CSCI 403: Databases
12 - Functional Dependencies and Normalization

Introduction

The point of this lecture material is to discuss some objective measures of the "goodness" of a database schema. The method relies on strong theory based in relational algebra and the notion of *functional dependency*. While it is possible to apply the theory (in theory) to generate a complete schema from some unnormalized starting point, for us the goal is to acquire some practical measures of "goodness" combined with an algorithm that can be easily applied to achieve a top-down design of our own schemas.

Informal guidelines

Some informal guidelines on what makes a schema "good":

- Clear semantics. Basically, the relations you create should make sense as independent units. You generally want clear entity distinctions and separation of concerns. This should mostly fall out naturally if you've done ER modeling first.

- Reducing redundancy. Data should be stored once and only once in the database (excluding foreign keys). This is not just a storage space consideration; redundancy in the database also leads to modification anomalies (more on this soon). For an example of redundancy in the data, see figure 1; for each instructor, office and email is repeated several times; for each course_id, title is repeated several times.

- Reducing NULLs. NULL values are undesirable for several reasons - they can be ambiguous in meaning, they don't play well with aggregate functions and joins, etc.

- Disallowing spurious tuple generation in joins. This has to do with not creating meaningless tuples when doing natural joins on tables.

Modification Anomalies

The redundancy in figure 1 occurs mainly because two (or more) concepts have been bundled together into one table. Specifically, instructor information is in the table together with course information, and these are independent and somewhat orthogonal pieces of data. Redundancy often goes hand in hand with a problem known as "modification anomalies", in which modifications to the database result in constraint violations or inconsistent data. Figure 1 demonstrates many opportunities for anomalies.

The various anomalies are easier illustrated than defined. An insertion anomaly in the table in figure 1 would occur, for instance, when we try to add a new faculty member who has not yet been assigned any courses. What values do we put in for course_id, section, and title? If we put in NULL values, we are possibly violating a key constraint (the only real key for this table is {course_id, section, instructor}). Also, when we later have course information for the faculty member, we would have to remember that there is NULL information in the table that needs to be overwritten.

Deletion anomalies are the inverse of insertion anomalies. What happens, for example, if we remove the last course taught by an instructor? The faculty member then ceases to exist! Similarly, if a faculty member leaves Mines, his or her courses vanish. This occurs because we are essentially bundling two separate entities into one table.

Finally, update anomalies are directly related to redundancy; if we try to update a faculty mem-

instructor	course_id	section	title	office	email
Painter-Wakefield, Christopher	CSCI403	A	Database Management	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	A	Data Structures	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	B	Data Structures	BB 280I	cpainter@mines.edu
Mehta, Dinesh	CSCI406	A	Algorithms	BB 280J	dmehta@mines.edu
Mehta, Dinesh	CSCI561	A	Theory of Computation	BB 280J	dmehta@mines.edu
Hellman, Keith	CSCI101	A	Intro to Computer Science	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI101	B	Intro to Computer Science	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI101	C	Intro to Computer Science	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI274	A	Intro Linux OS	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI274	B	Intro Linux OS	BB 310F	khellman@mines.edu

Figure 1: One possible relation with data about Mines courses and Mines faculty

ber's office information in the table in figure 1, we must make sure to update their information *everywhere* it occurs, not just in one or some rows, otherwise we introduce inconsistency into the data.

Spurious Tuples

Spurious tuples will occur primarily when a table has been factored incorrectly, or when a relation has been insufficiently or incompletely specified. For example, suppose we had two tables:

mines_courses
(instructor, course_id, section)

and

mines_faculty
(instructor, course_id, office, email)

In this example, if we join on the shared columns (instructor and course_id), we end up with tuples matching instructors to sections they do not actually teach.

Functional Dependencies

Our primary tool in the effort to eliminate redundancies and related anomalies from our database is something called a *functional dependency* (FD). A functional dependency is a 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 $X \rightarrow Y$ (read: X functionally determines Y, or Y is functionally dependent on X) exists if, whenever tuples t_1 and t_2 are two tuples from any relation $r(R)$ such that $t_1[X] = t_2[X]$, it is also true that $t_1[Y] = t_2[Y]$.

In words, if it is always true that whenever two tuples agree on attributes X they must also agree on attributes Y, then $X \rightarrow Y$.

For example, one functional dependency in the table in figure 1 is
instructor $\rightarrow$ {office, email}
if we assert that an instructor is always associated with only one office and email in our database.

