50000, we insert 500 tuples into the dataset sequentially. The update time of Static、I3M^{+} and InsertLeafNode algorithm in the *k*-anonymous dataset are shown in Fig. 4.

Figure 4 shows that while the tuples inserted is fixed, the time expended by Static algorithm increases sharply with the tuple increase, which relies on Static algorithm recalculating the *k*-anonymous dataset, and the expended time is linear with the number of tuples in the dataset. With the InsertLeafNode and I3M^{+} algorithms, the updating time is relatively small and increases slowly. The reason is InsertLeafNode and I3M^{+} algorithms adopt incremental updates and the execute time relates to the insert tuples, *QI* group count and k value, but not the tuples in original dataset. In addition, the execute time of InsertLeafNode algorithm is less than I3M^{+}. This means that the InsertLeafNode algorithm has significantly improved the performance with the help of similar-B-tree. The performance comparison in the case of fixing deleted tuples and increasing tuples in the original dataset is similar.

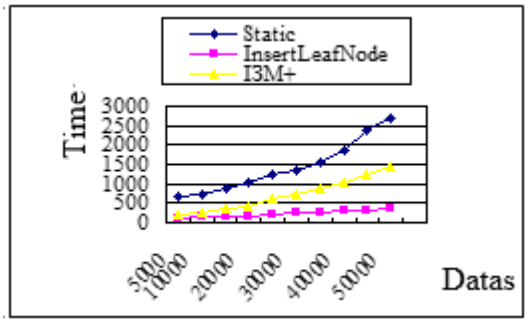

Fig.4. Inserting Time with Variation of Dataset Size

(2) Let the tuples in the original dataset be 20000 and k=5; the update time and data quality comparison of Static、I3cM$^+$ and InsertLeafNode algorithms when the inserted tuples varies from 1% to 50% are shown in Fig.5 and 6 respectively. The update time and data quality comparison when the deleted tuples vary from 1% to 50% are shown in Fig.7 and 8 respectively.

Fig.5 and 7 illustrates that the InsertLeafNode/DeleteLeafNode algorithm is clearly more efficient than both the I3M$^+$/I3M and Static algorithms. In addition, when the inserted/deleted tuples is fewer less, the execution time of the InsertLeafNode/DeleteLeafNode algorithm changes slowly, but the execution time increases quickly when the amount of inserted/deleted tuples increases to a certain number, indicating that the InsertLeafNode/DeleteLeafNode algorithm is suited to small amount of update data but not a large amount. Fig.6 and 8 show that with the increasing of inserted/deleted tuples, the difference between the level of information loss in the k-anonymous dataset with InsertLeafNode/DeleteLeafNode and I3M$^+$/I3M$^-$ algorithms is close, and the information loss is clearly similar with the Static algorithm when the amount of inserted/deleted tuples is small, indicating that the InsertLeafNode/DeleteLeafNode algorithm can guarantee the data quality of the k-anonymous dataset when the amount of update data is small.

