Chapter 15: Relational Database Design Algorithms and Further Dependencies

top-down design: designing a conceptual schema in a high-level data model (ER model) and mapping to a set of relations

relational systhesis: viewing relational database schema design in terms of functional and other dependencies - this is what this chapter is all about

Strict decomposition: start with one giant relation schema - universal relation:

$R = \{A_1, A_2, ..., A_n\}$ (all attributes in DB)

repeatedly decompose until cannot any longer, or no longer desireable

$D = \{R_1, R_2, ..., R_m\}$ (decomposed relations)

F is the set of functional dependencies used for decomposition

Goals:

1) each relation R_i in R to be in BCNF or 3NF (not good enough by itself for good design)

2) attribute preservation: make sure no attributes are lost in the decomposition, i.e.

R = union of all R_i

3) dependency preservation: all dependencies in F are represented in the decomposition D

- sufficient that the union of the dependencies in D be equivalent to F

- more formally:

projection of F on R_i:

$\Pi_F(R_i) = \{X \text{ -> } Y \text{ in } F^+ \mid (\text{all } A_j \text{ in } X \cup Y) \text{ contained in } R_i\}$

decomposition D of R is dependency preserving with respect to F if the union of the projections of F on each Ri in D is equivalent to F

$$((\Pi_F(R_1)) \cup ... \cup (\Pi_F(R_m)))^+ = F^+$$

- It is always possible to find a dependency preserving decomposition D with respect to F such that every relation in R_i in D is in 3NF (Algorithm 13.1 in book shows how)

  - based on the idea that we find with a minimal cover G for F
    - a set of dependencies that is equivalent to F with properties that minimize the set (see Ch 12 for formal definition)
  - all dependencies are preserved by equivalence
  - in 3NF because of minimal properties

4) lossless (nonadditive) join property: no spurious tuples are generated when a natural join is applied to the relations of the decomposition

- decomposition $D = \{R_1, R_2, ..., R_m\}$ of R has the lossless (nonadditive) join property with respect to F on R if for every relation state r of R that satisfies F we have:

  $* (\pi_{<R1>}(r), ..., \pi_{<Rm>}(r)) = r$

  the natural join of all projections gives the same state

- recall the example that we saw in which spurious tuples resulted:

  (pp. 42-43 of notes)

- Algorithm 13.2 in book tests for the lossless join property

- Other algorithms for finding decompositions:
  13.3 - lossless join decomposition into BCNF
  13.4 - lossless join, dependency-preserving decomposition into 3NF

  - Cannot guarantee all goals, do the best we can and deal with resulting anomalies when they arise

Null Values: all above algorithms assume no null values
- can cause problems with joins

  ex: if there is a toy in the TOY relation with a null MAN_ID, a query involving a join of the TOY with the MANUFACTURER relation would leave out that TOY

- an outer join could solve this problem - tuples with null values on join attributes still appear - but may give you more info than you want

- nulls can also cause problems with aggregate functions - how to interpret them
- Be cautious when assigning null to an attribute - especially foreign keys

Dangling Tuples: Assume that some entity is represented in more than one relation (may happen if relations are fragmented on distributed dbs) - if a tuple exists in one but not another, it is called a dangling tuple

- this could happen if we choose another alternative to using null values - leaving out the tuple

ex: break up the TOY relation as follows:

  TOY_1(TOY_NUM, MSRP, AGE_GROUP, NUM_IN_STOCK, SOLD_YTD)

  TOY_2(TOY_NUM, MAN_ID)

  - the first relation keeps all toy info, the second keeps pointer to manufacturer relation

- if we have toys with no man_id as in above example, they may be left out of TOY_2, and exist in TOY_1

Discussion:

- problem with above approach: difficult to specify all functional dependencies for a database
- algorithms are not deterministic: ex: minimal cover is not necessarily unique - so there may be more than one result depending on how minimal cover is defined
- this approach has not been as popular as top-down design
- combination approach: define relations in high-level model (like ER model) and decompose those based on the algorithms of this chapter

More Normal Forms:

multivalued dependencies:

- whenever two independent 1:N relationships A:B, and A:C are mixed in the same relation, an MVD may arise

Ex: Assume that each TOY relation contains information about the warehouses in which it is stored (each with a code name W1, W2,...), and that the same toy may go by more than one name:

TOY(TOY_NUM, NAME, WAREHOUSE, ...)

TOY_NUM	NAME	WAREHOUSE
011	FARM HOUSE	W1
011	FARM HOUSE	W2
011	BARN	W1
011	BARN	W2
302	TODDLER TABLE	W1
302	TODDLER TABLE	W2

The multivalued dependencies are: TOY_NUM ->> NAME
TOY_NUM ->> WAREHOUSE

Must represent all combinations of these values in tuples in order to avoid update anomalies - e.g. If Warehouse W1 were phased out, and we only represented its relationship with 011 using the name BARN, then deleting the tuple 011 BARN W1 would also delete info about the relationship between 011 and BARN.

4NF - For every (nontrivial) MVD X ->> Y, X must be a supkey of R

In above example, TOY_NUM is not a superkey of TOY, so to put into 4NF, we decompose TOY into TOY_NAME and TOY_WAREHOUSE

Then we still have TOY_NUM ->> NAME in TOY_NAME relation, but the dependency is trivial (Y subset X; or X U Y = R)

Algorithm 13.5 produces 4NF relations with the lossless join property
- does not necessarily preserve functional dependencies
- always replaces a non-4NF relation schema with 2 new ones and iterates

5NF - permits lossless decomposition into more than two relations (see book for more if interested)

Inclusion dependencies: (referential integrity constraints)
- interrelation dependencies like foreign key dependencies

- Let X be a set of attributes of relation schema R, Y be a set of attribs of relation schema S; r be an instance of R, s be an instance of S
  - an inclusion dependency R.X < S.Y specifies the constraint:

$\Pi_{<X>}(r)$ contains $\Pi_{<Y>}(s)$

ex: TOY.MAN_ID < MANUFACTURER.MAN_ID

- no normal forms developed for these

What do we get out of all of this:

- An idea of what a good design is by examining inconsistencies that could arise under normal operations (insert, delete, modify)

- Concepts of FDs and MVDs formalize ideas - permitted definition of algorithms

- Decomposition of relations

- Can cause extreme decomposition - have to use too many joins. What about more physical relationships between tables? More on this later.