集合数据相交查询的高效处理方法*

汪 卫[1+], 谢闽峰[1], 刘国华[1,2], 庞引明[1], 施伯乐[1]

[1](复旦大学 计算机与信息技术系,上海 200433)
[2](燕山大学 计算机科学与工程系,河北 秦皇岛 066004)

Efficient Processing of Large Intersection Queries on Set_Valued Data

WANG Wei[1+], XIE Min-Feng[1], LIU Guo-Hua[1,2], PANG Yin-Ming[1], SHI Bai-Le[1]

[1](Department of Computing and Information Technology, Fudan University, Shanghai 200433, China)

[2](Department of Computer Science and Engineering, Yanshan University, Qinhuangdao 066004, China)

+ Corresponding author: E-mail: weiwang1@fudan.edu.cn, http://www.fudan.edu.cn

Received 2003-06-09; Accepted 2004-05-08

Wang W, Xie MF, Liu GH, Pang YM, Shi BL. Efficient processing of large intersection queries on set_valued data. *Journal of Software*, 2004,15(Suppl.):53~67.

Abstract: Set is a common data type in database system today. But there is no efficient index structure for set type data to support the queries relate to it. This paper presents a structure called Settrie. The structure is built based on the common prefix patterns in database. Unlike invert file, the sets with same value are well organized. So the size of the data accessed by a query is smaller than that of invert file. This feature will cause the improvement of the selection operation's performance. The experiments support this result. In this paper we also discuss several optimizations approaches to Settrie.

Key words: index of set; Settrie; invert file; select operation; intersection

摘 要: 集合类型是面向对象数据库和对象-关系数据库中的一种重要的数据类型,但是目前还缺少支持相关查询的有效的索引结构.提出了集合类型数据的一种索引结构:Settrie,这种结构是基于数据库中数据的公共前缀构造的,与 Invert file 不同,在 Settrie 中重复的数据得以合理地组织,所以查询中访问的数据量比 Invert file 小,提高了选择操作的性能.通过实验证明:这种方法相比 Invert file 提高了集合数据上的各种相交选择操作的性能,同时还讨论了对 Settrie 的几种优化方法.

关键词: 集合索引;Settrie;倒排文件;选择操作;相交

---

* Supported by the National Natural Science Foundation of China under Grant Nos.69933010, 60303008 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2002AA4Z3430(国家高技术研究发展计划(863))

WANG Wei was born in 1963. He is a professor at the Department of Computing and Information Technology, Fudan University. His researches areas are database, data mining, digital library. **XIE Min-Feng** was born in 1980. He is a Master student at the Department of Computing and Information Technology, Fudan University. His current research interest is the storage of XML. **LIU Guo-Hua** was born in 1966. He is a professor and doctoral supervisor at Department of Computer Science and Engineering, Yanshan University. His researches areas are database, XML. **PANG Yin-Min** was born in 1969. He is a Ph.D. candidate at the Department of Computing and Information Technology, Fudan University. His current research interests include database and XML. **SHI Bai-Le** was born in 1935. He is a professor and doctoral supervisor at the Department of Computing and Information Technology, Fudan University. His researches areas are database, data mining, digital library and security database.

© 中国科学院软件研究所 http://www.jos.org.cn

1 Introduction

Set_valued data is a very popular concept to model entities in the real world. For example, there is a set of keywords for each paper published in journal. There is a set of children for each family. So it is an important data type in database management system[1]. The semantic of query on set is studied in Object_Oriented da tabase[2,3]. But there is still no efficient index schema to manage it and help to improve the performance of query operations.

We are concerned at the query operations looking for the sets that contain all or most of the elements of a given set. This operation is useful in many applications, such as digital library system. User in a digital library system usually gives a set of keywords and asks for papers, the keywords of which cover most elements in the given keyword set.

Like most database management systems, building index is one of the best ways to improve the performance of query. In Refs.[4,5] Sven Helmer and Guido Moerkotte studied four kinds of index structures for set_valued attribute, and found that invert file was the most efficient one. For text document invert_file structure have also been proved to be one of the most efficient index structure. But it has three weakness.

(1) The sequences of the elements which appear frequently in database in the invert file are very long.

(2) Invert_file does not consider efficient storage for sets with the same value, which appear in real data set frequently.

(3) In most of the real set_valued data sets, common patterns appear frequently. Some phrases such as “query optimization” and “access plan” always appear in the keyword list of a database paper together.

In Refs.[6,7] the authors studied the join operation on set_valued data. Two algorithms are all based on the method of translating the set_valued data into signature first, then joining them on the signatures.

In this paper, an index structure for set_valued data: Settrie is presented. The index structure is based on the common prefix patterns in the dataset. So the number of nodes accessed in the index structure is smaller for a query operation. It will improve the performance of query operations. In this structure all common prefix patterns and repeat data are merged.

The problem will be described in Section 2. In Section 3, the structure of Settrie will be discussed. The search algorithms on the index structure will be studied in section 4. In section 5, the performance of the algorithm will be shown via some experiments. We will review some related work in Section 6, and draw conclusions in Section 8.

2 Problem Formulation

In this section, we will give some definitions of the problem.

Definition 1. Suppose E is the set of elements appeared in DataBase. A *Set-database* is a set of set_value tuples, denoted $\{T_1,\ldots,T_n\}$, $T_i=\{id_i,S_i\}(i=1..n)$, id_i is the identify of the tuple in the database. S_i is the set_valued data, which is subsets of E.

In this paper the main topic is about the query on the columns which are set type. To emphasize the set retrieval condition we use the term “Set-database” to denote the database discussed in this paper. For a set value data S, $|S|$ is used to denote the cardinality.

Definition 2. Given a real number δ and a subset Q_s of E, the answer of a *large intersection query* of Q_s on the database D is the sets S_i in D which satisfying $|Q_s \cap S_i| \geq \delta * |Q_s|$.

For example user want to find all books which key words includes at least two words in {Query language, Index, Query Algebra }.

