# BCNF&3NF教学微信小程序

罗云扬，李宇扬，全秦霄，李伟杰

目录

[BCNF&3NF教学微信小程序 1](#_Toc135908645)

[一、计划实现的功能： 1](#_Toc135908646)

[二、项目进度安排： 2](#_Toc135908647)

[三、已经完成的功能： 2](#_Toc135908648)

[1.前端（李宇扬） 2](#_Toc135908649)

[2.基本函数（罗云扬） 3](#_Toc135908650)

[2.1 计算属性闭包集函数 3](#_Toc135908651)

[2.2计算函数依赖闭包集合F+ 6](#_Toc135908652)

[2.3计算候选码 12](#_Toc135908653)

[2.4计算正则覆盖集 14](#_Toc135908654)

[3.BCNF分解（李伟杰）： 18](#_Toc135908655)

[3.1判断属于BCNF 18](#_Toc135908656)

[3.2 BCNF分解 19](#_Toc135908657)

[3.3 展示计算步骤 22](#_Toc135908658)

[4.3NF分解（全秦霄）： 22](#_Toc135908659)

[4.1 判断是否有函数依赖破坏3NF 22](#_Toc135908660)

[4.2 整体代码 26](#_Toc135908661)

1. 计划实现的功能：

（1）BCNF：

开发软件实现BC范式分解, 用户输入

* 关系模式R=(ABCDE),
* 函数依赖集合F.

软件实现如下功能:

1. 计算任意属性闭包.
2. 计算函数依赖闭包F+.
3. 计算候选码.
4. 判断R是否属于BCNF.
5. BCNF分解.
6. 展示计算步骤, 以便用于教学.

（2）3NF：

开发软件实现BC范式分解, 用户输入

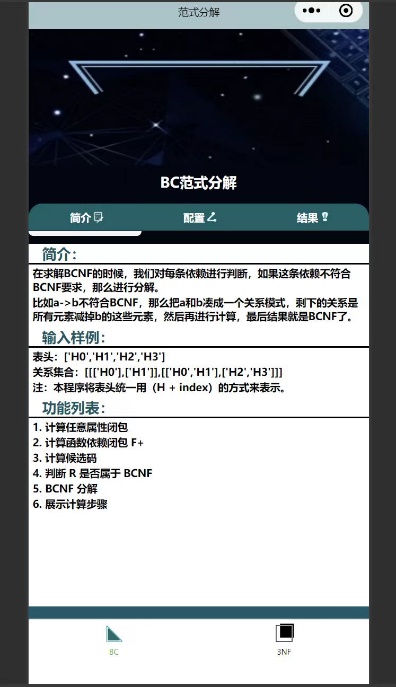
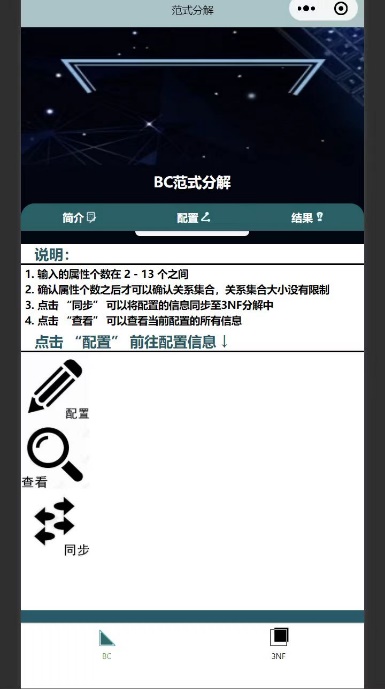
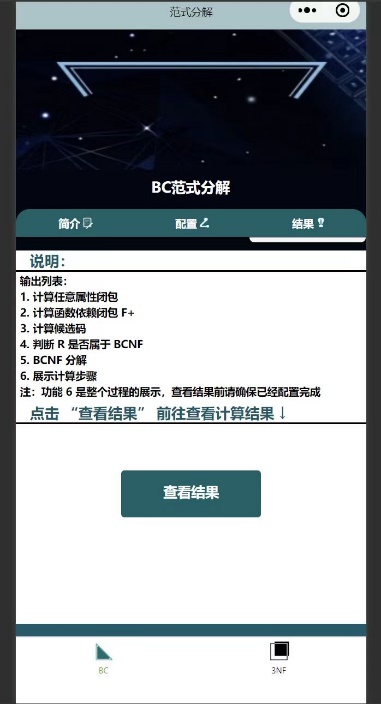
* 关系模式R=(ABCDE),
* 函数依赖集合F.

软件实现如下功能:

1. 计算任意属性闭包.
2. 计算Canonical Cover Fc..
3. 计算候选码.
4. 判断R是否属于3NF.
5. 3NF分解.
6. 展示计算步骤, 以便用于教学.
7. 项目进度安排：

|  |  |  |  |
| --- | --- | --- | --- |
| 主题 | 日期 | 计划实现 | 实际实现 |
| 确定题目 | 2023/4/16-2023/4/28 | 确定题目；提前学习bcnf和3nf的知识，JavaScript和微信小程序；建立Github仓库，方便代码协作 | 确定题目；学习bcnf和3nf的知识，JavaScript和微信小程序；建立Github仓库和微信小程序开发者账号 |
| 完成基本软件功能 | 2023/4/29-2023/5/18 | 完成软件的任务要求的后端算法和前端页面雏形 | 实现计算任意属性闭包，九三函数依赖集闭包，计算正则覆盖，计算候选码等基本函数依赖；实现判断R是否属于BCNF和BCNF分解；实现判断R是否属于3NF和3NF分解；完成微信小程序前端页面框架雏形 |
| 优化软件页面和算法性能 | 2023/5/19-2023/5/25 | 优化BCNF分解的计算和前端页面美观 | 通过计算属性闭包来近似获得函数依赖集闭包的方法，优化BCNF分解算法；前端页面简约美观 |