This brings up an important point about functional dependencies; they are not properties of the *data*, but rather properties of the *world* which we impose on the data. That is, determining functional dependencies is a design activity (possibly informed by available data), which imposes a constraint on our database. A specific relation instance $r(R)$ may coincidentally have the property that all tuples agreeing on some attribute set X also agree on attribute set Y, but unless this *should hold true for any* $r(R)$ *that can exist*, we cannot say $X \rightarrow Y$.

Types of FDs

Just following the definition of an FD, we can make the observation that $X \rightarrow X$, trivially. More generally, if $Y \subseteq X$, then $X \rightarrow Y$ is called a *trivial* functional dependency. If $Y \nsubseteq X$ and $X \rightarrow Y$, this is called a *nontrivial* FD. Finally, if there is no overlap in attributes between X and Y, i.e., $X \cap Y = \varnothing$, then $X \rightarrow Y$ is called a *completely nontrivial* FD. We are mostly interested in completely nontrivial FDs.

Some nontrivial FDs in our example:
instructor $\rightarrow$ office
instructor $\rightarrow$ email
{course_id, section} $\rightarrow$ instructor
course_id $\rightarrow$ title

Properties

FDs can be viewed as a generalization of the notion of a superkey. Recall that a superkey is a set of attributes of a relation schema for which all tuples will be unique. A superkey is thus a subset of attributes of a relation that functionally determines the remaining attributes. Or the other way around, if $X \rightarrow Y$ and $X \cup Y = R$, then X is a superkey of R.

There are some inference rules (easily proved) that let us infer other FDs from an existing set of FDs:

- Splitting rule: If $A \rightarrow \{B_1, B_2\}$, then $A \rightarrow B_1$ and $A \rightarrow B_2$.
- Combining rule: If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow \{B, C\}$.
- Transitive rule: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

Additional rules can be found in the textbook. These inference rules, particularly the combining and transitive rules, are mostly of value to use in computing the closures of sets of attributes.

Closures

Given some set of functional dependencies F on a relation schema R, and some subset of attributes A, then the set $\{B_i : A \rightarrow B\}$ is called the *closure* of A and is denoted A^+. The closure plays a role in normalization (as we shall see), and can also be used in an algorithm to find all superkeys of a relation schema.

The closure A^+ is easily computed using the following algorithm:
Start with $S = A$. Clearly $A \rightarrow S$.
Repeat until no change:
if there exists an FD $X \rightarrow B$ in F such that $X \subset S$, then let $S = S \cup B$.

When no more expansions can be made to S, then $A^+ = S$. This algorithm simply applies the combining rule and transitive rules to expand the set S.

We can use the closure algorithm to find all superkeys of a relation schema as follows: simply construct every subset of the attributes of the relation schema and compute their closure. If the closure of a subset is equal to the entire set of attributes of the relation schema, then the subset is a superkey. While this may seem impractical, we can compute all superkeys relatively efficiently by starting with the smallest attribute subsets first; any superset of a superkey is a superkey, so we don't need to test supersets.

For our purposes, this algorithm has small utility, as we can typically identify the keys in our relation schemas by intuition. However, it is feasible to construct a normalized database from a single unnormalized relation schema automatically, if all functional dependencies are provided as part of the input. The algorithm to do this must compute keys, so the above algorithm plays a part in the larger algorithm.

Boyce-Codd Normal Form

The concept of normalization was developed to address the question of what a good relational design looks like. There are a series of *normal forms* which describe certain properties of a schema: first normal form (1NF), for instance eliminates any multivalued or compound (non-atomic) atrributes from the relation schema. Second normal form (2NF) and third (3NF) have their own rules, but these are mostly stepping stones on the way to the stronger normal forms which are our main focus: Boyce-Codd Normal Form (BCNF) and fourth normal form (4NF). Schemas in BCNF may be considered "good" for most purposes; 4NF is even stronger than BCNF, and thus are also "good".

Definition: A relation R is in Boyce-Codd Normal Form (BCNF) if for every nontrivial functional dependency $X \rightarrow A$ on R, X is a superkey of R.

Consider the example table in figure 1. If we start to examine some of the FDs we listed for this relation schema, we can quickly see that it is not in BCNF. For instance, consider the FD
instructor $\rightarrow$ office.

It is not difficult to see just from the sample data that instructor by itself cannot be a superkey of the relation. Therefore, we say that the FD
instructor $\rightarrow$ office
violates BCNF.