Definition 3. For a large intersection query, if $\delta=1$, then the query is called a full containment query.

© 中国科学院软件研究所 http://www.jos.org.cn

3 Index Structure

The index structure is divided into two layers. The upper layer is a trie. The low layer is a structure similar to invert file on the nodes of the trie.

3.1 Data trie

Given a set-database, we use a trie to represent it. Assume there is a total order on E, and the elements in each set S_i of the set database are sorted by the total order of E. A trie on E is a tree structure over elements in E as follows:

(1) Each node, except the root contains one element of E. Conversely, for every element in E there is at least one node contains that element.

(2) Any two children of a node contain different elements. The order of the child nodes of a node follows the total order on E.

(3) The element of each node in the trie is larger than that of its parent.

(4) Each node contain five *fields: Id*, *Range*, *Label*, *Pointer* to its sons and pointer *P*. *Id* is used to identify each node. The order of *id* is base on the preorder traversal of the trie. *Range* is the largest *id* of its descendent. *Label* is the element in *E*. *P* point to the address of this set in data entry.

(5) Each node in the trie has only one parent.

Definition 4. This trie is said to represent the set-database. If,

(1) For each set in the given set-database, there exists a node v on the trie such that v represents the set. The value of the set is the set of the elements on the path from the root to the node.

(2) For each leaf node v, the set represented by v is in the given set-database.

For example in the Fig.1, node 3 represent the set_value $\{a,b,c\}$. Following the path $a.b.c$ you will find the node 3. From property (4), the trie has an interesting character.

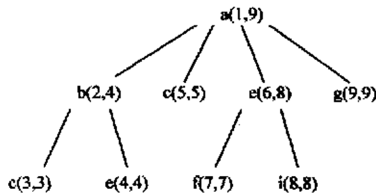

Fig.1 Sample of trie

Theorem 1. For two nodes n_1,n_2,

(1) If n_2 is n_1's descent, then $n_1.id<n_2.id$ and $n_2.range<n_1.rang$.

(2) If n_2 isn't n_1's descent, at the same time n_1 isn't n_2's descent, then either $n_1.id> n_2.rang$ or $n_2.id>n_1.rang$.

Proof. (1) Because n_2 is n_1's descent, and the count of id followed the preorder traversal of the tree, so $n_1.id<n_2.id$. From (4) of the describe of the structure of trie, we have $n_1.range>n_2.range$.

(2) Suppose the range $\{n_1.id,n_1.range\}$ and $\{n_2.id, n_2.range\}$ are overlap for two nodes n_1 and n_2, and n_1 are not n_2's direct relation. Suppose the *id* of node m is in range $\{n_1.id,n_1.range\}$ and $\{n_2.id,n_2.range\}$. From (1) m is one of n_1's descent. Atthe same time m is one of n_2's descent. But from the definition of trie, it is wrong. In Settrie, no node has more than one father.

Theorem 2. Given an element a in E, assume the *id* and *range* of the set of nodes which label is a are $i_1,\ldots,i_k$, and $j_1,\ldots,j_k$. Then a set S in the set-database contains a(its *id* is i) if and only if exists i_t and j_t ($1\le t\le k$), for set tuple (i,S) in set database, satisfying $i_t\le i\le j_t$.

© 中国科学院软件研究所 http://www.jos.org.cn

Proof. Since S contain element a, there is one node n labelled a on the path from the root to the node m which represent S. Suppose n represent set S'. S' contain the elements in S, which is smaller than a. From the definition of Settrie, node m is node n's descent. Suppose the id and range of n is i_t and j_t. From Theorem 1, we have $i_t \le i \le j_t$.

From the trie, it is easy to find the sets contain a set of elements. For example, in Fig.1 from the subtree of node $e(6,8)$, we will find all sets with prefix $\{a,e\}$. that is $\{a,e,i\}$ and $\{a,e,f\}$.

The following algorithm is used to build a trie. In the algorithm the sets in the set-database are added to the trie one by one. For each set, find a path from the *root*'s son node matching the set. If a path entirely matching the set cannot be found, find the path matching as many prefix elements as possible of the set, then build a new path with the rest elements.

Algorithm 1. Build the trie for the a set-database.

INPUT: The set-database *SDB*.

OUTPUT: The trie

```
{
    For each set s_i in SDB
        {Sort the elements in s_i; Put the elements in array EL;
        if (Root=NULL) // Root is the root node of the trie;
        Create the node Root;
        if((the element of node n is EL[1])and(n is a son of Root))
            currnode=n;
        else
            {Create a new node m; Add m to Root's son and m.element=EL[1];
            currnode=m;}
        For (i from 2 to length of EL)
            {if((the element of a node n is EL[i])and(n is a son of currtnode))
                currnode=n;
            else
                {Create a new node m; Add m to currnode's son and m.element=EL[i];
                currnode=m;}}
            return root;}
        Travers the tree with preorder, and set the value for id and range;
}
```

For set_data database $\{\{a,b,c\},\{a,c\},\{a,g\},\{a,b,e\},\{a,b\},\{a,e,i\},\{a,e,f\}\}$. Figure 2 shows the process of generating the trie. At first there is only one root node. With more and more sets are inserted into the trie, the trie grow up. Figure 1 shows the result of the algorithm.

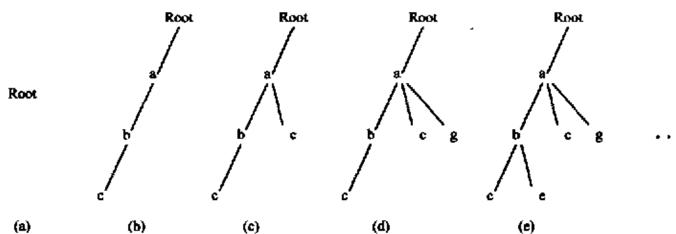

Fig.2 The process of building a trie

© 中国科学院软件研究所 http://www.jos.org.cn

3.2 Invert_system

In Settrie, the invert_system is a set of tuples like $\{ES,TNS\}$, *ES* is a set with one element(All elements appear in the data set) or two elements(two elements are generated by the material optimization in 5.7). *TNS* is a set of sequences. The nodes have two attributes *id* and *range*. *Id* of the nodes in the trie follow the preorder. *Range* is largest *id* of its descent. The sequence is called invert_sequence. Each node in the Settrie corresponds to a node in the invert_sequence. The nodes with same label locates in the same sequence. For the sequence which *ES* contain only one element, the sequence is the list of the nodes labeled the element in *ES*. For the sequence which *ES* contain two elements, the sequence is the list of the nodes labeled with the larger element in *ES*. These nodes are son of a node labelled with the smaller element in ES.

