·Functional Dependenciy –FD

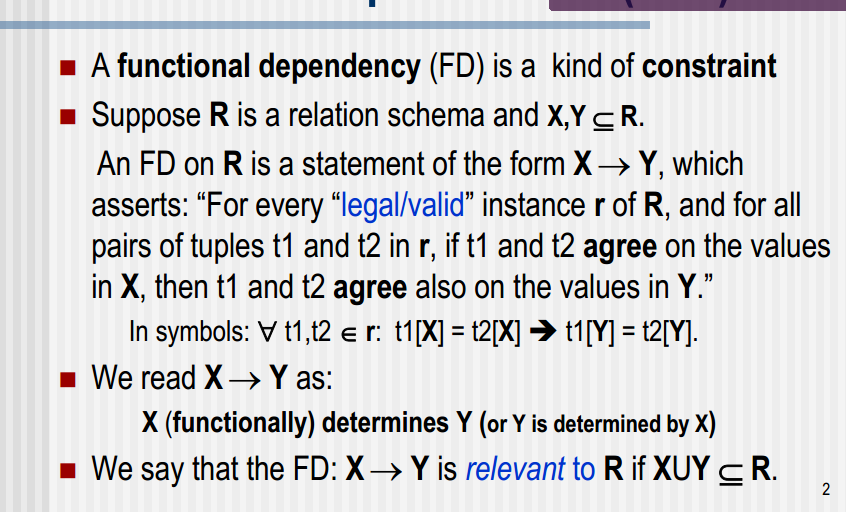
一个functional dependency(FD) 是一种constraint限制

relation schema就是个relation

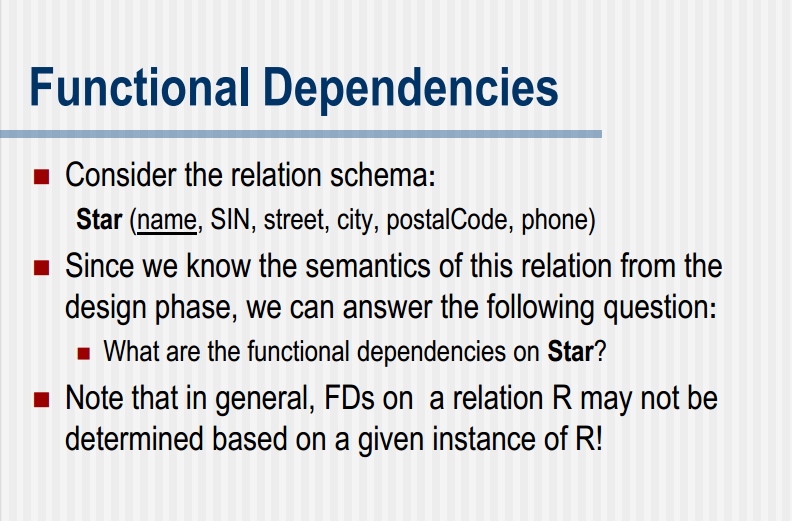
假设R是一个table/relation schema

那么对于R任何一个 有效instance(instance就是某一个时间点的状态)， 如果X->Y //X与Y是属于R的两个attributes

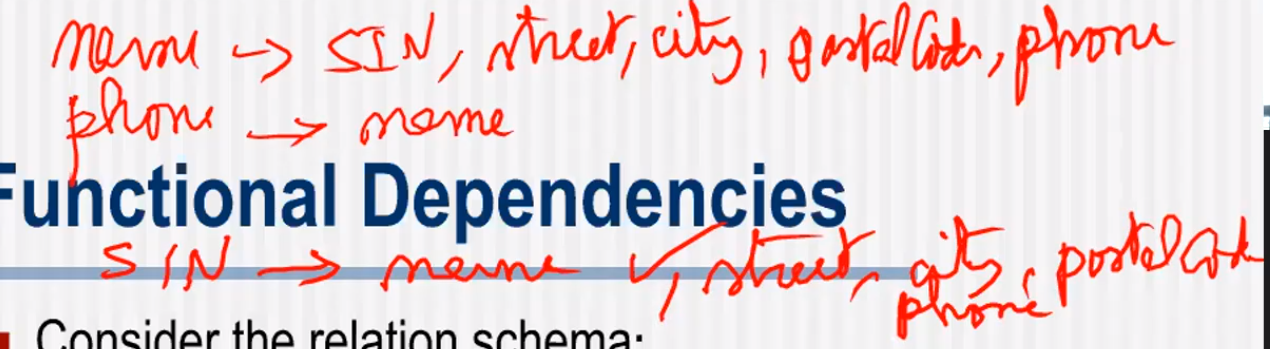
那么对于任意一对tuple，t1t2,如果t1 t2 agree X的值，那么agree Y的值 //换句话说每个X稳定对应一个Y