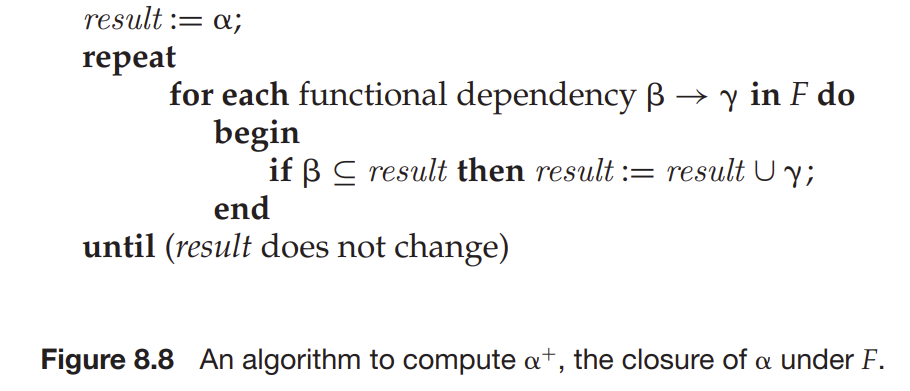
1. 已经完成的功能：
   * + 1. 前端（李宇扬）

* + - 1. 基本函数（罗云扬）

2.1 计算属性闭包集函数



根据书本Database System Concepts (6e) Figure 8.8，计算属性闭包函数的伪代码可以简述为上图。下面是使用JavaScript实现函数的一种方式

/\*\*

 \* @description 用于计算单个参数的属性闭包

 \* @param {array} alpha - 需要计算闭包的alpha

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回属性闭包alpha+

 \*/

function calculateSingleAttributeClosure(alpha, F) {

    // 复制原始的 alpha 数组

    alpha = alpha.slice();

    if (alpha.length === 0) {

      return [];

    }

    // 初始化结果数组为 alpha

    let result = alpha.slice();

    while (true) {

      for (let [alpha\_i, beta\_i] of F) {

        // 如果 alpha\_i 是 result 的子集

        if (isSubset(alpha\_i,result)) {

          // 将 beta\_i 中不在 result 中的元素加入 result

          for (let b of beta\_i) {

            if (!result.includes(b)) {

              result.push(b);

            }

          }

        }

      }

      // 如果结果不再改变，说明计算完成

      if (isSubset(result,alpha)) {

        break;

      }

      // 否则更新 alpha，并继续计算和比较

      alpha = result.slice();

    }

    return result;

  }

例子：若求属性A的闭包展示

R = ['A','B','C','D','E','F']

F = [[['A'],['B']],[['B'],['C','D']],[['D'],['E']],[['C','E'],['F']]]

closure\_of\_A = calculateSingleAttributeClosure(['A'],F)

console.log('closure of A is ',closure\_of\_A,'\n')

函数的输出为如下

[Running] node "c:\Users\lyy\Desktop\vscode\attributeClosureSet.js"

closure of A is  [ 'A', 'B', 'C', 'D', 'E', 'F' ]

但由于我们需要计算所有属性的闭包，因此需要循环遍历所有的属性闭包进行计算，定义函数calculateAttributeClosureSet来实现此功能

/\*\*

 \* @description 此函数用来判断例如[['A'],['B']]是不是[[['A'],['B']],[['B'],['C']]]的子集

 \* @param {array} item - F中的item

 \* @param {array} F - 函数依赖集合

 \* @returns {bool} 如果为的子集，返回true; 反之返回false

 \*/

function isInAttributeClosureSet(item, attributeClosureSet){

  return attributeClosureSet.some(subArr => JSON.stringify(subArr) === JSON.stringify(item))

}

/\*\*

 \* @description 用来计算所有的属性闭包

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回属性闭包集合{alpha+}

 \*/

function calculateAttributeClosureSet(F) {

    // to store all attribute closures 用与存储所有的属性闭包

    let attributeClosureSet = [];

    for (let [alpha, beta] of F) {

        let result = calculateSingleAttributeClosure(alpha, F);

        // to eliminate duplicate attribute closure 若属性闭包未重复，则添加

        if (! isInAttributeClosureSet([alpha,result],attributeClosureSet)){

          attributeClosureSet.push([alpha,result]);

        }

    }

    return attributeClosureSet;

}

例子：求所有属性的闭包集合

R = ['A','B','C','D','E','F']

F = [[['A'],['B']],[['B'],['C','D']],[['D'],['E']],[['C','E'],['F']]]

// calculate all attribute closures

let attributeClosureSet = calculateAttributeClosureSet(F);

// print all attribute closures

console.log('all attribute closures are')

for (let [alpha, alphaClosure] of attributeClosureSet) {

    console.log(alpha, " --> ", alphaClosure);

}

[Running] node "c:\Users\lyy\Desktop\vscode\attributeClosureSet.js"

all attribute closures are

[ 'A' ]  -->  [ 'A', 'B', 'C', 'D', 'E', 'F' ]

[ 'B' ]  -->  [ 'B', 'C', 'D', 'E', 'F' ]

[ 'D' ]  -->  [ 'D', 'E' ]

[ 'C', 'E' ]  -->  [ 'C', 'E', 'F' ]

2.2计算函数依赖闭包集合F+

为了使用Armstrong’s axioms 来计算函数依赖闭包集合，我们先分别定义三个要用到的定理：reflexivity，augmentation和transitivity。

* Reflexivity rule

/\*\*

 \* @description 此函数通过transitiviry rule来计算F+

 \* @param {array} R - 关系模式

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回应用transitivity rule后的F+

 \*/

function transitivityRule(R, F) {

    const newF = [...F];

    for (const [alpha, beta] of F) {

      for (const [alpha\_i, beta\_i] of F) {

        if (isArraySubset(beta,alpha\_i) && isArraySubset(alpha\_i,beta)){

          const transitiveItem = [alpha, beta\_i];

          if (! transitiveItem\_is\_subset\_of\_newF(transitiveItem, newF)){

            newF.push(transitiveItem);

          }

        }

      }

    }

    return newF;

  }

//测试transitivity rule

F\_new = transitivityRule(R,F)

console.log('F+ after transitivity rule ',F\_new,'\n')

此函数的输出为：

R = ['A','B','C']

