Database Design Issues

Hugh Darwen

hd@thethirdmanifesto.com
www.thethirdmanifesto.com

CS233.HACD: Introduction to Relational Databases
Section 9: Database Design Issues

A “Reducible” Relation

WIFE_OF_HENRY_VIII

Wife#	FirstName	LastName	Fate
1	Catherine	of Aragon	divorced
2	Anne	Boleyn	beheaded
3	Jane	Seymour	died
4	Anne	of Cleves	divorced
5	Catherine	Howard	beheaded
6	Catherine	Parr	survived

(Note the underscoring of the primary key attribute.)

A predicate for WIFE_OF_HENRY_VIII:

“The first name of the *Wife#*-th wife of Henry VIII is *FirstName* and her last name is *Lastname* and *Fate* is what happened to her.”

The appearances of the word “and” in this predicate indicate that it is “decomposible” into two or more simpler predicates:

1. “The first name of the *Wife#*-th wife of Henry VIII is *FirstName*.”

2. “The last name of the *Wife#*-th wife of Henry VIII is *Lastname* and *Fate* is what happened to her.”

The relations corresponding to predicates 1 and 2 are shown on the next slide.

"Decomposing" H8's Wives

W_FN

Wife#	FirstName
1	Catherine
2	Anne
3	Jane
4	Anne
5	Catherine
6	Catherine

W_LN_F

Wife#	LastName	Fate
1	of Aragon	divorced
2	Boleyn	beheaded
3	Seymour	died
4	of Cleves	divorced
5	Howard	beheaded
6	Parr	survived

Notice:

•W_FN = WIFE_OF_HENRY_VIII { Wife#, FirstName }

•W_LN_F = WIFE_OF_HENRY_VIII { Wife#, LastName, Fate }

•WIFE_OF_HENRY_VIII = W_FN **JOIN** W_LN_F

In other words, we use projection to perform the decomposition and JOIN to undo the decomposition and revert to the original.

The property by which WIFE_OF_HENRY_VIII can be decomposed this way, without any loss or gain of information, is known as a **join dependency**.

You've probably noticed that W_LN_F can be further decomposed. I'll come back to this point on a later slide. W_FN, however, cannot be further decomposed. We say that W_FN is an *irreducible relation.* Equivalently, we say that the relvar W_FN is in *sixth normal form* (6NF). The significance of the number 6 here will become apparent eventually.

Join Dependency

The *join dependency* (JD) that holds in WIFE_OF_HENRY_VIII and allows us to decompose in W_FN and W_LN_F is written like this:

* { { Wife#, FirstName }, { Wife#, LastName, Fate } }

- The star indicates a JD.
- The operands of a JD are written inside braces.
- Each operand is a set of attributes, hence the inner braces.

If a given JD holds in relation *r*, then *r* = the join of the projections indicated by the operands of the JD.

A Join Dependency That Does Not Hold

Although W_FN is irreducible, we can of course take several projections of it, the following two in particular:

W_FN { Wife# }

Wife#
1
2
3
4
5
6

W_FN { FirstName }

FirstName
Catherine
Anne
Jane

But the JOIN of these two does not yield W_FN, so the JD *{ { Wife# }, { FirstName} } does not hold in W_FN.

You might like to write down the result of (W_FN { Wife# }) JOIN (W_FN { FirstName }) if you need to convince yourself that it is not equal to W_FN.

Decomposition of W_LN_F

W_LN_F can be further decomposed:

W_LN

Wife#	LastName
1	of Aragon
2	Boleyn
3	Seymour
4	of Cleves
5	Howard
6	Parr

W_F

Wife#	Fate
1	divorced
2	beheaded
3	died
4	divorced
5	beheaded
6	survived

We have decomposed predicate 2 ("The last name of the *Wife#*-th wife of Henry VIII is *Lastname* and *Fate* is what happened to her") into:

3. "The last name of the *Wife#*-th wife of Henry VIII is *Lastname*."
4. "*Fate* is what happened to the *Wife#*-th wife of Henry VIII."