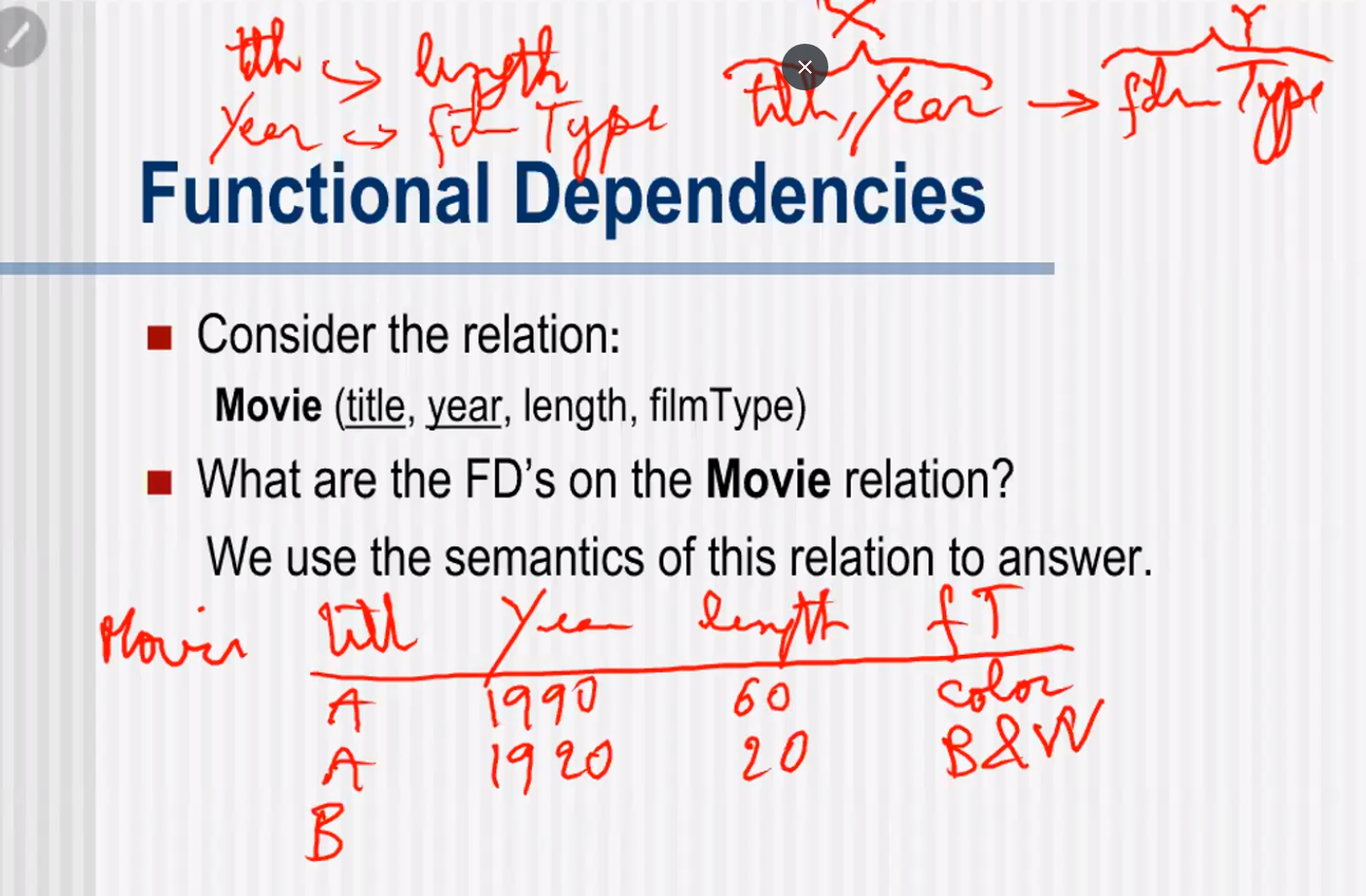
如果X->Y对R成立，我们说X->Y IS RELEVENT TO R



一个relation schema或许可以有多个FDs



通常情况下，你无法通过一个instance就决定FDs



凭借一个title或者一个year无法定位

那么我们也可以吧title,year作为一个X

kEYS

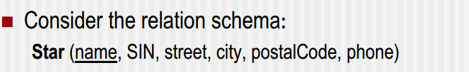
FD的概念天然覆盖了Key的概念



假设X是key（或是candidate候选，例如上面的star\_SIN），只要我们确定了key，其他的必然是固定的

X的子集不能再是key，换句话说candidate key必须是minimal最小的单位//理解成原子

//例如这里的title,year, (title,year)这里可以作为X，是因为他们提取任意一个都不能作为key，如果能包含Key



那么(name,street)

(name,city)

可以无限组合没完没了，而实际上STREET,CITY并没有推出关系

因此

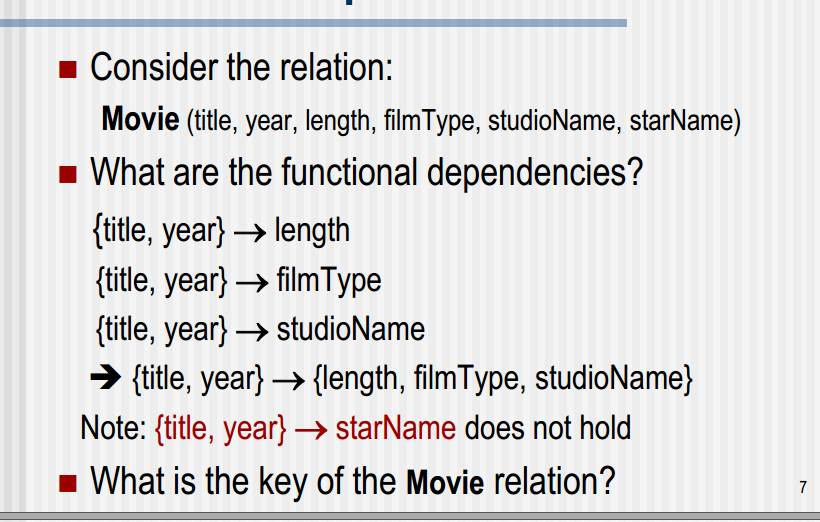


并不是relation,

SuperKey: 一组attribute，包含了至少一个key

每一个minimal key都是superkey，但superkey不见得就是key

X->Y被叫做functional dependency 因为只要你决定了X集合中的一个value，就会return最多一个或者没有Y 中的一个value

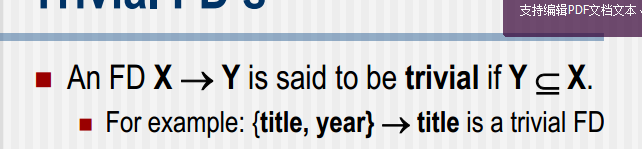


都可以，因为title,year是Key，根据定义就行

实际上这个starName有点蠢，他想表达的意思是，starName一个电影有多个·，因此不是一对一关系，所以不行，但矛盾之处在于，如果starName不是一对一，那他就应该重新创建一个新表，理解意思就好