F = [[['A'],['B']],[['B'],['C']]]

[Running] node "c:\Users\lyy\Desktop\vscode\functionDependencyClosureSet.js"

F+ after reflexive rule [

  [ [ 'A' ], [ 'B' ] ],

  [ [ 'B' ], [ 'C', 'D' ] ],

  [ [ 'D' ], [ 'E' ] ],

  [ [ 'C', 'E' ], [ 'F' ] ],

  [ [ 'A' ], [ 'A' ] ],

  [ [ 'B' ], [ 'B' ] ],

  [ [ 'D' ], [ 'D' ] ],

  [ [ 'C', 'E' ], [ 'C' ] ],

  [ [ 'C', 'E' ], [ 'E' ] ],

  [ [ 'C', 'E' ], [ 'C', 'E' ] ]

]

* Augmentation rule

/\*\*

 \* @description 此函数通过augmentation rule来计算F+

 \* @param {array} R - 关系模式

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回应用augmentation rule后的F+

 \*/

function augmentationRule(R, F) {

    //求R的非空子集

    const RSubsets = getSubsets(R).filter(subset => subset.length > 0);

    const newF = F.slice();

    for (const [alpha, beta] of F) {

      for (const subset of RSubsets) {

        const augmentationAlpha = [...new Set([...subset, ...alpha])].sort();

        const augmentationBeta = [...new Set([...subset, ...beta])].sort();

        const augmentItem = [augmentationAlpha, augmentationBeta];

        //如果augmentItem不在newF中

        if (!augmentItem\_is\_subset\_of\_newF(augmentItem,newF)){

          newF.push(augmentItem);

        }

      }

    }

    return newF;

  }

//测试augmentation rule

F\_new = augmentationRule(R,F)

console.log('F+ after augmentation rule',F\_new,'\n')

此函数的输出为：

R = ['A','B','C']

F = [[['A'],['B']]]

[Running] node "c:\Users\lyy\Desktop\vscode\functionDependencyClosureSet.js"

F+ after augmentation rule [

  [ [ 'A' ], [ 'B' ] ],

  [ [ 'A' ], [ 'A', 'B' ] ],

  [ [ 'A', 'B' ], [ 'B' ] ],

  [ [ 'A', 'B' ], [ 'A', 'B' ] ],

  [ [ 'A', 'C' ], [ 'B', 'C' ] ],

  [ [ 'A', 'C' ], [ 'A', 'B', 'C' ] ],

  [ [ 'A', 'B', 'C' ], [ 'B', 'C' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'B', 'C' ] ]

]

* Transitivity rule

/\*\*

 \* @description 此函数通过transitiviry rule来计算F+

 \* @param {array} R - 关系模式

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回应用transitivity rule后的F+

 \*/

function transitivityRule(R, F) {

    const newF = [...F];

    for (const [alpha, beta] of F) {

      for (const [alpha\_i, beta\_i] of F) {

        if (isArraySubset(beta,alpha\_i) && isArraySubset(alpha\_i,beta)){

          const transitiveItem = [alpha, beta\_i];

          if (! transitiveItem\_is\_subset\_of\_newF(transitiveItem, newF)){

            newF.push(transitiveItem);

          }

        }

      }

    }

    return newF;

  }

//测试transitivity rule

F\_new = transitivityRule(R,F)

console.log('F+ after transitivity rule ',F\_new,'\n')

此函数的输出为：

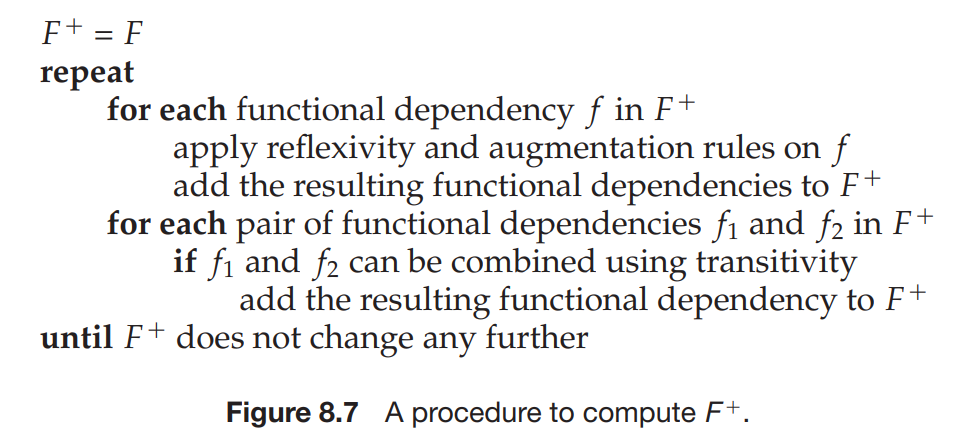
R = ['A','B','C']

F = [[['A'],['B']],[['B'],['C']]]

[Running] node "c:\Users\lyy\Desktop\vscode\functionDependencyClosureSet.js"

F+ after transitivity rule  [ [ [ 'A' ], [ 'B' ] ], [ [ 'B' ], [ 'C' ] ], [ [ 'A' ], [ 'C' ] ] ]

然后依据书本Database System Concepts (6e) Figure8.7给出的伪代码



我们给出计算F+的函数实现如下

/\*\*

 \* @description 此函数通过Armstrong Axiom来计算F+

 \* @param {array} R - 关系模式

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回应用Armstrong Axioms后的F+

 \*/

function calculateFunctionDependencyClosureSetByArmstrongAxioms(R, F){

    let FOriginal = [...F];

    while(true){

      F = reflexiveRule(R, F);

      F = augmentationRule(R, F);

      F = transitivityRule(R, F);

      if (JSON.stringify(F) === JSON.stringify(FOriginal)) {

          break;

      } else {

          FOriginal = [...F];

      }

    }

    F.sort();

    return F;

}

F\_new = calculateFunctionDependencyClosureSetByArmstrongAxioms(R,F)

console.log(F\_new);

此函数的输出如下

R = ['A','B','C','G','H','I']