3-Way Join Dependency

So the following JD holds in W_LN_F:

* { { Wife#, LastName }, { Wife#, Fate } }

and we can conclude that the following 3-way JD holds in WIFE_OF_HENRY_VIII:

* { { Wife#, FirstName }, { Wife#, LastName }, { Wife#, Fate } }

i.e., WIFE_OF_HENRY_VIII =
WIFE_OF_HENRY_VIII { Wife#, FirstName } JOIN
WIFE_OF_HENRY_VIII { Wife#, LastName } JOIN
WIFE_OF_HENRY_VIII { Wife#, Fate }

Design Comparison

Does the decomposed design have any advantages?

The single relvar design carries an implicit constraint to the effect that every wife has a wife number, a first name, a last name and a fate.

This constraint is not implicit in the decomposed design.

In fact, to enforce it, each of W_FN, W_LN and W_F needs a foreign key referencing each of the other two.

But then the first attempt to insert a tuple into any of them must fail (unless multiple assignment is available).

Conclusion

In the example at hand, the single relvar design is preferred, *so long as the constraint implied by it truly reflects the real world.*

(For example, if it turns out that in fact not every wife has a last name, then we should separate out W_LN.)

But the example at hand is rather special:
- Each operand of the 3-way JD that holds in WIFE_OF_HENRY_VIII includes a key of that relvar
- and it is the same key in each case, viz. {Wife#}

We need to look at some not-so-special examples.

We note in passing that WIFE_OF_HENRY_VIII has another key, {LastName}, if we can assume that, being dead, Henry cannot possibly have any more wives. So the following JD also holds in WIFE_OF_HENRY_VIII:

* { { Lastname, Wife# }, { LastName, FirstName }, { LastName, Fate } }

A Not-So-Special JD

ENROLMENT

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Davinder	C1

Recall that we decided to "split" (i.e., decompose) this one, as follows …

Splitting ENROLMENT

IS_CALLED

StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Davinder

IS_ENROLLED_ON

StudentId	CourseId
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

Notice the JD: *{ { StudentId, Name }, { StudentId, CourseId } } that holds in ENROLMENT.

Note that the key {StudentId, CourseId} of ENROLMENT is retained as the key of IS_ENROLLED_ON, but is not a key of IS_CALLED. The special thing about our WIFE_OF_HENRY_VIII example was that a key of WIFE_OF_HENRY_VIII was retained as a key of each of the three relvars resulting from its decomposition.

Although {StudentId} is not a key of ENROLMENT, it *is* a key of ENROLMENT { StudentId, Name } (and therefore of the relvar IS_CALLED).

Functional Dependency

Because {StudentId} is a key of the projection ENROLMENT{StudentId, Name}, we say that the following *functional dependency* (FD) holds in ENROLMENT:

{ StudentId } ↦ { Name}

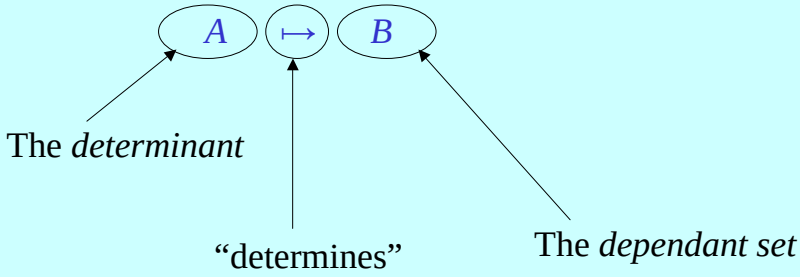
- The arrow, pronounced “determines”, indicates an FD.
- Each operand is a set of attributes (hence the braces).

Name is a function of StudentId.
For each StudentId there is exactly one Name.