For example in Fig.1 the invert_system is:

$\{a\}$:(1,9);$\{b\}$:(2,4);$\{c\}$:(3,3)(5,5);$\{e\}$:(4,4)(6,8);$\{f\}$:(7,7);$\{i\}$:(8,8);$\{g\}$:(9,9).

There is a data entry part in Settrie. It stores the pointers to the tuples in the database ordered by the id of the node in the trie which represent the value. From it we can find the id or addresses of corresponding tuples in the database.

Figure 3 is a sample of Settrie. In this figure set $\{a,c\}$ appear three times in the database. From the node c(5,5) we will find all of them in data entry.

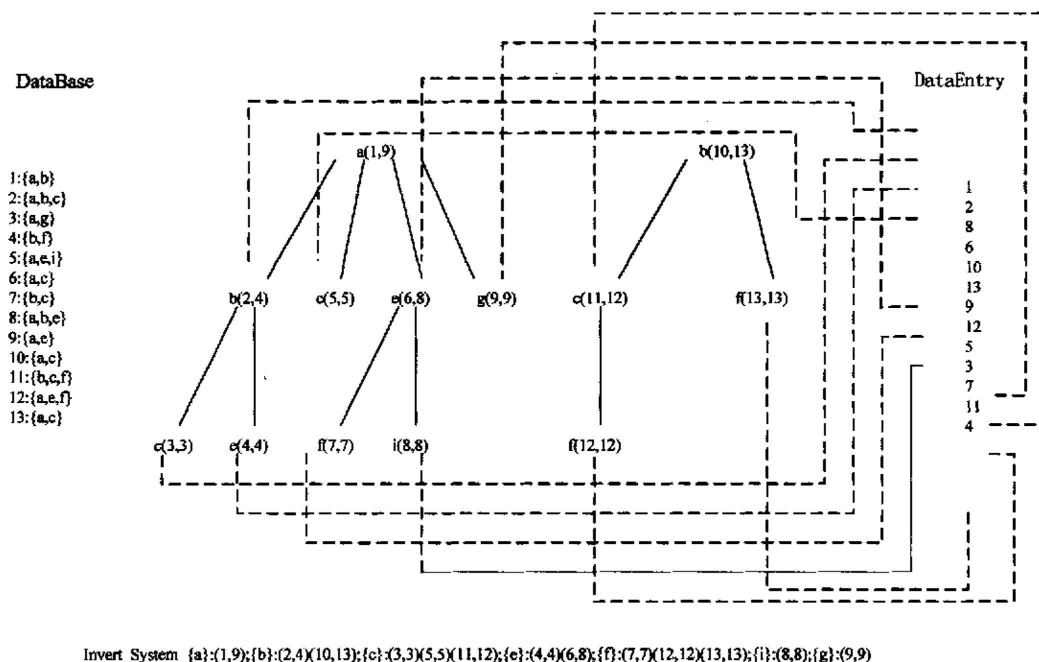

Fig.3 Sample of trie

3.3 Comparison of Settrie and invert file

The size of the invert file is directly proportional to the total number of the elements of all sets. In Fig.3, the size of the invert file is 31 nodes. Meanwhile the size of the invert_system is directly proportional to the total nodes of the data trie. While the data trie can be considered as a compressed version of the source data. In the trie the sets with common prefix are merged. The sets with same value are also merged. The number of nodes in the trie will less than the number of elements in the invert file. For example, in Fig.1 the dataset has 18 elements, but in the data trie,

© 中国科学院软件研究所 http://www.jos.org.cn

there are only 9 nodes. For a new set data to be inserted, which has already appeared in the Settrie, the data trie will not be updated. It means that if many records have same set field, the size of index structure is much less than that of the invert file, and this case often happens in real world.

The Settrie can also be used to reduce the cost of selection operations. For example, if you want to find the sets contain $\{a,b\}$ in Fig.1. You only need to get the nodes labeled b, which is a son of a node labeled a. Then all nodes in b's subtree are result. While with invert file, you must check all sets contain a or b.

3.4 Optimization to the structure

3.4.1 Frequent element

As we described above, the length of the sequence of element a in the invert_system is equal to the number of the nodes in the Settrie labeled a. If the nodes labeled a appear many times in the data trie, the length of the sequence of element a is great. It is not efficient to store and access these long sequences. It only contributes a little to constraint the search scope. With the idea of material view, our invert_system is not a normal invert file system. Not all elements have their invert sequence. While some two value set_values, which are accessed frequently have their invert_sequence. For example, if element a appear in almost all sets in set_value database. The sequence correspond to element a will be very long. So it is not efficient to execute a query operation involving element a. But we found the set $\{a,b\}$ appear only in very few sets. While set $\{a,b\}$ is accessed frequently in users query. So the sequence of set $\{a,b\}$ will improve the performance of users query. The rest of the problem is how to find the frequently accessed elements pair.

Definition 5. Suppose L is the number of leaf in the trie, LS is the size of the node_code sequence. The *selectivity* of the element set is The LS/L.

Definition 6. Given a threshold ε, an element set ES is called *high-selective* if the selectivity of ES, s is less than ε. If s is greater than ε, ES is called *low-selective*.

The definition of selectivity also considers the structure of trie. Some low-selective element in invert file became high selective, because of the position of the element in the order for the elements. For example in Fig.1, element a is low-selective in the original database. But in the trie there is only one node labeled a.