Decomposition algorithm

Fortunately, there are easy steps we can take to move closer to BCNF in our relation. There is a basic algorithm which starts with a relation schema not in BCNF and brings it to BCNF, which we detail now.

While not in BCNF:
- choose some R not in BCNF
- compute (super)keys of R
- choose some FD ($X \rightarrow Y$) in R violating BCNF
- (optional) Expand Y to be X^+
- Let $Z = R - (X \cup Y)$ be remaining attributes
- Replace R with two new relations R1 and R2:
— $R1 = \{X, Y\}$
— $R2 = \{X, Z\}$

The decomposition into R1 and R2 are accomplished by simply projecting R onto the attributes specified. Note that as usual, projection may result in fewer tuples in the resulting tables as duplicate tuples are removed.

As a final step, the FDs of R1 and R2 need to be recomputed. This step and the computing of superkeys of R is assumed to be (mostly) intuitive for a human, but if being performed by a machine, the key computation algorithm mentioned earlier and the FD inference rules can be used for these steps.

Note the optional step expanding Y; this tends to create larger/better relations, so it is a recommended step. For instance, suppose we used the FD we already noted violated BCNF in our example:
instructor $\rightarrow$ office.

If we follow the decomposition step above without the optional expansion step, we will create a table with just instructor and office, and another table with instructor and everything else. However, this still leaves the violating FD
instructor $\rightarrow$ email.

It makes more sense to keep all instructor info together, and that is in fact what will happen in this cases if we first computer instructor$^+$ = {instructor, office, email}. We will decompose one table with these three fields, and another table that now no longer has BCNF violating FDs with instructor on the left hand side. (We still have a violating FD, just not one involving instructor on the LHS.)

Example

Let's work through our example in more detail. We have the example table in figure 1, and a set of FDs:
instructor $\rightarrow$ office
instructor $\rightarrow$ email
{course_id, section} $\rightarrow$ instructor
course_id $\rightarrow$ title.

The only key for our table is {course_id, section, instructor}; any superset of attributes containing this set will be a superkey.

Applying the algorithm from the previous section, we choose a violating FD:
instructor $\rightarrow$ office.

Expanding the RHS as much as possible, we get
instructor $\rightarrow$ {office, email}.
(Technically the algorithm as I described would put instructor on the RHS as well; but since that is a trivial dependency, and a redundant column, I've left it out. Same end result.)

Now we decompose into two tables, shown in figures 2 and 3. Note how all the redundant information about instructors has been neatly eliminated by this step.

Now it is straightforward to show that the table in figure 2 is in BCNF. However, we still have a violating FD for the table in figure 3:
course_id $\rightarrow$ title.

Note that this FD corresponds pretty directly to the only real remaining redundancy in the table (aside from key values): the course titles. Again we continue through with the algorithm; the closure of course_id doesn't net us any additional columns, so we decompose into the two tables show in figures 4 and 5, which are now both in BCNF.

At this point, we are done, and our resulting schema has three tables in it: the ones in figures 2, 4, and 5. All inessential redundancy has been removed; we have a table of instructor information; a table with course information (course_id and title - but this could easily include things like number of hours, course description, etc.); and a table with actual schedule/section information for the current semester.

Note that the algorithm is not completely deterministic - there are points at which we get to make choices. The result is that there may be multiple correct normalizations of a complex

instructor	office	email
Painter-Wakefield, Christopher	BB 280I	cpainter@mines.edu
Mehta, Dinesh	BB 280J	dmehta@mines.edu
Hellman, Keith	BB 310F	khellman@mines.edu

Figure 2: One table resulting from the first decomposition of the example table.

instructor	course_id	section	title
Painter-Wakefield, Christopher	CSCI403	A	Database Management
Painter-Wakefield, Christopher	CSCI262	A	Data Structures
Painter-Wakefield, Christopher	CSCI262	B	Data Structures
Mehta, Dinesh	CSCI406	A	Algorithms
Mehta, Dinesh	CSCI561	A	Theory of Computation
Hellman, Keith	CSCI101	A	Intro to Computer Science
Hellman, Keith	CSCI101	B	Intro to Computer Science
Hellman, Keith	CSCI101	C	Intro to Computer Science
Hellman, Keith	CSCI274	A	Intro Linux OS
Hellman, Keith	CSCI274	B	Intro Linux OS

Figure 3: The other table resulting from the first decomposition of the example table.