Here is a list of all the FDs that hold in ENROLMENT (sorry about the FD arrow, not printing properly in 8-point, needed to fit these on one page):

{ StudentId } ↦ { StudentId} (this is a trivial FD: everything determines itself)
{ Name } ↦ { Name }
{ CourseId } ↦ { CourseId }
{ StudentId, CourseId } ↦ { StudentId, CourseId }
{ StudentId, CourseId } ↦ { StudentId }
{ StudentId, CourseId } ↦ { CourseId }
{ StudentId, Name } ↦ { StudentId, Name }
{ StudentId, Name } ↦ { StudentId }
{ StudentId, Name } ↦ { Name }
{ Name, CourseId } ↦ { Name, CourseId }
{ Name, CourseId } ↦ { Name }
{ Name, CourseId } ↦ { CourseId }
{ StudentId, Name, CourseId } ↦ { StudentId, Name, CourseId }
{ StudentId, Name, CourseId } ↦ { StudentId, Name }
{ StudentId, Name, CourseId } ↦ { StudentId, CourseId }
{ StudentId, Name, CourseId } ↦ { Name, CourseId }
{ StudentId, Name, CourseId } ↦ { StudentId }
{ StudentId, Name, CourseId } ↦ { Name }
{ StudentId, Name, CourseId } ↦ { CourseId }
{ StudentId } ↦ { } (this is trivial too: everything determines nothing, so to speak)
{ Name } ↦ { }
{ CourseId } ↦ { }
{ StudentId, CourseId } ↦ { }
{ StudentId, Name } ↦ { }
{ Name, CourseId } ↦ { }
{ StudentId, Name, CourseId } ↦ { }
{ StudentId } ↦ { Name } (this is the only nontrivial FD that holds in ENROLMENT)
{ StudentId, CourseId } ↦ { Name } (this is a trivial consequence of our only nontrivial FD)

Anatomy of an FD



Reminder: The determinant is a *set* of attributes, and so is the dependant set.

P.S. "dependant" is not a misspelling! It's the noun, not the adjective.

FDs That Do Not Hold in ENROLMENT

{ Name } ↦ { StudentId }
{ Name } ↦ { CourseId }
{ CourseId } ↦ { StudentId }
{ CourseId, Name } ↦ { StudentId }
{ StudentId } ↦ { CourseId }
{ StudentId, Name } ↦ { StudentId }
{ CourseId, Name } ↦ { StudentId }

also: { x } ↦ { StudentId }, because x is not an attribute of ENROLMENT.

Theorems About FDs

Assume $A \mapsto B$ holds in r. Then:

Left-Augmentation: If A' is a superset of A, then $A' \mapsto B$ holds in r.

Right-reduction: If B' is a subset of B, then $A \mapsto B'$ holds in r.

Transitivity: If $A \mapsto B$ and $B \mapsto C$ hold in r, then $A \mapsto C$ holds in r.

In general: If $A \mapsto B$ and $C \mapsto D$ hold in r,
then:

$$A \cup (C - B) \mapsto B \cup D$$

holds in r.

That final theorem (its proof is left as an exercise for those who are into such things) is sometimes referred to in the literature as “pseudotransitivity”.

Left-Irreducibility

If $A \mapsto B$ holds in r and there is no proper subset A' of A such that $A' \mapsto B$ holds in r, then $A \mapsto B$ is a *left-irreducible* FD (in r).
In this case, B is sometimes said to be *fully* dependent on A.

Conversely, if there *is* such a subset A', then $A \mapsto B$ is a *left-reducible* FD (in r), and B is therefore *not fully* dependent on A.

FDs and Keys

If $A \mapsto B$ is an FD in r and $A \cup B$ constitutes the entire heading of r, then A is a *superkey* of r.

If $A \mapsto B$ is a left-irreducible FD in r and $A \cup B$ constitutes the entire heading of r, then A is a *key* of r.

(The longer term *candidate key* is often used instead of *key*, for historical reasons.)