F = [[['A'],['B']], [['A'],['C']], [['C','G'],['H']], [['C','G'],['I']], [['B'],['H']]]

[Running] node "c:\Users\lyy\Desktop\vscode\functionDependencyClosureSet.js"

[

  [ [ 'A' ], [ 'A' ] ],

  [ [ 'A' ], [ 'A', 'B' ] ],

  [ [ 'A' ], [ 'A', 'B', 'C' ] ],

  [ [ 'A' ], [ 'A', 'B', 'C', 'H' ] ],

  [ [ 'A' ], [ 'A', 'B', 'H' ] ],

  [ [ 'A' ], [ 'A', 'C' ] ],

  [ [ 'A' ], [ 'A', 'C', 'H' ] ],

  [ [ 'A' ], [ 'A', 'H' ] ],

  [ [ 'A' ], [ 'B' ] ],

  [ [ 'A', 'B' ], [ 'A' ] ],

  [ [ 'A', 'B' ], [ 'A', 'B' ] ],

  [ [ 'A', 'B' ], [ 'A', 'B', 'C' ] ],

  [ [ 'A', 'B' ], [ 'A', 'B', 'C', 'H' ] ],

  [ [ 'A', 'B' ], [ 'A', 'B', 'H' ] ],

  [ [ 'A', 'B' ], [ 'A', 'C' ] ],

  [ [ 'A', 'B' ], [ 'A', 'C', 'H' ] ],

  [ [ 'A', 'B' ], [ 'A', 'H' ] ],

  [ [ 'A', 'B' ], [ 'B' ] ],

  [ [ 'A', 'B' ], [ 'B', 'C' ] ],

  [ [ 'A', 'B' ], [ 'B', 'C', 'H' ] ],

  [ [ 'A', 'B' ], [ 'B', 'H' ] ],

  [ [ 'A' ], [ 'B', 'C' ] ],

  [ [ 'A', 'B' ], [ 'C' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'B' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'B', 'C' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'B', 'C', 'H' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'B', 'H' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'C' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'C', 'H' ] ],

  [ [ 'A', 'B', 'C' ], [ 'A', 'H' ] ],

  [ [ 'A', 'B', 'C' ], [ 'B' ] ],

  [ [ 'A', 'B', 'C' ], [ 'B', 'C' ] ],

  [ [ 'A', 'B', 'C' ], [ 'B', 'C', 'H' ] ],

  [ [ 'A', 'B', 'C' ], [ 'B', 'H' ] ],

  [ [ 'A', 'B', 'C' ], [ 'C' ] ],

  [ [ 'A', 'B', 'C' ], [ 'C', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C', 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C', 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C', 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C', 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'C', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'B', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C', 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C', 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C', 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C', 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'C', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'A', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C', 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C', 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C', 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C', 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'C', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'B', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C', 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C', 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C', 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C', 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'C', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'G' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'G', 'H' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'G', 'H', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'G', 'I' ] ],

  [ [ 'A', 'B', 'C', 'G' ], [ 'H' ] ],

  [ [ 'A', 'B', 'C', 'G', 'H' ], [ 'A' ] ],

  [ [ 'A', 'B', 'C', 'G', 'H' ], [ 'A', 'B' ] ],

  ... 1568 more items

]

F+的计算是个非常耗时的工作，时间复杂度可以达到指数级。因此虽然我们实现了计算F+的算法，但目的是凸显其应用的不便性，在日后的教学展示中作为劝大家选择属性闭包而非F+的理由。

2.3计算候选码

要判断属性alpha是否为候选码，需要使用计算属性闭包函数。如果alpha的属性闭包是R的超集而alpha子集的属性闭包不是R的超集，那么alpha是候选码

判断alpha是否为候选码的一种实现方式如下

/\*\*

 \* @description 此函数用来判断alpha是不是candidate key

 \* @param {array} alpha - 需要判断的属性集合

 \* @param {array} R - 关系模式

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回s的所有真子集的集合

 \*/

function determineCandidateKey(alpha, R, F){

    alphaClosure = calculateSingleAttributeClosure(alpha,F);

    for(item of alphaClosure){

      //如果alpha为super key

      if(isSubset(R, alphaClosure)){

          notCandidateKey = false;

          //判断子集是否为super key

          alphaProperSubsets = getProperSubsets(alpha);

          for(subset of alphaProperSubsets){

              subsetClosure = calculateSingleAttributeClosure(subset,F);

              //子集存在super key，alpha就不是candidate key

              if(isSubset(R,subsetClosure)){

                  notCandidateKey = ture;

              }

          }

          if(!notCandidateKey)

              return true;

          return false;

      }

    }

    return false;

}

输出测试如下

R = ['A','B','C','D','E','F']

F = [[['A'],['B']],[['B'],['C','D']],[['D'],['E']],[['C','E'],['F']]]

//测试candidate key的计算

console.log('determine if A is a candidate key',determineCandidateKey(['A'],R,F),'\n');

console.log('determine if B is a candidate key',determineCandidateKey(['B'],R,F),'\n');

[Running] node "c:\Users\lyy\Desktop\vscode\candidateKey.js"

determine if A is a candidate key true

determine if B is a candidate key false

有了判断候选码的函数，我们就可以对F中所有函数依赖左侧的属性进行判断，找出候选码

一种实现方式如下

/\*\*

 \* @description 此函数计算所有的candidate key

 \* @param {array} R - 关系模式

 \* @param {array} F - 函数依赖集合

 \* @returns {array} 返回candidate key set

 \*/

function calculateCandidateKeySet(R, F){

    let candidateKeySet = [];

    for([alpha,beta] of F){

        if(determineCandidateKey(alpha,R,F)){

            candidateKeySet.push(alpha);

        }

    }

    return candidateKeySet;

}

输出测试如下

R = ['A','B','C','D','E','F']

