CSE 132B
Database Systems Applications

Alin Deutsch

Database Design and Normal Forms

Some slides are based or modified from originals by
Sergio Lifschitz @ PUC Rio, Brazil
and
Victor Vianu @ CSE UCSD
and
Database System Concepts, McGraw Hill 5th Edition
© 2005 Silberschatz, Korth and Sudarshan

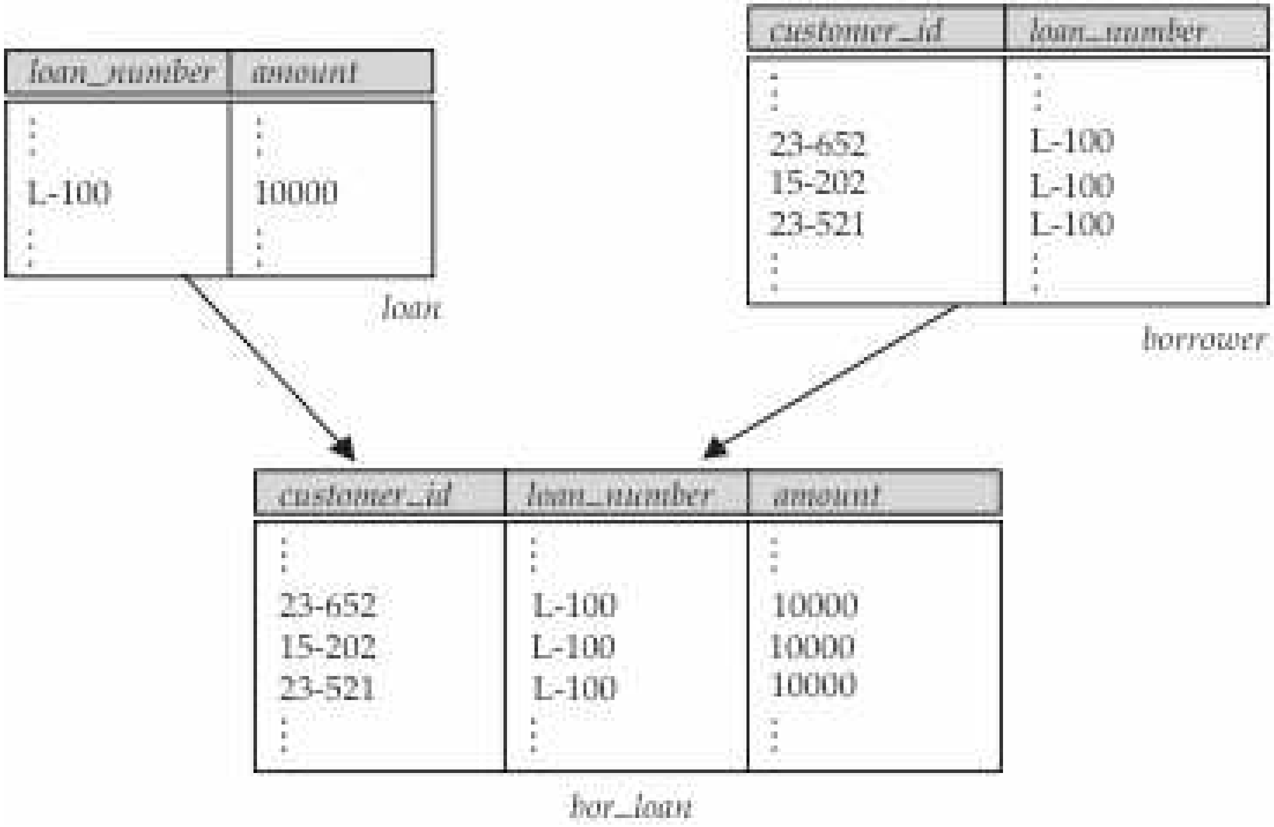
The Banking Schema

- *branch = (branch_name, branch_city, assets)*
- *customer = (customer_id, customer_name, customer_street, customer_city)*
- *loan = (loan_number, amount)*
- *account = (account_number, balance)*
- *employee = (employee_id. employee_name, telephone_number, start_date)*
- *dependent_name = (employee_id, dname)*
- *account_branch = (account_number, branch_name)*
- *loan_branch = (loan_number, branch_name)*
- *borrower = (customer_id, loan_number)*
- *depositor = (customer_id, account_number)*
- *cust_banker = (customer_id, employee_id, type)*
- *works_for = (worker_employee_id, manager_employee_id)*
- *payment = (loan_number, payment_number, payment_date, payment_amount)*
- *savings_account = (account_number, interest_rate)*
- *checking_account = (account_number, overdraft_amount)*

Combine Schemas?

