Database Design

Normalization

Database Design

1. *Requirements Analysis:* in this step, we must point out

2. *Conceptual Database Design:* develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold on this data.

3. *Logical Database Design*: convert the conceptual database design into a database schema within the data model of the chosen DBMS.

4. ***Schema Refinement*:** the schemas developed in step 3 are analyzed for potential problems, then are ***normalized***.

5. *Physical Database Design:* potential workloads and access patterns are simulated to identify potential weaknesses in the conceptual database. This will often cause the creation of additional indices and/or clustering relations.

6. *Security Design:* Different user groups are identified and their different roles are analyzed so that access patterns to the data can be defined.

Normalization

Normalization is a technique for producing a set of relations with desirable properties, given the data requirements of the enterprise being modeled.

The process of normalization was first developed by Codd in 1972.

Normalization is often performed as a series of tests on a relation to determine whether it satisfies or violates the requirements of a given normal form.

Codd initially defined three normal forms called first (1NF), second (2NF), and third (3NF). Boyce and Codd together introduced a stronger definition of 3NF called Boyce-Codd Normal Form (BCNF) in 1974.

Normalization

Attributes may be added to a relational schema based largely on the common sense of the database designer, or by mapping the relational schema from an ERD.

A formal method is often required to help the database designer identify the optimal grouping of attributes for each relation schema.

The process of normalization is a formal method that identifies relations based on their primary or candidate keys and the functional dependencies among their attributes.

Normalization is a series of tests, which can be applied to individual relations so that a relational schema can be normalized to a specific form to prevent the possible occurrence of update anomalies.

Why Normalization?

The major aim of relational database design is to group attributes into relations to minimize data redundancy and thereby reduce the file storage space required by the implemented base relations, and to avoid any anomaly.

In the following relation schema, anomalies may appear during any data manipulation processes: modification, deletion, addition

The Need of Normalization

staffbranch

staff#	sname	position	salary	branch#	baddress
SL21	Kristy	manager	30000	B005	22 Deer Road
SG37	Debi	assistant	12000	B003	162 Main Street
SG14	Alan	supervisor	18000	B003	163 Main Street
SA9	Traci	assistant	12000	B007	375 Fox Avenue
SG5	David	manager	24000	B003	163 Main Street
SL41	Anna	assistant	10000	B005	22 Deer Road

In the “staffbranch” relation, it is hard to avoid anomalies during data manipulation processes

The Need of Normalization

Staff

staff#	sname	position	salary	branch#
SL21	Kristy	manager	30000	B005
SG37	Debi	assistant	12000	B003
SG14	Alan	supervisor	18000	B003
SA9	Traci	assistant	12000	B007
SG5	David	manager	24000	B003
SL41	Anna	assistant	10000	B005

Branch

branch#	baddress
B005	22 Deer Road
B003	163 Main Street
B007	375 Fox Avenue

In the two separated relations “Staff” and “Branch” above, the redundancy is removed, and all anomalies can be avoided.

Normalization

The examples of three types of update anomalies suffered by the ***staffbranch*** relation demonstrate that its decomposition into the ***Staff*** and ***Branch*** relations avoids such anomalies.

Two important properties associated with the decomposition of a larger relation into a set of smaller relations must be considered:

- The lossless-join property ensures that any instance of the original relation can be identified from corresponding instances of the smaller relations.
- The dependency preservation property ensures that a constraint on the original relation can be maintained by simply enforcing some constraint on each of the smaller relations. In other words, the smaller relations do not need to be joined together to check if a constraint on the original relation is violated.

The Lossless-join Property

SP

s#	p#	qty
S1	P1	10
S2	P2	50
S3	P3	10

Decomposition

S1

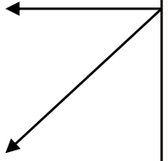
s#	qty
S1	10
S2	50
S3	10

S2

p#	qty
P1	10
P2	50
P3	10

$S1 \triangleright\triangleleft S2$

s#	p#	qty
S1	P1	10
S1	P3	10
S2	P2	50
S3	P1	10
S3	P3	10



These are extraneous tuples which did not appear in the original relation. However, now we can't tell which are valid and which aren't. Once the decomposition occurs the original SP relation is lost.

Preservation of Functional Dependencies

Example

R = (A, B, C)

F = {AB $\rightarrow$ C, C $\rightarrow$ A}

If R is decomposed into two relations RR = {(B, C), (A, C)}

Clearly C $\rightarrow$ A can be enforced on schema (A, C).

How can AB $\rightarrow$ C be enforced without joining the two relation schemas in K?

Answer, it can't, therefore the functional dependencies are not preserved in K.

Functional Dependency

All four of the normal forms, 1NF, 2NF, 3NF, BCNF, are based on functional dependencies among the attributes of a relation.

A functional dependency describes the relationship between attributes in a relation.

> For example, if A and B are attributes or sets of attributes of relation R, B is functionally dependent on A (denoted A $\rightarrow$ B), if each value of A is associated with exactly one value of B.

A functional dependency is a property of the semantics of the attributes in a relation, that indicate how attributes relate to one another.

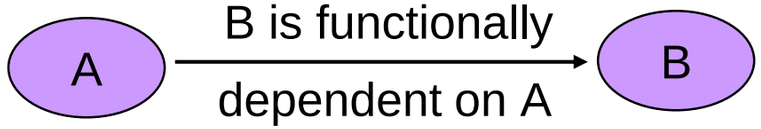
A functional dependency specifies a constraint between the attributes.