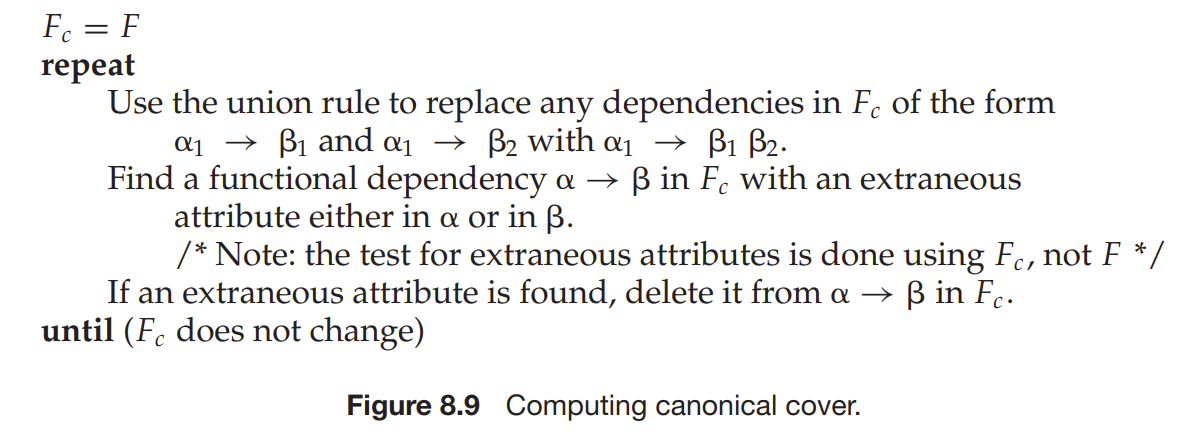
F = [[['A'],['B']],[['B'],['C','D']],[['D'],['E']],[['C','E'],['F']]]

[Running] node "c:\Users\lyy\Desktop\vscode\candidateKey.js"

candidate key set is  [ [ 'A' ] ]

2.4计算正则覆盖集

依据书本Database System Concepts (6e) Figure8.9给出的伪代码



上述函数的一种实现方式如下

/\*\*

 \* @description 此函数用来计算canonical cover set

 \* @param {array} F - 初始化的函数依赖集合F

 \* @returns {array} 返回canonical cover set

 \*/

function computeCanonicalCoverSet(F){

    F = functionDependencyUnion(F);

    F = eliminateLeftExtraneous(F);

    F = functionDependencyUnion(F);

    F = eliminateRightExtraneous(F);

    return F;

  }

* 函数functionDependencyUnion将函数依赖中左侧属性相同的依赖合并，实现如下

/\*\*

 \* @description 此函数返回合并后的函数依赖集合,

 \* Example: F 中有 A-->B, B-->C, A-->C

 \* 返回 A-->(B,C) , B-->C

 \* @param {array} F - 函数依赖集合F

 \* @returns {array} 返回处理后的unionF

 \*/

function functionDependencyUnion(F){

    let unionF = [];

    for([alpha,beta] of F){

        let betaUnion = beta.slice();

        for([alpha\_i, beta\_i] of F){

            if(alphaEqualsAlpha\_i(alpha,alpha\_i)){

                for(elem of beta\_i){

                    if(!betaUnion.includes(elem)){

                        betaUnion.push(elem);

                    }

                }

            }

        }

        let item = [alpha,betaUnion.sort()];

        if(!itemIsSubsetOfF(item,unionF)){

            unionF.push(item);

        }

    }

    return unionF;

}

输出测试如下

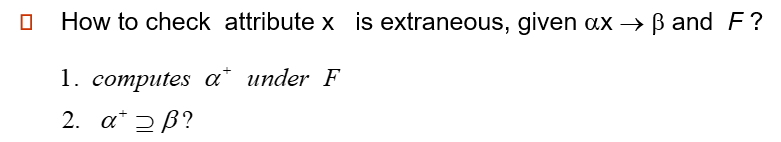
//测试

console.log('function dependency union test',functionDependencyUnion([[['A'],['B']],[['A'],['C']]]),'\n');

[Running] node "c:\Users\lyy\Desktop\vscode\canonicalCoverSet.js"

function dependency union test [ [ [ 'A' ], [ 'B', 'C' ] ] ]

* 剔除左侧extraneous item的函数eliminateLeftExtraneous实现了以下原理



/\*\*

 \* @description 此函数用来去除左侧的extraneous attribute

 \* @param {array} unionF - 经过union处理后的F

 \* @returns {array} 返回去除左侧extraneous attribute后的F

 \*/

function eliminateLeftExtraneous(unionF){

    for(let i in unionF){

        replace = unionF[i];    //replace = [alpha,beta]

        alpha = replace[0];

        beta = replace[1];

        alphaSubsets = getProperSubsets(alpha);

        for(subset of alphaSubsets){

            if(subset !== [] && (subset.length < alpha.length)){

                newClosure = calculateSingleAttributeClosure(subset, unionF);

                    if(beta\_is\_subset\_of\_newClosure(beta, newClosure)){

                        replace = [subset, beta];

                    }

            }

        }

        unionF[i] = replace;

    }

    return unionF;

}

输出测试

//test

unionF = [[['A'],['B']],[['B'],['C']],[['A','C'],['D']]];

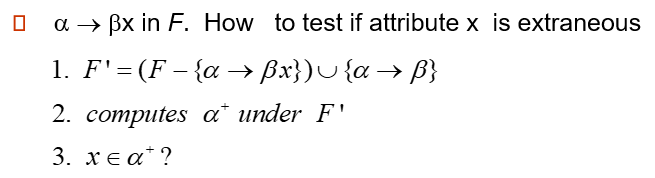
console.log('left extranuous elimination',eliminateLeftExtraneous(unionF),'\n');

[Running] node "c:\Users\lyy\Desktop\vscode\canonicalCoverSet.js"

function dependency union test [ [ [ 'A' ], [ 'B', 'C' ] ] ]

left extranuous elimination [ [ [ 'A' ], [ 'B' ] ], [ [ 'B' ], [ 'C' ] ], [ [ 'A' ], [ 'D' ] ] ]