E.F. Codd introduced the term *candidate key* to indicate candidacy (of a set of attributes) for being the primary key. We have since decided that although the concept of a primary key is useful, it is not fundamentally important. For example, SQL allows several keys to be declared for a table without any one of them being singled out as primary. In that case, any foreign key referencing such a table must include an explicit specification of the referenced columns (which in SQL default to being the columns of the primary key). But mere matters of syntax and convenience are not fundamental!

Normal Forms

Arising from the study of JDs in general and FDs in particular, various “normal forms” have been defined:

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- **Boyce/Codd Normal Form (BCNF)**
- Fourth Normal Form (4NF)
- **Fifth Normal Form (5NF)**
- **Sixth Normal From (6NF)**

Each of these is in a sense stricter than its immediate predecessor. The ones shown in bold are particularly important. The others were early attempts that eventually proved inadequate.

Normalisation

Normalisation is the act of decomposing a relvar that fails to satisfy a given normal form (e.g., BCNF) such that the result is an equivalent set of two or more “smaller” relvars that do satisfy that normal form.

Normalisation to a given NF can be done in steps through the NFs in order (2NF to 3NF to BCNF and so on, as required).

Purposes of Normalisation

A database all of whose relvars satisfy 6NF has the following possibly desirable properties:

- No redundancy (i.e., no recording of the same information more than once)
- No “update anomalies” (to be explained later)
- Orthogonality (independent recording of the simplest facts)

But 5NF is usually sufficient (and 6NF is sometimes problematical with existing technology), as we shall see …

First Normal Form (1NF)

On this course we shall assume that every relation, and therefore every relvar, is in 1NF.

The term (due to E.F. Codd) is not clearly defined, partly because it depends on an ill-defined concept of “atomicity” (of attribute values).

Some authorities take it that a relation is in 1NF iff none of its attributes is relation-valued or tuple-valued. It is certainly recommended to avoid use of such attributes (especially RVAs) in database relvars.

Second Normal Form (2NF)

This is the lowest NF that ENROLMENT fails to satisfy!

> Relvar *R* with primary key *K* is in 2NF if and only if, for every nonkey attribute *a* of *R*, $K \mapsto \{ a \}$ is a left-irreducible FD.

In ENROLMENT, { StudentId, CourseId } ↦ { Name} holds but is not left-irreducible, because { StudentId } ↦ { Name} also holds.

The definition of 2NF assumes the primary key is the only key. Shouldn't all keys have the same property?
("Nonkey" here means "not a member of the primary key".)

2NF is sometimes defined using the term "fully dependent". A relvar is in 2NF if and only if every nonkey attribute is fully dependent on the primary key.

Splitting ENROLMENT (bis)

IS_CALLED

StudentId	Name
S1	Anne
S2	Boris
S3	Cindy
S4	Davinder
S5	Boris

IS_ENROLLED_ON

StudentId	CourseId
S1	C1
S1	C2
S2	C1
S3	C3
S4	C1

The attributes involved in the “rogue” FD have been separated into IS_CALLED, *and now we can add student S5*!

IS_CALLED and IS_ENROLLED_ON are in 2NF. As it happens, they are also in 3NF, BCNF, 4NF, 5NF and 6NF, so we need to look at some different examples to discover cases where a lower NF holds while a higher one is violated.

Advantages of 2NF

With reference to our ENROLMENT example, decomposed into the 2NF relvars IS_CALLED and IS_ENROLLED_ON:

- Anne's name is recorded twice in ENROLMENT, but only once in IS_CALLED. In ENROLMENT it might appear under different spellings (Anne, Ann), unless the FD { StudentId } ↦ { Name} is declared as a constraint. **Redundancy** is the problem here.
- With ENROLMENT, a student's name cannot be recorded unless that student is enrolled on some course, and an anonymous student cannot be enrolled on any course. Lack of **orthogonality** is the problem here.

BUT 2NF doesn't always solve all problems of this nature.