Functional Dependency

Consider a relation with attributes A and B:

If attribute B is functionally dependent on attribute A (denoted A $\rightarrow$ B), then if we know the value of A, we will find only one value of B in all of the tuples that have a given value of A, at any moment in time.

Note: for a given value of B there may be several different values of A.



The determinant of a functional dependency is the attribute or group of attributes on the left-hand side of the arrow in the functional dependency. The consequent of a FD is the attribute or group of attributes on the right-hand side of the arrow.

Functional Dependency

Revisit the ***Staff*** relation:

staff#	sname	position	salary	branch#

The functional dependency staff# → position holds on this relation instance since there is a 1:1 relationship from staff# to position (for each staff member there is only one position).

Here, the reverse functional dependency position → staff# does not hold, since the relationship between position and staff# is 1:M (there are several staff numbers associated with a given position).

For the purposes of normalization we are interested in identifying functional dependencies between attributes of a relation that have a 1:1 relationship.

Functional Dependency

A functional dependency is a property of a relational schema (its intension) and not a property of a particular instance of the schema (extension).

When identifying fds between attributes in a relation it is important to distinguish clearly between the values held by an attribute at a given point in time and the set of all possible values that an attributes may hold at different times.

This represents the types of integrity constraints that we need to identify. Such constraints indicate the limitations on the values that a relation can legitimately assume. In other words, they identify the legal instances which are possible.

Functional Dependency

Let's identify the functional dependencies that hold in the relation staffbranch.

staff#	sname	position	salary	branch#	baddress

In order to identify the time invariant fds, we need to clearly understand the semantics of the various attributes in the relation

staff# $\rightarrow$ sname, position, salary, branch#, baddress

branch# $\rightarrow$ baddress

baddress $\rightarrow$ branch#

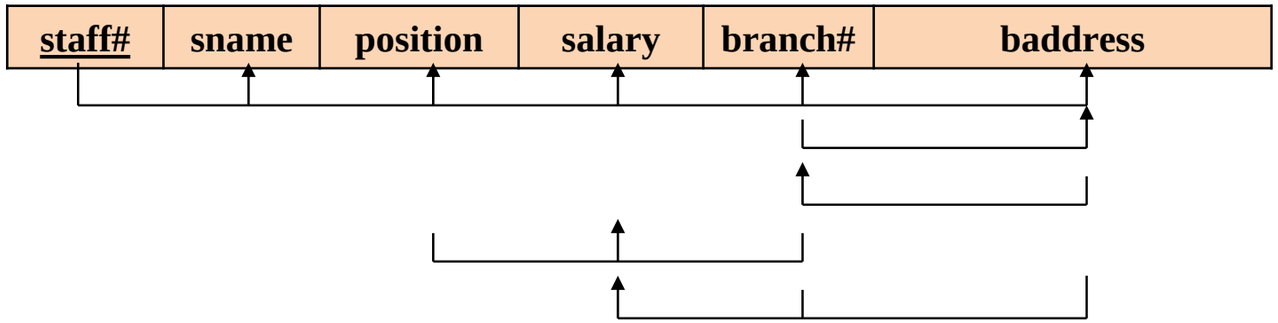
branch#, position $\rightarrow$ salary

baddress, position $\rightarrow$ salary

Functional Dependency

There is another way to present the 5 fds above as follows.

staffbranch



Trivial Functional Dependency

A functional dependency is trivial if and only if the consequent is a subset of the determinant. Here, it is impossible for it not to be satisfied.

Example: Using the relation ***Staff***, the trivial dependencies include:

$$\{ \text{staff\#}, \text{sname} \} \rightarrow \text{sname}$$

$$\{ \text{staff\#}, \text{sname} \} \rightarrow \text{staff\#}$$

Although trivial fds are valid, they offer no additional information about integrity constraints for the relation. As far as normalization is concerned, trivial fds are ignored.

Characteristics of Functional Dependency

In summary, the main characteristics of functional dependencies that are useful in normalization are:

1. There exists a 1:1 relationship between attribute(s) in the determinant and attribute(s) in the consequent.
2. The functional dependency is time invariant, i.e., it holds in all possible instances of the relation.
3. The functional dependencies are nontrivial. Trivial fds are ignored.

Inference Rules for Functional Dependency

IR1: reflexive rule – if $Y \subseteq X$, then $X \rightarrow Y$

IR2: augmentation rule – if $X \rightarrow Y$, then $XZ \rightarrow YZ$

IR3: transitive rule – if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

IR4: projection rule – if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

IR5: additive rule – if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

IR6: pseudo-transitive rule – if $X \rightarrow Y$ and $YZ \rightarrow W$, then $XZ \rightarrow W$

The first three of these rules (IR1-IR3) are known as Armstrong's Axioms and constitute a necessary and sufficient set of inference rules for generating the closure of a set of functional dependencies.

Example Proof using Inference Rules

Given R = (A, B, C, D, E, F, G, H, I, J) and F = {AB → E, AG → J, BE → I, E → G, GI → H}

does F ⊨ AB → GH?

Proof:

1. AB → E, given in F
2. AB → B, projective rule IR4
3. AB → BE, additive rule IR5 from steps 1 and 2
4. BE → I, given in F
5. AB → I, transitive rule IR3 from steps 3 and 4
6. E → G, given in F
7. AB → G, transitive rule IR3 from steps 1 and 6
8. AB → GI, additive rule IR5 from steps 5 and 7
9. GI → H, given in F
10. AB → H, transitive rule IR3 from steps 8 and 9
11. AB → GH, additive rule IR5 from steps 7 and 10 – proven!

