压缩态内存数据库实时算法设计与实现

详细设计说明书

V1.0

小组名称：Never give up
小组口号：Make the change
指导教师：赵振刚老师
文档撰写人：蓝鸿翔
文档撰写时间：2014 年 6 月 5 日

团队分工记录表

项目名称	学号	姓名	分工
压缩态内存数据库实时算法设计与实现	SG13225025	蓝鸿翔	详细设计报告
	SA13226282	刘勇	详细设计报告
	SG13225022	范亚林	

目录

1. 概述

列数据库是对应并区别于行数据库的概念。行数据库就是我们所熟知的传统关系型数据库,即数据按记录存储，每一条记录的所有属性都存储在一起，如果要查询一条记录的一个属性值，需要先读取整条记录的数据。而列数据库是按数据库记录的列来组织和存储数据的，数据库中每个表由一组页链的集合组成,每条页链对应表中的一个存储列，而该页链中每一页存储的是该列的一个或多个值。

列数据库技术有它独有的学术价值,近些年来在国际一流的数据库 会议上 频频有关于这个领域的优秀论文出现,他们主要围绕其商业价值以及主要关键技术，包括基于其主要存储原理的存储压缩、 延时物化、 成组叠代、 查询优化、 索引及加密等进行研发。

1.1 系统需求概述

本系统是基于服务器平台或者嵌入式平台的内存态压缩数据库系统。主要应用在对内存使用要求苛刻，查询密集型用户，对系统操作功能需求单一。

1.2 系统功能概述

系统功能如下图所示：

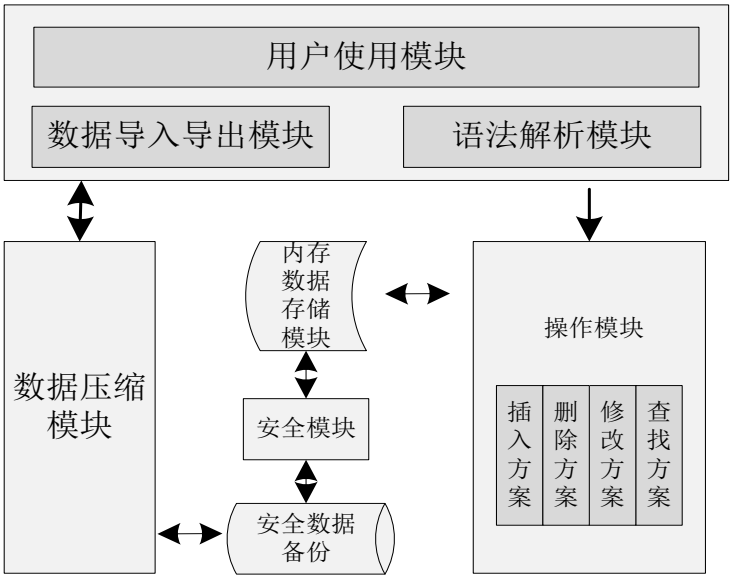

图 1：系统功能图

2. 系统架构

2.1 框架描述

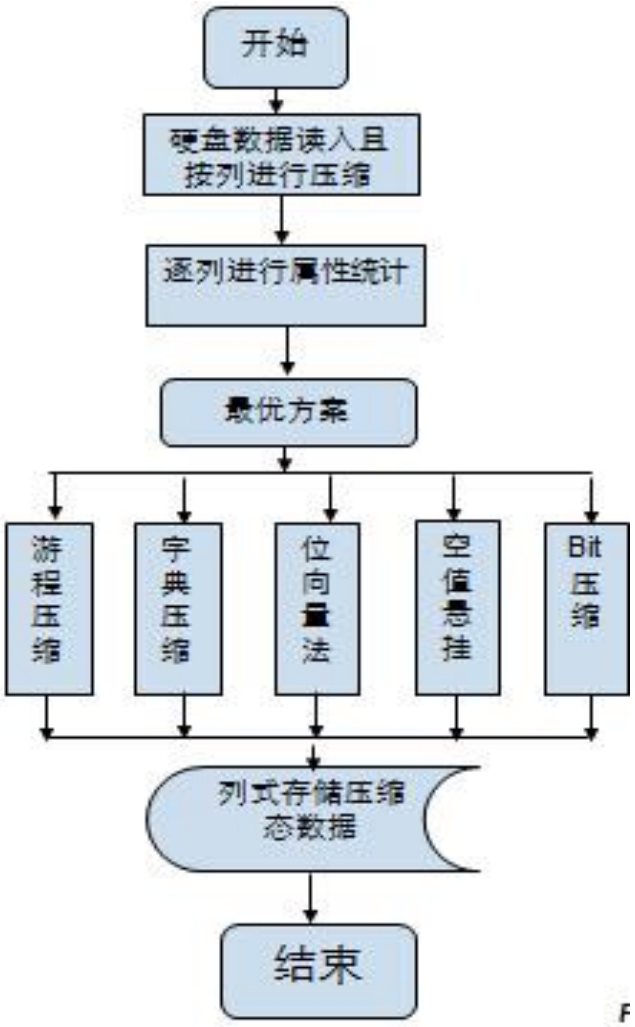

图 2：系统业务逻辑流程图