trivial

X->Y 而Y 属于X的一部分（等于或子集）



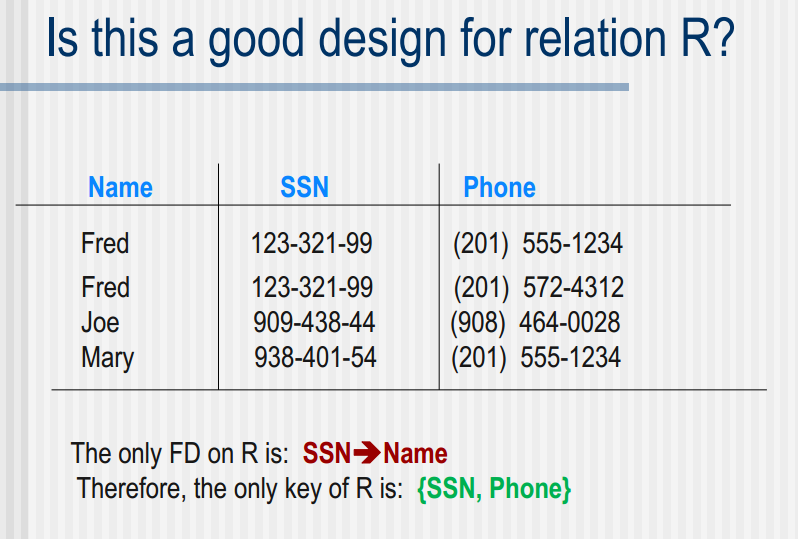
那么这个FD就叫做trivial FD

否则就是nontrivial FD



Redundancy Problem冗余问题

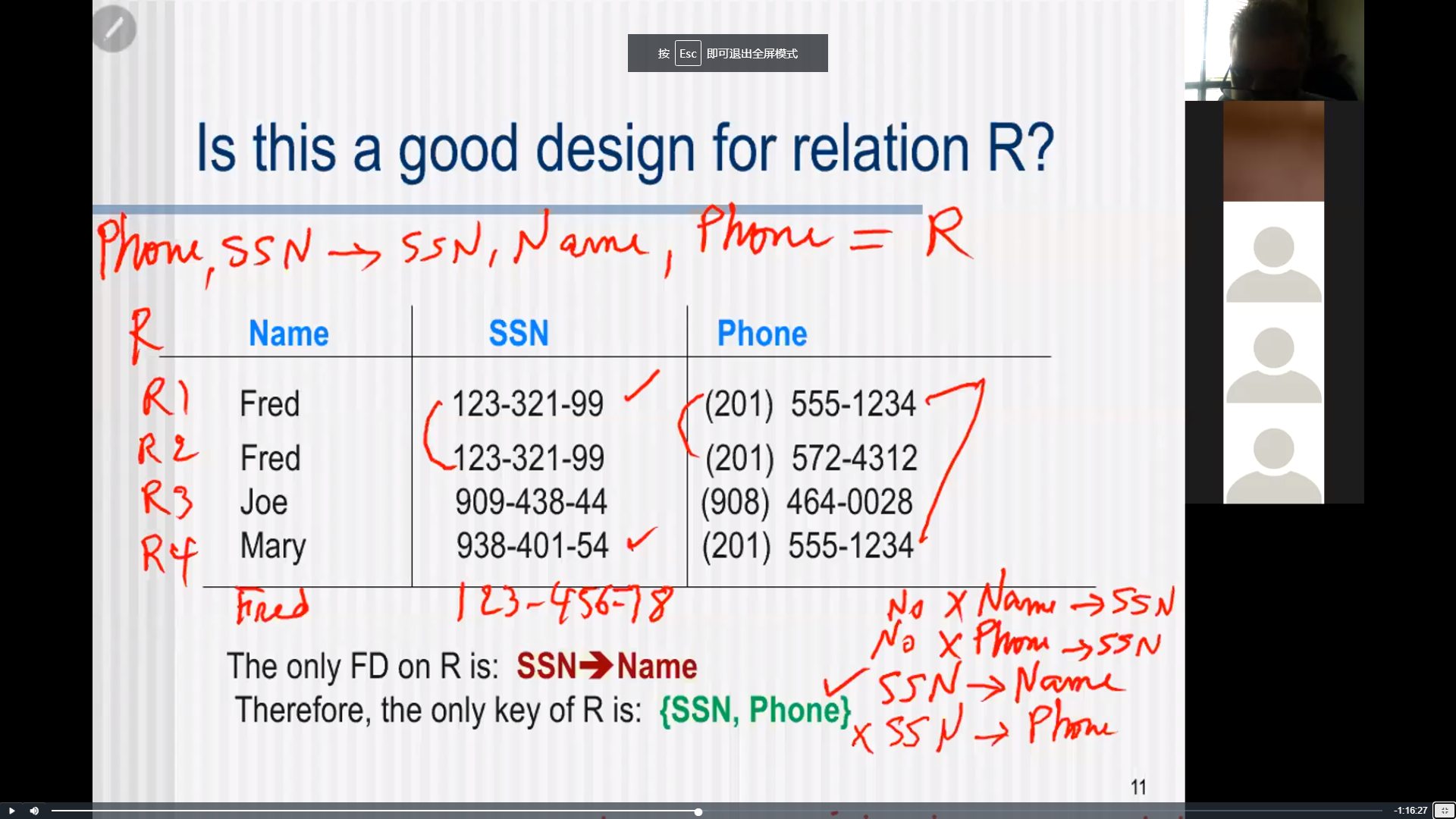
redundancy 代表多个tuples没必要的重复了同一段信息



为啥SSN不能作为单独的一个key

因为第一个第二个SSN不仅重复，而且同样的key推导不出同样的phone

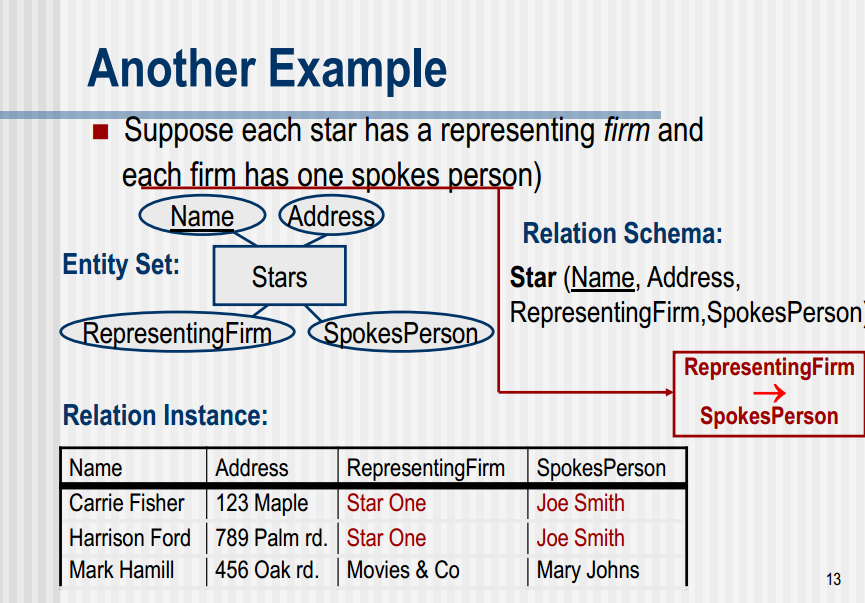
