Databases - Redundancy

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

This lecture

Redundancy in a DBMS refers to the storage of the same piece of data in multiple places.

While *controlled redundancy* (for example, system backups) are necessary, dealing with *uncontrolled redundancy* is a major issue in any database management system.

The concepts of *functional dependencies* and the associated theory of *normalization* is a mathematical theory dealing with redundancy.

Redundancy

One of the main reasons for using relational tables for data is to avoid the problems caused by *redundant storage* of data.

For example, consider the sort of general information that is stored about a student:

- Student Number
- Name
- Address
- Date of Birth

Different parts of the university may keep different additional items of data regarding students, such as grades, financial information and so on.

Repeating Data

Suppose that marks are kept in the following format:

Student Number	Name	Unit Code	Mark
14058428	John Smith	CITS1402	72
14058428	John Smith	CITS1401	68
14058428	John Smith	CITS2200	68
15712381	Jill Tan	CITS1401	88
15712381	Jill Tan	CITS1402	82

Then this table contains *redundant data*, because the student's name is repeated in numerous different rows.

If the financial system also stores student numbers and names, then there is redundancy *between* tables as well as *within* tables.

Problems with redundancy

Apart from unnecessary storage, redundancy leads to some more significant problems:

- *Update Anomalies*
  If one copy of a data item is *updated* — for example, a student changes his or her name — then the database becomes inconsistent unless *every copy* is updated.
- *Insertion Anomalies*
  A new data item — for example, a new mark for a student — cannot be entered without adding some other, potentially unnecessary, information such as the student's name.
- *Deletion Anomalies*
  It may not be possible to delete some data without losing other, unrelated data, as well (an example is on the next slide).

Deletion Anomalies

A *deletion anomaly* occurs when a table storing redundant information becomes a proxy for storing that information properly.

For example, suppose that a company pays fixed hourly rates according to the level of an employee:

Name	Level	Rate
Smith	10	55.00
Jones	8	30.00
Tan	10	55.00
White	9	42.00
⋮	⋮	⋮

This table contains not only the employee data, but also the *association* between the level of an employee and the rate for that level.

What if Jones leaves?

If Jones happens to be the *only* employee currently at level 8, and he leaves and is deleted from the database, then the more general information that "*The hourly rate for Level 8 is \$30.00*" is also lost.

In this situation a better approach is to keep a *separate table* that relates levels and rates.

Level	Rate
⋮	⋮
8	30.00
9	42.00
10	55.00
⋮	⋮

Name	Level
Smith	10
Jones	8
Tan	10
White	9
⋮	⋮

Separating the student tables

The redundancy problems with the student information can also be resolved by creating a separate table with *just* the basic student information

Student Number	Name
14058428	John Smith
15712381	Jill Tan

and then the marks in a separate table.

Student Number	Unit Code	Mark
14058428	CITS1402	72
14058428	CITS1401	68
14058428	CITS2200	68
15712381	CITS1401	88
15712381	CITS1402	68

Decomposition

Both of these examples were improved by replacing a table with redundancy with *two tables*, each containing *a subset* of the original attributes (columns).

This leads to the following definition:

A *decomposition* of a relation schema R is a set of two (or more) relation schemas, each containing a subset of the attributes of R, such that together, the replacement schemas contain all the attributes of R.

Note that the idea of a decomposition of a relation (and the normalization of a DB) relates to the *structure* of the relations, not the *contents* of the relations.

(In other words, we are dealing with *relation schemas* rather than *relation instances*.)

Example

Suppose that R is the original "Student Number / Name / Unit Code / Mark" schema above — we'll abbreviate this to

$$R = SNUM$$

(S =**S**tudent Number, N = **N**ame, U = **U**nit Code, M = **M**ark).

Then the decomposition suggested above would decompose R into

$$R_1 = SN \qquad R_2 = SUM$$

> Is it better to use *one relation* R with attributes $SNUM$ or *two relations* R_1 and $R_2$with attributes SN and SUM?

Which is better

Before we can answer this, or even think about it clearly, we need some more concepts.

If we replace R by R_1 and R_2, how would the *data* stored in R be split up?

A moment's thought tells us that the *only possible* thing that makes sense is for R_1 and R_2 to each be defined as the *projection* of R onto the relevant subset of attributes.

$$R_1 = \pi_{SUM}(R)$$
$$R_2 = \pi_{SN}(R)$$

In MySQL

```
CREATE TABLE R (S INT, N VARCHAR(16), U VARCHAR(8), M INT);

INSERT INTO R VALUES(14058428,"John Smith","CITS1401",72);
INSERT INTO R VALUES(14058428,"John Smith","CITS1402",68);
INSERT INTO R VALUES(14058428,"John Smith","CITS2200",68);
INSERT INTO R VALUES(15712381,"Jill Tan","CITS1401",88);
INSERT INTO R VALUES(15712381,"Jill Tan","CITS1402",68);

SELECT * FROM R;
+----------+------------+----------+------+
| S        | N          | U        | M    |
+----------+------------+----------+------+
| 14058428 | John Smith | CITS1401 |   72 |
| 14058428 | John Smith | CITS1402 |   68 |
| 14058428 | John Smith | CITS2200 |   68 |
| 15712381 | Jill Tan   | CITS1401 |   88 |
| 15712381 | Jill Tan   | CITS1402 |   68 |
+----------+------------+----------+------+
```