Definition 7. A selective subset is a minimum one if it does not properly contain a high selective subset.

Definition 8. For a threshold δ, if a set is minimum selective subset and it appear more than δ times in users query. Then the set is called *good_material_subset*.

With good_material_subset, we will find out the sets which are high selective, but it is accessed frequently. These sets are good candidates for material. It will improve the queries efficiently.

3.4.2 Material optimization

This step is to find the good_material_subsets, then generate the invert_sequence for it. From the definition of high selective, it is not difficult to find the low-selective elements from set database. The aim of this step is to find the sets relate to those low-selective elements. Since high selective element should no be considered. We hope to find those high selective sets which consist of low-selective elements. For example, the set $\{a\}$ and $\{b\}$ are low-selective, while the set $\{a,b\}$ is high selective. Another requirement for the set is it is frequently accessed. So these sets come from the frequent subset of the users query log. Generally two elements subset will contribute more to users queries than large sets. The following algorithm is only used to find the good_material_subset with two elements.

Algorithm 2. Find the high selective sets which infect the query performance.

INPUT: The element set low-selective, and the set of queries

OUTPUT: The element pair which has high selective.

© 中国科学院软件研究所 http://www.jos.org.cn

```
{
    Get invert sequence for each element in set database, find the high selective elements.
    For each low-selective element a generate a_project_database QPDB and DPDB,
     //QPDB is the set of query which contain element a; DPDB is the set of elements which is descent of
     //element a in Settrie
    For each project_database PDB, put the top m frequent elements in PDB to Candidate
    For each low-selective element b in Candidate, suppose b appeared t_b times in DPDB.
        if (t_b ≥ δ) // δ is the threshold to judge whether the element is high selective.
            Put (a,b) in Result.
    Return Result;
}
```

From the definition of good_material_subset, it must appear frequently in the query but not very frequently in the Settrie. For this reason, two project_databases are used, one is for the query set, another is for the Settrie.

The trie has a very interesting character. For different total order of the elements, we will get different trie. Because in difference order, the number of common prefix pattern are different. In Fig.4 there are two tries for database $\{\{a,b,c\},\{a,b,c,f\},\{b,c,d\}\{b,e,f\}\}$ with different order of the elements. The order for (a) is a,b,c,d,e,f. That for (b) is b,c,a,e,d,f. Then which order is better? There are 9 nodes in (a), but only 7 nodes in (b). So the order of (b) is more efficient than (a). The next problem is how to find the better order. With the better order, the size for the structure will be smaller and query performance will be better. In the trie the sets with common prefix pattern will be merged. These merges will cause the drop of the size of the trie. Based on the assumption that frequently appeared elements have large possibility belong to the same common prefix. The following algorithm generates the order based on the support of the elements in the set_database.

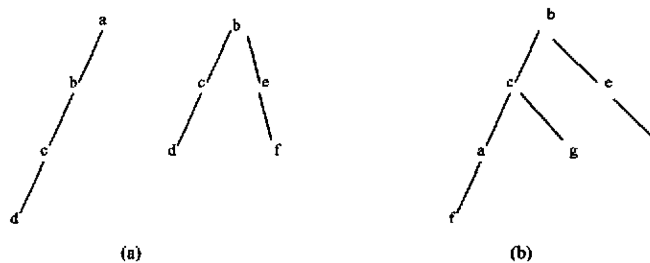

Fig.4 The order optimization

Algorithm 3. Find a better total order on the elements in the database.

INPUT: The sample database S.

OUTPUT: The total order on the elements in S.

```
{
    for each set s_i in S, and each elements a_j in s_i
        count[a_j]=count[a_j]+1;
    S=sort[count];
    return S.
}
```

In this algorithm count is an array to record the count of the elements in the database. The elements are sorted by count. The result of sorting is the new order. Since the elements frequently appear in the database are put in the front part of the order. The size of the trie will be reduced. But the order of the elements will be updated the trie will be rebuild if the distribution the elements changed a lot.

© 中国科学院软件研究所 http://www.jos.org.cn

3.5 Update operations

There are two update operations in set_database: insert and delete.

3.5.1 Insert

The insert operation consists of two steps. In the first step, the algorithm tries to find out whether the set data to be inserted will add new node in the data trie. If a new node is added, it will be inserted to the invert_system. Otherwise, we only need to add the data entry.

3.5.2 Delete

The delete operation also consists of two steps. In the first step, we try to find whether the set data to be deleted will cause a node be deleted from the data trie. If a node is deleted, the node will also be deleted from the invert system. Otherwise, only need to delete the data entry.

The algorithms for the two operations are similar. The algorithms consist of two steps. The first step is traveling from the root to the node represent the set in the trie. So the time complexity for the first step is $o(n)$, n is the cardinality of the set. For a set database, n can be considered as a constant. The second step is to modify the invert_system. Because the nodes in the sequence are sorted by the id of the node in the trie, It is possible to find the node in the sequence in $o(\log(m))$. m is the length of the sequence. The second step is similar to the update operation on invert file. So the complexity is similar to that of invert file.

4 Search Algorithm

Users selection query on set-database can be categorized into three classes, the first one is full containment query. The second is general large intersection queries. The third is equal query. The query statement for the first one is "find the sets which contain the set $\{a_1,a_2,\ldots,a_n\}$". That of the second one is "find the sets which contain p% elements in the set $\{a_1,a_2,\ldots,a_n\}$". The last one is "find the sets which value is $\{a_1,a_2,\ldots,a_n\}$". Users selection query on set-database can be categorized into three classes, the first one is full containment query. The second is general large intersection queries. The third is equal query. The query statement for the first one is "find the sets which contain the set $\{a_1,a_2,\ldots,a_n\}$". That of the second one is "find the sets which contain p% elements in the set $\{a_1,a_2,\ldots,a_n\}$". The last one is "find the sets which value is $\{a_1,a_2,\ldots,a_n\}$".