本系统核心数据流由特定格式的 csv 数据表格作为系统的输入，首先是系统的数据导入模块会把数据的以列存储形式导入到内存，方便后期的处理。接下来有系统的列属性统计模块来对数据的每一列进行统计生成对应的属性描述段。得到所有列的属性描述段后，由系统的算法的选择模块，为每一列选择最佳的压缩算法，并且把所有适合相同压缩算法的列打包在一块，最后有调度模块调用每一种算法来进行数据压缩。压缩态的列数据因为不存在关联，所以仍然是打包在一起存储的。

2.2 系统软硬件平台

本系统目前只支持 Linux 环境使用，对内存要求大于 1GB

2.3 开发环境和工具

计算机型号：联想 Z475
CPU：AMD Llano APU A4-3300M ，1.4GHZ
内存：4GB DDR3 1333MHz
操作系统：linux ubuntu 12.04
开发工具：vim

3．模块详细设计

3.1 体系结构

数据导入模块：

导入测试数据，或者是系统其他数据，对于导入的数据有表格使用本系统规定的格式，否则系统会无法执行。导入的表格一列存储的形式存储于内存，方便其他模块使用。

列属性统计模块：

利用 Hash 统计表格的每一列数据，得到列数据类型、数据的长度、该列的记录数、每个数据的重复次数，数据的样本数，连续长度平均值、空值数目等列属性。

压缩算法调度模块：

在得到每列属性的前提下，根据每列的属性进行使用本系统支持的算法进行空间预测，得到假设使用该算法需要的内存空间和不压缩情况下的内存空间，从而选择最适合该列的压缩算法。并且把所有列根据使用的压缩算法进行分类打包，方便其他模块处理。

数据压缩模块：

这个是系统核心模块，系统支持：游程压缩、字典压缩、位图压缩、比特压缩、空值悬挂、原始数据保留等压缩算法。算法对整个包进行压缩（每个算法和包进行绑定）。

数据操作模块：

数据操作模块对象分为两个，一个是压缩态的数据、另一个是费压缩态的组织在红黑树中的原始表格数据。对于压缩态数据，根据其选择的压缩算法选择对应的数据操作模块、对于原始态的表格数据、则使用同一的数据操作。