- Suppose we combine *borrow* and *loan* to get

  bor_loan = (*customer_id*, *loan_number*, *amount*)
- Result is possible repetition of information (L-100 in example below)



loan_number	*amount*
⋮	⋮
L-100	10000
⋮	⋮

loan

customer_id	*loan_number*
⋮	⋮
23-652	L-100
15-202	L-100
23-521	L-100
⋮	⋮

borrower

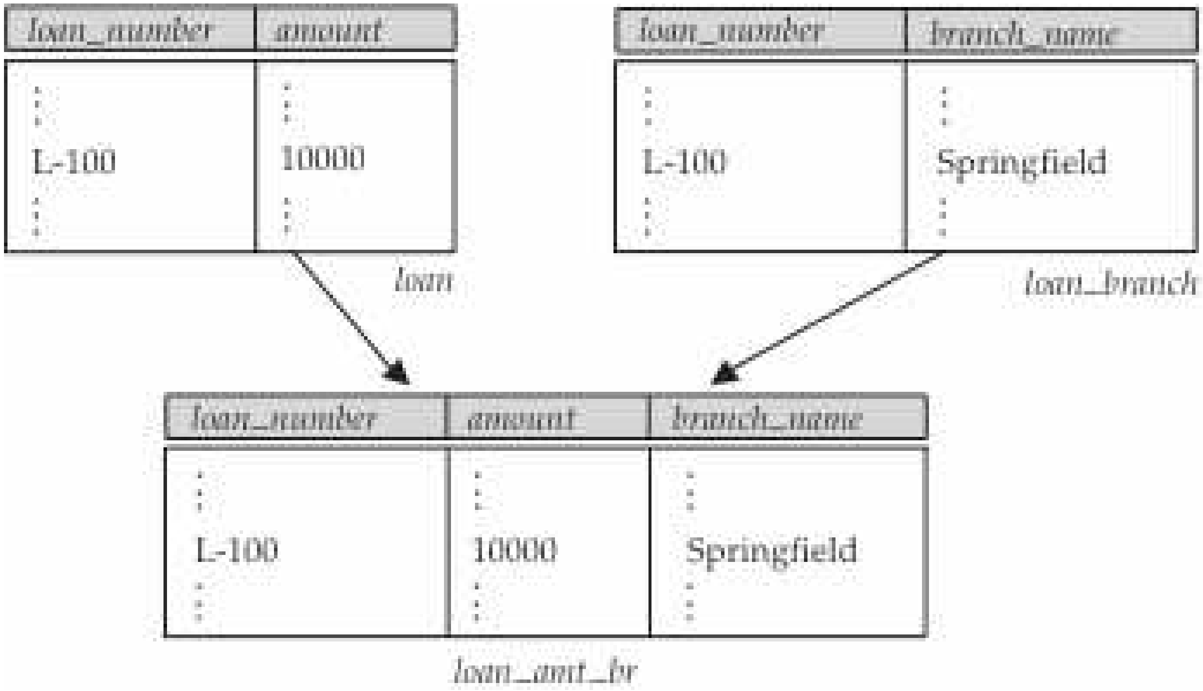
customer_id	*loan_number*	*amount*
⋮	⋮	⋮
23-652	L-100	10000
15-202	L-100	10000
23-521	L-100	10000
⋮	⋮	⋮

bor_loan

A Combined Schema Without Repetition

- Consider combining *loan_branch* and *loan*

  loan_amt_br = (*loan_number*, *amount*, *branch_name*)
- No repetition (as suggested by example below)



loan_number	*amount*
⋮	⋮
L-100	10000
⋮	⋮

loan

loan_number	*branch_name*
⋮	⋮
L-100	Springfield
⋮	⋮

loan_branch

loan_number	*amount*	*branch_name*
⋮	⋮	⋮
L-100	10000	Springfield
⋮	⋮	⋮

loan_amt_br

What About Smaller Schemas?