course_id	section
CSCI403	Database Management
CSCI262	Data Structures
CSCI406	Algorithms
CSCI561	Theory of Computation
CSCI101	Intro to Computer Science
CSCI274	Intro Linux OS

Figure 4: First result of second decomposition.

instructor	course_id	section
Painter-Wakefield, Christopher	CSCI403	A
Painter-Wakefield, Christopher	CSCI262	A
Painter-Wakefield, Christopher	CSCI262	B
Mehta, Dinesh	CSCI406	A
Mehta, Dinesh	CSCI561	A
Hellman, Keith	CSCI101	A
Hellman, Keith	CSCI101	B
Hellman, Keith	CSCI101	C
Hellman, Keith	CSCI274	A
Hellman, Keith	CSCI274	B

Figure 5: Second result of the second decomposition.

database.

Correctness of Decomposition

There are two requirements which must be met in any decomposition of relations, which primarily add up to the statement that we must be able to recover the original relations exactly by natural joins.

The first requirement is simply that a natural join of the decomposed relations must include all of the attributes in the original relation. This is trivially satisfied in our decomposition algorithm by construction; when we decomposed, we partitioned the attributes so that a natural join gives us back exactly the attributes we started with.

The second requirement is called the *lossless join property*, and basically it says that when we natural join our decomposed tables together, we should regain exactly the tuples we started with, that is, all of the original tuples and no spurious tuples. I showed a proof of that this property is satisfied by our decomposition algorithm in class, which I omit here - it is not a difficult proof to reconstruct for yourself.

Multivalued Dependencies and Fourth Normal Form

Functional dependencies let us ferret out violations of BCNF and, as in the preceding example, this is often enough to attain a good database design. However, there are stronger normal forms than BCNF: 4NF, 5NF, and more besides. However, normal forms higher than 4NF involve rather exotic conditions, and we won't explore them in this class. We will take a brief look at 4NF, however. The tool that lets us understand 4NF is the multivalued dependency (MVD).

A multivalued dependency can be defined as follows:

A MVD $X \twoheadrightarrow Y$ exists on a relation R if whenever there are two tuples t_1 and t_2 which agree on the attributes in X, then there also exists tuple t_3 (possibly the same as t_1 or t_2) such that the following is true:

- $t_3[X] = t_1[X] = t_2[X]$
- $t_3[Y] = t_2[Y]$
- $t_3[Z] = t_1[Z]$,

where Z is everything that is not X or Y. By symmetry, there must also exist a tuple t_4 which also agrees on X with the other three tuples, but which reversed agreement on Y and Z with t_2 and t_1.

The notation $X \twoheadrightarrow Y$ can be read as "X multidetermines Y".

The situation can be summarized in the table below, where X, Y, and Z are our sets of attributes, and the lower-case x, y, and z's represent settings of these attributes.

	X	Y	Z
t_1	x	y_1	z_1
t_2	x	y_2	z_2
t_3	x	y_2	z_1
(t_4	x	y_1	z_2)

Note that not only does X multidetermine Y, X also necessarily multidetermines Z as well; it is common to write $X \twoheadrightarrow Y|Z$ to represent this.

This kind of situation arises primarily when there are two independent concepts bundled together in a table together. For example, consider a table which tracks an instructor's courses as well as their hobbies (what a strange relation!):

instructor	course_id	hobby
CPW	CSCI 403	reading sci-fi
CPW	CSCI 403	gardening
CPW	CSCI 262	reading sci-fi
CPW	CSCI 262	gardening

The hobbies and the course ids are completely independent, so you get a cross product situation happening where the table has to hold every pairing of these independent ideas. Note that the above table is in BCNF; indeed, there are no nontrivial FDs in the table at all!

Not surprisingly, the definition of fourth normal form uses MVDs in pretty much the same way as BCNF uses FDs:

Definition: A relation R is in 4NF with respect to some set of functional and multivalue dependencies if, for every nontrivial MVD $X \twoheadrightarrow Y$, X is a superkey in R.

Also not surprisingly, the resolution of the problem tracks identically as well - find a violating MVD $X \twoheadrightarrow Y|Z$ and decompose into tables $R1 = \{X, Y\}$ and $R2 = \{X, Z\}$.

As a final note, any FD also qualifies as an MVD; therefore, if all violating MVDs are re-

moved (putting us in 4NF), then all violating FDs are necessarily removed, and we are also in BCNF.