* 剔除右侧extraneous item的函数eliminateRightExtraneous实现了以下原理



/\*\*

 \* @description 此函数用来去除右侧的extraneous attribute

 \* @param {array} unionF - 经过union处理后的F

 \* @returns {array} 返回去除右侧extraneous attribute后的F

 \*/

function eliminateRightExtraneous(unionF){

    for(let i in unionF){

        replace = unionF[i]; //replace = [alpha, beta]

        alpha = replace[0];

        beta = replace[1];

        betaSubsets = getProperSubsets(beta);

        for(subset of betaSubsets){

            if(subset !== [] && (subset.length < beta.length)){

                tempUnionF = unionF.slice();

                tempUnionF[i] = [alpha, subset];

                newClosure = calculateSingleAttributeClosure(alpha, tempUnionF);

                if(beta\_is\_subset\_of\_newClosure(beta, newClosure)){

                    replace = [alpha, subset];

                }

            }

        }

        unionF[i] = replace;

    }

    return unionF;

}

输出测试如下

//test

unionF = [[['A'],['B']],[['B'],['C']],[['A'],['C','D']]];

console.log('right extranuous elimination',eliminateRightExtraneous(unionF),'\n');

[Running] node "c:\Users\lyy\Desktop\vscode\canonicalCoverSet.js"

function dependency union test [ [ [ 'A' ], [ 'B', 'C' ] ] ]

right extranuous elimination [ [ [ 'A' ], [ 'B' ] ], [ [ 'B' ], [ 'C' ] ], [ [ 'A' ], [ 'D' ] ] ]

最后，计算正则覆盖集的输出测试如下

F = [[['A'], ['B','C']], [['B'], ['C']], [['A'], ['B']], [['A','B'], ['C']]]

canonicalCoverF = computeCanonicalCoverSet(F);

console.log('canonical cover set F is',canonicalCoverF,'\n');

[Running] node "c:\Users\lyy\Desktop\vscode\canonicalCoverSet.js"

canonical cover set F is [ [ [ 'A' ], [ 'B' ] ], [ [ 'B' ], [ 'C' ] ] ]

3.BCNF分解（李伟杰）：

### 3.1判断属于BCNF

方法：对F中每个非平凡依赖且，检查其是否违反BCNF，即属性集R是的属性闭包的子集。

伪代码：

 代码：

/\*\*

 \* @description 检查不常用（non-trivial）依赖s是否导致不符合BCNF

 \* @param {Array(Array())} s:F中的一个在schemaR上的非平凡依赖，2-dimension Array;ex:[['A'],['B']]

 \* @param {Array} schemaR:list,表的属性;ex:['A','B','C','D','E','F']

 \* @param {Array(Array(Array()))} F :函数依赖集，3-dimension Array

 \* @returns Boolean,True:符合BC范式；False：违背BC范式

 \*/

function check\_NontrivialDependencyCaseViolationOfBCNF(s,schemaR,F){

      var s\_plus\_set = new Set(calculateSingleAttributeClosure(s[0],F));

      // console.log(s,'s+:',s\_plus\_set);

      var R\_set = new Set(schemaR);

      // console.log('isSuperset(s\_plus\_set, R\_set)---',isSuperset(s\_plus\_set, R\_set));

      return isSuperset(s\_plus\_set, R\_set);

  }

 /\*\*

 \*@description 检检查schena是否符合BC范式

 \* @param {Array} schemaR:list,表的属性;ex:['A','B','C','D','E','F']

 \* @param {Array(Array(Array()))} F :函数依赖集，3-dimension Array

 \* @returns {Boolean}result:boolean,True:符合BC范式；False：违背BC范式

 \*          {Array} vio:违背BC范式的函数依赖，2-dimension Array;ex:[['A'],['B']]

 \*/

function check\_schemaInBCNF(schemaR,F){

      var result = true;

      var vio =[] ;

      for (i in F){

          var alpha\_set = new Set(F[i][0]);

          var beta\_set = new Set(F[i][1]);

          // 保证该依赖是非平凡依赖且在Ri中，可以被分解

          if (!(isSuperset(alpha\_set,beta\_set)) &&  isSuperset(new Set(schemaR),union(alpha\_set,beta\_set))){

              // console.log(i,'\t',F[i],'\t',!(check\_NontrivialDependencyCaseViolationOfBCNF(F[i],schemaR,F)));

              if (!(check\_NontrivialDependencyCaseViolationOfBCNF(F[i],schemaR,F))){

                  vio = F[i];

                  result = false;

                  break;

              }

          }

      }

      // console.log(result,'\t',vio);

      return [result,vio];

  }

* 1. BCNF分解

伪代码：

Result := {R};

Done := false;

Compute F+;

While(not done){

If (there is a schema Ri in result that is not in BCNF){

Let be a nontrivial functional dependency that holds on Ri

Such that is not in F+,and ;

Result := (result-Ri)∪(Ri-)∪();

}

Else {done := true;}

}

代码：

  /\*\*

   \* description:BC范式分解

   \* @param {Array} R-属性集

   \* @param {Array} F-依赖集

   \* @param {Boolean} F\_plus\_need-需要计算F\_plus

   \*/

  function BCNF(R,F,F\_plus\_need=true){

      var result = [R];

      var done = false;

      var step = {'root':R};

      var decompose\_times = 0;

      var F\_plus =[];

      console.log('F\_plus start');

      if (F\_plus\_need){

          F\_plus = calculateFunctionDependencyClosureSetByArmstrongAxioms(R,F);

      }

      else{

        var F\_closure\_set = calculateFunctionDependencyClosureSetByAttributeClosureSet(F);

        for (var i = 0;i<F\_closure\_set.length;i++){

          var temp = true;

          for (var j = 0 ;j<F\_plus.length;j++){

            if (F\_plus[j][0]==F\_closure\_set[i][0]){

              F\_plus[j][1].push(F\_closure\_set[i][1][0]);

              temp = false;

              break;

            }

          }

          if(temp){

            F\_plus.push(F\_closure\_set[i]);

          }

        }

      }

      console.log('F\_plus end');

      var out = 1;

      while(! done){

          var allBCNF = true;

          for(var i = 0;i<result.length;i++){

              var temp\_R = result[i]; //list

              var temp\_R\_set = new Set(temp\_R);

              var temp = check\_schemaInBCNF(temp\_R,F);

              var bc = temp[0];

              var vio = temp[1];

