February 7, 2019

# Database Normalization

From Lab 1:

**Why is this table bad design?**

* What happens when a person changes address?
  + If we want to change someone’s address, we would need to change their address as many times as their name appears in the relation
* Etc.

SO, **never repeat a fact in two places in a database**

Not following the above rule and not following the rule of storing complex info in a column will result in significant errors in our application down the road and critical failures in our database and application code

The end result is creating a good methodology that can be used to create good design for any database

* Drive redundancy out of our data by identifying dependencies between attributes in a relation

For example: are **city** and **state** dependent on one another?

* Yes: if we have city, we don’t necessarily know state

**Normal Forms**

* ‘Levels’ of proper design

1st Normal Form: A relation contains only **elementary** attributes

Boyce-Codd Normal Form (BCNF): Each functional dependency between attributes represent a key in the relation

* What is a functional dependency?
  + If any functional dependency does not form a key, we have a problem
* What is a key? (Duh, set of columns that uniquely identify a row)
* Why does it matter?
* How can we fix it?

**Functional Dependency**

* If two instance of a relation agree on a set of attributes, then they must also agree on some other set of attributes
* Long winded way of saying dependencies defined above

*For example:*

(title, year) => (length)

***There cannot be two lengths for the same movie***

If you find a functional dependency that does **not** include all columns, that relation is not in BCNF and we need to fix it

Abstraction of Functional Dependencies:

**Relation**: R(A, B, C, D, E)

**Dependencies:** B => C, D

D => A

A= > B, E

***Transitive dependencies allow this to be BCNF and so each dependency is a key***

This is called **closure** of a functional dependency

Functional Dependency Decomposition:

* It is easier to decompose the FD first before computing the closure
  + Helps manage complexity
* We **only** decompose on the right-hand side of the dependency

**How do we resolve a functional dependency?**

* We are going to create a new relation
  + Need to make sure that we can still tie these two data sets together

1. Create a table containing all columns within the functional dependency
2. Create a new table containing all items from the left side of the functional dependency and any columns not in the functional dependency

Even after all of this, **it is possible to have a loss of functional dependency**

* After a decomposition, the relevant columns still need to appear together somewhere
* There are some situations where we follow the rules of BCNF, convert to BCNF and something doesn’t get preserved

For example:

R(jkl)

JK=>L

L=>K

L=>K is a violation: if we decompose to LK and LJ the FD JK=>L is gone

**We didn’t do anything wrong, BCNF just didn’t apply here and that’s okay.**

So,

**Resolving a lost FD**:

3rd Normal Form

* Option 1: enforce the constraint we lost in code, but this can result in a mess
* Option 2: relax the BCNF constraint
  + For every FD, the left side is a key (BCNF) *or*
  + Each attribute on the right is a **member of any other key**

4th Normal Form:

**Multivalued Dependencies**

* If you know what you have in one column, there is a set of options for other columns
* Essentially, there are no functional dependencies available
* **Will not have multiple multivalued dependencies in the same table**

Steps to make a good design

1. Locate/Define FDs
2. Decompose into BCNF
3. Check for FD loss
   1. If lost, try 3rd Normal Form starting from Step 1
4. Check for multivalued dependencies
   1. Decompose to 4th Normal form if needed