Practice Problem

Using the same set F,
Prove that F ⊨ BE → H

Closures

The notation: $F \models X \rightarrow Y$ denotes that the functional dependency $X \rightarrow Y$ is implied by the set of fds F.

Formally, $F^+ \equiv \{X \rightarrow Y \mid F \models X \rightarrow Y \}$

A set of inference rules is required to infer the set of fds in F^+.

For example, if I tell you that Kristi is older than Debi and that Debi is older than Traci, you are able to infer that Kristi is older than Traci. How did you make this inference? Without thinking about it or maybe knowing about it, you utilized a transitivity rule to allow you to make this inference.

Determining Closures

Another way of looking at the closure of a set of fds F is: F^+ is the smallest set containing F such that Armstrong's Axioms cannot be applied to the set to yield an fd not in the set.

F^+ is finite, but exponential in size in terms of the number of attributes of R.

> For example, given R = (A,B,C) and F = {AB $\rightarrow$ C, C $\rightarrow$ B}, F^+ will contain 29 fds (including trivial fds).

Thus, to determine if a fd X $\rightarrow$ Y holds on a relation schema R given F, what we really need to determine is does F $\models$ X $\rightarrow$ Y, or more correctly is X$\rightarrow$Y in F^+? However, we want to do this without generating all of F+ and checking to see if X $\rightarrow$ Y is in that set.

Algorithm Closure

The technique for this is to generate not F^+ but rather X^+, where X is any determinant from a fd in F. An algorithm for generating X^+ is shown below.

X^+ is called the closure of X under F (or with respect to F).

Algorithm Closure

```
Algorithm Closure  {returns X+ under F}
input:  set of attributes X, and a set of fds F
output: X+ under F
Closure (X, F)
{
      X+ ← X;
      repeat
            oldX+ ← X+;
            for every fd W→ Z in F do
                  if W ⊆ X+ then X+ ← X+ ∪ Z;
      until (oldX+ = X+);
}
```

Example Using Algorithm Closure

Given F = {A $\rightarrow$ D, AB $\rightarrow$ E, BI $\rightarrow$ E, CD $\rightarrow$ I, E $\rightarrow$ C}, Find (AE)+

pass 1

X^+ = {A, E}

using A $\rightarrow$ D, A $\subseteq$ X^+, so add D to X^+, X^+ = {A, E, D}

using AB $\rightarrow$ E, no change

using BI $\rightarrow$ E, no change

using CD $\rightarrow$ I, no change

using E $\rightarrow$ C, E $\subseteq$ X^+, so add C to X^+, X^+ = {A, E, D, C}

changes occurred to X^+ so another pass is required

pass 2

X^+ = {A, E, D, C}

using A $\rightarrow$ D, yes, but no change

using AB $\rightarrow$ E, no change

using BI $\rightarrow$ E, no change

using CD $\rightarrow$ I, CD $\subseteq$ X^+, so add I to X^+, X^+ = {A, E, D, C, I}

using E $\rightarrow$ C, yes, but no changes

changes occurred to X+ so another pass is required

Example Using Algorithm Closure

pass 3
X+ = {A, E, D, C, I}
using A $\rightarrow$ D, yes, but no changes
using AB $\rightarrow$ E, no
using BI $\rightarrow$ E, no
using CD $\rightarrow$ I, yes, but no changes
using E $\rightarrow$ C, yes, but no changes
no changes occurred to X+ so algorithm terminates

(AE)+ = {A, E, C, D, I}

This means that the following fds are in F+: AE $\rightarrow$ AECDI

Algorithm Member

Once the closure of a set of attributes X has been generated, it becomes a simple test to tell whether or not a certain functional dependency with a determinant of X is included in F^+.

Algorithm Member

```
Algorithm Member  {determines membership in F+}
input:  a set of fds F, and a single fd X → Y
output: true if F ⊨ X → Y, false otherwise
Member (F, X → Y)
{
    if Y ⊆ Closure(X,F)
        then return true;
        else return false;
}
```

Covers and Equivalence of FD Sets

A set of fds F is covered by a set of fds G (alternatively stated as G covers F) if every fd in F is also in G^+.

- That is to say, F is covered if every fd in F can be inferred from G.

Two sets of fds F and G are equivalent if $F^+ = G^+$.

- That is to say, every fd in G can be inferred from F, and every fd in F can be inferred from G.
- Thus $F \equiv G$ if F covers G, and G covers F.

To determine if G covers F, calculate X+ with respect to G for each $X \rightarrow Y$ in F. If $Y \subseteq X^+$ for each X, then G covers F.

Why Covers?

- Algorithm Member has a run time which is dependent on the size of the set of fds used as input to the algorithm. Thus, the smaller the set of fds used, the faster the execution of the algorithm.
- Fewer fds require less storage space and thus a corresponding lower overhead for maintenance whenever database updates occur.

There are many different types of covers ranging from non-redundant covers to optimal covers.

Essentially the idea is to ultimately produce a set of fds G which is equivalent to the original set F, yet has as few total fds as possible.

Non-redundant Covers

A set of fds is non-redundant if there is no proper subset G of F with $G \equiv F$. If such a G exists, F is redundant.

F is a non-redundant cover for G if F is a cover for G and F is non-redundant.

