Project Report

Jxx133

Project Goal

1. Implement a Bed tree based on two kinds of String order. i.e. dictionary order and gram counting order
2. Use range search query and two different datasets (one consist of long string and another consists of short string ) to measure the execution performance of bed tree in difference string order.

Datasets

1. All SIGMOD paper titles from DBLP

There are 8734 titles and many of them are duplicate titles.

1. All author names from DBLP

There are 1211034 names and none of them are duplicates.

As duplicate names are filtered during my implementation.

Implementation details

My implementation codes consists of five parts logically:

1.The DBLP xml file (which consists of all data in DBLP sites) parser

It retrieves the two datasets used in my implementation.

2.The conventional B plus tree, which can be of any degree

Inserting data from step 1 is implemented here.

3.The dictionary order implementation

It consists of range query function, upper bound function and edit distance dynamic programming calculation function.

4.The gram counting order implementation

It consists of range query function, upper bound function, z-order value calculation function (Raw String -> generate 2-gram->calculate hash(each gram)->Generate 4 bucket integer vector->generate z-order value)

5. Test case functions

It consists four test cases.

Detail description in every part:

1.DBLP xml parser

The parser library I used is Apache Xerces2 Java instead of the one in java standard library because the one in standard library don’t support for such a big xml file.

The core part code of my parser to parse the author names is the following:

**private** **class** ConfigHandler **extends** DefaultHandler {

(some codes omitted here)

**public** **void** startElement(String namespaceURI, String localName,

String rawName, Attributes atts) **throws** SAXException {

String k;

**if** (insidePerson = (rawName.equals("author") )) {

Value = "";

**return**;

}

}

**public** **void** endElement(String namespaceURI, String localName,

String rawName) **throws** SAXException {

**if** (rawName.equals("author")) {

Person p;

**if** (Person.*searchPerson*(Value)==**false**) {

p = **new** Person(Value);

}

**return**;

}

}

**public** **void** characters(**char**[] ch, **int** start, **int** length)

**throws** SAXException {

**if** (insidePerson)

Value += **new** String(ch, start, length);

}

(some codes omitted here ...)

}

The idea is simple here. Find every content in tag like this “<author>Herón Molina Lozano</author>”. Use the content to construct a object person.

At the same time the content will be inserted into a hash table, which consists of more than 1million names finally. Before inserting any name entry into the hash table, I will check if there is such an entry already there.

The person class is defined like this.

**public** **class** Person {

**private** **static** **int** *maxPublCount* = 0;

**private** **static** **int** *maxNameLength* = 0;

**private** **static** HashSet<String> *personMap* = **new** HashSet<String>();

**public** Person(String name) {

*personMap*.add(name);

}

**static** **public** Iterator<String> iterator() {

**return** *personMap*.iterator();

}

**static** **public** **boolean** searchPerson(String name) {

**return** *personMap*.contains(name);

}

**static** **public** **int** numberOfPersons() {

**return** *personMap*.size();

}

}

personMap is the hash table to store 1 million names.

The way to extract the paper titles are similar, and class Publication is used to deal with and store all the SIGMOD paper titles strings.

2. The conventional B plus tree details.

First the skeleton of my tree node definition is given:

**class** treeNode

{

E[] key;

treeNode[] child;

**int** ksize;

**int** csize;

**boolean** leaf;

**int** height;

**public** treeNode(**boolean** leaf)

{

key=(E[])**new** Object[2\*degree];

child=(treeNode[])**new** BPlusTree.treeNode[2\*degree];

ksize=0;

csize=0;

height=0;

**this**.leaf=leaf;

}

**public** String toString()

{

codes omitted here

}

}

Among the codes above, ksize and csize are the current key number and child pointer number in the tree node. There is check mechanism in the insert function to make sure child pointer number cannot exceed the degree and key number cannot exceed the degree minus 1.

The following is the B plus tree skeleton, let’s focus on the constructor function.

**public** **class** BPlusTree<E> {

codes omitted here

LinkedList<String> result=**new** LinkedList<String>();

LinkedList<GCinstance> result2=**new** LinkedList<GCinstance>();;

**public** BPlusTree(**int** degree, Comparator<? **super** E> comparator,**int** queryType)

{

**this**.comparator=comparator;

**this**.degree=degree;

minDegree=(degree+1)/2;

minKeys=minDegree-1;

root=**new** treeNode(**true**);

**this**.queryType=queryType;

}

codes omitted here

}

degree is the most number of child pointer in a tree node. Comparator is used to compare the keys. Keys type is <E>. And keys are defined different order so the comparator needs to be different. result are the container to store the results in dictionary order scenario. result2 is the container to store results in gram counting order scenario.