- Suppose we had started with *bor_loan.* How would we know to split up (**decompose**) it into *borrower* and *loan*?
- Write a rule “if there were a schema (*loan_number, amount*), then *loan_number* would be a candidate key”
- Denote as a **functional dependency**:

  $$loan\_number \rightarrow amount$$

- In *bor_loan*, because *loan_number* is not a candidate key, the amount of a loan may have to be repeated. This indicates the need to decompose *bor_loan*.

What About Smaller Schemas? (cont.)

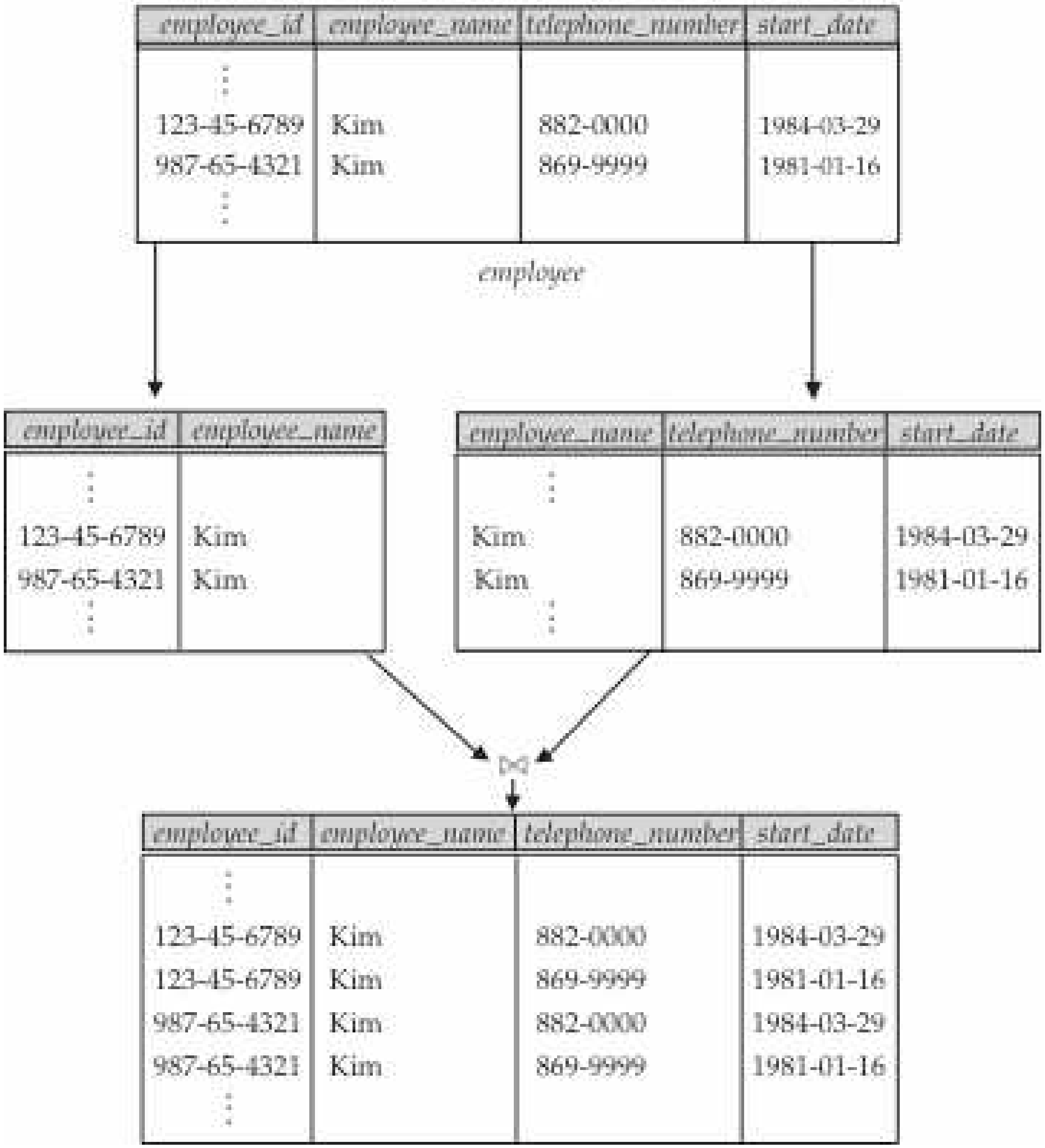
- Not all decompositions are good. Suppose we decompose *employee* into

  employee1 = (*employee_id*, *employee_name*)

  employee2 = (*employee_name*, *telephone_number*, *start_date*)