Algorithm Non-redundant

```
Algorithm Non-redundant  {produces a non-redundant cover}
input:  a set of fds G
output: a non-redundant cover for G
NonRedundant (G)
{
    F ← G;
    for each fd X → Y ∈ G do
        if Member(F – {X → Y}, X → Y)
         then F ← F – {X → Y};
     return (F);
}
```

Example Using Algorithm NonRedundant

Let G = {A → B, B → A, B → C, A → C}, find a non-redundant cover for G.

F ← G

Member({B → A, B → C, A → C}, A → B)

Closure(A, {B → A, B → C, A → C})

A^+ = {A, C}, therefore A → B is not redundant

Member({A → B, B → C, A → C}, B → A)

Closure(B, {A → B, B → C, A → C})

B^+ = {B, C}, therefore B → A is not redundant

Member({A → B, B → A, A → C}, B → C)

Closure(B, {A → B, B → A, A → C})

B^+ = {B, A, C}, therefore B → C is redundant F = F – {B → C}

Member({A → B, B → A}, A → C)

Closure(A, {A → B, B → A})

A+ = {A, B}, therefore A → C is not redundant

Return **F = {A → B, B → A, A → C}**

Example Using Algorithm NonRedundant

If G = {A → B, A → C, B → A, B → C}, the same set as before but given in a different order. A different cover will be produced!

F ← G

Member({A → C, B → A, B → C}, A → B)

Closure(A, {A → C, B → A, B → C})

A^+ = {A, C}, therefore A → B is not redundant

Member({A → B, B → A, B → C}, A → C)

Closure(A, {A → B, B → A, B → C})

A^+ = {A, B, C}, therefore A → C is redundant F = F – {A → C}

Member({A → B, B → C}, B → A)

Closure(B, {A → B, B → C})

B^+ = {B, C}, therefore B → A is not redundant

Member({A → B, B → A}, B → C)

Closure(B, {A → B, B → A})

B^+ = {B, A}, therefore B → C is not redundant

Return **F = {A → B, B → A, B → C}**

Non-redundant Covers

The previous example illustrates that a given set of functional dependencies can contain more than one non-redundant cover.

It is also possible that there can be non-redundant covers for a set of fds G that are not contained in G.

For example, if

$G = \{A \rightarrow B, B \rightarrow A, B \rightarrow C, A \rightarrow C\}$

then $F = \{A \rightarrow B, B \rightarrow A, AB \rightarrow C\}$ is a non-redundant cover for G

however, F contains fds that are not in G.

Extraneous Attributes

If F is a non-redundant set of fds, there are no “extra” fds in F and thus F cannot be made smaller by removing fds. If fds are removed from F then a new set G would be produced where $G \not\equiv F$.

However, it may still be possible to reduce the overall size of F by removing attributes from fds in F.

If F is a set of fds over relation schema R and $X \rightarrow Y \in F$, then the attribute A is extraneous in $X \rightarrow Y$ wrt F if:

1. $X = AZ$, $X \neq Z$ and $\{F - \{X \rightarrow Y\}\} \cup \{Z \rightarrow Y\} \equiv F$, or
2. $Y = AW$, $Y \neq W$ and $\{F - \{X \rightarrow Y\}\} \cup \{X \rightarrow W\} \equiv F$

In other words, an attribute A is extraneous in $X \rightarrow Y$ if A can be removed from either the determinant or consequent without changing F^+.

Extraneous Attributes

Example:

Let $F = \{A \rightarrow BC, B \rightarrow C, AB \rightarrow D\}$

attribute C is extraneous in the consequent of $A \rightarrow BC$ since

$A^+ = \{A, B, C, D\}$ when $F = F - \{A \rightarrow C\}$

similarly, B is extraneous in the determinant of $AB \rightarrow D$ since

$AB^+ = \{A, B, C, D\}$ when $F = F - \{AB \rightarrow D\}$

Left and Right Reduced Sets of FDs

Let F be a set of fds over schema R and let $X \rightarrow Y \in F$.

$X \rightarrow Y$ is left-reduced if X contains no extraneous attribute A.

- A left-reduced functional dependency is also called a full functional dependency.

$X \rightarrow Y$ is right-reduced if Y contains no extraneous attribute A.

$X \rightarrow Y$ is reduced if it is left-reduced, right-reduced, and Y is not empty.

Algorithm Left-Reduce

Algorithm Left-Reduce

```
Algorithm Left-Reduce  {returns left-reduced version of F}
input:  set of fds G
output: a left-reduced cover for G
Left-Reduce (G)
{
      F ← G;
      for each fd X→ Y in G do
            for each attribute A in X do
                  if Member(F, (X - A) → Y)
                        then remove A from X in X→ Y in F
      return(F);
}
```

Algorithm Right-Reduce

Algorithm Right-Reduce

```
Algorithm Right-Reduce  {returns right-reduced version of F}
input:  set of fds G
output: a right-reduced cover for G
Right-Reduce (G)
{
      F ← G;
      for each fd X→ Y in G do
            for each attribute A in Y do
                  if Member(F – {X→ Y} ∪ {X → (Y- A)}, X → A)
                        then remove A from Y in X→ Y in F
      return(F);
}
```

Algorithm Reduce

Algorithm Reduce

```
Algorithm Reduce  {returns reduced version of F}
input:  set of fds G
output: a reduced cover for G
Reduce (G)
{
      F ← Right-Reduce( Left-Reduce(G));
      remove all fds of the form X→ null from F
      return(F);
}
```

If G contained a redundant fd, X→ Y, every attribute in Y would be extraneous and thus reduce to X → null, so these need to be removed.