数据导出模块：

数据导出模块的导出数据可分为:原始数据、查询数据、压缩态数据、Hash 统计得到的列属性集合。

3.1.1 数据导入模块

模块名：csvfile_loader
数据处理流：

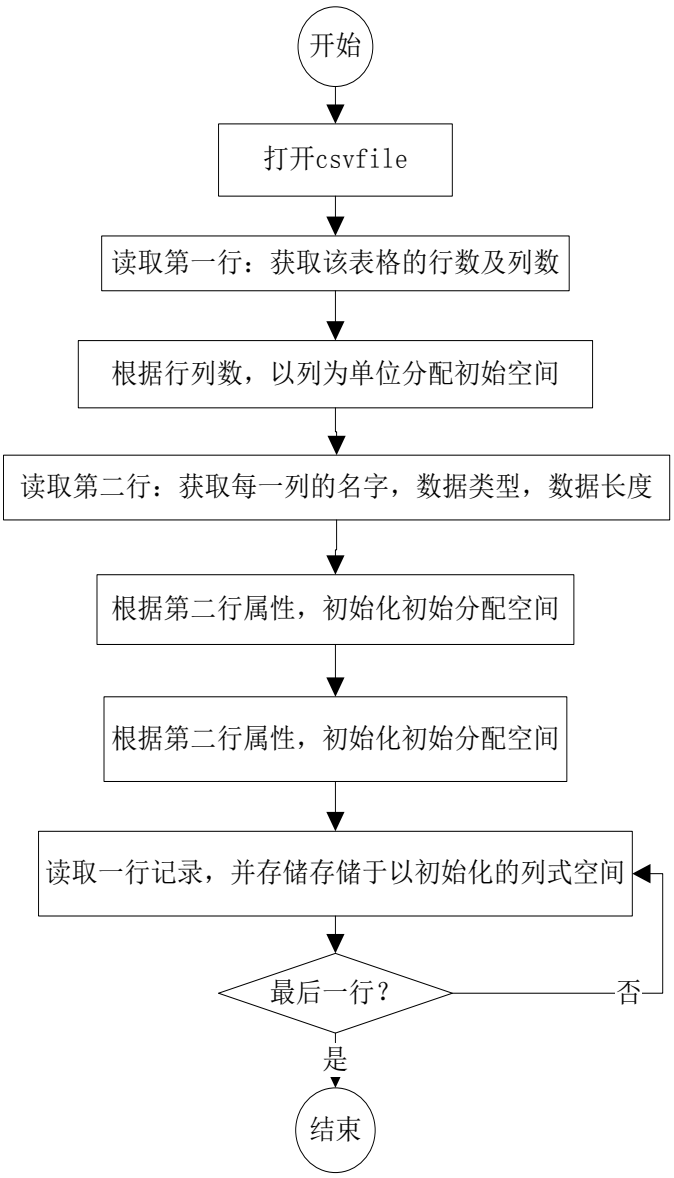

图 3：数据处理流

本系统使用的 csv 文件满足一下条件：
首行：存放该 csv 文件的行数 rows 和列数 cols；
次行：存放每列数据的属性 name1:datatype<[data_len]>,…；
剩下：都是数据本身 field1,field2…；

导入数据库文件到内存的相关代码如下：

```
pItem memcsv = DBcalloc(col*row,sizeof(Item));
    int i ;
    for( i = 0 ; i < col ; i++) {
        mfile->colarray[i] = &memcsv[i*row];
    }
    memcpy(mfile->attlist,attlist,sizeof(CAttrtList));
    mfile->memcsv = memcsv;//set memcsv field of memory csv file
    mfile->len      = col ;   //set column number field of   memory csv file
    mfile->rows     = row ;   // set rows   number field of   memory csv file
```

3.1.2 列属性统计模块

统计目标：

```
/*
 * 列属性结构体
 * 例如十个 items 的例子：eg:1,1,1,2,2,3,4,5,6,7
*/
typedef struct Segment
{
    unsigned int rows ;       /*该段中包含的 item 数目---eg:10*/
    /*该段中包含相同值的 item(必须两个以上的值，并且不同的) ---eg:2*/
    unsigned int same_items ;
    /*该段中包含相同值的 item 总数目(两个以上的值)----eg:3+2=5*/
    unsigned int all_same_items;
    /*该段中包含不同值的数目----eg:'3' '4' '5' '6' '7' =5*/
    unsigned int diff_items;
    /*该段中所有不相同的数目*/
    unsigned int all_diff_items;
    /*该段中包含空值的数目---eg:0*/
    unsigned int null_items;
    /*该段中相同值连续的平均数目----(all_same_items/same_items)---eg:5/2*/
    unsigned int avrg_items;
    /*该段中各个 item 的平均长度------eg:rows/(same_items+diff_items)= 10/(2+5)*/
    unsigned int len_item;
    /*数据的长度(1)，如果是字符串那么应该是字串的长度,*/
    /*数据的字节长度=len*sizeof(type)*/
    unsigned int len ;
    /*该段所属的数据类型*/
    DBtype type;
    char name[32];
}Segment,*pSegment;
```

统计方法：
利用 hash 表格完加快属性的统计：

```
typedef struct HashNode
{
    /*第一个出现的值*/
    pItem pitem ;
    /*该值重复出现的长度*/
    unsigned int repeat_items ;
    /*该值有几个连续的段*/
    unsigned int times ;
struct HashNode * pnext ;
}HashNode,*pHashNode;
```