These problems give rise to what have been called "update anomalies":

1. Inability to record some fact (in this case, somebody's name) without at the same time recording some other fact (in this case, an enrolment).
2. Loss of a fact (in this case, somebody's name) when some other recorded fact ceases to be true (in this case, the only enrolment for the student in question).
3. The same update (e.g., correction of somebody's name) might need to be done in more than one place (so to speak).
4. When a fact is recorded (in this case, an enrolment), the accompanying fact, if already recorded (e.g. S!'s name), must be copied faithfully.

In 2NF But Still Problematical

TUTORS_ON

StudentId	TutorId	TutorName	CourseId
S1	T1	Hugh	C1
S1	T2	Mary	C2
S2	T3	Lisa	C1
S3	T4	Fred	C3
S4	T1	Hugh	C1

Assume the FD { TutorId } ↦ { TutorName} holds.

Every nonkey attribute (TutorId and TutorName) is irreducibly dependent on { StudentId, CourseId }, so this is in 2NF. However, the nonkey attribute TutorName is dependent on TutorId, also a nonkey attribute.

We have the same problems as with our non-2NF relvar: T1's name is recorded twice (so we need to make sure it is always spelled the same way and not, for example, Hugh in one place and Huw in another), and we cannot record the name of a tutor who hasn't been assigned for any enrolment yet.

Third Normal Form (3NF)

Note that 2NF doesn't cater for an FD whose determinant is not a subset of the primary key. Therefore we need 3NF:

> Relvar *R* with primary key *K* is in 3NF if and only if it is in 2F and no FD $NK \mapsto \{ a \}$ holds in *R* such that *K*, *NK* and $\{ a \}$ are disjoint, where *a* is an attribute or *R*.

Because $K \mapsto NK$ and $NK \mapsto \{ a \}$, *a* is sometimes said to be "transitively dependent" on the primary key *K*.

Like 2NF, 3NF is problematical because it assumes the primary key is the only key.

Note that *a* here is necessarily a nonkey attribute of *R*, where a nonkey attribute is one that is not a member of the primary key.

In TUTORS_ON we have an FD, { TutorId } ↦ { TutorName }, such that its determinant, its dependant set and the primary key are disjoint, thus violating 3NF. TutorName is "transitively dependent" on the primary key.

Splitting TUTORS_ON

TUTOR_NAME

TutorId	TutorName
T1	Hugh
T2	Mary
T3	Lisa
T4	Fred
T5	Zack

TUTORS_ON_3NF

StudentId	TutorId	CourseId
S1	T1	C1
S1	T2	C2
S2	T3	C1
S3	T4	C3
S4	T1	C1

Now we can put Zack, who isn't assigned to anybody yet, into the database.

Boyce/Codd Normal Form (BCNF)

Unlike 2NF and 3NF, BCNF caters for relvars with more than one key (and subsumes both 2NF and 3NF):

> Relvar *R* is in BCNF if and only if for every nontrivial FD $A \mapsto B$ satisfied by *R*, *A* is a superkey of *R*.

More loosely, “every nontrivial determinant is a [candidate] key”.

BCNF addresses all JDs that are consequences of FDs.

But it doesn’t address all JDs, nor even all harmful JDs …

TUTOR_FOR is not in BCNF because { TutorId } is the determinant of a nontrivial FD and yet is not a superkey.

Not In 3NF But Not Problematical

EMP

Employee#	Name	NatIns#
001	Black	YH123456A
002	White	ZQ654321B
003	Green	QQ456781C
004	Brown	YH333333B
005	Indigo	ZH444444D

Assume the FD { NatIns# } ⟼ { Name, Employee# } holds.

Under the assumed FD we have { Employee# } ⟼ { NatIns# } ⟼ { Name }, an apparent violation of 3NF. And yet we have no redundancy, not updating problems. We do lack orthogonality, but that is not a problem if it really is the case that every employee has both a name and a national insurance number.

The reason why EMP is not a problem in spite of apparently violating 3NF is that EMP has two keys, {Employee#} and {NatIns#}. The determinant of the FD, { NatIns# } ⟼ { Name, Employee# }, that makes it violate 3NF, is a key (though not the one arbitrarily chosen to be the primary key). Because it is a key, the problems normally arising from not being in 3NF do not arise.