              // console.log(bc,vio);

              if (!bc){

                  allBCNF = false;

                  var alpha = vio[0];

                  var beta = vio[1];

                  var alpha\_set = new Set(alpha);

                  var beta\_set = new Set(beta);

                  // console.log(isdisjoint(alpha\_set,beta\_set));

                  if (isdisjoint(alpha\_set,beta\_set)){

                      var decompose = true;

                      if(F\_plus\_need){

                          for (j in F\_plus){

                              if( F\_plus[j][0] == alpha && F\_plus[j][1] == temp\_R){

                                  decompose = false;

                                  break;

                              }

                          }

                      }

                      else{

                        for (j in F\_plus){

                          if( F\_plus[j][0] == alpha && isSuperset(new Set(F\_plus[j][1]),temp\_R\_set )){

                              decompose = false;

                              break;

                          }

                        }

                      }

                      if(decompose){

                          var temp\_step = step;

                          for(var j =0;j<decompose\_times;j++){

                              temp\_step = step['right'];

                          }

                          decompose\_times++;

                          console.log(temp\_step['decompose\_dependency']);

                          temp\_step['decompose\_dependency'] = vio;

                          console.log(temp\_step['decompose\_dependency']);

                          del\_index = result.findIndex(element =>element == temp\_R);

                          // console.log(del\_index);

                          result.splice(del\_index,1);

                          result.push(Array.from(union(alpha\_set,beta\_set)));

                          temp\_step['left'] = Array.from(union(alpha\_set,beta\_set));

                          result.push(Array.from(difference(temp\_R\_set,beta\_set)));

                          temp\_step['right']= {'root':Array.from(difference(temp\_R\_set,beta\_set))};

                      }

                  }

              }

          }

          // console.log(allBCNF,'\t',out)

          if(allBCNF){

              done = true;

          }

      }

      // console.log(result,'\t',step);

      return [result,step];

  }

* 1. 展示计算步骤

使用二维数组存储分解结果，字典存储分解过程。

例题：

R = [1,2,3,4,5],

F = { 12,3

3,45

24

51

}

 result-----

 BC分解结果： [ [ 2, 4 ], [ 1, 2, 3, 5 ] ]

分解过程：

 {

  root: [ 1, 2, 3, 4, 5 ],

  decompose\_dependency: [ [ 2 ], [ 4 ] ],

  left: [ 2, 4 ],

  right: { root: [ 1, 2, 3, 5 ] }

}

1. 3NF分解（全秦霄）：
   1. 判断是否有函数依赖破坏3NF

方法：对于任意一组函数依赖，若同时不满足以下三个条件，则该依赖破坏3NF。

* + 1. α⊇β
    2. α是R的超码
    3. β−α中含有R的候选码

伪代码：

For each non-trival α🡪β,

Test α is a superkey: α+⊇R

If α is not a superkey, we have to verify if each attribute in β is contained a candidate key of R.

代码：

/\*\*

  \* @description 此函数用来判断beta是否属于alpha

  \* @param {array} alpha

  \* @param {array} beta

  \* @returns {bool} 如果beta为alpha的子集，返回true; 反之返回false

  \*/

 function beta\_belong\_alpha(alpha,beta){

     return isArraySubset(beta,alpha);

 }

 /\*\*

  \* @description 此函数用来判断alpha是否为超码

  \* @returns {bool}

  \*/

 function alpha\_is\_superKey(alpha,F,R){

     alpha\_closure=calculateSingleAttributeClosure(alpha,F);

     //console.log(alpha\_closure);

     return isArraySubset(R,alpha\_closure);

 }

 /\*\*

  \* @description 此函数用来实现集合减法

  \* @returns {bool}

  \*/

 function subs(A,B){

     let C=A.filter(a => !B.includes(a));

     return C

 }

 /\*\*

  \* @description 此函数用来判断beta-alpha是否包含于某个候选码中

  \* @returns {bool}

  \*/

 function b\_minus\_a\_contain\_candidatKey(alpha,beta,F,R){

     b\_minus\_a=subs(beta,alpha);

     //console.log(b\_minus\_a);

     candidateK=calculateCandidateKeySet(R,F);

     //console.log(candidateK);

     return itemIsSubsetOfF(b\_minus\_a,candidateK);

 }

 /\*\*

  \* @description 此函数用来测试表是否符合3NF

  \* @returns {array}   破坏3NF的函数依赖关系;当输出为[]时，代表表符合3NF

  \*/

 function test3NF(F,R){

     let break\_F=[];

     // console.log(F)

     for (j=0;j<F.length;++j){

         let alpha=F[j][0];

         // console.log("alpha")

         // console.log(alpha)

         let beta=F[j][1];

         if(!beta\_belong\_alpha(alpha,beta) && !alpha\_is\_superKey(alpha,F,R) && !b\_minus\_a\_contain\_candidatKey(alpha,beta,F,R)){

             break\_F[break\_F.length]=[alpha,beta];

         }

     }

     return break\_F;

