This Lecture

Discuss “goodness” of a database design
- Informal guidelines
- Objective measures

Informal Guidelines

1. Clear semantics
   - Do your relations make sense as independent units?
   - Do you have a clear separation of concerns?
   - Did you do ER modeling beforehand?
2. Reducing redundancy
   - Data should be stored once and only once (excepting foreign keys)
   - Redundancy leads to modification anomalies
3. Reducing NULLs
4.Disallowing spurious tuple generation

Redundancy

- Example has multiple issues of redundancy:
  - Multiple sections, with redundant course id and title information
  - Instructor name and email repeated many times
- Cause:
  - Two (or more) concepts have been combined into one table
  - Instructor
  - Course info
  - Section info
  - These should be (somewhat) independent pieces of data

Modification Anomalies

- A consequence of bad design
- Goes hand-in-hand with redundancy issues
- Three types:
  - Insertion
  - Update
  - Deletion
Insertion Anomaly

Insert a new faculty member in example table – no course info yet

- What do we put in for course info?
  - NULL values?
    - Could violate constraints
  - What happens when we want to add a course for this faculty member?
  - Dummy data?

Deletion Anomaly

Inverse of insertion anomaly:
What happens if we delete the last course taught by an instructor?
Similarly, what happens to a faculty member’s courses when they leave/retire?

Update Anomaly

- When updating redundant data, must remember to update all instances
- E.g., suppose you are in an application updating course info for CSCI 403
  - You notice that CPW’s office info is wrong (e.g., maybe he moved)
  - You edit the record to correct his office info
  - Now, inconsistent data in different records! Which is correct?

Spurious Tuple Generation

- Happens when data has been incorrectly factored
  - There is no linking data (foreign keys)
  - The linking data is incomplete
  - (Somewhat contrived) example:
    - Table mines_courses (instructor, course_id, section)
    - Table mines_faculty (instructor, course_id, office, email)
    - Joining these tables on instructor and course_id will yield spurious combinations of instructors with sections they do not teach

Functional Dependencies

- Our primary tool for eliminating redundancy and modification anomalies
- A kind of constraint between two sets of attributes in a relation schema
- Definition:
  Given a relation schema R and sets of attributes X and Y, then we say a functional dependency X → Y exists if, whenever tuples t1 and t2 are two tuples from any relation r(R) such that t1[X] = t2[X], it is also true that t1[Y] = t2[Y].
  - The lingo: We say X functionally determines Y, or Y is functionally dependent on X.

Functional Dependencies 2

- In other words:
  - If it is always true that whenever two tuples agree on attributes X, they also agree on Y, then X → Y
- Example:
  - If we assert that an instructor is always associated with one office and email, then
    \[
    \{ \text{instructor} \} \rightarrow \{ \text{office, email} \}
    \]
  - is a functional dependency (FD) on the example table in figure 1.
Functional Dependencies

Note:
FD’s are properties of the world that we impose on the data, not properties of the data.
That is, finding FD’s is a design activity.
The result is a constraint on the data that is allowed in our database.

Example:
It may be that we have a particular set of courses data in which each course_id is associated with one instructor. Then, for that data, it is true that whenever a tuple agrees on course_id, it also agrees on instructor. However, unless this is required to be true for any set of data we can put in our database, we cannot say { course_id } → { instructor }.

Types of Functional Dependency

- Trivial FD’s
  - Trivially, X → X
  - More generally, if Y ⊆ X, then X → Y

- Non-trivial FD’s
  - X → Y
  - Y ⊈ X

- Completely non-trivial FDs
  - X → Y
  - X ∩ Y = ∅ (No overlap between X and Y)

Non-Trivial FDs

- We are primarily interested in non-trivial and completely non-trivial FD’s.
- In our figure 1 example, we might identify the following completely non-trivial FD’s:
  - instructor → office
  - instructor → email
  - { course_id, section } → instructor
  - course_id → title
  - Can you identify others?

Functional Dependencies and Superkeys

- FD’s can be viewed as a generalization of the notion of a superkey
- A superkey is a set of attributes which will contain a unique subset of values for any tuple in a relation.
- Thus, if X is a superkey of R, X → R.
- Alternately, if X → Y and X ∩ Y = R, then X is a superkey of R.

Inference Rules

- Allow us to infer additional FD’s from an existing set of FD’s
  - Splitting rule:
    - If A → {B₁, B₂} then A → B₁ and A → B₂
  - Combining rule:
    - If A → (B₃ and A → C then A → {B₃, C}
  - Transitive rule:
    - If A → B and B → C then A → C

Additional rules can be derived and can be found in your textbook.

Closures

Definition:
Given some set of functional dependencies F on a relation schema R, and some subset of attributes A, then the set { Bᵢ : A → Bᵢ } is called the closure of A and is denoted A⁺.

Closures are useful in:
- Normalization
- Finding all superkeys of a relation schema
Computing Closure

Algorithm:
Given set $F$ of functional dependencies, and some set of attributes $A$, compute $A^+$:

Start with $S = A$. Trivially, $A \rightarrow S$.

Repeat until no change:
  if there exists an FD $X \rightarrow Y$ in $F$ such that $X \subseteq S$,
  then let $S = S \cup Y$

$A^+ = S$

Expands $S$ while maintaining the invariant $A \rightarrow S$. The step follows from the three inference rules.

Finding All Superkeys

- In short:
  - Generate the power set of $R$ – all subsets of attributes
  - For each subset, compute the closure
  - If the closure $= R$, then the subset is a superkey of $R$

- This algorithm is mostly of academic interest to us, but could be used in automated software to build a normalized database, when the functional dependencies are inputted.

Next Time

- Normal forms & Boyce-Code normal form
- Decomposition algorithm