Next we look at a relvar that appears to be in 3NF and yet *is* problematical …

In 3NF But Problematical

TUTOR_FOR

StudentId	TutorId	CourseId
S1	T1	C1
S1	T2	C2
S2	T3	C1
S3	T4	C3
S4	T1	C1

Now assume the FD { TutorId } ↦ { CourseId } holds.

Under the assumed FD we have the possibly unrealistic situation, for the sake of an example, where no teacher teaches more than one course.

We have exactly the same problems of redundancy and lack of orthogonality, and yet TUTOR_FOR is in 3NF according to our definition, because TutorId is not a nonkey attribute.

Note that although each tutor teaches only one course, and exactly one tutor is assigned for a single enrolment, the same course can have several tutors (T1 and T3 both teach C1).

Note that the primary key, {StudentId, CourseId} is not the only key of TUTOR_FOR. Because {Tutor} ↦ {CourseId} holds, {Tutor, StudentId} ↦ {CourseId} also holds, under left-augmentation, therefore {Tutor, StudentId} is also a key.

But if {Tutor, StudentId} is chosen to be the primary key, then TUTOR_FOR is not in 2NF!!

So pursuit of 2NF and 3NF fails to achieve its objective in certain cases. Note that these “rogue” cases can only arise when there is more than one key and two of the keys “overlap” (have an attribute in common). However, having overlapping keys doesn’t *necessarily* lead to a violation of 3NF.

Splitting TUTOR_FOR

TUTORS

StudentId	TutorId
S1	T1
S1	T2
S2	T3
S3	T4
S4	T1

TEACHES

TutorId	CourseId
T1	C1
T2	C2
T3	C1
T4	C3

Note the keys.
Have we "lost" the FD { StudentId, CourseId } ↦ { Tutor } ?

Yes, we have lost the constraint that was expressed by the key, {StudentId, CourseId} of TUTOR_FOR. If we do not somehow reinstate that constraint, it will be possible to enrol student S1 of course C1 for a second time, with tutor T3.

Reinstating The Lost FD

Need to add the following constraint:

```
CONSTRAINT KEY_OF_TUTORS_JOIN_TEACHES
  WITH TUTORS JOIN TEACHES AS TJT :
    COUNT (TJT) = COUNT (TJT {StudentId, CourseId}) ;
```


Any functional dependency, $A \mapsto B$, in relvar R can be expressed as a constraint of the form:

COUNT (R { A , B }) = COUNT (R { A })

In BCNF But Still Problematical

TBC1

Teacher	Book	CourseId
T1	Database Systems	C1
T1	Database in Depth	C1
T1	Database Systems	C2
T1	Database in Depth	C2
T2	Database in Depth	C2

Assume the JD *{ { Teacher, CourseId }, { Book, CourseId } } holds.

Perhaps the predicate is "*Teacher* uses *Book* on course *CourseId*". In that case, the given JD tells us that if teacher *t* uses book *b* at all and *b* is used by anybody on course *c*, then *t* uses *b* on *c*.

The two books in the example are both used on C1 and C2 and teacher T1 uses both books. The appearance of the fourth tuple shown in the diagram is implied by the fifth (showing that somebody uses "Database in Depth" on C2) and the second (showing that T1 uses "Database in Depth" on some course). The JD also tells us that T2 does not use "Database Systems" at all, for otherwise that teacher would be compelled to use it on course C2.

Now, suppose we learn that T2 is to teach course C1 and has decided to use "Database Systems" on that course. Then under the JD (assuming it is declaed as a constraint) the DBMS must reject any attempt to insert the tuple <T2, Database Systems, C1> unless the tuple <T2, Database Systems, C2> is inserted at the same time.