4.1 Full containment queries

The algorithm of full containment query is to find all nodes locate in the same path from the root, labels on the path covers set in users query. We create an array named Compare_Bed, there are two attributes for each element in Compare_beds element. One is used to store the *ES* part of the invert_system. Another is a node of the sequence in the invert_system denoted nos.

Algorithm 4. Find all sets which contain the set $\{a_1,a_2,\ldots,a_n\}$.

INPUT: The Settrie of the database,Set query=$\{a_1,a_2,\ldots,a_n\}$ given by user;

OUTPUT: The ids of the sets contain query

{

from ES gather all one element sets or two elements set, which is subset of $\{a_1,a_2,\ldots,a_n\}$;

Put the two kinds of sets into two sets S_1 and S_2;

*query*1=*query*;

for each set s_i in S_2

if(*query*1 intersect with s_i)

*query*1=*query*1-s_i; add s_i to *querycandidate*; *querycandidate* is set of set;

for each set s_i in S_1

© 中国科学院软件研究所 http://www.jos.org.cn

```
            if(query1 intersect with s_i)
                query1=query1-s_i; add s_i to querycandidate;
        Sort all sets in querycandidate with the largest element in the sets. And put the sets into the
            Compare_Bed base on the order got in last step;
        Link each element in Compare_Bed to the sequence under corresponding ES;
        Get the first elements of each sequence. And put them into the nos field of correspond set
            in Compare_Bed;
        while none sequence meet the end sign
            {from Compare_Bed, get the element t. t.nos.range is the smallest one in Compare_Bed.
            Put all set t_i, that t_i.rang>t.rang into set Semi_result.
            if (The union of sets in Semi_result cover {a1,a2,... ,an}).......C
                then
                {from the subtree of node t_i, get the tuples contain{a1,a2,...,an}.
                get the next element in sequence related to t and put it to Compare_Bed.}. ...A
                else
                get the next element in sequence related to t and put it to compare_Bed. }
    }
```

The whole process can be divided into two steps. In the first step, the sequences in the invert_system used in the query are selected. Since this algorithm try to find all sets which contain the given set, in the first step, we need to find the ES part in the invert_system which cover the set in query condition. In the second step, the root of the result trees is found out. The data structure used in the algorithm is shown in Fig.5. There is an invert_sequence relate to each elements in Compare_bed. Each time an element in the sequence is get out and put into Compare_bed. For the elements in the Compare_bed, the algorithm will judge whether the elements are all in the same set.

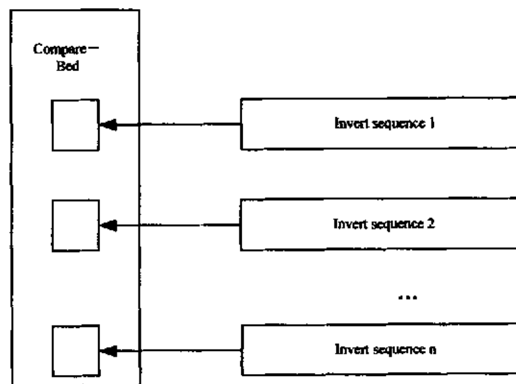

Fig.5 The data structure

Lemma 1. For two sets $A=\{a_1,a_2,\ldots,a_n\}$,$B=\{a_1,a_2,\ldots,a_n,a_{n+1},\ldots,a_m\}$,suppose node n_a,n_b are the two nodes in the trie, which represent set A and B, then $n_a.Id<n_b.Id$ and $n_a.Range>n_b.Range$.

Proof. From the definition of the trie, n_a is one of n_b's ancestor. From Theorem 1, we will get the following result $n_a.Id>n_b.Id$ and $n_a.Range>n_b.Range$.

Theorem 3. All sets that contain set $B=\{a_1,a_2,\ldots,a_n\}$ will be found by Algorithm 4.

Proof. For any set $B'=\{a_1',a_2',\ldots,a_m'\}$, B' contain the set $B=\{a_1,a_2,\ldots,a_n\}$. Suppose the elements in B and B' are sorted. Suppose $a_1'=a_n$. Suppose in the trie the node f_l represent the set $\{a_1',a_2',\ldots,a_l'\}$. The nodes on the path

© 中国科学院软件研究所 http://www.jos.org.cn

from the root to node f_l is $f_1.f_2...f_l$. Set $\{e_1,e_2,...,e_n\}$ are subset of $\{f_1,f_2,...,f_l\}$, and e_i labelled a_i.

At first lets prove when e_n left Compare_bed[n], $e_i(1\le i\le n)$ will be in Compare_Bed[i]. When e_n left Campare_Bed[n], suppose the element in Campare_Bed[i]($i<n$) is not e_i. Because set $\{a_1,a_2,...,a_n\}$ contain set $\{a_1,a_2,...,a_n\}$, from lemma 1 we have two cases: Campare_Bed[i].*Id*>e_n.*Range* or Campare_Bed[i].*Range*<e_n.*Id*. If Campare_Bed[i].*Id*>e_n.*Range*, then e_i have left Compare-bed. But in the algorithm each time we always select the node with smallest range. So e_i left Compare-Bed after e_n. It is conflict. If Campare_Bed[i].*Range*<e_n.*Id* then Campare_Bed[i].*Range*<e_n.*Id*<e_n.*Range*. It is also conflict. So when e_n left Campare_Bed[n], e_i is in Campare_Bed[i].

Because $\{e_1,e_2,...,e_n\}$ are the nodes on a path, all nodes which in the subtree of node e_n are returned as result. From the definition of Settrie, the sets contain set B will be returned as result.

4.2 General large intersection queries

The algorithm of full containment query is to find all nodes locate in the same path from the root, labels on the path covers part of the set in users query.