1、 把对应每个 field 都生成一个的 hash code,字段相同的 hash code 一样
2、 插入到对应的 hash 表中,每个 hash 值对应一个链表
3、 更新 hash 节点中的对应域
4、 更新 hashtable 的每一个域：

```
typedef struct HashTable
{
    /*the length of hash table */
    unsigned int len ;
/*the max sizeof of the data,measure by byte*/
    unsigned int max_len ;
pItem preInsertItem ;
    pItem ppreInsertItem ;
    //pItem ppreInsertItem ;
    /*该段中包含的 item 数目*/
    unsigned int rows ;
    /*该段中包含空值的数目*/
    unsigned int null_items;
    unsigned int diff ;
    unsigned int all_diff;
    /*node number in this hash table*/
    unsigned int nodes ;
    DBtype type ;
    /*the pointer of the hash table */
    pHashNode hasharray[1024];
}HashTable,*pHashTable;
/*end of hash table*/
```

5、 把每一列的属性以 segment 结构体导出。

3.1.3 压缩算法调度

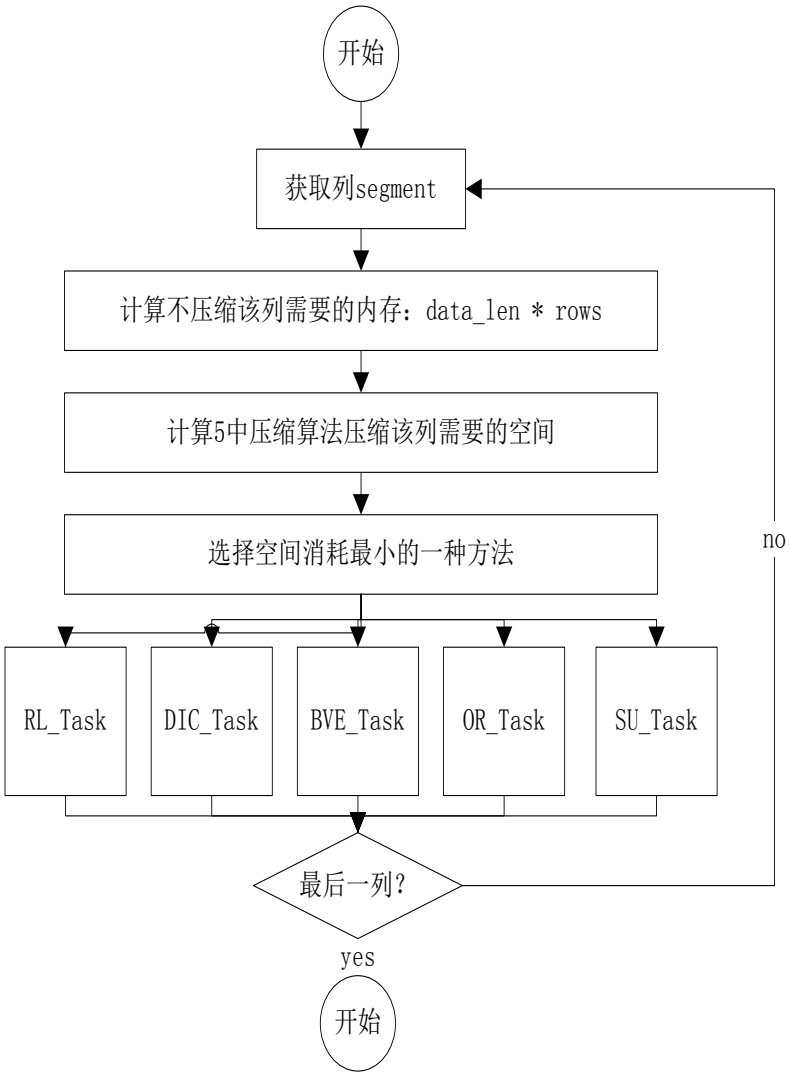

图 4：压缩算法的调度流程图

每列数据的属性特征，即属性统计模块 hash 统计的结果，计算采用不同压缩方案的空间消耗情况。

每种压缩算法，对应空间消耗计算公式如下所示：

原始数据：$S\_o=D\times(l-N)+l/8$

游程压缩：$S\_R=(4+D)\times(K+M)+l/8$

字典压缩：$S\_D=D\times L+2\times l$

位向量：　$S\_B=D\times L+l/8\times L$

空值悬挂：$S\_N=0$

注释：空值悬挂只适合 rows =null_items,即该列都为空

D:data_length　　l:rows　　L:all_diff_items　　M:diff_items

K:same_items　　N:null_items

3.1.4 数据压缩模块

该模块中，使用 5 中压缩算法，其中比特压缩可以和其他算法混合使用。

1）游程压缩思路：

游程编码算法是比较适合列数据库的压缩算法之一,即用一个三元组记录数据值、数据出现的起始位置和持续长度(即行程) ,以代替具有相同值的若干连续原始数据,使三元组的存储长度少于原始数据的长度。三元组描述为(X, Y, Z) : X:数据的值, Y: 起始位置, Z:长度。