Once again we have redundancy: that T1 uses Database Systems is recorded twice, and that Database in Depth is used on C2 is also recorded twice. And once again we have lack of orthogonality. The simple predicates "*Teacher* uses *Book*" and "*Book* is used on course *CourseId*" are not treated independently.

Fourth Normal Form (4NF)

4NF subsumes BCNF and caters for harmful JDs that have exactly two operands.

> Relvar *R* is in 4NF if and only if every nontrivial *binary* JD that holds in *R* is implied by the keys of *R*.

This is not the original definition, which used an arcane concept called multi-valued dependency (MVD) and was drafted before the more general concept of JD had been noted.

The definition of 4NF is unsatisfactory precisely because it fails to cover JDs of higher degree than 2.

TBC1 is not in 4NF because the nontrivial JD *{ { Teacher, CourseId }, { Book, CourseId } } is not implied by the only key of TBC1, which is { Teacher, Book, CourseId }, the entire heading.

A JD is “implied by the keys of *R*” if each of its projections includes the same key of *R*.

Normalising TBC1

TB

Teacher	CourseId
T1	C1
T1	C2
T2	C2

BC

Book	CourseId
Database Systems	C1
Database in Depth	C1
Database Systems	C2
Database in Depth	C2

But does a teacher really have to use a book just because he or she teaches a course that uses that book?
Meet 5NF …

In 4NF But Still Problematical

TBC2

Teacher	Book	CourseId
T1	Database Systems	C1
T1	Database in Depth	C1
T1	Database Systems	C2
T1	Database in Depth	C2
T2	Database in Depth	C2

Now assume the JD *{ { Teacher, Book }, { Book, CourseId }, { Teacher, CourseId } } holds.

Still taking the predicate to be "*Teacher* uses *Book* on course *CourseId*", the revised JD tells us that if teacher *t* uses book *b* at all and *b* is used by anybody on course *c*, and *t* teaches course *c*, then *t* uses *b* on *c*.

We are heading for a 3-way decomposition, as we were with the decomposition on WIFE_OF_HENRY_VIII, but this time there can be no intermediate step—we cannot do it by two 2-way decompositions. Our ternary JD here is what might called *essentially* ternary.

Fifth Normal Form (5NF)

5NF subsumes 4NF and caters for *all* harmful JDs.

> Relvar *R* is in 5NF iff every nontrivial JD that holds in *R* is implied by the keys of *R*.

Time to explain “nontrivial”: A JD is trivial if and only if one of its operands is the entire heading of *R* (because every such JD is clearly satisfied by *R*).

Normalising TBC2

TB

Teacher	Book
T1	Database Systems
T1	Database in Depth
T2	Database in Depth

BC

Book	CourseId
Database Systems	C1
Database in Depth	C1
Database Systems	C2
Database in Depth	C2

TC

Teacher	CourseId
T1	C1
T1	C2
T2	C2

Verify that TB JOIN BC JOIN TC = TBC.

Sixth Normal Form (6NF)

6NF subsumes 5NF and is the strictest NF:

> Relvar *R* is in 6NF if and only if every JD that holds in *R* is trivial.

6NF provides maximal orthogonality, as already noted, but is not normally advised. It addresses additional anomalies that can arise with temporal data (beyond the scope of this course—and, what's more, the definition of join dependency has to be revised).

Wives of Henry VIII in 6NF

W_FN

Wife#	FirstName
1	Catherine
2	Anne
3	Jane
4	Anne
5	Catherine
6	Catherine

W_LN

Wife#	LastName
1	of Aragon
2	Boleyn
3	Seymour
4	of Cleves
5	Howard
6	Parr

W_F

Wife#	Fate
1	divorced
2	beheaded
3	died
4	divorced
5	beheaded
6	survived

Not a good idea!

Use 6NF only for nonkey attributes that are “optional”. If every wife has a first name but not every wife has a last name and not every wife has a fate, then the above design is a good one.

Decomposing a 5NF relvar introduces orthogonality but does not eliminate any redundancy because a 5NF relvar is free of redundancy.