Algorithm Reduce

The order in which the reduction is done by algorithm Reduce is important. The set of fds must be left-reduced first and then right-reduced. The example below illustrates what may happen if this order is violated.

Example:

Let $G = \{B \rightarrow A , D \rightarrow A , BA \rightarrow D\}$

G is right-reduced but not left-reduced. If we left-reduce

G to produce $F = \{B \rightarrow A , D \rightarrow A , B \rightarrow D\}$

We have F is left-reduced but not right-reduced!

$B \rightarrow A$ is extraneous on right side since $B \rightarrow D \rightarrow A$

Minimum Cover

A set of functional dependencies F is minimal if

1. Every fd has a *single attribute for its consequent*.
2. F is *non-redundant*.
3. No fd $X \rightarrow A$ can be replaced with one of the form $Y \rightarrow A$ where $Y \rightarrow X$ and still be an equivalent set, i.e., F is *left-reduced*.

Example:

$G = \{A \rightarrow BCE,\ AB \rightarrow DE,\ BI \rightarrow J\}$

a minimum cover for G is:

$F = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, BI \rightarrow J\}$

Algorithm Mincover

Algorithm MinCover

```
Algorithm MinCover  {returns minimum cover for F}
input:  set of fds F
output: a minimum cover for F
MinCover (F)
{
      G ← F;
      replace each fd X → A1A2...An in G
                  by n fds X → A1, X → A2,..., X → An
      LeftReduce(G);
      NonRedundant(G);
       return(G);
}
```

Keys of a Relational Schema

If R is a relational schema with attributes $A_1, A_2, ..., A_n$ and a set of functional dependencies F where $X \subseteq \{A_1, A_2, ..., A_n\}$ then X is a key of R if:

- $X \rightarrow \{A_1, A_2, ... A_n\} \in F^+$, and
- no proper subset $Y \subseteq X$ gives $Y \rightarrow \{A_1, A_2, ...A_n\} \in F+$.

Basically, this definition means that you must attempt to generate the closure of all possible subsets of the schema of R and determine which sets produce all of the attributes in the schema.

Determining Keys - example

Find all keys for **R = (C, T, H, R, S, G)** with

F = {C → T, HR → C, HT → R, CS → G, HS → R}

Step 1: Generate $(A_i)^+$ for $1 \le i \le n$

$$\binom{6}{1} = \frac{6!}{1! \times (6-1)!} = \frac{720}{120} = 6$$

$C^+ = \{CT\}$, $T^+ = \{T\}$, $H^+ = \{H\}$

$R^+ = \{R\}$, $S^+ = \{S\}$, $G^+ = \{G\}$

no single attribute is a key for R

$$\binom{6}{2} = \frac{6!}{2! \times (6-2)!} = \frac{720}{48} = 15$$

Step 2: Generate $(A_iA_j)^+$ for $1 \le i \le n$, $1 \le j \le n$

$(CT)^+ = \{C,T\}$,	$(CH)^+ = \{CHTR\}$,	$(CR)^+ = \{CRT\}$
$(CS)^+ = \{CSGT\}$,	$(CG)^+ = \{CGT\}$,	$(TH)^+ = \{THRC\}$
$(TR)^+ = \{TR\}$,	$(TS)^+ = \{TS\}$,	$(TG)^+ = \{TG\}$
$(HR)^+ = \{HRCT\}$,	$(HS)^+ = \{HSRCTG\}$,	$(HG)^+ = \{HG\}$
$(RS)^+ = \{RS\}$,	$(RG)^+ = \{RG\}$,	$(SG)^+ = \{SG\}$

The attribute set (**HS**) is a key for R

Determining Keys - example

Step 3: Generate $(A_iA_jA_k)^+$ for $1 \le i \le n$, $1 \le j \le n$, $1 \le k \le n$

$(CTH)^+ = \{CTHR\}$,	$(CTR)^+ = \{CTR\}$
$(CTS)^+ = \{CTSG\}$,	$(CTG)^+ = \{CTG\}$
$(CHR)^+ = \{CHRT\}$,	$(CHS)^+ = \{CHSTRG\}$
$(CHG)^+ = \{CHGTR\}$,	$(CRS)^+ = \{CRSTG\}$
$(CRG)^+ = \{CRGT\}$,	$(CSG)^+ = \{CSGT\}$
$(THR)^+ = \{THRC\}$,	$(THS)^+ = \{THSRCG\}$
$(THG)^+ = \{THGRC\}$,	$(TRS)^+ = \{TRS\}$
$(TRG)^+ = \{TRG\}$,	$(TSG)^+ = \{TSG\}$
$(HRS)^+ = \{HRSCTG\}$,	$(HRG)^+ = \{HRGCT\}$
$(HSG)^+ = \{HSGRCT\}$,	$(RSG)^+ = \{RSG\}$

$$\binom{6}{3} = \frac{6!}{3! \times (6-3)!} = \frac{720}{36} = 20$$

Super keys are shown in red.