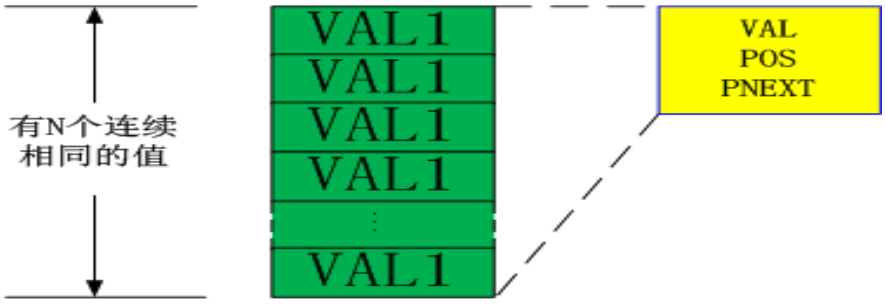

图 5：游程压缩原理图

设 pos 使用 int 的数据类型来表示，那么在 32 位机器上 int 占 32 个字节，但是 64 位机占 64bit 字节。这种不一致性，存在一个危险，使得 64 比特机器使用该压缩算法的效率降低。所以可以限制系统的每个表格的长度，使得表格的长度不超过 2W 行记录。当实际表格查过这个阀值时，把该表格切割成小的表格，进行管理。

每个节点都是三元组（x1, y1, z1） (x2, y2, z2),而 y1 = z2-z1，这种隐藏的算术关系可以让我们省略掉 Z 字段，或者用来表示其他用途。

由于列存储使得列之间缺少关联，我们利用 Z 字段来增加一个 pnext 域，来增加这种关联，使得游程压缩的列的关联性更好：

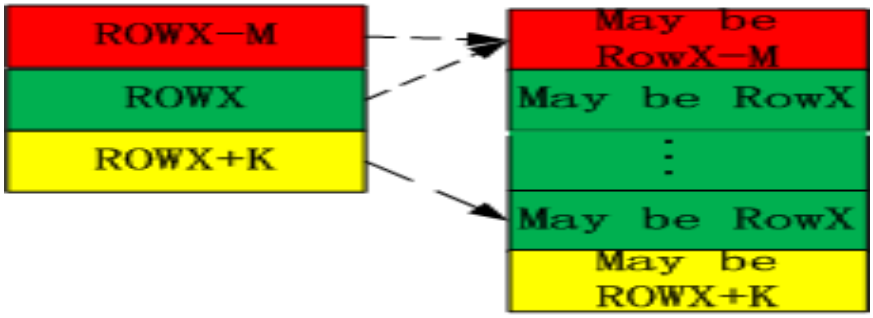

图 6：游程压缩的列的关联图

2）字典压缩思路：

词典编码就是生成一个 原始值 替代值 的对照词典。为了起到压缩的作用，替代值的长度小于原始值的长度。存储的时候,只存储替代值而不是原始值，从而压缩了存储空间。

算法描述如下:

T1: 原始表(其中, T1 有 n 行)

DIC: 词典表(关于表 T 1 的第 k 列的词典)

T2: 压缩存储表

N: 原始值

M: 替代压缩值

For(0 <= i < n)

Begin

扫描 T 1 中第 k 列的第 i 个值得到原始值 N ;

查找 DIC 中 N 对应的压缩值 M ;

将压缩值 M 作为 T 2 中第 k 列的第 i 个值存储;

End.

3）位图压缩思路：

位向量编码是为每一个不同的取值生成一个位向量，根据位向量(串)中不同的位置取值 0 或 1 来对应并确定不同的原始值。

算法描述如下:

T 1: 原始表中的某一列(其中, T 1 有 n 行)

V : T1 的取值空间(无重复元素)

M: 原始值

m: V 的模数(V 中的值的个数)

BVi (0<= i < m) : V 中不同取值对应的位向量

For(0 <= i < n)

Begin

扫描 T 1 中的第 i 个值 M ;

将 M 加入取值空间 V;

End;

计算 V 的模 m;

For(0<= i < m)

初始化位向量 BVi;

// m 个位向量分别对应 m 个不同的取值

For(0 <= i < n)

Begin

扫描 T 1 中的第 i 个值 M ;

确定 M 对应的位向量 BVx;

将位向量 BVx 的第 i 位值置为 1;

For(0 <= j < m)

if (j != x) then 将位向量 BVj 的第 i 位值置为 0;

End.