Algorithm 5. Suppose the query is "find all sets which contain δ% elements in the set $\{a_1,a_2,...,a_n\}$"

Input: The Settrie of the database, the threshold δ% of length of intersect part b, Set query=$\{a_1,a_2,...,a_n\}$ given by user;

OUTPUT: The ids of the sets contain more than δ% elements in the set query

```
{
    From ES gather all one element sets or two elements sets, which is subset of {a1,a2,...,an};
    Put the two kinds of sets into two sets S1 and S2;
    query1=query; elemater=0;
    for each elements ei in query that are low-selective
        if (for each elements fi, sets {ei,fi} ∈ S2 and Query)
            put these sets into querycandidate;
    for each set si in S1
        query1=query1-si; put si into querycandidate;
    Sort all sets in querycandidate with the largest element in the sets. And put the sets into the
        Compare_Bed base on the order got in last step.
    Get the first elements of each sequence. And put them into the nos field of correspond set
        in Compare_Bed.
    while none sequence meet the end sign
        {from Compare_Bed, get the set t. t.nos.range is the smallest in Compare_Bed.
        Put all set ti, that ti.rang>t.rang and ti.id<t.id into set Semi_result.
        if (The union of sets in Semi_result cover more than δ% elements in the set {a1,a2,...,an})....C
            then
            {From the nodes in the subtree of ti, get all tuples contain set {a1,a2,...,an}.
            Get the next element in sequence related to t and put it to compare_Bed.} ...A
            else
            {if(The union of sets in Semi_result may cover more than δ% elements of set{a1,a2,...,an})
                {Scan the nodes in the subtree of node ti. find all sets contain δ% {a1,a2,...,an}....B
                get the next element in sequence related to t and put it to compare_Bed.}}}
}
```

© 中国科学院软件研究所 http://www.jos.org.cn

Lets explain the meaning of "The union of sets in Semi_result may cover more than δ% elements in the set $\{a_1,a_2,\dots,a_n\}$" with an example. For set $\{a,b,c,d\}$, suppose element d is low-selective. For a node p in the sequence of $\{c\}$, if p have no node represent a and b in its ancestor, then the descendant of node p may not cover more than 75% elements in the set $\{a,b,c,d\}$). But for a node q in the sequence of $\{c\}$ if q have a node represent a in its ancestor , then the descendant of node q may cover more than 75% elements in the set $\{a,b,c,d\}$). The reason is a node represent d may in q's descendant. So the algorithm needs to traverse the subtree of q to get the right result. If the order optimization is used, we only need to check the path from the node to the *root*. The reason is that the low-selective element is always smaller than the high selective one in the new order. But if there is a sequence about $\{c,d\}$ is used, then we can tell whether d will be c's descent.

This algorithm looks like the algorithm 4. The difference is in Algorithm 5 all potential subset which contain more than δ% elements should be find out. So the sequence selection part is difference.

Theorem 4. all sets that contain $\delta*n$ elements in set $\{a_1,a_2,\dots,a_n\}$ will be found by Algorithm 5.

The proof of Theorem 5 is similar to that of Theorem 4. When we can not tell whether the set will cover δ% of the query, it will search the path from root or scan the subtree of the node. This step will make sure the algorithm will find the right result.

4.3 Equality queries

Another special query on set is equality query. Equality query are the queries, looks like "find all sets which equal to set $\{a_1,a_2,\dots,a_n\}$".

Algorithm 6. Suppose the query is "find all sets which equal to the set $\{a_1,a_2,\dots,a_n\}$"

Input: index trie, the root of the trie is R.

Output: sets which equal to the set $\{a_1,a_2,\dots,a_n\}$

```
{
    p is a point to the node in the trie.
    i is a integer variable;
    p=Root; i=1;
    while (i<n)
        {In the sons of node p, if there is a node q, q.lable=a_i;
        i=i+1;
        p=q;}
    From the data entry get all sets with the value {a_1,a_2,...,a_n}.
}
```

This algorithm is very simple, it just find a node which represent the given set in the Settrie.

Theorem 5. All sets which value is $\{a_1,a_2,\dots,a_n\}$ will be found by Algorithm 6.

Proof. At the end of the algorithm, we will arrive the node p with path $a_1,a_2,\dots,a_n$. From the definition of the trie, this node represents the set $\{a_1,a_2,\dots,a_n\}$. So from the data entry of node p, we will get the tuples, the value of which is $\{a_1,a_2,\dots,a_n\}$.

5 Performance Analysis

We implement several experiments to evaluate the performance of our algorithm. The data were synthetically generated, and allowed us to study the performance under different conditions.

All experiments were done on PIII 600 machines with 256MB main memory, running Windows 2000 Professional. And the algorithm is written with Standard C++ Library.

© 中国科学院软件研究所 http://www.jos.org.cn

5.1 Synthetic datasets

Because there is no set data set generator, we use the program referred by Ref.[8] to generate the data set.

Each sequence in the data set generated by the program is divided into several transactions. Each transaction is transferred into a set_value data. We also give an identity to each set_value data. Then the data set is transfer to a set_value database.

In the following experiments, we generate several data sets to test the performance of the algorithm in different conditions.

5.2 Compress rate

In our method, the invert system can be considered as the invert file on the nodes in the trie. Why our method is more efficient than that of invert file? The reason is the number of nodes accessed by a query will be smaller than that of the elements in the invert file. Based on the data set generated by the IBM's sequence data set generator, Fig.6 demonstrates the compress rate of the data trie. Compress rate is (number of elements in the invert file)/(the number of nodes in the trie). From it we can get the following result.

(1) The rate of data set with higher average cardinality will be lower.

(2) With the growing of the size of the data set, the rate will also grow.

The principle of trie is to merge the same prefix part of the sets. The prefix part of the set also depends on the total order on the elements. So only common patterns which appear as the prefix part of the sets will cause merge. These merge will cause the increase of compress rate. The Common patterns does not appear in the prefix part will not cause merge. So the compress rate for large cardinality data set is small. While in the large data set, it has more possibility to have common prefix pattern.