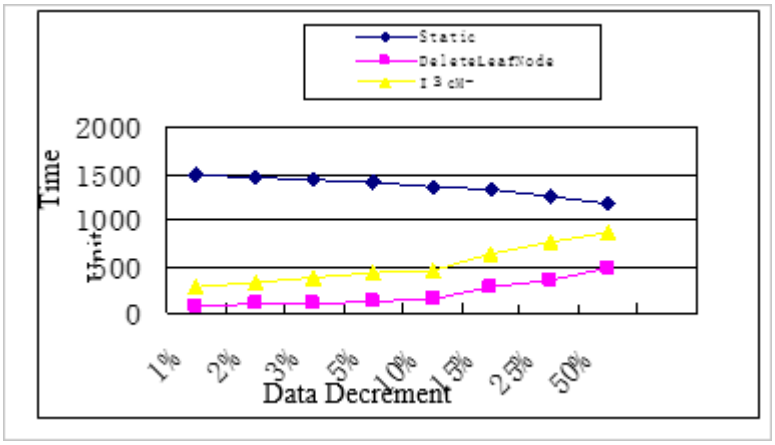

Fig.5. Time expended with Data Increment

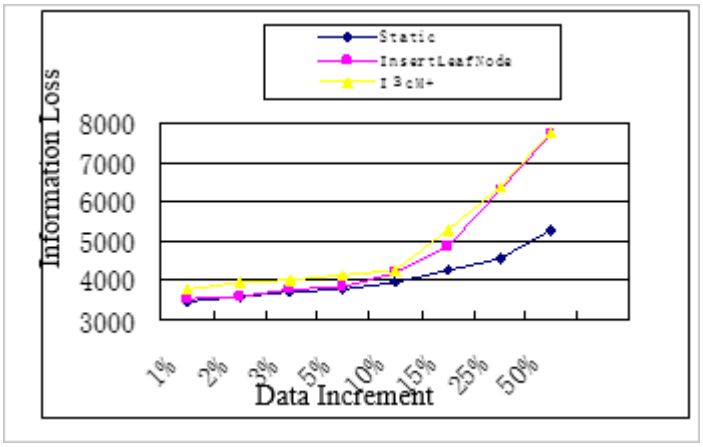

Fig.6. Information Loss with Data Increment

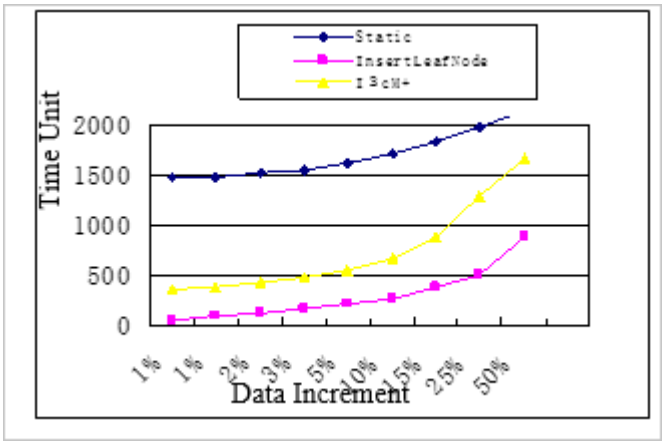

Fig.7. Time expended with Data Decrement

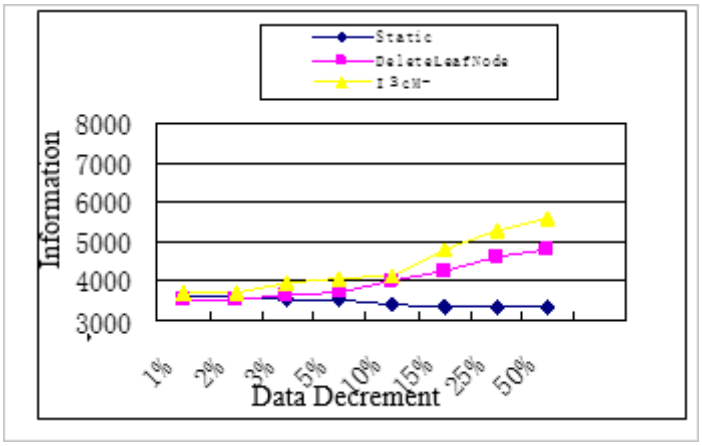

Fig.8. Information Loss with Data Decrement

9. Conclusion

In this paper, we propose an incremental updating method for a *k*-anonymous dataset based on the similar-B-tree. Later research will focus on verifying the extent of incremental updating of a *k*-anonymous dataset with high data quality and without new disclosure by other datasets, and improve the algorithm to fit other improved *k*-anonymous models.

Acknowledgement

This work is supported by The Natural Science Youth Foundation of Hebei Province (No. F2015407039), Hebei Province Department of science and technology Fund (NO.13227427), Doctoral Foundation of HeBei Normal University of Science & Technology (2013YB007, 2013YB001).