4）比特压缩思路：

对于某些列，例如有序，或者该列存在很多的空值，可以这样的策略，分配一个数组，长度 L = rows/8; 该数组的每一位表示一个该值。

对于有序的 int：bit=1 的位置说明该值存在，当 bit=0 说明位置对应的值不存在。

对于样本值很少的，使用 bit=1 来代表该值存在 bit=0 代表该值不存在。

5）空值悬挂

对于一整列为空值的列，我们使用一个特殊的字段来表示该列。

3.1.5 数据操作模块

本模块包括数据的增删改查，并且对于数据的查询操作需要分两种情况进行讨论，即压缩态的数据查询和原始态数据的查询。对于数据的增加操作。本系统采用一种预留空间的做法：使用红黑树来组织预留的空间（存储十条记录，并且是基于列式存储方案的），对于数据删除操作，系统采用在主键值位置的 bit 数组上 set bitlist[pos]=0.这样的方法来记录该记录被删除。对于数据的修改操作，采用的方法是先删除再添加的方法

- **数据的查询操作:**

（1） 对压缩态数据的查询：

对压缩态数据的查询，我们都采用主键值得形式进行数据的查询。压缩态数据查询的流程图如下图所示：

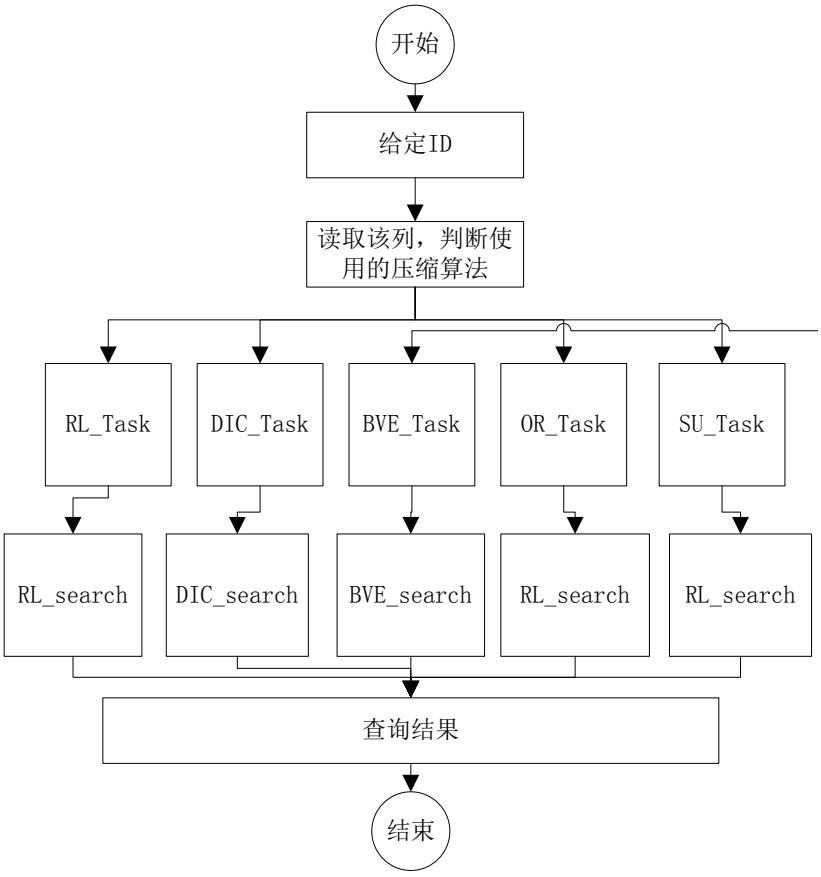

图 7：压缩态数据的查询流程图

■ 游程压缩：

对于第一列，使用二分算法来找到对饮的 pos 位置.并且返回对应 pos 的值，然后获取下一列的查找范围:start end.

