CS 2451
Database Schema Design

http://www.seas.gwu.edu/~bhagiweb/cs2541

Instructor: Dr. Bhagi Narahari

Building Database Applications: Steps

1. Start with a conceptual model
   - "On paper" using certain techniques
     E-R Model
   - ignore low-level details – focus on logical representation
   - "step-wise refinement" of design with client input
2. Design & implement schema
   - Design and codify (in SQL) the relations/tables
   - Refine the schema – *normalization*
   - Do physical layout – indexes, etc.
3. Import the data
4. Write applications using DBMS and other tools
   Many of the hard problems are taken care of by other people (DBMS, API writers, library authors, web server, etc.)

DBMS takes care of Query Optimization, Efficiency, etc.

Next:

Database Design Process- How to design a good schema ?

Relational Model: Definitions Review

- Relations/tables, Attributes/Columns, Tuples/rows
  - Attribute domains
- Superkey
- Key
  - No two tuples can have the same value in the key attribute
  - Primary key, candidate keys
  - No primary key value can be null
- Referential integrity constraints
  - Foreign key

Relational Schema Design

- Logical Level
  - Whether schema has intuitive appeal for users
- Manipulation level
  - Whether it makes sense from an *efficiency* or *correctness* point of view

Informal Design Guidelines for Relational Databases

- What is relational database design?
  - The grouping of attributes to form "good" relation schemas
- Two levels of relation schemas
  - The "user view" level
  - The storage "base relation" level
- Design is concerned mainly with base relations
- What are the criteria for "good" base relations?

Schema Design Decisions

- Guidelines for database schema design: how to design a "good" schema ?
- Example of a COMPANY database: two possible designs to represent Employees and Department information

S1: EMPLOYEE(LNAME,FNAME,SSN,DNO)
DEPT(DNUM, DNAME, MGRSSN)
S2:
EMPDEPT(LNAME,FNAME,SSN,DNUM,DNAME,MGRSSN)

Question: Which one is better ?? S1 or S2 ?

Schema Design & Functional Dependencies/Normal Forms

- Informal methods
  - Rules of thumb, intuitive reasoning, experience
- Formal methods
  - Provable properties
  - Involve concept of Functional Dependencies
  - Develop theoretical model to define what we mean by "good schema"
  - Normal forms are defined in terms of properties of the functional dependencies and link to 'good' or 'bad' schema design

    If a relation is in Third Normal Form it is a "good" design..etc.

Design Guidelines for Relational Databases

- We first discuss informal guidelines for good relational design
- Then we discuss formal concepts of functional dependencies and define normal forms
  - - 1NF (First Normal Form)
  - - 2NF (Second Normal Form)
  - - 3NF (Third Normal Form)
  - - BCNF (Boyce-Codd Normal Form)

Informal Guidelines: 1

- 1: Try to make user interpretation easy

S1: EMP(FNAME, LNAME, SSN)
WORKS_ON (SSN, PNO)
PROJECT_LOC(PNO, PLOC)

S2: EMP(FNAME,LNAME,SSN, PNO,PLOC)

- Perhaps S2 has too much information to absorb per tuple ?

Semantics of the Relational Attributes must be clear

- GUIDELINE 1 : Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).
  - Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation
  - Only foreign keys should be used to refer to other entities
  - Entity and relationship attributes should be kept apart as much as possible.
- Bottom Line: *Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.*

Informal Guidelines: 2

- Try to reduce redundancy
  - In S2, suppose only few projects
    PLOC is unnecessarily repeated too often
  - On the other hand, S1 repeats SSN in WORKS_ON
    But SSN is a smaller attribute than PLOC (which may be a large string)

S1: EMP(FNAME, LNAME, SSN)
WORKS_ON (SSN, PNO)
PROJECT_LOC(PNO, PLOC)

S2: EMP(FNAME,LNAME,SSN, PNO,PLOC)