Determining Keys - example

Step 4: Generate $(A_iA_jA_kA_r)^+$ for $1 \le i \le n$, $1 \le j \le n$, $1 \le k \le n$, $1 \le r \le n$

$$\binom{6}{4} = \frac{6!}{4! \times (6-4)!} = \frac{720}{48} = 15$$

$(CTHR)^+ = \{CTHR\}$,
$(CTHG)^+ = \{CTHGR\}$,
$(CHRG)^+ = \{CHRGT\}$,
$(THRS)^+ = \{THRSCG\}$,
$(TRSG)^+ = \{TRSG\}$,
$(CTRS)^+ = \{CTRS\}$,
$(CSHG)^+ = \{CSHGTR\}$,
$(CTRG)^+ = \{CTRG\}$

$(CTHS)^+ = \{CTHSRG\}$
$(CHRS)^+ = \{CHRSTG\}$
$(CRSG)^+ = \{CRSGT\}$
$(THRG)^+ = \{THRGC\}$
$(HRSG)^+ = \{HRSGCT\}$
$(CTSG)^+ = \{CTSG\}$
$(THSG)^+ = \{THSGRC\}$

Super keys are shown in red.

Determining Keys - example

Step 5: Generate $(A_iA_jA_kA_rA_s)^+$ for $1 \le i \le n$, $1 \le j \le n$, $1 \le k \le n$, $1 \le r \le n$, $1 \le s \le n$

$$\binom{6}{5} = \frac{6!}{5! \times (6-5)!} = \frac{720}{120} = 6$$

$(CTHRS)^+ = \{CTHSRG\}$
$(CTHRG)^+ = \{CTHGR\}$
$(CTHSG)^+ = \{CTHSGR\}$
$(CHRSG)^+ = \{CHRSGT\}$
$(CTRSG)^+ = \{CTRSG\}$
$(THRSG)^+ = \{THRSGC\}$

Super keys are shown in red

Determining Keys - example

Step 6: Generate (AiAjAkArAsAt)+ for $1 \le i \le n$, $1 \le j \le n$, $1 \le k \le n$, $1 \le r \le n$, $1 \le s \le n$, $1 \le t \le n$

$$\binom{6}{6} = \frac{6!}{6! \times (6-6)!} = \frac{720}{720} = 1$$

(CTHRSG)+ = {CTHSRG}

Super keys are shown in red.

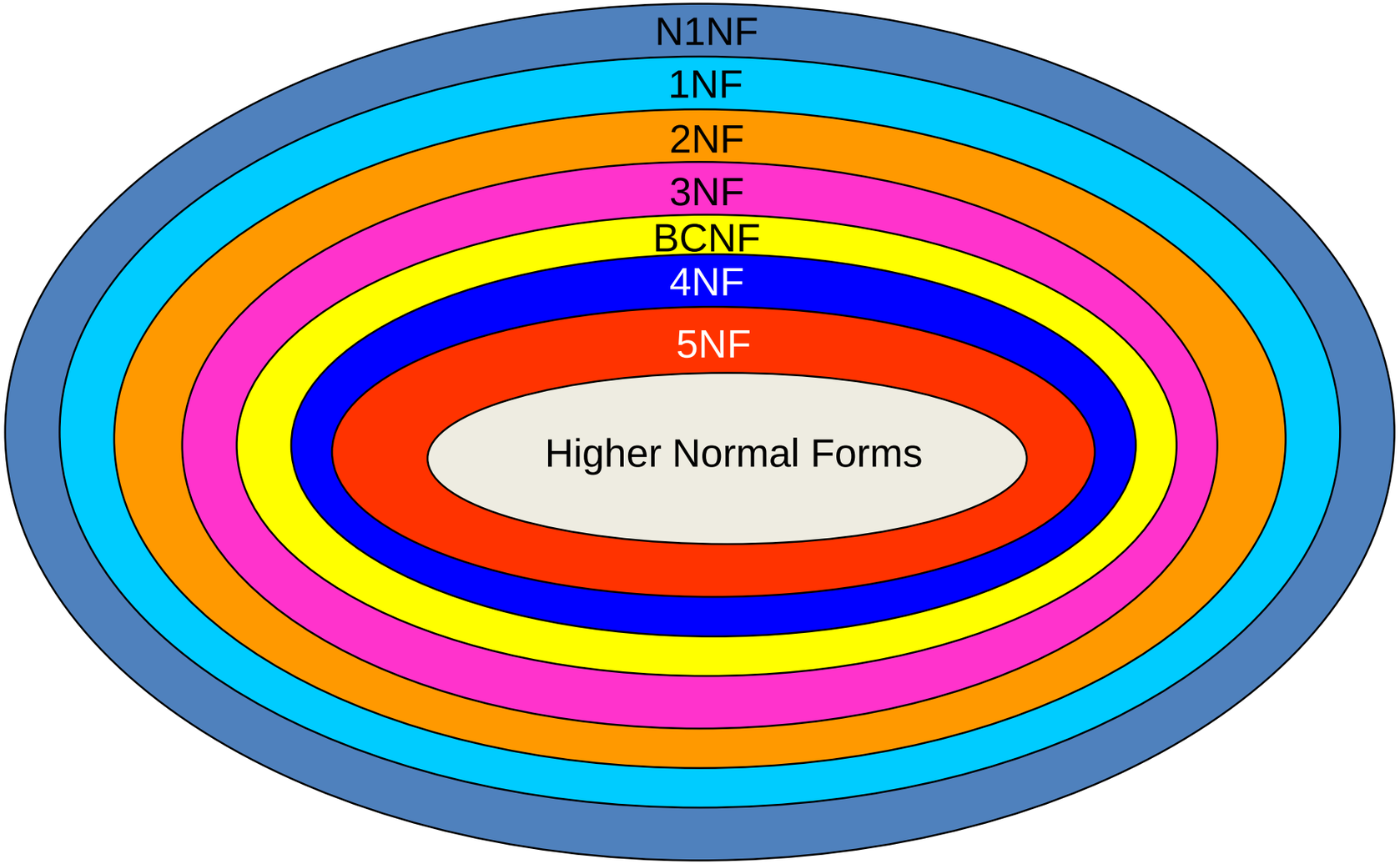
In general, for 6 attributes we have:

$$\binom{6}{1} + \binom{6}{2} + \binom{6}{3} + \binom{6}{4} + \binom{6}{5} + \binom{6}{6} = 6 + 15 + 20 + 15 + 1 = 63 \text{ cases}$$