具体流程如下：

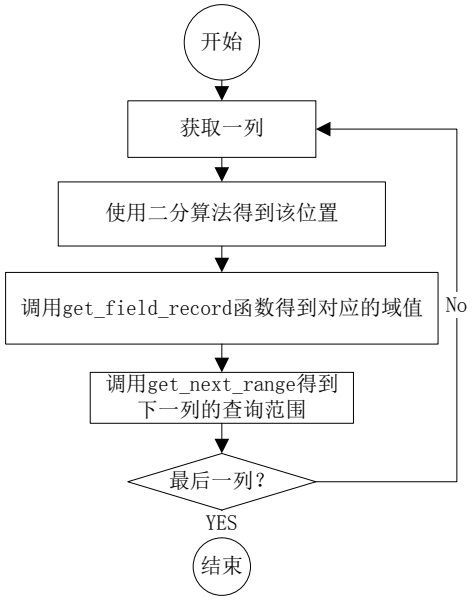

图 8：游程压缩的数据的查询流程图

关键代码如下：

```
static int get_field_record(const pKColumn current,const int id ,char * field , const int from, const int to)
{
    int res = ERR ;
    if(OK == IsInNullList(current,id)){
        field[0]='\0';
        return -2;
    }else{
        res = SearchPos(current,id,from,to);
        if(res != ERR){
            GetUnitDscr(current,field,res);
        }else{
            debug("error: search pos in ulist");
            return ERR;
        }
    }
    return res;
}
```

获取下一列的核心操作如下：

```
Type * _ulist = ulist;
    *from = _ulist[pos].pnext;
    if(pos==rows-1){
        *to =-1;
    }else{
        *to = _ulist[pos+1].pnext;
    }
```

■ 字典压缩：

从 Task 里面获取使用字典压缩的列，从而形成新的表格：

伪代码如下所示：

```
Begin:
For 0<=I < slen
   get_one_field(id ,field, ref_diction);
end
```

而 get_one_field 的核心思路是：

```
Input: id
Begin:
Dic_pos = dictable[id];
Value  = Diction[dic_pos];
End.
```

■ 位图压缩：

其操作和位图操作类似，首先需要确定主键值为 ID 的纪录是否存在：

伪代码如下所示：

```
    /*find if exits*/
    int i =0 ;
    for( i = 0 ; i < tlen ; i++){
        if(1==check_bit(vlist[i],id,vlen*8)){
            tid=i;
        }
    }
```

当确认该主键值存在的情况下，得到对饮的字典值。

（2） 对原始数据的查询：

对于原始数据的查询操作，我们采用红黑树的树性结构进行查询。输入主键值 ID，根据其 ID 查询该记录所在的子表，即红黑树相应节点中（ID/10 号节点），由于每个节点指向 10 条记录的子表中，再根据 ID%10 的结果查询到相应记录，返回查询结果。

关键代码如下：

```
//该函数是对红黑树内核代码的封装。
struct mynode * search(struct rb_root *root, char *string)
{
    struct rb_node *node = root->rb_node;

    while (node) {
        struct mynode *data = container_of(node, struct mynode, node);
        int result;

        result = strcmp(string, data->string);

        if (result < 0)
            node = node->rb_left;
        else if (result > 0)
            node = node->rb_right;
        else
            return data;
    }
    return NULL;
}
//该函数是对上面函数的简单封装。根据 id 查询相应记录。
struct mynode * my_search(struct rb_root *root, int id)
{
    //int id=records/10;
    char str[10];
    sprintf(str, "%d", id);
    struct mynode *data=search(&mytree,str);
    return data;

}
```