The core functions of B plus tree is insert(), insertIntoLeaf(), splitNode() their implementation codes are very long, and hard to compress, please see the in BPlusTree.java file. The main idea is every inserting entry first find which leaf and the position in the target leaf it should locate, insert it no matter whether the leaf overflows. Eevery parents during the search down process is pushed into a stack for the following split process. The splitNode() function is responsible for this job. For the leaf node, some key should be “copied up”. For internal node, some key should be “pushed up”.

3.The dictionary order implementation

The range query pseudo code and explicit explanation is given in the paper.So I don’t list the rangeQuery() codes here.

The lower bound implementation needs some illustration here. I use my real code to explain.

//calculate the lower bound value in dictionary order

**public** **boolean** lowerBound(**char**[] query, **char**[] min, **char**[] max, **int** threshold)

{

**int** i=0;

**char** lastMin;

**char** lastMax;

//the index that characters are first different in min and max

**int** smaller;

smaller=min.length<max.length?min.length:max.length;

**if**((**new** String(min)).equals(**new** String(max)))

{

**return** EDcheck(min,query,threshold);

}

**for**(i=0;i<smaller;i++)

{

**if**(min[i]!=max[i])

**break**;

}

//use lastMin as the lowest character in unicode

**if**(i==smaller)

{

lastMin='"';

lastMax=max[i];

}

**else**

{

lastMin=min[i];

lastMax=max[i];

}

//row is query string,column is prefix common string plus null and lastmin or last max

**int**[][] matrix=**new** **int**[i+2][query.length+1];

**int** rows=i+2;

**int** columns=query.length+1;

**int** d1,d2,d3;

**char**[] preComChar=**new** **char**[i];

**for**(i=0;i<preComChar.length;i++)

{

preComChar[i]=min[i];

}

**int** j=0;

**for**(j=0;j<columns;j++)

{

matrix[0][j]=j;

}

**for**(i=0;i<rows;i++)

{

matrix[i][0]=i;

}

//dynamic programming to calculate distance

**int**[] distArray=**new** **int**[3];

**for**(i=1;i<rows;i++)

{

**for**(j=1;j<columns;j++)

{

d1=matrix[i][j-1]+1;

**if**(i!=(rows-1))

{

**if**(preComChar[i-1]==query[j-1])

{

d2=matrix[i-1][j-1];

}

**else**

{

d2=matrix[i-1][j-1]+1;

}

}

**else**

{

**if**((query[j-1]<=lastMax)||(query[j-1]>=lastMin))

{

d2=matrix[i-1][j-1];

}

**else**

{

d2=matrix[i-1][j-1]+1;

}

}

d3=matrix[i-1][j]+1;

distArray[0]=d1;

distArray[1]=d2;

distArray[2]=d3;

matrix[i][j]=smallest(distArray);

}

}

**int**[] lastRow=**new** **int**[query.length+1];

**for**(i=0;i<query.length+1;i++)

{

lastRow[i]=matrix[rows-1][i];

}

**if**(smallest(lastRow)<=threshold)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

The main idea is first find the longest common prefix of the minimal key(min) and maximal key(max). And the first different characters should be recorded. That is lastMin and lastMax in my codes. Once max totally corporates min, then lastMin should use a smallest character in the dictionary order. Here I use double quotation mark as the smallest character in Unicode(though it’s actually not but small enoguth in this experiment). Then do normal dynamic programming to calculate edit distance with query string and this special string “prefix common string”+”lastMin or lastMax”(depends on which let the edit distance smaller). And use the smallest value in last row to compare with the threshold.

4. gram counting order implementation detail

For gram counting order, first I want to introduce class GCinstance, which used as key type”<E>” in the BPlusTree class when using gram counting order.

The constructor is given like this:

**public** GCinstance(String s)

{

store=**new** LinkedList<String>();

store.add(s);

gramNumber=gramHashVector(s,gramL);

zvalue=zorder(gramNumber);

}

In every GCinstance, we use a linkd list to store all strings which shares the same z-order value. And this list is initialized with one stirng. There is a merge function to merge GC instance with the same z order value.

