CTEC323 Lecture 8

Dan Zingaro
OISE/UT

October 30, 2008

What is Normalization?

- Remember all of the data redundancy problems with using files from Chapter 1 (update, insert, and delete anomalies)?
- Without good table structure, databases can have these problems, too!
- Normalization: process for evaluating and correcting table structures to minimize data redundancies, thereby reducing the likelihood of data anomalies
- The higher the normal form, the more join operations required to produce a specified output
- Sometimes, denormalizing a database is required to meet performance requirements (but, of course, this increases data redundancy)

Running Example

We begin with a table with the following attributes.

proj_num	proj_name	emp_num	emp_name
job_class	chg_hour	hours	

- Assume each project is completed by many employees, and each employee can work on multiple projects
- Each employee has exactly one job_class and each job_class has one chg_hour
- hours represents the hours worked on a particular project by a particular employee

Problems with this Table

- This structure invites data inconsistencies
  - What if we enter Elect. Engineer for the job_class of one employee and Elect. Eng for others?
- It also promotes anomalies, such as
  - Updating an employee's job_class may require changes in many rows
  - Updating a job's chg_hour may require changes in many rows
  - If we insert a new employee who has no project yet, we have to fill in the project information with phantom data
  - If we delete the sole employee of a project, we lose the project information too

Objectives of Normalization

- Produce tables that represent single subjects (courses, students, etc.),
- Avoid storing data unnecessarily in more than one table
- Ensure that attributes in a table are dependent on
  - the primary key (1NF)
  - the entire primary key (2NF)
  - nothing but the primary key (3NF)

1NF

- All relational tables are already in 1NF (first normal form)
- All we have to do is identify a PK
- The combination of proj_num and emp_num gives us a PK for our running example

Identifying Dependencies

1. proj_num, emp_num $\rightarrow$ proj_name, emp_name, job_class, chg_hour, hours
2. proj_num $\rightarrow$ proj_name
3. emp_num $\rightarrow$ emp_name, job_class, chg_hour
4. job_class $\rightarrow$ chg_hour

- Partial dependency: dependency based on only a part of a PK (dependencies 2 and 3 above)
- Transitive dependency: dependency of one non-PK attribute on another non-PK attribute (dependency 4 above)
- If we start with a 1NF table and remove the partial dependencies, we have a 2NF table

2NF

- To convert from 1NF to 2NF, we take each component of the PK that functionally determines other attributes, copy that part of the PK to a new table, and move the attributes it determines from the original table to that new table
- Each component of the original key becomes the PK of its new table
- For our example, this results in three tables (PK attributes in bold)
- project (**proj_num**, proj_name)
- employee (**emp_num**, emp_name, job_class, chg_hour)
- assignment (**proj_num**, **emp_num**, hours)
- Note: this format of writing tables with PK's in bold is called a relational schema

3NF

- A table in 2NF may still have transitive dependencies; we eliminate these when we convert a 2NF table into 3NF
- To convert a table from 2NF to 3NF, begin by identifying each determinant of a transitive dependency and the attributes it determines
- Then, we take each determinant, copy it to a new table, and move the attributes it determines from the original table to that new table
- Each determinant becomes the PK of its new table

3NF...

- The project and assignment tables have no determinants, so they are already in 3NF
- Employee has a determinant: job_class $\longrightarrow$ chg_hour
- Converting to 3NF, we split employee into two tables
- employee (**emp_num**, emp_name, job_class)
- job (**job_class**, chg_hour)
- Our final database with all tables in 3NF thus contains four tables

3NF...

- assignment (**proj_num**, **emp_num**, hours)
- project (**proj_num**, proj_name)
- employee (**emp_num**, emp_name, job_class)
- job (**job_class**, chg_hour)

Numeric Keys

- Using job_class as the PK in the employee table forces us to enter the text for the job class (such as Database Designer) for each new employee
- If we make a spelling mistake on the job_class of a new employee, it will not match a job_class in job (referential integrity problem)
- An alternative is to add a job_code attribute which becomes the new PK of the job table (and a new FK in the employee table)
- employee (**emp_num**, emp_name, job_code)
- job (**job_code**, job_class, chg_hour)
- This introduces a transitive dependency (which we could eliminate)
- job_code is called a surrogate key because it is an artificial key introduced to simplify the assignment of PK's to tables

Historical Accuracy

- We can add the job charge per hour to the assignment table to capture the amount charged per hour at the time the project was completed
- If we don't store this value and the assignment table and the job charge later changes, we will incorrectly calculate the total charge of a project (we will not use the historically accurate value for the hourly rates)
- assignment (**proj_num**, **emp_num**, hours, assign_chg_hour)

Denormalization

- As tables are decomposed in the normalization process, the number of tables expands
- Retrieving meaningful information often requires multiple joins and therefore increased system resources
- We occasionally denormalize to increase processing speed
- Unnormalized tables have the following drawbacks
  - Greater possibility for data anomalies
  - Updates are less efficient because larger tables have to be processed