Since additional information is added, the size of the index file may be larger than the original data set. In the first experiment, we generate several data set with different sizes. There are no sets with same value in these data sets. Figure 7 gives the ratio of index size and that of data set generated. In large data set, the size of the index is less than twice of the size of data set. As we discussed in previous experiment, the ration for large data set is smaller than that of small one. In the second experiment, we generate several data sets with 12 000 sets, but the duplicate rates are different. From Fig.8, we find the size of index is smaller than the original data set if the duplicate rate is larger than 2. (duplicate rate is the average number of tuples in the data set which have the same value). The reason is duplicate tuples in data set are merged into one tuple in Settrie.

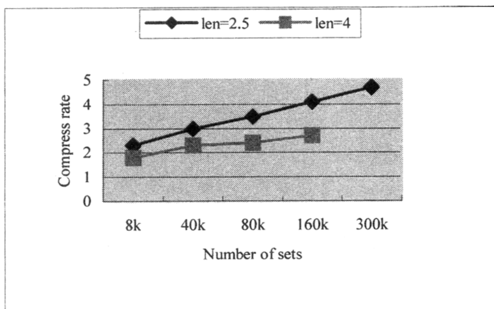

Fig.6 Compress rate of trie

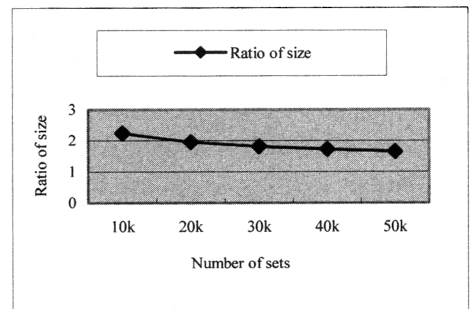

Fig.7 The size of index and original file

© 中国科学院软件研究所 http://www.jos.org.cn

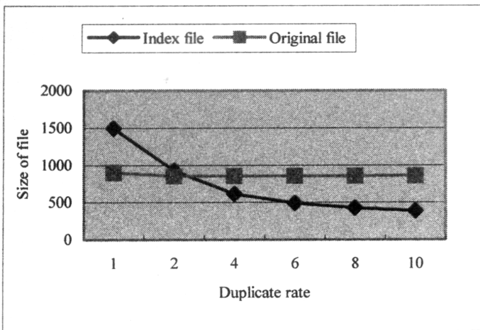

Fig.8 The size of index and original file with duplicate

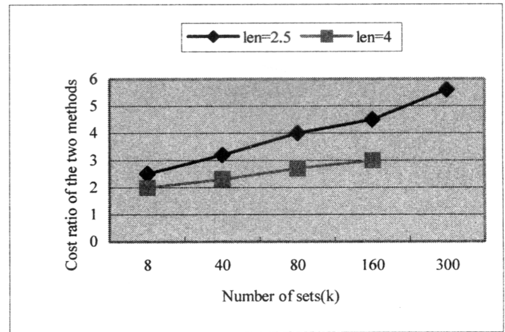

Fig.9 Comparison of query performance

5.3 Query performance

Because the number of the nodes in the trie is smaller than the number of elements in the invert file, the size of the invert_system part in our structure is smaller than the size of invert file. Our query algorithm is similar to that on the invert file. The I/O cost for the query on trie will be smaller than that on invert file directly. Fig.9 show the relationship of the I/O cost for set query on invert file and our method. As what we can get from Fig.6, the large data sets always have high compress rate. The reason is there are more common prefix patterns in larger data set. So the performance is high. In Fig.9 the "Ratio for the cost of the two query method" is (the number of read operations of ourmethod)/(The number of read operations of Invert file).

5.4 Performance of equal query

The algorithm for equal query is totally different to that of invert file. Our algorithm only needs to travel from the root of the trie following the path corresponding to the set given by user. So the cost for equal query relate to the cardinality of the set given by the user. While the I/O cost for equal query on invert file relate to the size of the data set. Figure 10 shows the I/O cost of the equal query on invert file and our method.

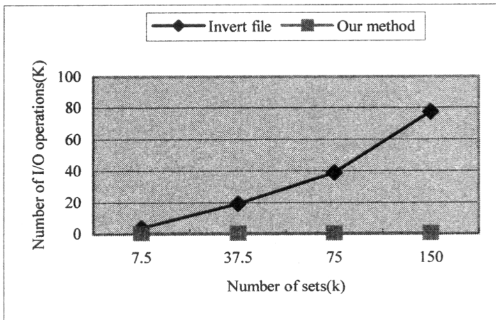

Fig.10 Cost of equal query

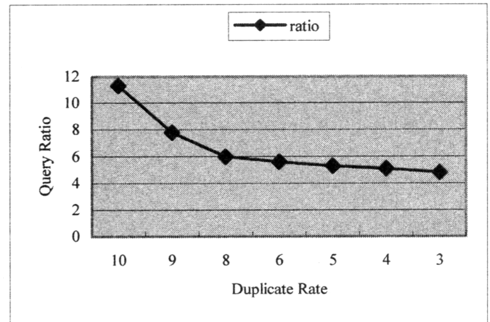

Fig.11 Relationship between the duplicate rate and the size of trie

5.5 Duplicate data

Comparing to invert file, one of our method's benefit is to reduce to size for the sets with same value. As we discussed in previous experiment, in our method only one copy of the duplicate data should be processed. As what we see in Fig.11, if the duplicate rate is high, the size of the trie will be smaller. Each of them have different duplicate rate. Because the size of the data set and the average cardinality of the sets in the data set are same, the number of elements in the invert file is similar. Because the size of the trie is smaller, the query performance will improve. As the size of the data set is smaller, the size of the invert system for the data set is also smaller. The query

© 中国科学院软件研究所 http://www.jos.org.cn

performance will be better. Figure 11 is the result of our experiment. At first, we generate a set of query statements on set, Then run the query statements on invert file and our method. Figure 11 is the curve of ratio of the run time of the two methods. The "Query ratio" is (the number of read operations of out method)/(the number of read operations of Invert file).

5.6 Order optimization

In our method the order of the elements in the set_database influences the structure of the data trie. Based on assumption that frequent elements have large possibility appear in the same frequent two elements pattern. The order optimization step will reduce the size of the trie. Figures 12, 13 and 14 show three group of experiments. The average cardinality of the data set in the first group is 2.5. That of the second group is 4. That of the third group is 8. These figures show the order optimization step will reduce the 10 percent to 20 percent of the size of the date trie.