Normalization based on Keys

- Normalization is a formal technique for analyzing relations based on the candidate keys and functional dependencies.
- The technique involves a series of rules that can be used to test individual relations so that a database can be normalized to any degree.
- When a requirement is not met, the relation violating the requirement is decomposed into a set of relations that individually meet the requirements of normalization.
- Normalization is often executed as a series of steps. Each step corresponds to a specific normal form that has known properties.

Different Normal Forms



Normalization

- In relational model, only the first normal form (1NF) is critical in creating relations. All the subsequent normal forms are optional.
- To avoid the update anomalies, it is normally recommended that the database designer proceed to at least 3NF.
- In fact, some 1NF relations are also in 2NF and some 2NF relations are also in 3NF, and so on.
- As we proceed, we'll look at the requirements for each normal form and a decomposition technique to achieve relation schemas in each of normal form.

First Normal Form (1NF)

- A relation in which every attribute value is atomic is in 1NF.
- We have only considered 1NF relations for the most part in this course.
- When dealing with multi-valued attributes at the conceptual level, recall that in the conversion into the relational model created a separate table for the multi-valued attribute.

Some Additional Terminologies

- A key is a superkey such that the removal of any attribute from the key will cause it to no longer be a superkey. In other words, the key is minimal in the number of attributes.
- The candidate key for a relation is one of minimal superkeys of the relation schema.
- The primary key for a relation is a selected candidate key. All of the remaining candidate keys (if any) become secondary keys.
- A prime attribute is any attribute of the schema of a relation R that is a member of any candidate key of R.
- A non-prime attribute is any attribute of R which is not a member of any candidate key.

Full Functional Dependency

- Second normal form (2NF) is based on the concept of a full functional dependency.
- A functional dependency $X \rightarrow Y$ is a full functional dependency if the removal of any attribute A from X causes the fd to no longer hold.
  - for any attribute $A \in X$, $X - \{A\} \not\rightarrow Y$
- A functional dependency $X \rightarrow Y$ is a partial functional dependency if some attribute A can be removed from X and the fd still holds.
  - for some attribute $A \in X$, $X - \{A\} \rightarrow Y$

Second Normal Form (2NF)

- A relation scheme R is in 2NF with respect to a set of functional dependencies F if every non-prime attribute is fully dependent on every key of R.
- Another way of stating this is: there does not exist a non-prime attribute which is partially dependent on any key of R. In other words, no non-prime attribute is dependent on only a portion of the key of R.

2NF Identification

A relation scheme R is in 2NF if it satisfies one of those conditions:

- It has single-attribute keys (simple keys)
- It doesn’t have any non-prime attribute
- All non-prime attribute are fully functional dependent on the keys.

2NF Normalization

- Remove each non-prime attribute that is functional dependent on a proper subset of a candidate key and form a new relation with a new key is the proper subset.
- All remaining attributes form a new relation with the key is the original one.

2NF Normalization - Example

Given R = (A, D, P, G),

F = {AD $\rightarrow$ PG, A $\rightarrow$ G}

Then R is not in 2NF because G is partially dependent on the key AD since AD $\rightarrow$ G yet A $\rightarrow$ G.

Decompose R into:

R1 = (A, D, P)	R2 = (A, G)
K1 = {AD}	K2 = {A}
F1 = {AD $\rightarrow$ P}	F2 = {A $\rightarrow$ G}

Transitive Dependency

Third Normal Form (3NF) is based on the concept of a transitive dependency.

- Given a relation scheme R with a set of functional dependencies F and subset $X \subseteq R$ and an attribute $A \in R$. A is said to be transitively dependent on X if there exists $Y \subseteq R$ with $X \rightarrow Y$, $Y \not\rightarrow X$ and $Y \rightarrow A$ and $A \notin X \cup Y$.
- An alternative definition for a transitive dependency is: a functional dependency $X \rightarrow Y$ in a relation scheme R is a transitive dependency if there is a set of attributes $Z \subseteq R$ where Z is NOT a subset of any key of R and yet both $X \rightarrow Z$ and $Z \rightarrow Y$ hold in F.

Third Normal Form (3NF)

A relation scheme R is in 3NF with respect to a set of functional dependencies F, if whenever $X \rightarrow A$ holds, either:

(1) X is a superkey of R or

(2) A is a prime attribute.

Alternative definition: A relation scheme R is in 3NF with respect to a set of functional dependencies F if it satisfies:

- It is in 2NF
- No non-prime attribute is transitively dependent on any key of R.

3NF Normalization

Method 1:

Input a relation R and a functional dependencies set F hold for R.

- Find a minimal cover, G, of the set of functional dependencies F.
- For each functional dependency X→A in G, form (X, A) as the schema of one of the relations in the decomposition.
- If non of the resulting relation schemas from step 2 contains a candidate key for R, add another relation whose schema is a candidate key of R

3NF Normalization

Method 2:

Input a relation R and a functional dependencies set F hold for R.