- However, we lose information
  - we cannot reconstruct the original *employee* relation
  - this is called a *lossy decomposition*

A Lossy Decomposition



employee_id	employee_name	telephone_number	start_date
⋮			
123-45-6789	Kim	882-0000	1984-03-29
987-65-4321	Kim	869-9999	1981-01-16
⋮			

employee

employee_id	employee_name
⋮	
123-45-6789	Kim
987-65-4321	Kim
⋮	

employee_name	telephone_number	start_date
⋮		
Kim	882-0000	1984-03-29
Kim	869-9999	1981-01-16
⋮		

employee_id	employee_name	telephone_number	start_date
⋮			
123-45-6789	Kim	882-0000	1984-03-29
123-45-6789	Kim	869-9999	1981-01-16
987-65-4321	Kim	882-0000	1984-03-29
987-65-4321	Kim	869-9999	1981-01-16
⋮			

Goal — Devise a Theory for the Following

- Decide whether a particular relation R is in “good” form.
- In the case that a relation R is not in “good” form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on functional dependencies

Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key.*

Functional Dependencies (Cont.)

- Let *R* be a relation schema

$$\alpha \subseteq R \;\; \mathit{and} \;\; \beta \subseteq R$$

- The functional dependency

$$\alpha \rightarrow \beta$$

holds on *R*

if and only if for any legal relations *r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes β.

That is,

$$t_1[\alpha] = t_2\,[\alpha] \;\; \Rightarrow \;\; t_1[\beta\,] = t_2\,[\beta\,]$$

Functional Dependencies (cont.)

- Dependencies: statements about properties of valid data
  - e.g.: “Every student is a person”
    - inclusion dependency
  - “Each employee works in no more than one department”
    - NAME → DEPARTMENT
    - functional dependency
- Use of dependencies:
  - check data integrity
  - query optimization
  - schema design → “normal forms”

Functional Dependencies (Cont.)

- Example: Consider *r*(A,*B*) with the following instance of *r.*

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold
  - but $B \rightarrow A$ does hold.

Functional Dependencies (cont.)

- Functional dependency over R:
  - expression $X \rightarrow Y$ where $X, Y \subseteq att(R)$
- A relation R satisfies $X \rightarrow Y$ iff
  whenever two tuples in R agree on X, they also agree on Y

e.g.

SCHEDULE	THEATER	TITLE
	la jolla	aviator
	hillcrest	crash

Satisfies **THEATER → TITLE**

SCHEDULE	THEATER	TITLE
	la jolla	aviator
	hillcrest	crash
	hillcrest	matrix

Violates **THEATER → TITLE**, satisfies **TITLE → THEATER**

Functional Dependencies (Cont.)

- *K* is a **superkey** for relation schema *R* if and only if $K \rightarrow R$
- *K* is a **candidate key** for *R* if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$

- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  bor_loan = (*<u>customer_id, loan_number</u>, amount*).

  We expect this functional dependency to hold:

  loan_number → *amount*

  but would not expect the following to hold:

  amount → *customer_name*

Functional Dependencies (Cont.)

- *A* functional dependency is trivial if it is satisfied by all instances of a relation
  - Example*:*
    - *customer_name, loan_number* → *customer_name*
    - *customer_name* → *customer_name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Use of Functional Dependencies

- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* satisfies *F.*
  - specify constraints on the set of legal relations
    - We say that *F* holds on *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *loan* may, by chance, satisfy

    $amount \rightarrow customer\_name.$

Goals of Normalization

- Let *R* be a relation scheme with a set *F* of functional dependencies.
- Decide whether a relation scheme *R* is in “good” form.
- In the case that a relation scheme *R* is not in “good” form, decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

Normal Forms

- Terminology:
  - Let R be a relation schema and F a set of FD’s over R.
  - Key: $X \subseteq att(R)$ such that $X \rightarrow att(R)$.
  - Minimal key: $X \subseteq att(R)$ s.t. $X \rightarrow att(R)$

    and there is no $Y \subset\neq X$ such that $Y \rightarrow att(R)$.
  - $A \in att(R)$ is prime: $A \in X$ where X is a minimal key
  - A is non-prime: A is not a member of any minimal key.
- Obs. We could have used above
  - Super Key (instead of Key) and
  - Candidate Key (instead of Minimal Key)
- Purpose of normal forms:
  - Eliminate problems of redundancy and anomalies.