这个SSN仅能推出Name



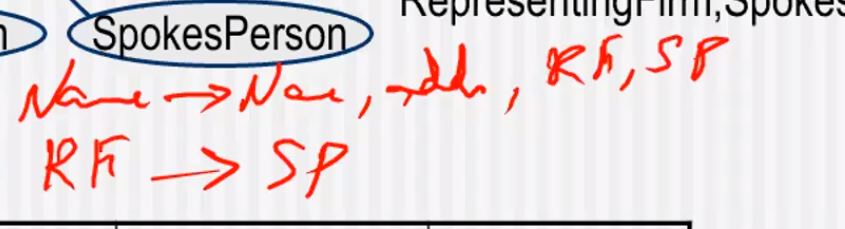
SSN->SSN,NAME

并没有覆盖整个R

因此要在左右同时加上Phone



这里的FD关系式RepresentingFirm->SpokesPerson



判断有没有冗余：

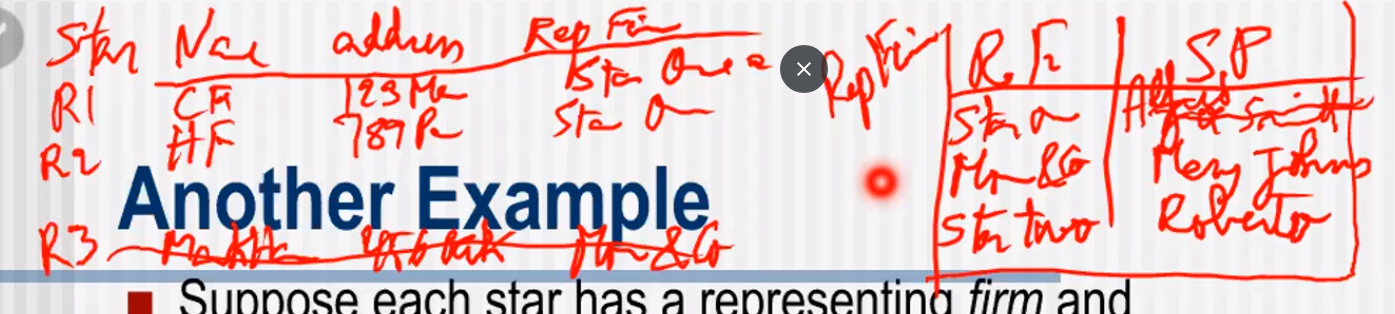
1.所有的FD列出来，看看左边有没有不是KEY的

这里RF不是key

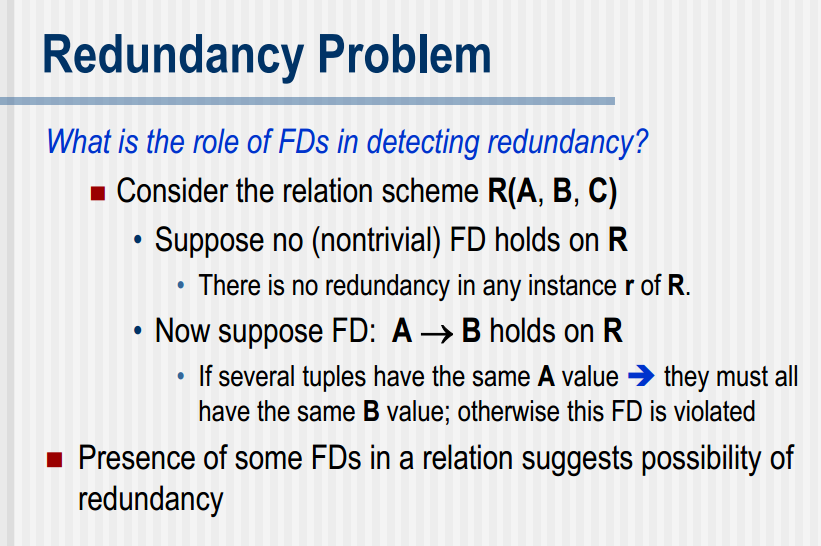
2.删去第三个tuple，我们发现我们流失了信息Movies->Mary Johns