This is the hash function used in my implementation. It hashed all 2-gram of one string into 4 buckets.

**public** **int** innerHash(String gram)

{

**int** i=0;

**char**[] temp=gram.toCharArray();

**int** hashValue=temp[0];

**for**(i=1;i<temp.length;i++)

{

hashValue=37\*hashValue+temp[i];

}

**return** hashValue%4;

}

The function to generate four gram counts bucket is easy, so its omitted to list here. It’s called gramHashVector(String s, int n) in the gramInstance class.

The zvalue calculate is also easy but there is one tricky part—the length of binary presentation of each bucket is different. For eample, <5,3,7,15>, the fisrt three integer needs three bits to represent each, the last one need 4 bits. So we need to fulfill zeros at the head of first three binary presentation. That is 4 bits to represent hem each. There is similar problem in the lower bounding function, different vector can have different length z-value binary presentation and we need to fulfill zeros into the shorter z-value binary presentation also.

You can check zorder(int[] buckets) in GCinstance class for detail, it is a bit long.

Now I want to introduce this pair of functions:

**public** **int**[] hbVector(**char**[] comPrefix, **int** bitsPerBucket)

**public** **int**[] lbVector(**char**[] comPrefix, **int** bitsPerBucke)

This can be used when we ensure the low z value and high z value in a pair of keys in one B plus node have the same bits to represent.(through the means of fulfill head with 0s). bitsPerBucket are then the length of binary presentation z value divided by 4. comPrefix is the common prefix binary sequence of these two z value binary presentation. What these two function to do is fufill the uncertain position with all 0s or 1s, and return the decimal presentation of gram counting buckets. i.e. integer vectors of 4 buckets. Here is the lbVector code as an example:

**public** **int**[] lbVector(**char**[] comPrefix, **int** bitsPerBucket)

{

**if**(comPrefix==**null**)

{

**return** **new** **int**[4];

}

**char**[][] lbBuckets=**new** **char**[4][bitsPerBucket];

**int** i=0;

**int** j=0;

**int** row;

**int** column;

**for**(i=0;i<4;i++)

{

**for**(j=0;j<bitsPerBucket;j++)

{

lbBuckets[i][j]='0';

}

}

**for**(i=0;i<comPrefix.length;i++)

{

row=i%4;

column=i/4;

lbBuckets[row][column]=comPrefix[i];

}

**int**[] gramSignature=**new** **int**[4];

String intoBucket;

**for**(i=0;i<4;i++)

{

intoBucket=**new** String(lbBuckets[i]);

gramSignature[i]=Integer.*parseInt*(intoBucket,2);

}

**return** gramSignature;

}

lbBuckets are the gramSignature ‘s binary presentation. gramSignature is the 4 gram counting buckets.

Another critical function is the following:

//check minimal edit distance between query vector and low vector, high vector pair

**public** **int** minPossibleED(**int**[] low, **int**[] high, **int**[] q, **int** n)

{

**int** i=0;

**int** ed=0;

**int** diff,diff1,diff2;

**for**(i=0;i<4;i++)

{

**if**((q[i]<low[i])||(q[i]>high[i]))

{

diff1=Math.*abs*(low[i]-q[i]);

diff2=Math.*abs*(q[i]-high[i]);

diff=diff1<diff2?diff1:diff2;

}

**else**

{

diff=0;

}

ed=ed+diff;

}

**return** ed/n;

}

The function is to calculate the minimal possible edit distance with a pair of keys.

Low vector and high vectoris their corresponding gram counting vectors which can be computed use lbVector and hbVector I talke before.

Now the gram counting order ‘s lower bounding function can be introduced. I don’t list it here as it is long. The specific of this function is to invoke the above functions to judge whether we need to search the child. Also before invoking the above function it also ensure all pair of z values will be translated into binary presentation of same length.

The range search function of gram counting order is very similar to the pseudo code in the paper, so it’s not listed here.

5. Test case part

Test.class and Test1.class use SIGMOD titles as its dataset. Test uses string order. Test1 uses gram counting order.

Test and Test1 use four different queries with different range query threshold.

