实验十四 范式分解

姓名	学号	学院	日期
臧祝利	202011998088	人工智能学院	2022.11.22

实验目的

加深理解 `3NF` 范式分解。

实验内容

编写程序实现 `3NF` 范式分解。

程序输入：关系模式 `R` 及 `R` 上的函数依赖集 `F`

程序输出：`R` 的 `3NF` 无损连接和保持函数依赖的分解结果

实验思路

全局变量：

```
typedef pair<string, string> PSS;
string R;                       //关系模式
vector<PSS> F;                  //函数依赖集
vector<string> subset;          //关系模式R的所有子集
vector<string> candidate_key;   //候选键
vector<string> super_key;       //超键
char *tmp;   //辅助变量，用于创建数组
```

范式分解过程如下：

- 计算 `F` 的最小函数依赖集
    - 将所有函数依赖右边变成单个属性
    - 去掉F中多余的函数依赖
    - 去掉各函数依赖左边多余的属性

```
vector<PSS> get_min_relyset(const vector<PSS> &F)
{
    vector<PSS> G = F;
    //第一步：使得右边元素都为单属性；
    for (int i = 0; i < G.size(); i++)
    {
        if (G[i].second.size() > 1)  //如果长度大于1，进行分解
        {
            string f = G[i].first, s = G[i].second;
            string tmp;
            G[i].second = s[0];  //当前函数依赖变成第一个属性
            for (int j = 1; j < s.size(); j++)
            {
                tmp = s[j];
                G.push_back(make_pair(f, tmp));  //加入函数依赖
            }
        }
    }

    int Max = 0;
```

```
for (int i = 0; i < G.size(); i++)
{
    Max = max((int)G[i].first.size(), Max);   //找到最长的左侧依赖长度，方便申请空间；
}
bool *del = new bool[Max];

//第二步：去掉左边多余的属性
for (int i = 0; i < G.size(); i++)
{
    if (G[i].first.size() > 1)   //左侧不是单属性
    {
        fill(del, del + G[i].first.length(), 0);   //初始化数组
        for (int j = 0; j < G[i].first.size(); j++)   //遍历每个单属性
        {
            string tmp;
            del[j] = 1;   //标记，判断是否
            for (int k = 0; k < G[i].first.size(); k++)
            {
                if (!del[k])
                {
                    tmp += G[i].first[k];
                }
            }
            if (!iscontain(get_closure(tmp, G), G[i].second))
            {
                del[j] = 0;
            }
        }
        string tmp;
        for (int j = 0; j < G[i].first.size(); j++)
        {
            if (!del[j])
            {
                tmp += G[i].first[j];
            }
        }
        G[i].first = tmp;
    }
}
delete[] del;
del = NULL;

sort(G.begin(), G.end());
G.erase(unique(G.begin(), G.end()), G.end());

//第三步：去除冗余的函数依赖
vector<PSS> ans;
for (int i = 0; i < G.size(); i++)
{
    vector<PSS> tmp = G;
    tmp.erase(tmp.begin() + i);
    if (!iscontain(get_closure(G[i].first, tmp), G[i].second))
    {
        ans.push_back(G[i]);
    }
}
return ans;
}
```

- 合并左部相同的依赖
- 每个函数依赖构成模式
- 在构成的模式集中，如果每个模式都不包含候选键，那么把候选键当作模式放入模式集中

```
vector<string> Transform_3NF(const string &R, const vector<PSS> &F)
{
    vector<PSS> FF = get_min_relyset(F);
    map<string, string> mp;
    for (int i = 0; i < FF.size(); i++)
    {
        if (mp.find(FF[i].first) == mp.end())
        {
            mp[FF[i].first] = FF[i].second;
        }
        else
        {
            mp[FF[i].first] += FF[i].second;
        }
    }
    FF.resize(mp.size());
    int idx = 0;
    map<string, string>::iterator it;
    for (it = mp.begin(); it != mp.end(); it++)
    {
        FF[idx].first = it->first;
        FF[idx++].second = it->second;
    }

    vector<string> P;
    for (int i = 0; i < FF.size(); i++)
    {
        P.push_back(FF[i].first + FF[i].second);
    }
    get_all_candidate_key(R, F);
    for (int i = 0; i < candidate_key.size(); i++)
    {
        int flag = 0;
        for (int j = 0; j < P.size(); j++)
        {
            if (iscontain(P[j], candidate_key[i]))
            {
                flag = 1;
                break;
            }
        }
        if (!flag)
        {
            P.push_back(candidate_key[i]);
        }
    }
    sort(P.begin(), P.end());
    P.erase(unique(P.begin(), P.end()), P.end());
    return P;
}
```