Redundant Information in Tuples and Update Anomalies

- Information is stored redundantly
  - Wastes storage
    Size of file relates to time to retrieve data
  - Causes problems with update anomalies
    Insertion anomalies
    Deletion anomalies
    Modification anomalies

Informal Guidelines: 3

- Try to avoid update anomalies
  - Avoid having to search through entire table during update operation
    Insert, delete, update/modify
    Searching using a non-key attribute may require searching the entire table
  - Avoid losing information
- This is an important criteria
  - efficiency

EXAMPLE OF AN UPDATE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Update Anomaly:
  - Changing the name of project number P1 from “Billing” to “Customer-Accounting” may cause this update to be made for all 100 employees working on project P1.

EXAMPLE OF AN INSERT ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Insert Anomaly:
  - Cannot insert a project unless an employee is assigned to it.
- Conversely
  - Cannot insert an employee unless an he/she is assigned to a project.

EXAMPLE OF A DELETE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Delete Anomaly:
  - When a project is deleted, it will result in deleting all the employees who work on that project.
  - Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

Guideline for Redundant Information in Tuples and Update Anomalies

- GUIDELINE 2:
  - Design a schema that does not suffer from the insertion, deletion and update anomalies.
  - If there are any anomalies present, then note them so that applications can be made to take them into account.

Informal Guidelines: 4

- Avoid too many NULL values
  - Space is wasted…why is this a problem ?
  - Problems occur when using aggregate functions like count or sum
  - NULLs can have different intentions

    Attribute does not apply

    Value unknown and will remain unknown

    Value unknown at present

What are Lossless Joins & Spurious Tuples ?

- Split a table into smaller tables (with fewer columns in each) – sometimes a better design in our examples
  - How to split ?
- Must be able to reconstruct the 'original' table
- When reconstructing the "original" data, should not introduce spurious tuples
  - Also called non-additive joins

Informal Guidelines 5: Generation of Spurious Tuples – avoid at any cost

- Bad designs for a relational database may result in erroneous results for certain JOIN operations
- The "lossless join" property is used to guarantee meaningful results for join operations

- GUIDELINE :
  - The relations should be designed to satisfy the lossless join condition.
  - No spurious tuples should be generated by doing a natural-join of any relations.

Decomposition of a Table into smaller tables

- There are two important properties of decompositions:
  a) Non-additive or losslessness of the corresponding join
  b) Preservation of the functional dependencies.

- Note that:
  - Property (a) is extremely important and ***cannot*** be sacrificed.
  - Property (b) is less stringent and may be sacrificed. (Or can enforce using Triggers and constraints.)

Informal Guidelines: 6

- Do not lose the dependencies
  - Will define this after we define functional dependencies….

- Informally:
  - Do not lose constraints and business rules when we decompose tables into smaller tables

Example: Decomposing a table into smaller tables

Original Table S has 3 columns

Decompose into:
S1: CAR (ID, Make, Color)

S2: CAR1 (ID, Color)
CAR2 (Color, Make)

123	Toyota	Blue
456	Audi	Blue
789	Toyota	Red

Question: Is there a problem with this decomposition ?

S2-Car 1

123	Blue
456	Blue
789	Red

S2-Car 2

Blue	Toyota
Blue	Audi
Red	Toyota

Spurious tuples…We do not get back the original table

123	Blue	Toyota
123	Blue	Audi
456	Blue	Toyota
456	Blue	Audi
789	Red	Toyota

Summary of Problems

- Insertion, Deletion, modification anomalies
- Too many NULLs
- Spurious tuples – called non-additive join
- We need a theory of schema design
  - Functional dependencies and normalization
- Using functional dependencies define "normal forms" of schema
  - A schema in a "Third Normal Form" will avoid certain anomalies

Normalization

- ***Normalization*** is a technique for producing relations with desirable properties.
- Normalization decomposes relations into smaller relations that contain less redundancy. This decomposition requires that no information is lost and reconstruction of the original relations from the smaller relations must be possible.
- Normalization is a bottom-up design technique for producing relations. It pre-dates ER modeling and was developed by Codd in 1972 and extended by others over the years.
  - Normalization can be used after ER modeling or independently.
  - Normalization may be especially useful for databases that have already been designed without using formal techniques.

Normalization Motivation

- The goal of normalization is to produce a set of relational schemas R_1, R_2, …, R_m from a set of attributes A_1, A_2, … ,A_n.
  - Imagine that the attributes are originally all in one big relation R= {A_1, A_2, .., A_n} which we will call the ***Universal Relation***.
  - Normalization divides this relation into R_1, R_2, …, R_m.

Desirable Relational Schema Properties

- Relational schemas that are well-designed have several important properties:
  - 1) The most basic property is that relations consists of attributes that are logically related.
    The attributes in a relation should belong to only one entity or relationship.
  - 2) ***Lossless-join property*** ensures that the information decomposed across many relations can be reconstructed using natural joins.
  - 3) ***Dependency preservation property*** ensures that constraints on the original relation can be maintained by enforcing constraints on the normalized relations.
  - 4) Avoid update anomalies

Formal Model for Schema Design: Functional Dependencies

- Functional dependencies (FDs)
  - Are used to specify *formal measures* of the "goodness" of relational designs
  - And keys are used to define **normal forms** for relations
  - Are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes
- A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y
- Functional dependencies represent constraints on the values of attributes in a relation and are used in normalization

Functional Dependencies: Definition

- A ***functional dependency*** (abbreviated ***FD***) is a statement about the relationship between attributes in a relation. We say a set of attributes *X* functionally determines an attribute *Y* if given the values of *X* we always know the only possible value of *Y*.
  - Notation: $X \rightarrow Y$
  - *X* functionally determines *Y*
  - *Y* is functionally dependent on *X*
- Example:
  - eno $\rightarrow$ ename
  - eno, pno $\rightarrow$ hours

Defining Functional Dependencies

- X → Y holds if whenever two tuples have the same value for X, they *must have* the same value for Y
  - For any two tuples t1 and t2 in any relation instance r(R): If t1[X]=t2[X], *then* t1[Y]=t2[Y]
- X → Y in R specifies a *constraint* on all relation instances r(R)
- Written as X → Y; can be displayed graphically on a relation schema (denoted by the arrow).
  - FDs are derived from the real-world constraints on the attributes

Notation for Functional Dependencies

- A functional dependency has a left-side called the ***determinant*** which is a set of attributes, and one attribute on the right-side.

$$\text{eno, pno} \rightarrow \text{hours}$$

determined attribute

- Strictly speaking, there is always only one attribute on the RHS, but we can combine several functional dependencies into one:

$$\text{eno, pno} \rightarrow \text{hours}$$
$$\text{eno, pno} \rightarrow \text{resp}$$

$$\text{eno, pno} \rightarrow \text{hours, resp}$$

- Remember that this is really short-hand for two functional dependencies.

Why the Name "Functional" Dependencies?

- Functional dependencies get their name because you could imagine the existence of some function that takes in the parameters of the left-hand side and computes the value on the right-hand side of the dependency.
- Example: $\text{eno, pno} \rightarrow \text{hours}$

  $\text{f(eno, pno)} \rightarrow \text{hours}$

```
int f (String eno, String pno)
{       // Do some lookup...
        return hours;
}
```

- Remember that no such function exists, but it may be useful to think of FDs this way.

Defining FDs from instances

- Note that in order to define the FDs, we need to understand the meaning of the attributes involved and the relationship between them.
- An FD is a property of the attributes in the schema R
- Given the instance (population) of a relation, all we can conclude is that an FD *may exist* between certain attributes.
- What we can definitely conclude is – that certain FDs *do not exist* because there are tuples that show a violation of those dependencies.

Question

- We have attributes (A1, A2, A3, A4) in a table
- We have functional dependencies

$$A1 \rightarrow A2$$
$$A1 \rightarrow A3$$
$$A1 \rightarrow A4$$

Question: What property does A1 have ?

Next: Functional Dependencies & Normal Forms

- Normalization requires decomposing a relation into smaller tables
- Normal forms are properties of relations
- We say a relation is in xNF if its attributes satisfy certain properties
  - Properties formally defined using functional dependencies
  - For example, test the relation to see if it is in 3NF
  - If not in 3NF, then change design…how ?
    Decomposition

How to go about designing a good schema ?

- How to create a 3NF database schema ? (i.e., a good design) ?
- Ad-hoc approach
  - Create relations intuitively and hope for the best!
- Formal method – procedure Start with single relation with all attributes
  - Systematically decompose relations that are not in the desired normal form
  - Repeat until all tables are in desired normal form
  - Can decomposition create problems if we are not careful ?
    Yes: (i) Spurious tuples and (ii) lost dependencies
- Can we automate the decomposition process…
  Input: Set of attributes and their functional dependencies
  Output: A 'good' schema design

General Thoughts on Good Schemas

Ideally we want all attributes in every tuple to be determined only by the *superkey* (for key $X \rightarrow Y$, a superkey is a "non-minimal" X)

What does this say about redundancy?

But:

- *What about tuples that don't have keys (other than the entire value)?*

Sets of Functional Dependencies

- Relation EMP-DEPT(SSN, NAME, ADDRESS, DNUMBER, DNAME, MGRSSN}
  - Employee info; the dept they are assigned to; their manager's ssn
  - Key is SSN
- Some obvious functional dependencies
  - {SSN} → {NAME, ADDRESS, DNUMBER}
  - {DNUMBER} → {DNAME, MGRSSN}
- What else can we infer from above dependencies ?

Sets of Functional Dependencies

- Some obvious functional dependencies
  - {SSN} → {NAME, ADDRESS, DNUMBER}
  - {DNUMBER} → {DNAME, MGRSSN}
- From above dependencies, we can infer
  - {SSN} → {DNAME, MGRSSN}
- Concept of a set of dependencies that can be inferred from the given set
  - Inference rules ?
  - Closure: F^+ is all dependencies that can be inferred from F

Some Questions:

- Given a set of functional dependencies (properties on the data), what other properties can we infer ?
- What is the formal definition of a key ?
- How can we use the formal framework of Functional dependencies to define a 'good schema design" ?
- Can we automate the process (develop algorithms) ?

First question:

- Given a set F of functional dependencies, what are all the properties, i.e. dependencies, we can infer ?
- Do two sets of functional dependencies, F and G, imply the same set of properties ?
- How to formally define this property ?

Armstrong's Axioms: Inferring FDs

- Some FDs exist due to others; can compute using Armstrong's axioms:
- Reflexivity: If $Y \subseteq X$ then $X \rightarrow Y$ (*trivial dependencies*)
  name, sid → name
- Augmentation: If $X \rightarrow Y$ then $XW \rightarrow YW$
  cid → subj so cid, exp-grade → subj, exp-grade
- Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$
  cid→ crnum and crnum → subj
  so cid → subj

Armstrong's Axioms Lead to…

- Union: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$
- Pseudotransitivity: If $X \rightarrow Y$ and $WY \rightarrow Z$ then $XW \rightarrow Z$
- Decomposition: If $X \rightarrow Y$ and $Z \subseteq Y$ then $X \rightarrow Z$

Can prove these from Armstrong's Axioms…

Why Armstrong's Axioms?

- Why are Armstrong's axioms (or an equivalent rule set) appropriate for FD's? They are:
  - *Consistent*: any relation satisfying FD's in F will satisfy those in F^+
  - *Complete*: if an FD $X \rightarrow Y$ cannot be derived by Armstrong's axioms from F, then there exists some relational instance satisfying F but not $X \rightarrow Y$
- In other words, Armstrong's axioms derive ***all*** the FD's that should hold

Closure of a Set of FD's

Defn. Let F be a set of FD's.
Its *closure*, F^+, is the set of all FD's:
{$X \rightarrow Y$ | $X \rightarrow Y$ is derivable from F by Armstrong's Axioms}

Which of the following are in the closure of our Student-Course FD's?

name → name	F={	sid → name
crnum → subj		cid → crnum
cid → subj		(sid,cid) → expgrade
crnum, sid → subj		crnum → subject}
crnum → sid		

Computing the Closure of FDs

- The transitivity rule of FDs can be used for three purposes:
  - 1) To determine if a given FD $X \rightarrow Y$ follows from a set of FDs F.
  - 2) To determine if a set of attributes X is a superkey of R.
  - 3) To determine the set of all FDs (called the ***closure*** F^+) that can be inferred from a set of initial functional dependencies F.
- The basic idea is that given any set of attributes X, we can compute the set of all attributes X^+ that can be functionally determined using F. This is called the ***closure of X under F***.
  - For purpose #1, we know that $X \rightarrow Y$ holds if Y is in X^+.
  - For purpose #2, X is a superkey of R if X^+ is all attributes of R.
  - For purpose #3, we can compute X^+ for all possible subsets X of R to derive all FDs (the ***closure*** F^+) .

Computing the Attribute Closure

- The algorithm is as follows:
  - Given a set of attributes X.
  - Let $X^+ = X$
  - Repeat
    Find a FD in F whose left side is a subset of X^+.
    Add the right side of F to X^+.
  - Until (X^+ does not change)
- After the algorithm completes you have a set of attributes X^+ that can be functionally determined from X. This allows you to produce FDs of the form:
  - $X \rightarrow A$ where A is in X^+

Computing Attribute Set Closure

- For attribute set X, compute closure X^+ by:

Closure $X^+ := X$;
repeat until no change in X^+ {
  if there is an FD $U \rightarrow V$ in F
    such that U is in X^+
      then add V to X^+}

Attribute Closure: Example

- Let F be:
  - SSN → EName
  - PNUMBER → PNAME, PLOCATION
  - SSN, PNUMBER → HOURS
  - What is the closure of {SSN, PNUMBER}

Attribute Set Closure and Keys

- If X is a key over relation scheme R, then what is X^+
  - Formal definition of a Key
- How to determine the keys for relation R ?
  - R is a set of attributes $\{A_1, A_2, \ldots, A_n\}$
  - For each subset S of R, compute S^+
    If $S^+ = R$ then S is Key
  - What is the "catch" here ?
  - Can you improve this ?

Example

- R= (C,T,H,R,S)
  - Course (C), Time (T), Hour (H), Room (R), Section (S), Grade (G)

$C \rightarrow T$ $\quad CS \rightarrow G$

$HS \rightarrow R$ $\quad HR \rightarrow C$

$HT \rightarrow R$

Find all keys for this relation

Hint: What is the smallest attribute set that must be part of the key ?

Attribute Set Closures

- If attribute A does not appear on RHS of any FD, then any key must contain A
- If X is a key, then anything containing X is a superkey
- If X is a key, and $Y \rightarrow X$ is a FD then Y is a key

Example 2: Find All keys of R

- R= (A,B,C,D,E)
- $A \rightarrow BC$
- $CD \rightarrow E$
- $B \rightarrow D$
- $E \rightarrow A$

Next….Normal Forms

- Now we are ready to formally define the normal forms
  - And procedure to decompose a relation into one of the normal forms

- Informal definitions:
- 1st normal form
  - All attributes depend on **the key**
- 2nd normal form
  - All attributes depend on **the whole key**
- 3rd normal form
  - All attributes depend on **nothing but the key**