Original String: Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

Four fuzzy query String:

1. Proceedings of the ACM SIGMOD international Conference on Management of Data, Beijing, China, June 12-14, 2007

Proceedings of the AAM SIGMOD Internetional Conference on Management of Data, Beijing, China, June 12-14, 2007

3. Proceedings of the AAM SIGMOD nternational Conferenca on Management of Data, Beijing, China, June 22-14, 2007

4. Procaedings f the ACM SIGMQD Internttional Confarence on Management of Data, Beijing, China, June 12-14, 2907

Edit distance above are 1,2,4,6 respectively

Test2.class and Test3.class use DBLP names as its dataset. Test2 uses string order and Test3 uses gram counting order.

Test2 and Test3 use four different queries with same threshold 1.

Origianl string Timo Partala

Fuzzy query:

Tamo Partala

Timo Partalb

TimoPartala

Timo Paartala

Experiment raw data, statistic and analysis

Raw data

Test

starting time: 1429906961933

number of titles inserted: 8734

(duplicate titles will try to be inserted, but will not be inserted at leaf level)

Building time: 38

This range query execution time: 103

leafVisited: 277

fuzzy query: Proceedings of the ACM SIGMOD international Conference on Management of Data, Beijing, China, June 12-14, 2007

fuzzy query result:

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

This range query execution time: 502

leafVisited: 483

fuzzy query: Proceedings of the AAM SIGMOD Internetional Conference on Management of Data, Beijing, China, June 12-14, 2007

fuzzy query result:

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

This range query execution time: 103

leafVisited: 715

fuzzy query: Proceedings of the AAM SIGMOD nternational Conferenca on Management of Data, Beijing, China, June 22-14, 2007

fuzzy query result:

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

This range query execution time: 214

leafVisited: 864

fuzzy query: Procaedings f the ACM SIGMQD Internttional Confarence on Management of Data, Beijing, China, June 12-14, 2907

fuzzy query result:

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

Test1

Starting time: 1429907556320

8734

Building time: 273

this range query execution time: 0

fuzzy query: Proceedings of the ACM SIGMOD international Conference on Management of Data, Beijing, China, June 12-14, 2007

fuzzy query result:

2016861

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

leaves visited: 12

number of Keys(zorder value) matches the query: 1 number of strings in the match leaves: 1

this range query execution time: 3

fuzzy query: Proceedings of the AAM SIGMOD Internetional Conference on Management of Data, Beijing, China, June 12-14, 2007

fuzzy query result:

2016861

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

5908616

leaves visited: 33

number of Keys(zorder value) matches the query: 2 number of strings in the match leaves: 4

this range query execution time: 1

fuzzy query: Proceedings of the AAM SIGMOD nternational Conferenca on Management of Data, Beijing, China, June 22-14, 2007

fuzzy query result:

1007610

2016861

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

5900845

5908616

leaves visited: 40

number of Keys(zorder value) matches the query: 4 number of strings in the match leaves: 6

this range query execution time: 1

fuzzy query: Procaedings f the ACM SIGMQD Internttional Confarence on Management of Data, Beijing, China, June 12-14, 2907

fuzzy query result:

884727

1007431

1007610

1039575

1046216

1984634

1990856

2015522

2016861

Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007

4799415

4920093

4920186

4953264

4954207

4960735

4961624

4968760

5900661

5900845

5900872

5902037

5908616

5931760

8869221

12783872

leaves visited: 100

number of Keys(zorder value) matches the query: 25 number of strings in the match leaves: 58

Test2

Starting time: 1429907169765

Total number of names inserted: 1211034

Building time: 5784

This range query execution time: 122

leafVisited: 2245

fuzzy query: Tamo Partala

fuzzy query result:

Timo Partala

This range query execution time: 30

leafVisited: 2230

fuzzy query: Timo Partalb

fuzzy query result:

Timo Partala

This range query execution time: 27

leafVisited: 2106

fuzzy query: TimoPartala

fuzzy query result:

Timo Partala

This range query execution time: 34

leafVisited: 2230

fuzzy query: Timo Paartala

fuzzy query result:

Timo Partala

Test3

Starting time: 1429907441772

Total number of names inserted: 1211034

Building time: 4204

This range query execution time: 5

fuzzy query: Tamo Partala

fuzzy query result:

1183

1205

1207

1213

Timo Partala

1215

1272

1448

3095

3125

leaves visited: 153

number of Keys(zorder value) matches the query: 9 number of strings in the match leaves: 27152

This range query execution time: 3

fuzzy query: Timo Partalb

fuzzy query result:

1087

1205

1207

1213

Timo Partala

1215

1266

1442

1565

1685

leaves visited: 159

number of Keys(zorder value) matches the query: 9 number of strings in the match leaves: 30464