实验结果

> 样例1：

> 输入：
>
> ABCDE
> 3
> A->BC
> ABD->CE
> E->D
>
> 输出：
>
> ABC
> ADE
> ED

> 样例2：
>
> 输入：
>
> ABCDEF
> 2
> AB->CD
> BC->EF
>
> 输出：
>
> ABCD
> BCEF

源代码

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<string, string> PSS;

string R;                       //关系模式
vector<PSS> F;                  //函数依赖集
vector<string> subset;          //关系模式R的所有子集
vector<string> candidate_key;   //候选键
vector<string> super_key;       //超键
char *tmp;                      //辅助变量，用于创建数组

/*
function:查看s2是否包含在s1中
params:2 string
return: 0/1;
*/
bool iscontain(string s1, string s2)
{
    map<char, int> mp1, mp2;
    for (int i = 0; i < s1.size(); i++)
    {
        mp1[s1[i]]++; //统计s1出现的字符及其个数
    }
    for (int i = 0; i < s2.size(); i++)
    {
        mp2[s2[i]]++; //统计s2
    }
    for (int i = 0; i < s2.size(); i++)
    {
        if (mp2[s2[i]] > mp1[s2[i]]) //如果s2在s1内，应该满足s2的任何一个字符个数都小于s1;
        {
            return false;
        }
```

```
    }
    return true;
}

/*
fuction:比较两个集合是否相同
params:2 string
return: y/n
*/
bool isequal(string s1, string s2)
{
    set<char> set1, set2; //使用集合set比较
    for (int i = 0; i < s1.size(); i++)
    {
        set1.insert(s1[i]);
    }
    for (int i = 0; i < s2.size(); i++)
    {
        set2.insert(s2[i]);
    }
    return set1 == set2;
}

/*
fuction: 得到x的闭包
params: 属性集+函数依赖集
return: 闭包string
*/
string get_closure(const string &x, const vector<PSS> &F)
{
    string ans = x;
    string tmp;
    bool vis[F.size()];
    memset(vis, 0, sizeof vis);
    do
    {
        tmp = ans;
        for (int i = 0; i < F.size(); i++)
        {
            if (!vis[i] && iscontain(ans, F[i].first))
            {
                vis[i] = 1;
                ans += F[i].second;
            }
        }
    } while (tmp ≠ ans);
    sort(ans.begin(), ans.end());
    ans.erase(unique(ans.begin(), ans.end()), ans.end());
    return ans;
}
/*
fuction:深搜求所有子集
params:pos为R的字符下标，cnt为子集的下标，num为还可选的字符数量
*/
void all_subset(int pos, int cnt, int num)
{
    if (num ≤ 0)
    {
        tmp[cnt] = '\0';
        subset.push_back(tmp);
```

```
        return;
    }
    tmp[cnt] = R[pos];
    all_subset(pos + 1, cnt + 1, num - 1);
    all_subset(pos + 1, cnt, num - 1);
}
/*
fuction:求所有子集
params:关系模式
*/
void get_subset(const string &R)
{
    subset.clear();
    tmp = NULL;
    tmp = new char[R.size()];
    all_subset(0, 0, R.size());
    delete[] tmp;
    tmp = NULL;
}
/*
fuction:判断s是否为候选键
params:字符串s
*/
bool iscandidate_key(const string &s)
{
    for (int i = 0; i < candidate_key.size(); i++)
    {
        if (iscontain(s, candidate_key[i]))
        {
            return false;
        }
    }
    return true;
}

/*
fuction:排序的lambda函数
*/
bool cmp(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}

/*
fuction:求关系模式基于F的全部候选键
params:关系模式+函数依赖集
*/
void get_all_candidate_key(const string &R, const vector<PSS> &F)
{
    get_subset(R); //得到R的所有子集
    sort(subset.begin(), subset.end(), cmp);
    candidate_key.clear();
    super_key.clear();

    for (int i = 0; i < subset.size(); i++)
    {
        if (iscontain(get_closure(subset[i], F), R))
        {
            super_key.push_back(subset[i]);
            if (iscandidate_key(subset[i]))
```

```
            {
                candidate_key.push_back(subset[i]);
            }
        }
    }
}

/*
fuction:得到F的最小依赖集
params:函数依赖集F
*/
vector<PSS> get_min_relyset(const vector<PSS> &F)
{
    vector<PSS> G = F;
    //第一步：使得右边元素都为单属性；
    for (int i = 0; i < G.size(); i++)
    {
        if (G[i].second.size() > 1)
        {
            string f = G[i].first, s = G[i].second;
            string tmp;
            G[i].second = s[0];
            for (int j = 1; j < s.size(); j++)
            {
                tmp = s[j];
                G.push_back(make_pair(f, tmp));
            }
        }
    }

    int Max = 0;
    for (int i = 0; i < G.size(); i++)
    {
        Max = max((int)G[i].first.size(), Max);
    }
    bool *del = new bool[Max];