References

1. L. Sweeney, K-Anonymity: a model for protecting privacy, 2002, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, vol. 10, no. 5, pp. 557-570.
2. X.K. Xiao, Y.F. Tao, Dynamic Anonymization: Accurate Statistical Analysis with Privacy Preservation. 2008, SIGMOD, 2008, Vancouver, Canada, pp. 107-120.
3. L. Sweeney, Achieving k-anonymity privacy protection using generalization and suppression, 2002, Int'l Journal on Uncertainty, Fuzziness and Knowledge-Based Systems, vol. 10, no. 5, pp. 571-588.
4. K. Lefvre, D. DeWitt, Ramakrishnan. R, Incognito: Efficient full-domain k-anonymity, SIGMOD, 2005, Baltimore, USA, pp. 49–60.
5. K. LeFevre, D.J. DeWitt, Ramakrishnan Raghu, Mondrian Multidimensional K-Anonymity, 2006, ICDE, 2006, Atlanta, USA, pp. 25-35.
6. A. Machanavajjhala, J. Gehrke, D. Kifer, l-diversity: Privacy beyond k-anonymity, 2006, ICDE, 2006, Atlanta, USA, pp. 1-12.
7. G. Ghinita, P. Kalnis, Y.F. Tao, Anonymous Publication of Sensitive Transactional Data, 2011, IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 2, pp. 161-174.
8. X.K. Xiao, K. Yi, Y.F. Tao, The Hardness and Approximation Algorithms for L-Diversity, EDBT , 2010, Lausanne, Switzerland, pp. 135-146.
9. J.Q. Liu, K. Wang, On Optimal Anonymization for L+-Diversity, ICDE, 2010, Long Beach, USA, pp. 213-224.

10. K. Wang, Y.B. Xu, R.C.W. Wong, W.C. Fu, Anonymizing Temporal Data, 2010, the 10th International Conference on Data Mining, 2010, Sydney, Australia, pp. 1109-1114.
11. P. Shi, L. Xiong, B.C.W. Fung, Anonymizing Data with Quasi-Sensitive Attribute Values, 2010, 19th International Conference on Information and Knowledge Management, Toronto, Canada, pp. 1389-1392.
12. N. Mohammed, B.C.M. Fung, M. Debbabi, Anonymity Meets Game Theory: Secure Data Integration with Malicious Participants, 2011, The VLDB Journal, vol. 20, no. 4, pp. 567-588.
13. B. Wang, J. Yang, Personalized (α, k)-Anonymity Algorithm Based on Entropy Classification, 2012, Journal of Computational Information Systems, vol. 8, no. 1, pp. 259- 266.
14. Q.Y. Gong, M. Yang, J.Z. Luo, Data Anonymization Approach for Incomplete Microdata, 2013, Journal of Software, vol. 24, no. 12, pp. 2883-2896.
15. Z.Y. Zhang, J.M. Han, H.Y. Wang, X.C. Chen, (k,l,e)-Anonymity: A Resisting Approximate Attack Model for Sensitive Attributes, 2014, Journal of Chinese Computer Systems, vol. 35, no. 7, pp. 1491-1495.
16. J.W. Byun, Y. Sohn, Bertino E, N. Li, Secure Anonymization for Incremental Datasets, 2006, the 3rd VLDB Workshop on Secure Data Management, Seoul, Korea, pp. 48-63.
17. J. Pei, J. Xu, Z. Wang Z, W. Wang, Maintaining K-Anonymity against Incremental Updates, 2007, the 19th International Conference on Scientific and Statistical Database Management, Banff, Canada, pp. 5.
18. J.W. Byun, T.C. Li, B. Elisa, N. Li, Y. Sohn, Privacy-Preserving Incremental Data Dissemination, 2009, Journal of Computer Security, vol. 17, no. 1, pp. 43-68.
19. Y.J. Wu, Z.H. Sun, X.D. Wang, Privacy Preserving k-Anonymity for Re-publication of Incremental Datasets, 2009, WRI World Congress on Computer Science and Information Engineering, Los Angeles, California, pp. 53-60.
20. X. Xiao, Y. Tao, M-invariance: towards privacy preserving re-publication of dynamic datasets, 2007, SIGMOD, Beijing, China, pp. 689-700.
21. M.T. Traian, C. Alina, K-anonymization incremental maintenance and optimization techniques, 2007, ACM symposium on Applied computing, Seoul, Korea, pp. 380-387.
22. K. Guo, Q.S. Zhang, Fast Clustering-Based Anonymization Algorithm for Data Streams, 2013, Journal of Software, vol. 24, no. 8, pp. 1852-1867.

23. D.J. Newman, S. Hettich, C.L. Blake, C.J. Merz, UCI Repository of Machine Learning Databases, 1998, www.ics.uci.edu/~mlearn/MLRepository.html, University of California, Irvine.