**Database Chapter Seven Outline**

Trivial:

*A* functional dependency is trivial if it is satisfied by all instances of a relation

*Closure:*

The set of all functional dependencies logically implied by *F* is the *closure* of *F*.

We denote the *closure* of *F* by F*+.*

F+ is a superset of *F*.

BCNF:

A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in *F*+ of the form

α → *β*

where α ⊆ *R* and *β* ⊆ *R*,at least one of the following holds:

α → *β* is trivial (i.e., *β* ⊆ α)

α is a superkey for *R*

**Decomposing a Schema into BCNF**

Suppose we have a schema *R* and a non-trivial dependency α →*β* causes a violation of BCNF.

We decompose *R* into:

* + (α U β)
  + ( *R* - ( *β -* α ) )

可能会进行BCNF分解时会产生更多不属于BCNF的结果模式，在这种情况下要进行进一步的分解，直至最后产生的结果是一个BCNF模式的集合。

***dependency preserving：***

If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is ***dependency preserving.***

Decide *a decomposition is good or bad*

Lossless-join decide whether the relation is able todecompose or not

Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as ***third normal form.***

模式分解的目标：保持依赖的，无损分解，坚持BCNF模式的分解可能会导致非保持依赖的。

**Third Normal Form**

A relation schema *R* is in third normal form (3NF) if for all:

α → *β* in *F*+  
at least one of the following holds:

α → *β* is trivial (i.e., *β* ∈ α)

α is a superkey for *R*

Each attribute *A* in *β* – α is contained in a candidate key for *R.*

(**NOTE***:* each attribute may be in a different candidate key)

Third condition is a minimal relaxation of BCNF to ensure dependency preservation

**Closure of a Set of Functional Dependencies**

Given a relational schema *R*, a functional dependency *f* on *R* is **logically implied** by a set functional dependencies *F* on *R* if every relation instance *r(R)* that satisfies *F* also satisfies *f*.

if *β* ⊆ α, then α → *β* **(reflexivity)**

if α → *β,* then γ α → γ *β* **(augmentation)**

if α → *β,* and *β* → γ, then α → γ **(transitivity)**

If α → *β* holds *a*nd α→ γ holds, then α → *β* γ holds **(union)**

If α → *β* γ holds, then α → *β* holds and α→ γ holds **(decomposition)**

If α → *β* holds *a*nd γ *β* → δ holds, then α γ → δ holds **(pseudotransitivity)**

To compute the closure of a set of functional dependencies F:

*F* + = *F*  
**repeat**  
 **for each** functional dependency *f* in *F*+  apply reflexivity and augmentation rules on *f* add the resulting functional dependencies to *F* + **for each** pair of functional dependencies *f*1and *f*2 in *F* +  **if** *f*1 and *f*2 can be combined using transitivity  
 **then** add the resulting functional dependency to *F* +**until** *F* + does not change any further

**Closure of Attribute Sets**

Given a set of attributes a, define the *closure* of a under *F* (denoted by a+) as the set of attributes that are functionally determined by a under *F*

Algorithm to compute a+, the closure of a under *F  
 result* := a;  
 **while** (changes to *result*) **do  
 for each** β→ γ **in** *F* **do  
 begin  
 if** β⊆ *result* **then**  *result* := *result* ∪ γ   
 **end**

**Uses of Attribute Closure**

esting for superkey:

Testing functional dependencies

Computing closure of F

**Computation of candidate key**

If attribute A doesn’t appear in any side of *f*, then A should be part of candidate key

If attribute A appear only in the left side of *f*, then A should be part of candidate key

If attribute A appear only in the right side of *f*, then A would not be part of candidate key

If attribute A appear in both sides of *f*, then A should be tested further, and Attribute Closure could be used.

当属性A不出现在函数依赖两边，或者只出现在左边时候，A应该是候选码的一部分。

出现在两边时，应该进一步计算其闭包。

**Canonical Cover 正则闭包**

Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

To test if attribute A ∈ α is extraneous in α

* 1. compute ({α} – A)+ using the dependencies in *F 用原来函数依赖关系*
  2. check that ({α} – A)+ contains β; if it does, *A* is extraneous

To test if attribute *A* ∈ β is extraneous in β 用去除属性后的函数依赖关系

1. compute α+  using only the dependencies in   
    F’ = (*F* – {α → β}) ∪ {α →*(*β– *A*)},
2. check that α+  contains *A;* if it does*, A* is extraneous

**A *canonical cover* for *F* is a set of dependencies *Fc* such that**

*F* logically implies all dependencies in *Fc,* and

*Fc*logically implies all dependencies in *F,* and

No functional dependency in *Fc* contains an extraneous attribute

Each left side of functional dependency in *Fc* is unique

To compute a canonical cover for *F*:

**repeat** Use the union rule to replace any dependencies in *F* α1 → β1 and α1 → β2 with α1 → β1 β2   
 Find a functional dependency α → β with an   
 extraneous attribute either in α or in β   
 If an extraneous attribute is found, delete it from α → β  
**until** *F* does not change

**Lossless-join Decomposition**

A decomposition of *R* into *R*1 and *R*2 is lossless join if and only if at least one of the following dependencies is in *F*+:

*R*1 ∩ *R*2 → *R*1

*R*1 ∩ *R*2 → *R*2

*如果R*1 ∩ *R*2 是 *R*1 *或者R*2 *的超码，则是无损分解。*

**Dependency Preservation**

A decomposition is dependency preserving, if

(*F*1∪ *F*2 ∪ *…* ∪ *F*n )+ = *F +*

方法一：计算F'+ 与F+ 比较是否相等。

方法二：

To check if a dependency α → β is preserved in a decomposition of *R* into *R*1, *R*2, …, *R*n we apply the following test (with attribute closure done with respect to *F*)

*result* = α  
while (changes to *result*) do  
 for each *Ri* in the decomposition  
 *t* = (*result* ∩ *Ri*)+ ∩ *Ri result = result* ∪ *t*

If *result* contains all attributes in β, then the functional dependency α → β is preserved.

**Testing Decomposition for BCNF**

for every set of attributes α ⊆ *Ri*, check that α+ (the attribute closure of α under F) either includes no attribute of *Ri*- α, or includes all attributes of *Ri*.

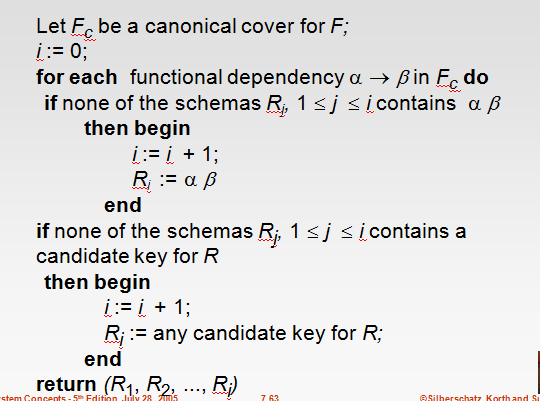
* + - If the condition is violated by some α → *β* in *F+*, the dependency  
       α → (α+ - α ) ∩ *Ri*can be shown to hold on *Ri*, and *Ri* violates BCNF.

**BCNF Decomposition Algorithm**

*result* := {*R* };  
*done* := false;  
compute *F* +;  
**while (not** *done)* **do  
 if** (there is a schema *Ri* in *result*  that is not in BCNF)  
 **then begin** let α → *β*  be a nontrivial functional dependency that holds on *Ri*

such that α → *Ri* is not in *F* +, and α ∩ *β =* ∅;  
 *result* := (*result – Ri )* ∪ (*Ri – β*) ∪ (α, *β );* **end  
 else** *done* := **true;**

**3NF Decomposition Algorithm**



Above algorithm ensures:

each relation schema *Ri* is in 3NF

decomposition is dependency preserving and lossless-join

Proof of correctness is at end of this file

**Comparison of BCNF and 3NF**

It is always possible to decompose a relation into a set of relations that are in 3NF such that:

the decomposition is lossless

the dependencies are preserved

It is always possible to decompose a relation into a set of relations that are in BCNF such that:

the decomposition is lossless

it may not be possible to preserve dependencies.

Goal for a relational database design is:

BCNF.

Lossless join.

Dependency preservation.

If we cannot achieve this, we accept one of

Lack of dependency preservation

Redundancy due to use of 3NF