3.我们不能创造单独的RF->SP关系

NULL NULL star\_two brian, 因为 key不能为null



把RFSP单独列为一个table才是最好的



假如R里没有FD（不算key，而是算额外的，例如上面的RF->SP）

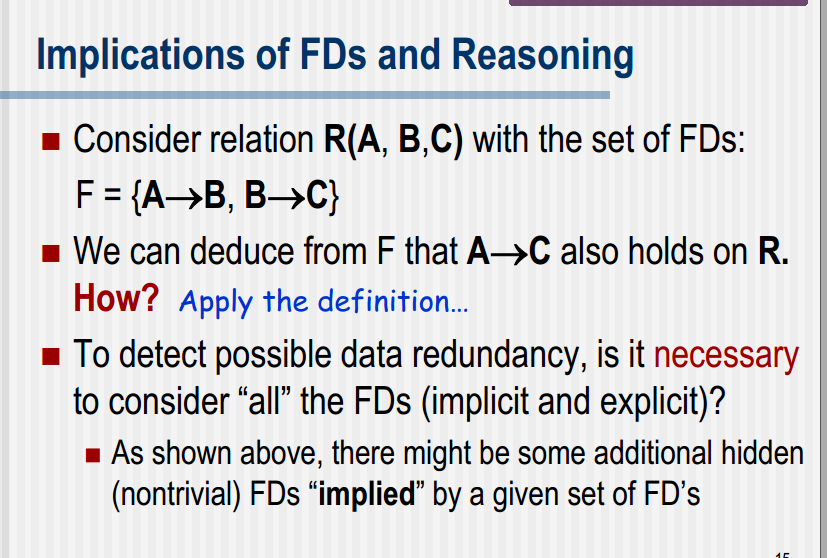
那么就没有redundancy

假如有类似的FD

那么有同样的A，就必然推出同样的B

FD的存在产生了redundancy的可能性

FD的implication隐性



如果A->B,B->C那么A->C

为了检测到可能的redundancy，有必要考虑所有隐含的与explicit的FD

Implications of FDs

r: relation instance

F:一组FD



R:relation, r是R 的一个instance

如果有一个r满足F里的每一个FD，那么我们说 r satisfies F

我们也说r是一个legal/valid instance

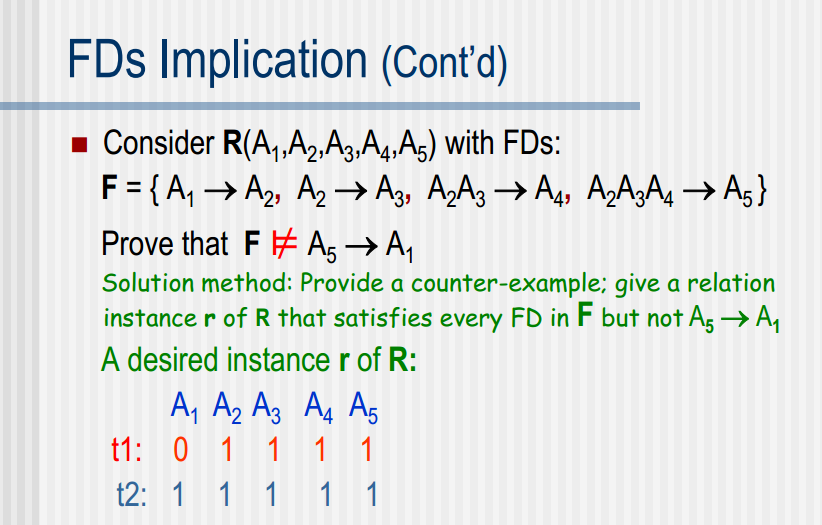
给你一个<R,F>，我们说F implies X->Y, 如果R的每个instance r都满足F， 那么每个r也就都满足X->Y