This range query execution time: 2

fuzzy query: TimoPartala

fuzzy query result:

1085

1087

1175

1183

1205

1207

1213

Timo Partala

1264

1440

leaves visited: 143

number of Keys(zorder value) matches the query: 9 number of strings in the match leaves: 24970

This range query execution time: 2

fuzzy query: Timo Paartala

fuzzy query result:

1207

1213

Timo Partala

1215

1274

1450

1685

1693

3125

3127

leaves visited: 167

number of Keys(zorder value) matches the query: 9 number of strings in the match leaves: 28984

Statistics and Analysis:

First is the building time analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Test(dict) | Test1(gram) | Test2(dict) | Test3(gram) |
| 38 | 273 | 5784 | 4204 |

Analysis

For test VS test1,

Dictionary order has shorter building time.

Possible reason:

Building Gram counting order based tree has great preprocessing overhead. This includes: Raw String -> generate n-gram->calculate hash(each gram)->

Generate 4 bucket integer vector->generate z-order value

For test2 VS test3,

Gram counting order has shorter building time.

Reason: names are short strings. Thus names has much less n-grams so that gram counting has much less overhead. While dictionary order still needs to compute edit distance for every pair of strings.

Range query time comparison(dict VS gram)

(Test VS Test1)

1. Proceedings of the ACM SIGMOD international Conference on Management of Data, Beijing, China, June 12-14, 2007

Proceedings of the AAM SIGMOD Internetional Conference on Management of Data, Beijing, China, June 12-14, 2007

3. Proceedings of the AAM SIGMOD nternational Conferenca on Management of Data, Beijing, China, June 22-14, 2007

4. Procaedings f the ACM SIGMQD Internttional Confarence on Management of Data, Beijing, China, June 12-14, 2907

|  |  |  |  |
| --- | --- | --- | --- |
| Fuzzy query # | Range query threshold | Dictionary time | Gram counting time |
| 1 | 1 | 103 | Less than 1 |
| 2 | 2 | 502 | 3 |
| 3 | 4 | 103 | 1 |
| 4 | 6 | 214 | 1 |

Unit :milliseconds

Test2 VS Test3

Origianl string Timo Partala

Fuzzy query :

5.Tamo Partala

6.Timo Partalb

7.TimoPartala

8.Timo Paartala

|  |  |  |  |
| --- | --- | --- | --- |
| Fuzzy query # | Range query threshold | Dictionary time | Gram counting time |
| 5 | 1 | 122 | 5 |
| 6 | 1 | 30 | 3 |
| 7 | 1 | 27 | 2 |
| 8 | 1 | 34 | 2 |

Unit :milliseconds

Analysis:

at every threshold, search time of dictionary order is much slower than that of gram counting order

Reason: Dynamic Programming to calculate edit distance between two strings VS integer comparison, the former takes much more time

Different thresholds apply to same fuzzy query(dictionary order)  
Proceedings of the AAM SIGMOD Internetional Conference on Management of Data, Beijing, China, June 12-14, 2007

|  |  |  |
| --- | --- | --- |
| threshold | Leaves visited | time |
| 2 | 481 | 135 |
| 4 | 715 | 373 |
| 8 | 991 | 126 |
| 12 | 1049 | 102 |

Higher threshold incurs that range search check more leaves

Ratio of the leaves containing the strings we want VS the total number of matched leaves

Remember Test1 uses 4 queries of titles with different threshold

Test 3 uses 4 queries of names with same threshold

Both in gram counting order, and have use different datasets

|  |  |  |  |
| --- | --- | --- | --- |
| Test1 | Threshold | Test3 | Threshold(Test3) |
| 100% | 1 | 11% | 1 |
| 25% | 2 | 11% | 1 |
| 17% | 4 | 11% | 1 |
| 4% | 6 | 11% | 1 |

The smaller ratio it is, the more leaves we find satisfies our query.

There is deseeding in the left column, the direct reason is the range query threshold is 1,2,4,6 in increasing order. As the threshold is larger, it’s not weird that more leaves satisfy the query.

The right column all has threshold 1. But we see, all of them only have 11% compared to the first item in left column(whose query also has threshold one).

Same threshold but lower ratio, that’s because left column uses long strings and right uses short strings. Long string have more variance in length than short string.

The gram counting bucket vector then has more variance in long string. Thus it’s less likely to find leaves we actually don’t want.