Boyce-Codd Normal Form

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

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for *R*

Example schema *not* in BCNF:

bor_loan = (*customer_id, loan_number, amount*)

because

loan_number → *amount* holds on *bor_loan* but *loan_number* is not a superkey

Another Example

- BAD(S#, P#, SNAME, PNAME, SCITY, PCITY, QTY)
  - not in BCNF wrt F:

    S# → SNAME SCITY

    P# → PNAME PCITY

    S# P# → QTY

  - S(S#, SCITY, SNAME) is in BCNF wrt S# → SNAME SCITY
  - P(P#, PCITY, PNAME) is in BCNF wrt P# → PNAME PCITY
  - SP(S# P# QTY) is in BCNF wrt S# P# → QTY

Decomposing a Schema into BCNF

- Suppose we have a schema *R* and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

  We decompose *R* into:

  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$

- In our previous banking example,

  - α = *loan_number*
  - β = *amount*

  and *bor_loan* is replaced by

  - $(\alpha \cup \beta)$ = (*loan_number, amount*)
  - $(R - (\beta - \alpha))$ = (*customer_id, loan_number*)

BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- 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****.*
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

Third Normal Formal (3NF)

- Problem with BCNF:
  - Not every relation schema can be decomposed into BCNF relation schemas which preserve the dependencies and have lossless join.

- Third Normal Form
  - A relation scheme R is in **Third Normal Form** wrt a set F of fd’s over R, if whenever $X \rightarrow A$ holds in R and $A \notin X$ then
either X is a key or A is prime

Weaker than BCNF

Third Normal Form (cont.)

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

  $$\alpha \rightarrow \beta \text{ on R}$$

  at least one of the following holds:

  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
  - α is a superkey for *R*
  - Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R.*

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

- If a relation is in BCNF it is in 3NF
  - since in BCNF one of the first two conditions above must hold.
- Third condition is a minimal relaxation of BCNF that ensures dependency preservation.

Functional-Dependency Theory

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.
- We then develop algorithms to generate lossless decompositions into BCNF and 3NF
- We then develop algorithms to test if a decomposition is dependency-preserving

Closure of a Set of Functional Dependencies

- Given a set *F* set of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.
  - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow$ C
- The set of all functional dependencies logically implied by *F* is the *closure* of *F*.
- We denote the *closure* of *F* by F^+.

Closure of a Set of Functional Dependencies (cont.)

- We can find all of F^+ by applying Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
  - sound
    - generate only FDs that actually hold; and
  - complete
    - generate all FDs that hold.

Example (Armstrong)

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,

      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,

      and then transitivity

Procedure for Computing F^+

- 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 $f_1$and $f_2$ 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
```

NOTE: There is an alternative (and more efficient) procedure for this task!

Closure of Attribute Sets

- Given a set of attributes α, define the *closure* of α under F (denoted by α^+) as the set of attributes that are functionally determined by α under F
- Algorithm to compute α^+, the closure of α under F