我们可以表示成F ,也就是说X->Y属于F的一部分，或是能从F推导出X->Y

或者说X->Y follows from F

为了展示

我们必须给出一个反例，例如一个r满足F但是不满足X->Y



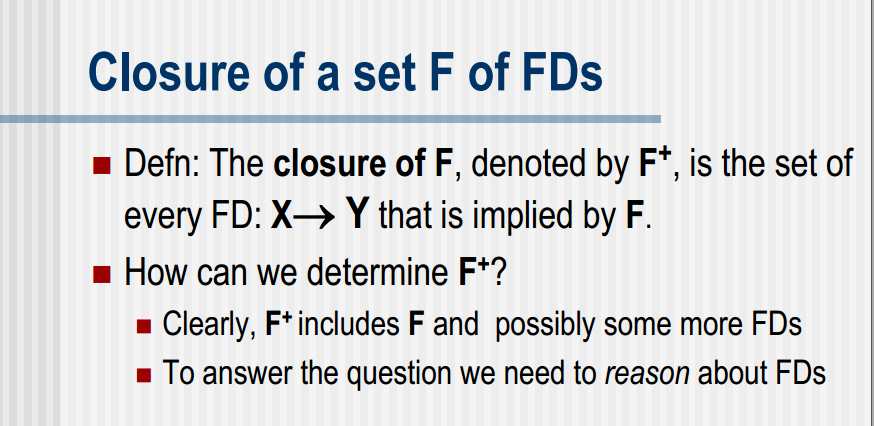
证明方法就是给一个反例

注意这里0和1并不代表true or false,他只是database里的值，类似ALICE,BRIAN这种

因此t1里面可以A1=0,A2=1他只是代表当A1=0时，A2是1，而不是false->true

因此A5=1时他并没有推出相等的1，因此A5不能决定A1

**上下两个tuple都要满足F**



F的closure，记作F^+, 它包含了F所能推导出来的所有X->Y

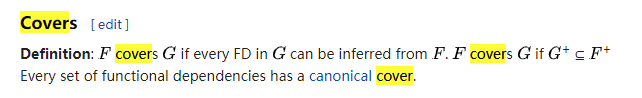
首先F+显然包含了F，或许还包含额外的可以推导出来但不在F之内的FD

两组FD相等问题

R:relation schema

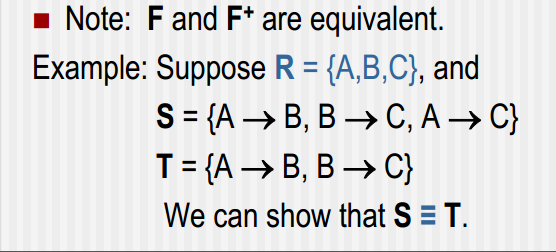
S,T: SETS of FDs on R

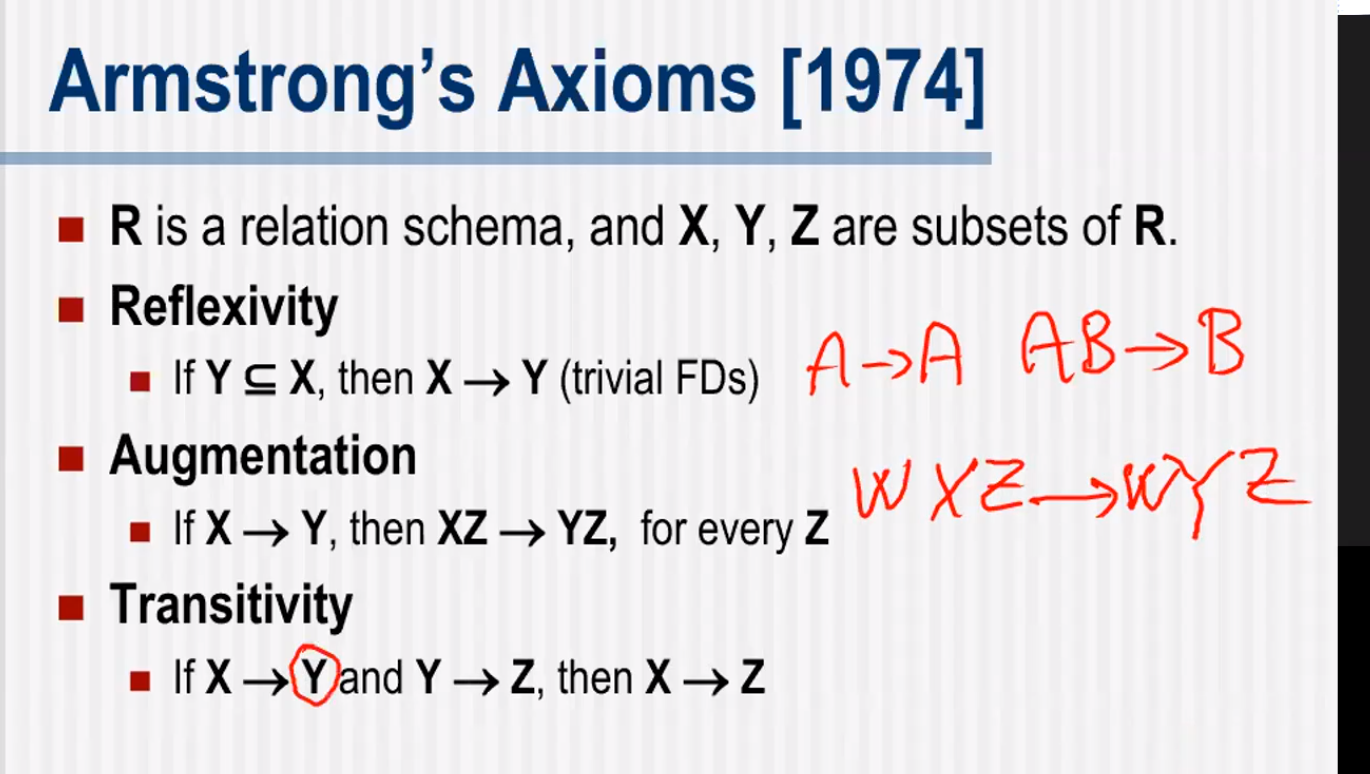
我们说S covers T,



EQUIVALEMT: S covers T且T covers S

注意F与F+是equivalent的//因为他们虽然表面不同，但是我们可以通过逻辑从F推导出完成体F+， 而COVERS也是建立在CLOSURE上的

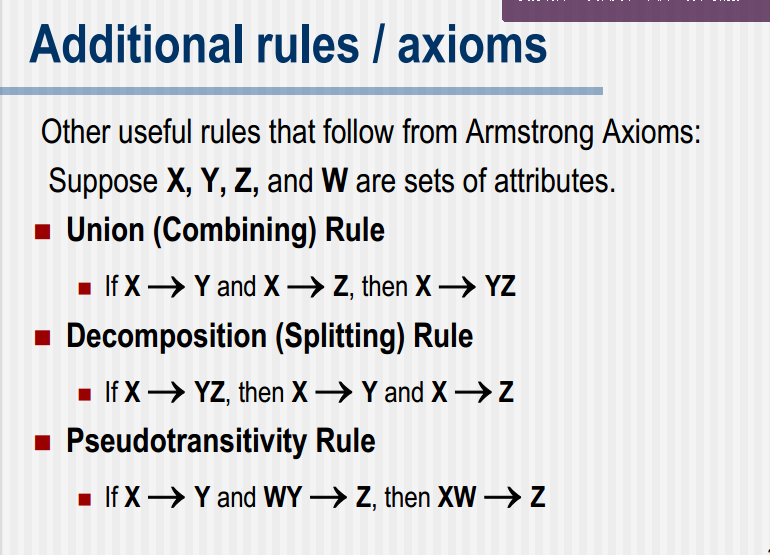




X, 一组attribute

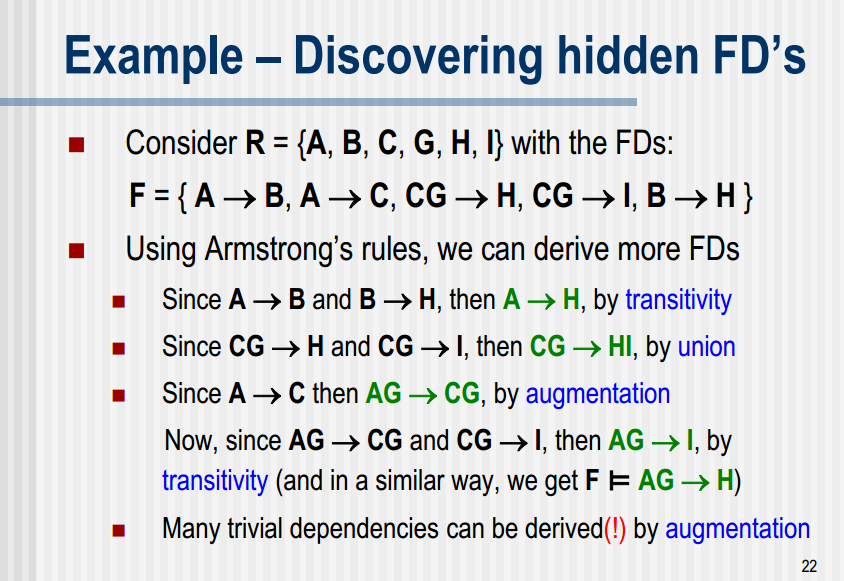
Y,Z同上

XZ代表两组attribute加在一起



伪递移法则

就是把X传到第二个里面



隐藏性问题：

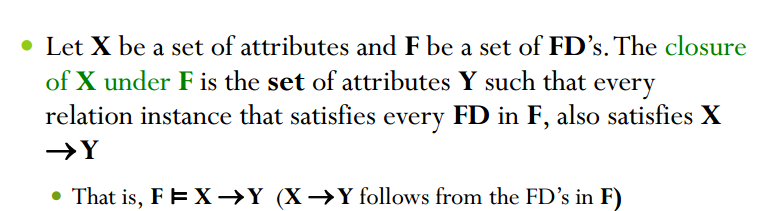
给你一组FD F, 我们怎么知道X->Y这个FD是否在F里，或者更准确的说，在F的closure F+L里

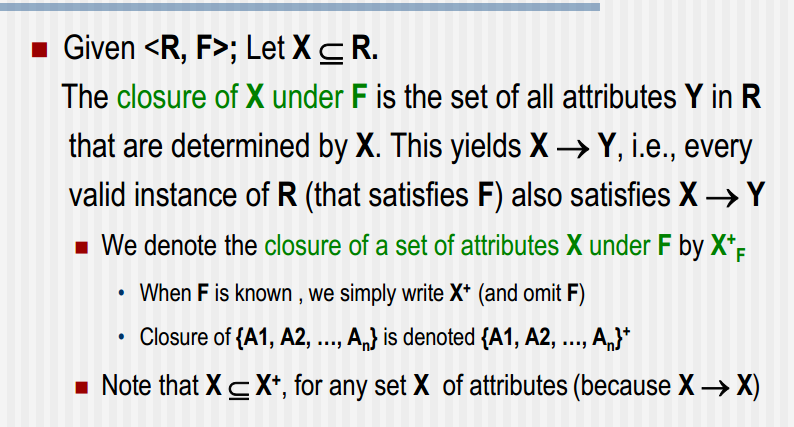
但是计算F+很耗时间

引入新概念：部分Closure

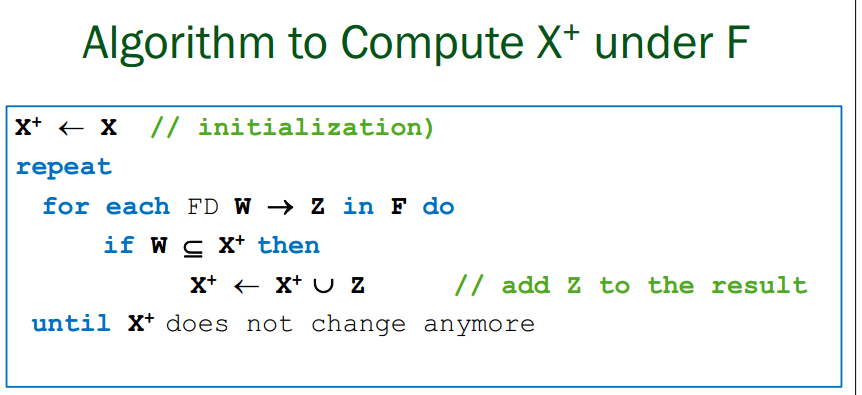
Closure of X under F, X是我们关注的attribute, F是所有FD的集合， //X可以实际和