5.7 Material optimization

Another optimization step is to material some two elements invert sequences. The sequences of low-selective elements in the invert system are very long. It will reduce the performance of query on the sets contain these kinds of elements. In this experience we generate several queries relate to the low-selective elements randomly. Based on the sets in the queries we generate the set of good_material_subset. Figure 15 shows the cost the queries after material optimization. In Fig.15, the X axis is the number of I/O before optimization for these queries. The Y axis is the ratio of the number of I/O of the queries before and after the optimization. From it we will find the material optimization will significantly improve the performance of the queries relate to low-selective element.

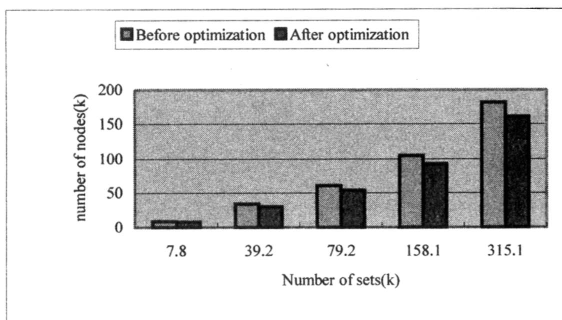

Fig.12 Result of optimization

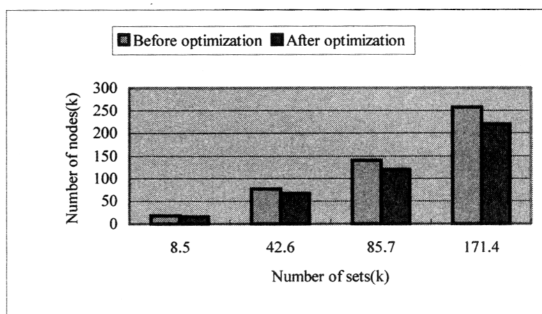

Fig.13 Result of optimization

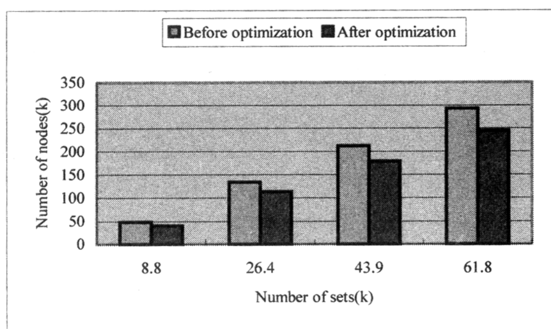

Fig.14 Result of optimization

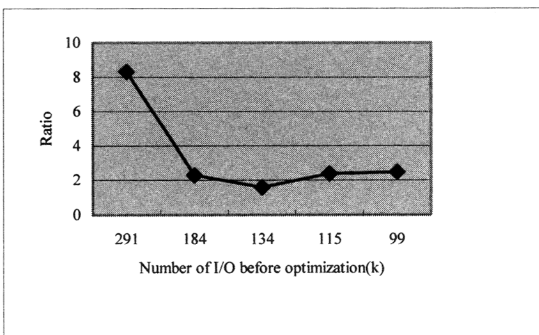

Fig.15 Compress ratio of trie

© 中国科学院软件研究所 http://www.jos.org.cn

6 Related Work

There is some work on the query operation of set_valued data. Reference [4] studied four index structures for set_valued data: Sequential Signature File, Signature Tree, Extendible Signature Hashing and Inverted File. Their work was focused on the select operations relate to set_valued data, such as superset query, subset query and equation query. In Signature based approaches, the set is transfer to a Signature with n bites. The set in the query will be also transfer to a signature. The sets which signature covers the signature of the user's query will be the candidate result. After the confirm step the real results are selected. Because of large number of false drop, they found signature-based index structure have difficulties with some frequently used query set, such as small sets for subset queries, large sets for superset queries. While invert file get much better performance on these operations. In the invert file approach, the sets are organized with invert file. There is a location sequence for each element in the data set. In Ref.[6] several join method are studied, such as tree join, nested-loop join and signature based join. In Ref.[7], partitioned signature based join operation is studied. In Ref.[9], an index structure like R_tree is presented, called RD-Tree. This paper also considered some basic query operations. In Ref.[10], a structure Fp-tree similar to Settrie is used to maintain the frequent item set. Based on FP-tree an efficient frequent pattern mining algorithm is proposed.

7 Conclusion

In this paper, we present an index structure for set type data. It can improve the performance of select operation on set data. In the future we will study other operations on set data such as join. We will study the block based storage of Settrie too.

References:

[1] Bancilhon F, Ferran G. The object database standard. PODS'92. 1992. 351~362.

[2] Beeri C. New data models and language—the challenge. PODS'92. 1992. 351~362.

[3] Tannen V. Languages for collection types. PODS'93. 1993.

[4] Helmer S, Moerkotte G. A study of four index structures for set-valued attributes of low cardinality. Technique Report, 1999.

[5] Helmer S, Moerkotte G. Index structures for databases containing data items with setvalued attributes. Technical Report, University of Mannheim, 1997.

[6] Helmer S, Moerkotte G. Evaluation of main memory join algorithms for joins with subset join predicates. VLDB'97. 1997.

[7] Ramasamy K, *et al.* Set containment joins: the good, the bad and the ugly VLDB'2000. 2000.

[8] Agrawal R, *et al.* Fast algorithms for mining association rules in large databases. VLDB94. 1994.

[9] Hellerstein JM, Pfeffer A. The RD-Tree: An index structure for sets. Technical Report, University of Wisconsin, Madison, 1997.

[10] Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In: Proc. of the 2000 ACM SIGMOD Int'l Conf on Management of Data (SIG-MOD 2000). Dallas, 2000.

© 中国科学院软件研究所 http://www.jos.org.cn