```
result := $\alpha$;
while (changes to result) do
        for each $\beta \rightarrow \gamma$ in $F$ do
            begin
                if $\beta \subseteq result$ then  $result := result \cup \gamma$
            end
```

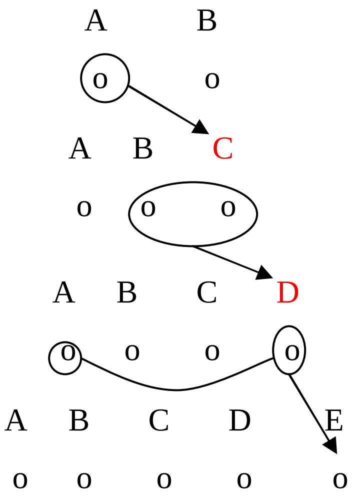
Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
     2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

Another Example of Closure Computation

- $R = ABCDEF$
- $F = \{A \rightarrow C, BC \rightarrow D\ AD \rightarrow E\}$
- $X = AB$

- $X^{(0)} = AB$
- $X^{(1)} = ABC$
- $X^{(2)} = ABCD$
- $X^{(3)} = ABCDE$
- $X^{(4)} = X^{(3)}$
- $X^{+} = ABCDE$



- To check if X is key in R: $X^{+} = R$

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute $\alpha^{+,}$ and check if α^+ contains all attributes of *R*.
- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
  - That is, we compute α^+ by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- **Computing closure of F**
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Extraneous Attributes

- Consider a set F of FDs and the functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow(\beta - A)\}$ logically implies F.

- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
  - B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (i.e. the result of dropping B from $AB \rightarrow C$).

- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - C is extraneous in $AB \rightarrow CD$ since
    $AB \rightarrow C$ can be inferred even after deleting C

Testing if an Attribute is Extraneous

- Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.
- To test if attribute $A \in \alpha$ is extraneous in α
  1. compute $(\{\alpha\} - A)^+$ using the dependencies in *F*
  2. check that $(\{\alpha\} - A)^+$ contains A; if it does, *A* is extraneous
- To test if attribute $A \in \beta$ is extraneous in β
  1. compute α^+ using only the dependencies in

     $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
  2. check that α^+ contains *A;* if it does, *A* is extraneous

Canonical (Minimal) Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to
      $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
    - E.g.: on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to
      $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
- 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

Canonical (minimal) Cover

- A *canonical cover* for *F* is a set of dependencies F_c such that
  - *F* logically implies all dependencies in $F_{c,}$ and
  - F_c logically implies all dependencies in *F,* and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique.

- To compute a canonical cover for *F*:

```
repeat
    Use the union rule to replace any dependencies in $F$
        $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
    Find a functional dependency $\alpha \rightarrow \beta$ with an
        extraneous attribute either in $\alpha$ or in $\beta$
    If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until $F$ does not change
```

- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes*:* using transitivity on $A \rightarrow B$ *and* $B \rightarrow$ C.
      - Can use attribute closure of A in more complex cases
- The canonical cover is: $A \rightarrow B$
  $B \rightarrow C$

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- 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^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
  - Can be decomposed in two different ways
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition:

    $$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

  - Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless-join decomposition:

    $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.
  - A decomposition is dependency preserving, if

$$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of *R* into $R_1, R_2, \ldots, R_n$ we apply the following test (with attribute closure done with respect to *F*)

  -
    ```
    result = $\alpha$
    while (changes to result) do
      for each $R_i$ in the decomposition
              $t = (result \cap R_i)^+ \cap R_i$
              $result = result \cup t$
    ```
  - If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved.
- We apply the test on all dependencies in *F* to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $\quad B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 and R_2 in BCNF
  - Lossless-join decomposition
  - Dependency preserving

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  1. compute α^+ (the attribute closure of α), and
  2. verify that it includes all attributes of *R*, that is, it is a superkey of *R*.
- Simplified test: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in F^+.
  - If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.
- However, using only *F* is incorrect when testing a relation in a decomposition of R
  - Consider $R = (A, B, C, D, E)$, with $F = \{ A \rightarrow B, BC \rightarrow D\}$
    - Decompose *R* into $R_1 = (A,B)$ and $R_2 = (A,C,D, E)$
    - Neither of the dependencies in *F* contain only attributes from (A,C,D,E) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

- To check if a relation R_i in a decomposition of R is in BCNF,
  - Either test R_i for BCNF with respect to the restriction of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
  - or use the original set of dependencies F that hold on R, but with the following test:
    - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.
  - If the condition is violated by some $\alpha \rightarrow \beta$ in F, the dependency

    $$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$

    can be shown to hold on R_i, and R_i violates BCNF.
  - We use above dependency to decompose R_i

BCNF Decomposition Algorithm

```
result := {R };
done := false;
compute $F^+$;
while (not done) do
   if (there is a schema $R_i$ in result  that is not in BCNF)
      then begin
            let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on $R_i$
                        such that $\alpha \rightarrow R_i$ is not in $F^+$,
                        and $\alpha \cap \beta = \emptyset$;
              result := $(result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
            end
      else done := true;
```

Note: each R_i is in BCNF, and decomposition is lossless-join.