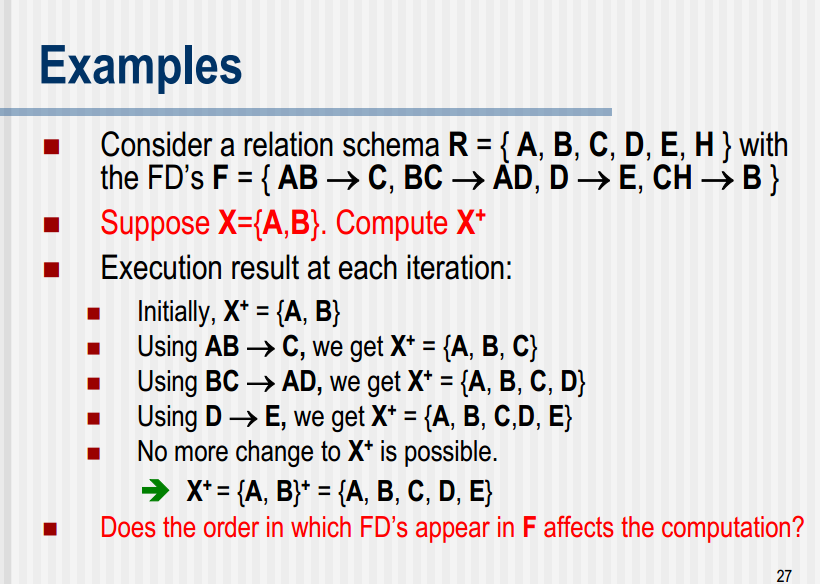
closure of X under F ： 只要是valid relation instance(满足 F中的每一个FD) ，那么必然满足X->Y // Y是所有X能推出来的东西的集合





那么我们怎么计算X+呢





就是不停的遍历FD集合F，

X+在这个过程中是不停变动的

第一步，初始化X+=X

=AB

遍历F，AB->C， 显然W也就是AB属于X+.

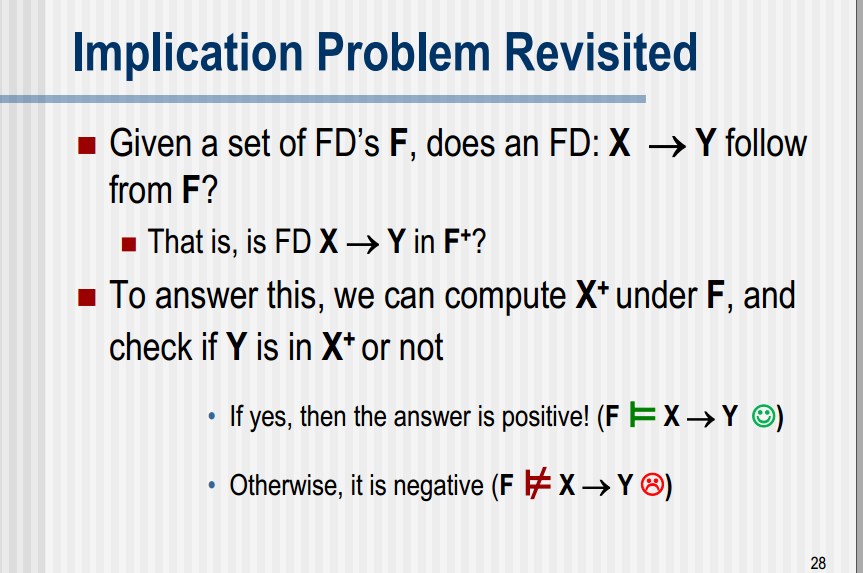
因此X+更新， X+=X+ **U** C=ABC //这个U是并列的意思

然后遍历F，BC->AD ，属于X+, X+=X+ U AD= ABCD

同理ABCDE

而CH不属于X+ ,无法遍历

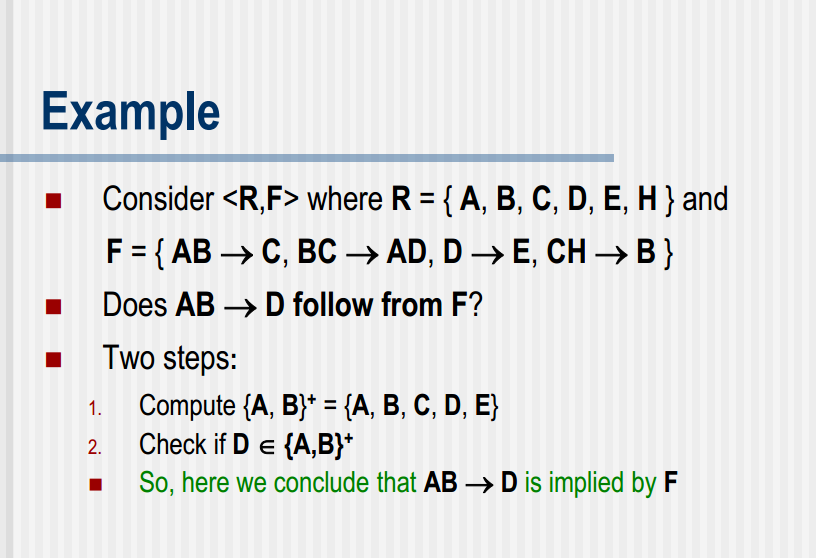
因此最终X+=ABCDE



所以证明X->Y是不是F+里的一份子的时候，

我们可以计算X+ under F, 如果Y在X+里

那么F就能推出X->Y



AB->D是不是F+里面的

第一计算AB+

看D在不在AB+里

Cloure and keys

假设X+包括了relation R的所有Attribute

那么X是一个（super） key of R

那么为了验证X是不是一个candidate key,我们需要检查两个事情

1.x+是否是superkey R 且X+=R

2. X的子集并不包含key， 任意A属于X， （X-A）+ ≠R// 意思是去除了A以后，剩下的无法再作为key，因为他们不能推导出所有attribute