采用红黑树的树形结构对表格进行管理如下图所示：

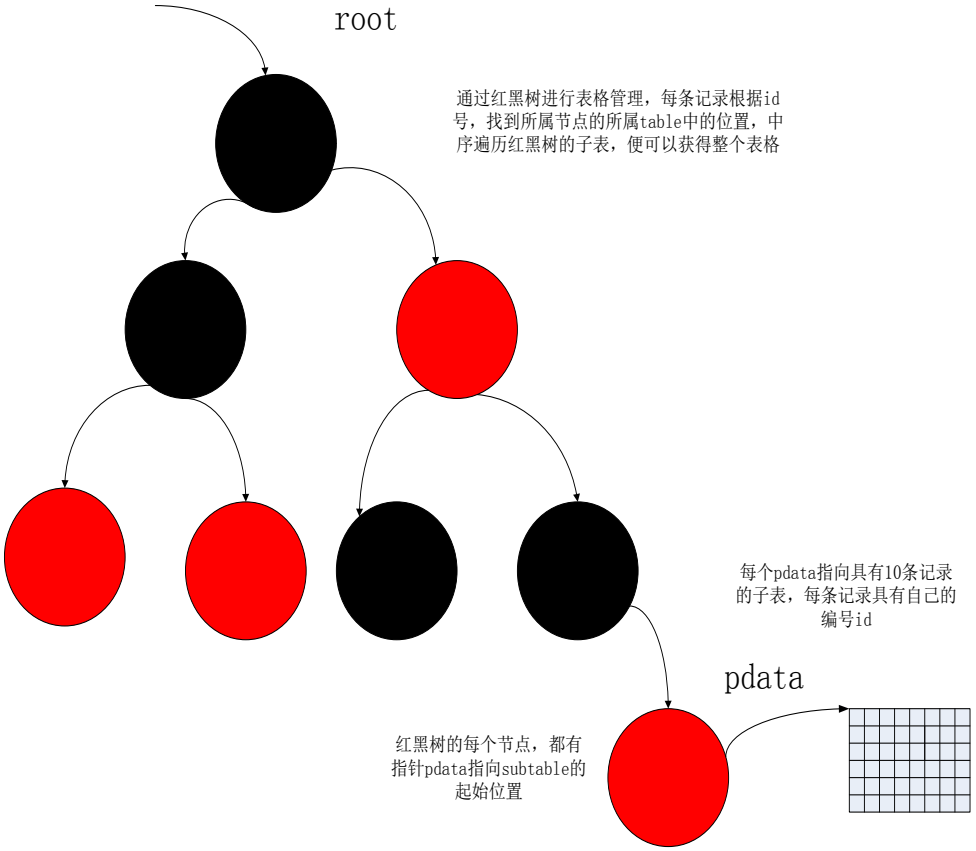

图 9：红黑树的表格管理

当我们中序遍历一遍红黑树所有节点指向的子表，我们就可以获得整个表格数据。同时通过该结构我们可以快速地查询和插入数据。

该红黑树的结构如下：

```
//红黑树节点的基本信息。
struct rb_node
{
    unsigned long  rb_parent_color;
#define  RB_RED      0
#define  RB_BLACK    1
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
```

```
//我们对红黑树节点信息域的扩展，增加了三个域，分别是：
int from，to:表示该节点的子表中可以存放的记录的范围;
void *pdata:指向子表的开始位置。

typedef struct mynode {
    struct rb_node node;
    char *string;

    int from;
    int to;
    void* pdata;        //pointer of   a subtable
}Node, *pNode;



//红黑树根节点的基本信息,我们也对根节点信息域扩展了三个域，分别是：
void *pcurrent：指向最新插入的子表；
int cols：      表示子表的列数；
int records：   当前表的记录总数。

struct rb_root
{
    struct rb_node *rb_node;

    void *pcurrent;        // address of latest added subtable
    int cols;              // column number of subtable
    int records;           // records of this table contains
};
```

- **数据的追加操作：**

当没有把 CSV 文件导入内存时，红黑树为空，没有任何节点；当数据导入内存，每条记录根据其 ID 判断插入到相应节点中（ID/10 号节点），由于每个节点指向 10 条记录的子表中，再根据 ID%10 的结果插入到子表的相应位置。

关键代码如下：

```
int insert(struct rb_root *root, struct mynode *data)
{
    struct rb_node **new = &(root->rb_node), *parent = NULL;

    /* Figure out where to put new node */
    while (*new) {
        struct mynode *this = container_of(*new, struct mynode, node);
        int result = strcmp(data->string, this->string);

        parent = *new;
        if (result < 0)
            new = &((*new)->rb_left);
        else if (result > 0)
            new = &((*new)->rb_right);
        else
            return 0;
    }

    /* Add new node and rebalance tree. */
    rb_link_node(&data->node, parent, new);
    rb_insert_color(&data->node, root);

    return 1;
}

//该函数是对上面函数的简单封装。
int my_insert(struct rb_root *root, int id)
{
    struct mynode *node;
    node= (struct mynode *)malloc(sizeof(struct mynode));
    node->string = (char *)malloc(sizeof(char) * 10);
    sprintf(node->string, "%d", id);
    if( insert(&mytree, node) )
    {
        root->pcurrent=node;
        node->from=root->records;
        node->to=node->from;
        //node->pdata=
        return 1;
    }
    else
        return 0;
}
```