Example 1 of BCNF Decomposition

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $\quad B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF ($B \rightarrow C$ but B is not a superkey)
- Decomposition
  - $R_1 = (B, C)$
  - $R_2 = (A,B)$

Example 2 of BCNF Decomposition

R = C T H R S G

C = course

T = teacher

H = hour

R = room

S = student

G = grade

F: $C \rightarrow T$

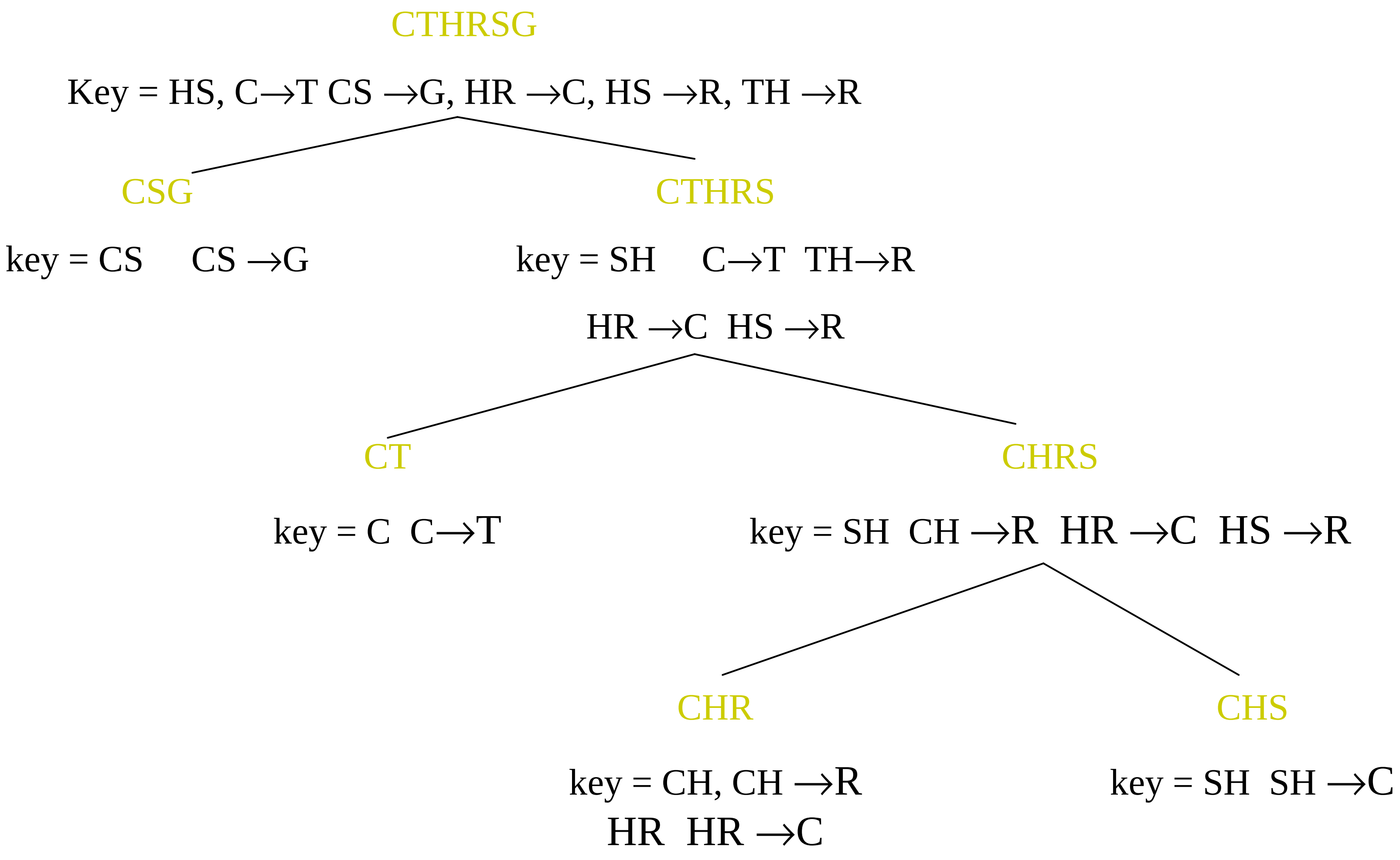
$HR \rightarrow C$

$HT \rightarrow R$

$CS \rightarrow G$

$HS \rightarrow R$ only key: HS

Example 2 of BCNF Decomposition (cont.)