    //第二步：去掉左边多余的属性
    for (int i = 0; i < G.size(); i++)
    {
        if (G[i].first.size() > 1)
        {
            fill(del, del + G[i].first.length(), 0);
            for (int j = 0; j < G[i].first.size(); j++)
            {
                string tmp;
                del[j] = 1;
                for (int k = 0; k < G[i].first.size(); k++)
                {
                    if (!del[k])
                    {
                        tmp += G[i].first[k];
                    }
                }
                if (!iscontain(get_closure(tmp, G), G[i].second))
                {
                    del[j] = 0;
                }
            }
            string tmp;
```

```
            for (int j = 0; j < G[i].first.size(); j++)
            {
                if (!del[j])
                {
                    tmp += G[i].first[j];
                }
            }
            G[i].first = tmp;
        }
    }
    delete[] del;
    del = NULL;

    sort(G.begin(), G.end());
    G.erase(unique(G.begin(), G.end()), G.end());

    //第三步：去除冗余的函数依赖
    vector<PSS> ans;
    for (int i = 0; i < G.size(); i++)
    {
        vector<PSS> tmp = G;
        tmp.erase(tmp.begin() + i);
        if (!iscontain(get_closure(G[i].first, tmp), G[i].second))
        {
            ans.push_back(G[i]);
        }
    }
    return ans;
}

/*
fuction:分解成3NF
params:关系模式+函数依赖集
*/
vector<string> Transform_3NF(const string &R, const vector<PSS> &F)
{
    vector<PSS> FF = get_min_relyset(F);
    map<string, string> mp;
    for (int i = 0; i < FF.size(); i++)
    {
        if (mp.find(FF[i].first) == mp.end())
        {
            mp[FF[i].first] = FF[i].second;
        }
        else
        {
            mp[FF[i].first] += FF[i].second;
        }
    }
    FF.resize(mp.size());
    int idx = 0;
    map<string, string>::iterator it;
    for (it = mp.begin(); it != mp.end(); it++)
    {
        FF[idx].first = it->first;
        FF[idx++].second = it->second;
    }

    vector<string> P;
    for (int i = 0; i < FF.size(); i++)
```

```
    {
        P.push_back(FF[i].first + FF[i].second);
    }
    get_all_candidate_key(R, F);
    for (int i = 0; i < candidate_key.size(); i++)
    {
        int flag = 0;
        for (int j = 0; j < P.size(); j++)
        {
            if (iscontain(P[j], candidate_key[i]))
            {
                flag = 1;
                break;
            }
        }
        if (!flag)
        {
            P.push_back(candidate_key[i]);
        }
    }
    sort(P.begin(), P.end());
    P.erase(unique(P.begin(), P.end()), P.end());
    return P;
}

void inputR()
{
    cout << "请输入关系模式 R:" << endl;
    cin >> R;
}
void inputF()
{
    int n;
    string temp;
    cout << "请输入函数依赖的数目: " << endl;
    cin >> n;
    cout << "请输入" << n << "个函数依赖:(输入形式为 a→b ab→c) " << endl;
    for (int i = 0; i < n; ++i)
    {
        pair<string, string> ps;
        cin >> temp;
        int j;
        for (j = 0; j != temp.length(); ++j)
        { //读入 ps.first
            if (temp[j] != '-')
            {
                if (temp[j] == '>')
                    break;
                ps.first += temp[j];
            }
        }
        ps.second.assign(temp, j + 1, string::npos); //读入 ps.second
        F.push_back(ps);                             //读入ps
    }
}
int main()
{
    freopen("in.txt", "r", stdin);
    R = "";
    F.clear();
```

```
    inputR();
    inputF();
    cout << R << endl;
    vector<string> ans = Transform_3NF(R, F);
    cout << ans.size() << endl;
    cout << "将关系模式 R 无损分解且保持依赖地分解成 3NF 模式集，如下：" << endl;
    for (int i = 0; i != ans.size(); ++i)
        cout << ans[i] << endl;
    return 0;
}
```

思考题

各类范式之间的区别是什么?

第1范式：

关系模式S的每个关系的每个属性值都是不可分的原子值；

> 第1范式的模式要求属性值不可再分裂成更小的部分；
>
> 是关系数据库的最基本要求；
>
> 容易出现数据冗余、插入异常、删除异常、修改异常的问题；

第2范式：

如果关系模式是第1范式，且每一个非主属性都不部分依赖于S的任何候选键，则为第2范式；

> 在第1范式的基础上消除了部份依赖

第3范式：

如果关系模式是第1范式。且每个非主属性都既不部分也不传递依赖于S的任何候选键，则为第3范式；

> 在第2范式的基础上消除传递依赖；
>
> 基本能解决第1范式中出现的问题；

BC范式：

满足以下条件均是BC范式：

- 如果关系模式是第三范式，它的任何一个 主属性 都既不部分也不传递依赖于关系模式的任何候选键，则为BC范式；
- 如果关系模式任何一个 属性 都既不部分也不传递依赖于任何候选键，则为BC范式
- 如果关系模式为第一范式，函数依赖集中的任意一个 非平凡函数依赖 的决定因素都包含键，则属于BC范式；