- Remove each non-prime attribute that is transitive dependent on a key of R and form a new relation with the key is the transitive attributes.
- Remaining attributes form a new relation with the key is the original one.

3NF - Example

Let R = (A, B, C, D)

and F = {AB $\rightarrow$ CD, C $\rightarrow$ D, D $\rightarrow$ C}

then R is not in 3NF since C $\rightarrow$ D holds and C is not a superkey of R.

Alternatively, R is not in 3NF since AB $\rightarrow$ C and C $\rightarrow$ D and thus D is a non-prime attribute which is transitively dependent on the key AB.

3NF – Exercise 1

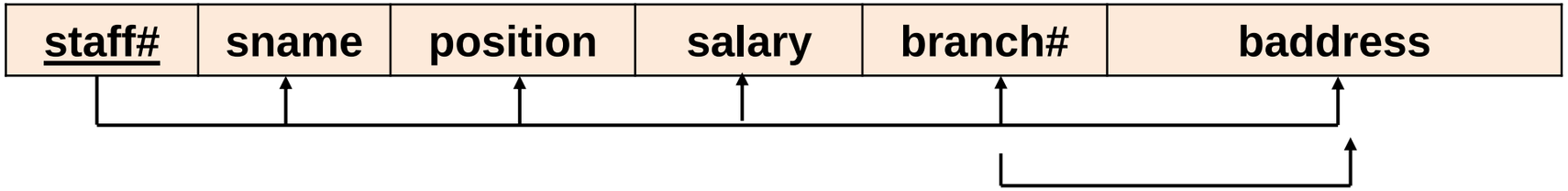
Let R = (A, B, C, D, E)

and F = {AB $\rightarrow$ C, C $\rightarrow$ B, A $\rightarrow$ D}

Convert R into 3NF.

3NF – Exercise 2

StaffBranch



Set of functional dependencies:

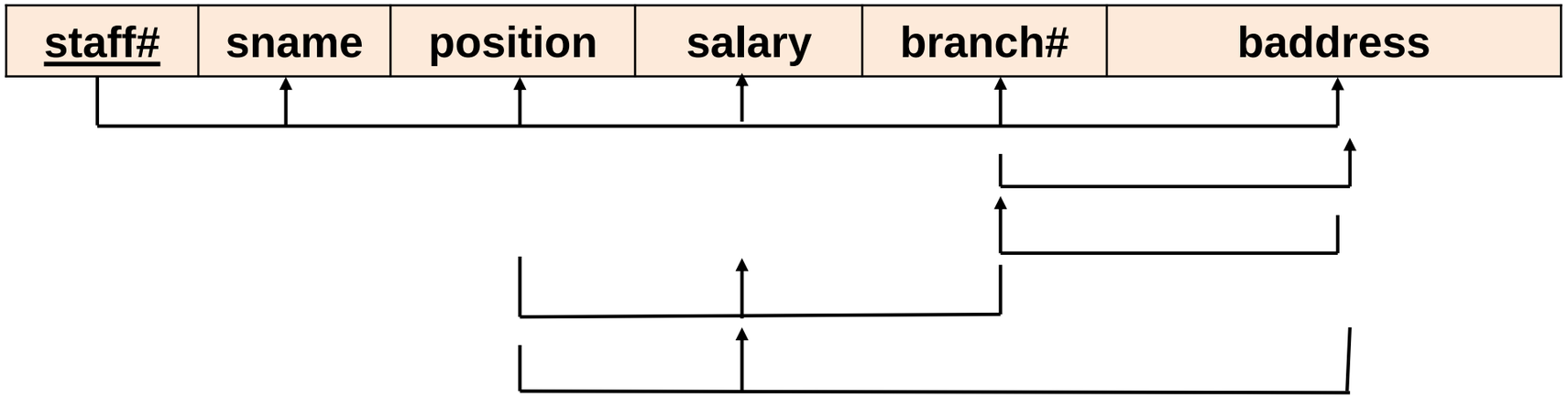
staff# $\rightarrow$ sname, position, salary, branch#, baddress

branch# $\rightarrow$ baddress

Convert StaffBranch into 3NF

3NF – Exercise 3

StaffBranch



Set of functional dependencies:

staff# → sname, position, salary, branch#, baddress

branch# → baddress

baddress → branch#

branch#, position → salary

baddress, position → salary

Convert StaffBranch into 3NF

Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form (BCNF) is a more stringent form of 3NF.

A relation scheme R is in Boyce-Codd Normal Form with respect to a set of functional dependencies F if whenever $X \rightarrow A$ holds and $A \notin X$, then X is a superkey of R.

Example: Let R = (A, B, C)

$F = \{AB \rightarrow C,\ C \rightarrow A\}$

K = {AB}

R is not in BCNF since $C \rightarrow A$ holds and C is not a superkey of R.

BCNF versus 3NF

- Notice that the only difference in the definitions of 3NF and BCNF is that BCNF drops the allowance for A in $X \rightarrow A$ to be prime.
- An interesting side note to BCNF is that Boyce and Codd originally intended this normal form to be a simpler form of 3NF. In other words, it was supposed to be between 2NF and 3NF. However, it was quickly proven to be a more strict definition of 3NF and thus it wound up being between 3NF and 4NF.
- In practice, most relational schemes that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in the schema where X is not a superkey and A is prime, will the schema be in 3NF but not in BCNF.