CTHRSG

Key = HS, C→T CS →G, HR →C, HS →R, TH →R

CSG

key = CS CS →G

CTHRS

key = SH C→T TH→R

HR →C HS →R

CT

key = C C→T

CHRS

key = SH CH →R HR →C HS →R

CHR

key = CH, CH →R

HR HR →C

CHS

key = SH SH →C

Remarks (drawbacks)

- Decomposition is not unique
  - CHRS could be decomposed into CHR and CHS or CHR and RHS
- The decomposition does not preserve TH→R
  - local fds:

$$G = \{CS \rightarrow G \quad C \rightarrow T \quad CH \rightarrow R \quad HR \rightarrow C \quad SH \rightarrow C\}$$

which does not imply $TH \rightarrow R$

Example 3 of BCNF Decomposition

- Original relation R and functional dependency F

  R = (*branch_name, branch_city, assets,*
  customer_name, loan_number, amount)

  F = {*branch_name* → *assets branch_city*
  loan_number → *amount branch_name* }

  Key = {*loan_number, customer_name*}

- Decomposition
  - R_1 = (*branch_name, branch_city, assets*)
  - R_2 = (*branch_name, customer_name, loan_number, amount*)
  - R_3 = (*branch_name, loan_number, amount*)
  - R_4 = (*customer_name, loan_number*)

- Final decomposition

  R_1, R_3, R_4

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

- $R = (J, K, L)$
  $F = \{JK \rightarrow L$
  $L \rightarrow K\}$
  Two candidate keys = *JK* and *JL*
- *R* is not in BCNF
- Any decomposition of *R* will fail to preserve

$$JK \rightarrow L$$

This implies that testing for $JK \rightarrow L$ requires a join

Third Normal Form: Motivation

- There are some situations where
  - BCNF is not dependency preserving, and
  - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy (with resultant problems; we will see examples later)
  - But functional dependencies can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example

- Relation R:
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$
  - Two candidate keys: *JK* and *JL*
  - *R* is in 3NF

    $JK \rightarrow L$ *JK* is a superkey
    $L \rightarrow K$ *K* is contained in a candidate key

Redundancy in 3NF

- There is some redundancy in this schema
- Example of problems due to redundancy in 3NF
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

- repetition of information (e.g., the relationship l_1, k_1)
- need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J).

Testing for 3NF

- Optimization: Need to check only FDs in *F*, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of *R*
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

3NF Decomposition Algorithm

```
Let $F_c$ be a canonical cover for $F$;
$i$ := 0;
for each functional dependency $\alpha \rightarrow \beta$ in $F_c$ do
  if none of the schemas $R_j$, $1 \leq j \leq i$ contains $\alpha\ \beta$
        then begin
              $i := i + 1$;
              $R_i := \alpha\ \beta$
          end
if none of the schemas $R_j$, $1 \leq j \leq i$ contains a candidate key for $R$
  then begin
          $i := i + 1$;
          $R_i$ := any candidate key for $R$;
        end
return $(R_1, R_2, ..., R_i)$
```

3NF Decomposition Algorithm (Cont.)

- 3NF decomposition algorithm ensures:
  - each relation schema R_i is in 3NF
  - decomposition is dependency preserving and lossless-join

Example 3NF decomposition

- Relation schema:

  cust_banker_branch = (*customer_id, employee_id*, *branch_name, type*)

- The functional dependencies for this relation schema are:

  customer_id, employee_id → *branch_name, type*
  employee_id → *branch_name*

- The **for** loop generates:

  (*customer_id, employee_id, branch_name, type*)

  It then generates

  (*employee_id*, *branch_name*)

  but does not include it in the decomposition because it is a subset of the first schema.

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.

Design Goals

- 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
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

  Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database

  classes (*course, teacher, book*)

  such that (*c, t, b*) $\in$ *classes* means that *t* is qualified to teach *c*, and *b* is a required textbook for *c*
- The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

How good is BCNF? (Cont.)

course	*teacher*	*book*
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	Pete	Stallings

classes

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted

(database, Marilyn, DB Concepts)
(database, Marilyn, Ullman)

How good is BCNF? (Cont.)

- Therefore, it is better to decompose *classes* into:

course	*teacher*
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	*book*
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF)