Chapter 14: Functional Dependencies and Normalization

When designing a DB, what is the best way to represent the real-world entities involved?
- common sense of the designer plays a big part
- here we formalize a measure of "goodness" of groupings of attributes in relations

Two levels to measure goodness:

logical level - how users interpret relation schemas - how well user understands meaning of data tuples

manipulation (storage) level - how tuples in base relations are stored and updated

Factors in designing database schemas:
1) Semantics of attributes
2) Reduce redundancy
3) Reduce null values
4) Eliminate spurious tuples

1) Semantics: meaning of the attributes in a tuple - how they relate to each other

ex: TOY database - the meaning of each relation schema is pretty straightforward

look at the TOY relation: TOY_NUM is a key (primary key)
NAME - the name of the toy
MAN_ID - a foreign key representing an implicit relationship between the TOY db and the MANUFACTURER db

- make semantics clear and easy to explain - one relation describes only one real world entity

2) Reduce redundancy:

ex: imagine that the TOY relation and the MANUFACTURER relation were all put into one relation:

TOY_MAN(TOY_NUM, NAME, MAN_NAME, ADDRESS, PHONE, SALES_CONTACT, MSRP, AGE_GROUP, NUM_IN_STOCK, SOLD_YTD)

- while this would reduce the number of relations, the number of tuples would increase because for every toy made by the same manufacturer, all of the manufacturer's data would have to be stored

TOY_MAN:

011	FARM HOUSE	FISCHER PRICE	111 MAIN ST... 29.95 ...
221	EXERSAUCER	FISCHER PRICE	111 MAIN ST... 45.00 ...

- Another problem with storing data in this way is the existence of update anomalies (3 types):

    - Insertion anomalies
        - every time a new tuple is added, all manufacturer data must be inserted - and must be consistent with the data that already exists for the same manufacturer in the other relations
        - how to insert a new manufacturer for which the catalog does not yet sell any toys - null values for toy info - but toy_num is a key and can't be null

    - Deletion Anomalies
        - if the last toy made by a manufacturer is deleted, then the manufacturer data is lost

    - Modification Anomalies
        - if value of data in a manufacturer changes, has to be changed in all tuples involving that manufacturer

  Avoid redundancy that can cause these anomalies - if any must exist to accomodate particular frequently used queries, document them so that they can be maintained

3) Null values: Avoid including attributes that are likely to have null values

- nulls waste space if used too often
- multiple interpretations of null
    - does not apply
    - unknown
    - known but missing - not yet recorded

If nulls must be used, make sure they apply only in rare cases.

4) Spurious tuples
- represent wrong information
- occur when joining two or more badly designed relations

ex: Take the TOY_MAN db from above and project it onto the following:

TOY_ADDR(NAME, ADDRESS) (primary key is (NAME,ADDRESS))

TOY_MAN1(TOY_NUM,MAN_NAME,ADDRESS, PHONE, SALES_CONTACT, MSRP, AGE_GROUP, NUM_IN_STOCK, SOLD_YTD)

- when these are joined together using ADDRESS, we end up with extraneous incorrect tuples - specifically:

    011 EXERSAUCER FISCHER PRICE 111 MAIN ST... 29.95 ...

- this is a rather contrived example - but we will discuss this topic more formally later when we discuss lossless joins

Functional Dependencies: constraint between two sets of attributes in a database

some formal notation:

R = {A_1,A_2,...A_n} is a single universal relation describing the whole database (assumption)

functional dependency: X->Y between two sets of attributes X and Y (subsets of R) - specifies a constraint on the possible tuples that can form a relation instance r of R.

for any tuples t1, t2 in r such that t1[X]=t2[X] we must have t1[Y]=t2[Y]
values of Y (right-hand side) depend on values of X (left-hand side)
or values of X functionally determine values of Y

X functionally determines Y in a realtion schema R iff whenever two tuples of r(R) agree on their X-value, they must necessarily agree on their Y-value
- note: X -> Y does not necessarily imply Y -> X

a legal extension (legal relation state) is one that satisfies functional dependency constraints

Ex: functional dependencies:
TOY_NUM -> NAME
- TOY_NUM uniquely defines the name of the toy

(draw picture depicting all FDs for toy catalog db using figure 12.3 - also draw the one from the book to use later)

- a functional dependency is a property of the schema and not of an instance - cannot be determined by looking at one tuple
- determined by semantics of the data
- individual instances can be used to demonstrate examples that are NOT functional dependencies
ex: both TODDLER TABLE and EXERSAUCER have MSRP=45.00, thus, we do not have MSRP -> NAME

- let F be the set of all functional dependencies specified on a relation schema R
- F^+ is the closure of F - set of all functional dependencies (those that may not be obvious semantically)
- such dependencies can be inferred from the ones in F - using a systematic set of inference rules

notation: F |= X -> Y
X -> Y is inferred from the dependencies in F

Ex: from above we have:

F = { SSN -> {ENAME, BDATE, ADDRESS, DNUMBER},
DNUMBER -> {DNAME, DMGRSSN} }

can infer the following:

SSN -> {DNAME, DMGRSSN}
SSN -> SSN
DNUMBER -> DNAME

Inference Rules:

IR1) Reflexive rule: If X is contained in Y, then X -> Y
IR2) Augmentation rule {X -> Y} |= XZ -> YZ
IR3) Transitive rule: {X -> Y, Y-> Z} |= X -> Z
IR4) Decomposition rule { X -> YZ} |= X -> Y
IR5) Union rule: { X -> Y, X -> Z} |= X -> YZ
IR6) Pseudotransitive rule { X -> Y, WY -> Z} |= WX -> Z

IR1) An attribute always determines itself

SSN -> SSN

IR2) Can add same set of attributes to both sides of a dependency and get another dependency

{SSN, ENAME} -> {BDATE, ENAME}

IR3) Transitivity

SSN -> {DNAME, DMGRSSN}

IR4) Can remove attributes from right-hand side of dependency

DNUMBER -> DNAME

IR5) Can combine sets of dependencies

{SSN -> ENAME , SSN -> BDATE} |= SSN -> {ENAME, BDATE}

IR6) Like transitivity

no example from above but look:

if we have X -> Y and WY -> Z, then using IR2 we can get WX -> WY and then using IR3 on we get WX -> Z

See proofs of these inference rules in book

Rules IR1 - IR3 are sound and complete -

sound: any dependency that can be inferred from IR1-IR3 is a valid dependency

complete: all dependencies can be inferred from IR1-IR3

For each set of attributes X , we determine X^+ - the set of attributes that are functionally determined by X (closure of X)

ex: from above F

$\{SSN\}^+ = \{SSN, ENAME, BDATE, ADDRESS, DNUMBER, DNAME, DMGRSSN\}$

$\{DNUMBER\}^+ = \{DNUMBER, DNAME, DMGRSSN\}$

$\{SSN, DNUMBER\}^+ = \{SSN, ENAME, BDATE, ADDRESS, DNUMBER, DNAME, DMGRSSN\}$

Two sets of functional dependencies, E and F, are equivalent if $E^+ = F^+$
i.e. every FD in E can be inferred from F and vice versa

Normal Forms:

- normalization - process by which unsatisfactory relation schemas are decomposed into smaller relation schemas that possess desireable properties
- one objective is to ensure that update anomalies do not occur

- formal framework for analyzing relation schemas
- series of tests to normalize schema to any degree - schema failing test must be further decomposed
- do not guarantee good db design by themselves - normalization must confirm existence of certain other properties:

lossless join property - guarantees no spurious tuples
dependency preservation property - all functional dependencies are represented

First Normal Form:

- domains of attributes must include only atomic (simple, indivisible) values
- values of any attribute must be a single value from domain
- disallows relations within relations

Ex: CUSTOMER relation - ages of children is a set of numbers and thus is not in 1NF

- break up CUSTOMER relation into two relations

CUST_CHILDREN(<u>CUST_NUM, CHILD_AGE</u>)

CUSTOMER(CUST_NUM, NAME, ADDRESS,
NUM_CHILDREN, LAST_ORDER_DATE)

- notice the primary key of CUST_CHILDREN is the combination of the two attributes

- could also have a distinct tuple in the CUSTOMER relation for each child with primary key being {CUST_NUM,AGE} but then there is a lot of redundancy

Second Normal Form:

based on the following idea:

a FD X -> Y is a full functional dependency if removal of any attribute from X means that the dependency does not hold

a FD is a partial FD if some attribute can be removed from the lhs and still get a FD

Ex: in the CUSTOMER relation

{NAME, ADDRESS} -> NUM_CHILDREN
- take out either NAME or ADDRESS and we do not have a FD
- full FD

{MAN_ID, NAME} -> NUM_CHILDREN
- can take out NAME, so a partial FD

- a relation schema R is in 2NF if every nonprime attribute (not a member of any key) A in R is fully functionally dependent on the primary key of R

Ex: Suppose the manufacturer relation did not have MAN_ID and instead used {MAN_NAME, PHONE} as a primary key

- the resulting relation would not be in 2NF because the nonprime attribute SALES_CONTACT would be partially dependent on the primarny key

i.e. FD: {MAN_NAME, PHONE} -> SALES_CONTACT
{PHONE, SALES_CONTACT}

- this new relation would have to be decomposed into the following to make it 2NF:

(MAN_NAME, PHONE, ADDRESS)

(PHONE, SALES_CONTACT)

Third Normal Form:

- based on the concept of transitive dependency

- A FD X -> Y is a transitive dependency if there is a set of attributes Z that is not a subset of any key and both X -> Z and Z -> Y hold

- a schema is in 3NF if it is in 2NF and no nonprime attribute of R is transitively dependent on the primary key

Ex: in the EMP_DEPT above, SSN -> DMGRSSN is a transitive dependency because we have SSN -> DNUMBER and DNUMBER -> DMGRSSN

- decompose into the following

ED1(ENAME, SSN, BDATE, ADDRESS, DNUMBER)

ED2(DNUMBER, DNAME, DMGRSSN)

- intuitively, we see that ED1 and ED2 represent separate concepts
  - ED1 - employee
  - ED2 - dept
- a natural join on the two would get the original relation we started with

More general definitions:

- the above definitions of 2NF and 3NF disallow partial and transitive dependencies involving primary keys - a more general definition involves all candidate keys

General definition of 3NF:

A relation schema R is in 3NF if every nonprime attribute of R is:
1) fully functionally dependent on every key of R and
2) nontransitively dependent on every key of R

Boyce-Codd Normal Form (BCNF)

- stricter than 3NF

A relation schema R is in BCNF if whenever a FD X -> A holds in R, then X is a superkey of R

- in practice, most schemas that re in 3NF are also in BCNF

BCNF is best - if not possible, use 3NF

- 1NF and 2NF were described to build 3NF and BCNF - not good for designing relation schemas by themselves