 }

3NF分解

方法：计算出canonical cover，并将他们都各自存在一张表中。求R的候选码，如果候选码没有单独的表的话，为候选码建表。

Compute the canonical cover: Fc={α1🡪β1, α2🡪β2,……, αn🡪βn}

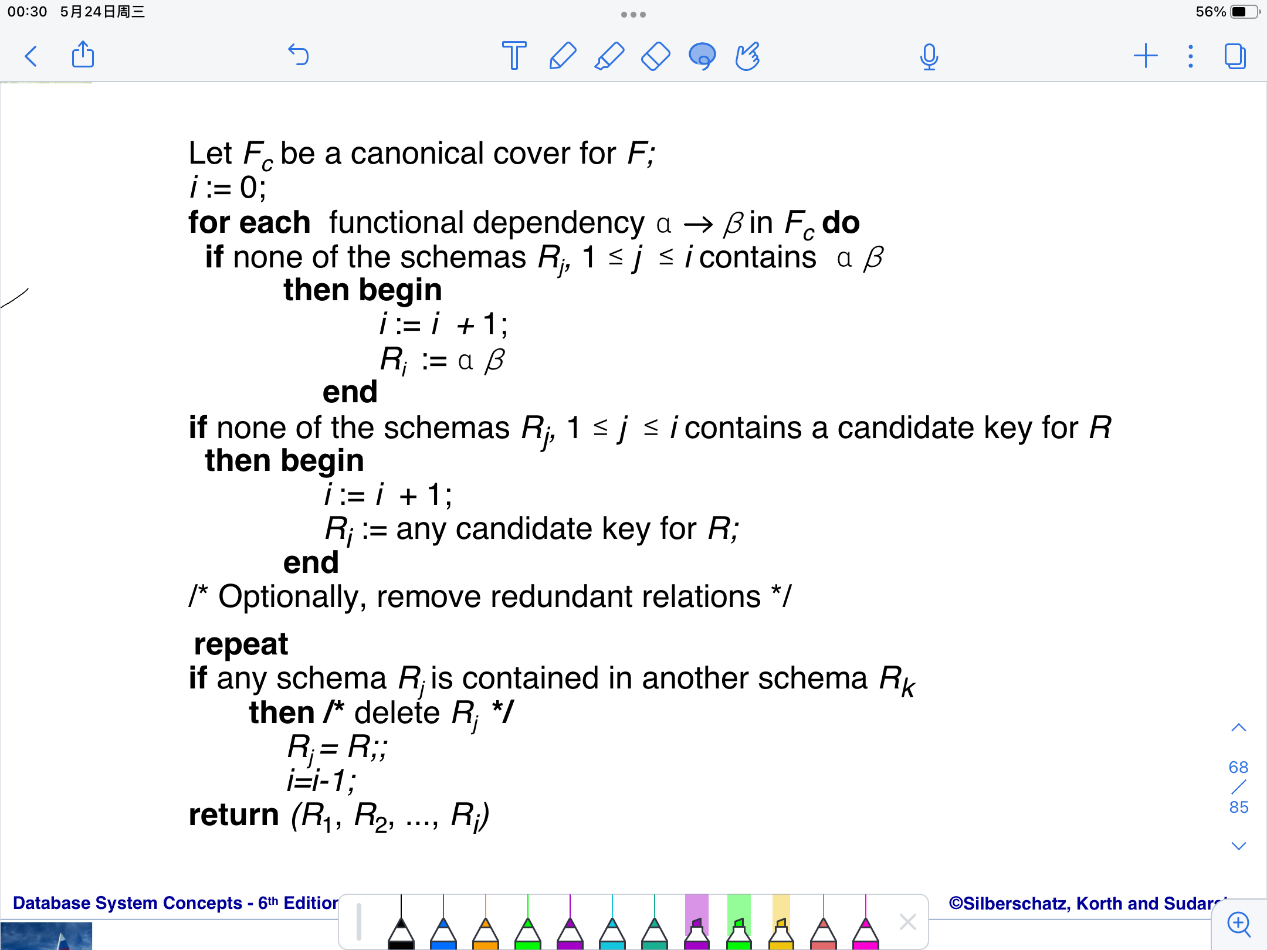
Generate new schemas:R1=α1β1,R2=α2β2,……,Rn=αnβn

Find R’s candidate keys: γ1, γ2,……,γn

If none of the schemas Rj contains a candidate key of R, generate new schemas:R1+n=γ1

outputs:R1,R2,……,Rn+1(if exists)

伪代码：



代码：

/\*\*

  \* @description 此函数用来按照元素的长度升序排序

  \* @returns {array} 排序后的表

  \*/

 function sort\_by\_length(result){

     result.sort(function(a,b){

         return a.length-b.length;

     })

     return result

 }

 /\*\*

  \* @description 此函数用来删除有重复元素的表

  \* @returns {array} 删后的表

  \*/

 function del\_element(result){

     sort\_by\_length(result);

     let del\_index=0;

     while(del\_index<result.length){

         let temp=result.slice(0);//浅拷贝

         result.splice(del\_index,1);

         let state=false;

         for(subs\_i=0;subs\_i<result.length;++subs\_i){

             // console.log("temp[del\_index]");

             if(isArraySubset(temp[del\_index],result[subs\_i])){

                 state=true;

             }

         }

         if(! state){

             result=temp;

             del\_index+=1;

         }

     }

     return result;

 }

 /\*\*

  \* @description 此函数用来根据3NF拆表

  \* @returns {array} 拆后的表

  \*/

 function decomposition3NF(F,R){

     let result\_d=[];

     Fc=computeCanonicalCoverSet(F);

     for(fc\_i=0;fc\_i<Fc.length;++fc\_i){

         result\_d[result\_d.length]=Fc[fc\_i][0].concat(Fc[fc\_i][1]);

     }

     cKey=calculateCandidateKeySet(R,F);

     for(cand\_i=0;cand\_i<cKey.length;++cand\_i){

         result\_d[result\_d.length]=cKey[cand\_i];

     }

     result\_d=del\_element(result\_d);

     return result\_d;

 }

### 4.2 整体代码

代码：

 /\*\*

  \* @description 此函数整合3NF的函数，先检验，再生成符合3NF的表。输出是否破坏3NF，如果破坏，哪些函数依赖破坏3NF。

  \* @returns {array} 符合3NF的表

  \*/

 function ThreeNF(F,R){

     let breakList=test3NF(F,R);

     if(breakList!=[]){

         console.log("存在函数依赖破坏3NF。破坏3NF的函数依赖：");

         console.log(breakList);

         return decomposition3NF(F,R);

     }

     else{

         console.log("该表符合3NF。");

         return R;

     }

 }

整体输出

输入：

  F=[[[1,2],[3,4]],[[2],[3]],[[1,3],[2]]]

  R=[1,2,3,4]

  console.log(ThreeNF(F,R) )

输出：

存在函数依赖破坏3NF。破坏3NF的函数依赖：

[ [ [ 2 ], [ 3 ] ] ]

[ [ 1, 2, 4 ], [ 1, 3, 2 ] ]