Projection

We want $R_1 = \pi_{SN}(R)$, so

```
INSERT INTO R1
        (SELECT DISTINCT S, N FROM R);
```

We need the `SELECT DISTINCT` to force `MySQL` to remove duplicates.

```
mysql> SELECT * FROM R1;
+----------+------------+
| S        | N          |
+----------+------------+
| 14058428 | John Smith |
| 15712381 | Jill Tan   |
+----------+------------+
```

```
INSERT INTO R2 (SELECT DISTINCT S, U, M FROM R);
```

Recovering data

Can we *recover* the original relation R from its replacements R_1 and R_2?

What happens when we *join* R_1 and R_2 matching up the attributes they have in common?

```
SELECT * FROM R1 NATURAL JOIN R2;
+----------+------------+----------+------+
| S        | N          | U        | M    |
+----------+------------+----------+------+
| 14058428 | John Smith | CITS1401 |   72 |
| 14058428 | John Smith | CITS1402 |   68 |
| 14058428 | John Smith | CITS2200 |   68 |
| 15712381 | Jill Tan   | CITS1401 |   88 |
| 15712381 | Jill Tan   | CITS1402 |   68 |
+----------+------------+----------+------+
```

For this *particular instance* of R, we can equally well store R or R_1 and R_2 and create one from the other and vice versa.

Lossless-join decomposition

If a relation R is decomposed into relations R_1, R_2 such that *for every legal instance* r of R

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$$

then the *decomposition itself* is said to be a *lossless-join* decomposition.

You can view this as a sort of *minimum requirement* for a decomposition to be acceptable.

How can a decomposition not be lossless-join?

We'll take the same relation $R = SNUM$, but this time we'll try taking

$$S_1 = SM \quad S_2 = NUM$$

and see what happens.

Remember that a trial with *one particular instance* might show that a decomposition is *not* lossless-join, but not that it is!

Project onto the two relations

```
INSERT INTO S1 (SELECT DISTINCT S, M FROM R);
INSERT INTO S2 (SELECT DISTINCT N, U, M FROM R);

SELECT * FROM S1;
+----------+------+
| S        | M    |
+----------+------+
| 14058428 |   72 |
| 14058428 |   68 |
| 15712381 |   88 |
| 15712381 |   68 |
+----------+------+
SELECT * FROM S2;
+------------+----------+------+
| N          | U        | M    |
+------------+----------+------+
| John Smith | CITS1401 |   72 |
| John Smith | CITS1402 |   68 |
| John Smith | CITS2200 |   68 |
| Jill Tan   | CITS1401 |   88 |
| Jill Tan   | CITS1402 |   68 |
+------------+----------+------+
```

Join back together

```
SELECT * FROM S1 NATURAL JOIN S2;
+------+----------+------------+----------+
| M    | S        | N          | U        |
+------+----------+------------+----------+
|   72 | 14058428 | John Smith | CITS1401 |
|   68 | 14058428 | John Smith | CITS1402 |
|   68 | 15712381 | John Smith | CITS1402 |
|   68 | 14058428 | John Smith | CITS2200 |
|   68 | 15712381 | John Smith | CITS2200 |
|   88 | 15712381 | Jill Tan   | CITS1401 |
|   68 | 14058428 | Jill Tan   | CITS1402 |
|   68 | 15712381 | Jill Tan   | CITS1402 |
+------+----------+------------+----------+
```

This is not the original instance of R, and so the decomposition into S_1 and S_2 is not suitable.

Which are lossless?

A decomposition of a relational schema R into R_1 and R_2 is lossless-join if and only if the set of attributes in $R_1 \cap R_2$ contains a *key* for R_1 or R_2.

For the example above that worked, $R_1 \cap R_2$ is the single attribute S (student number) which is a key for $R_2 = SN$ and hence the decomposition is lossless-join.

For the example that did not work, $S_1 \cap S_2 = M$ and M is not a key for either S_1 or S_2.

Other decompositions

In general, an arbitrary decomposition of a schema *will not* be lossless join.

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_1	c_3

Instance r

A	B
a_1	b_1
a_2	b_2
a_3	b_1

Instance $\pi_{AB}(r)$

B	C
b_1	c_1
b_2	c_2
b_1	c_3

Instance $\pi_{BC}(r)$

Here B is not a key for either AB or BC, so the condition for lossless join is not met.

Lossy join

Now consider the join $\pi_{AB}(r) \bowtie \pi_{BC}(r)$

A	B	C
a_1	b_1	c_1
a_1	b_1	c_3
a_2	b_2	c_2
a_3	b_1	c_3
a_3	b_1	c_1

This contains two tuples that were not in the original relation — because b_1 is associated with both a_1 and a_3 in the first relation, and c_1 and c_3 in the second.

Problems with decomposition

Some types of redundancy in (or between) relations can be resolved by decomposition.

However decomposition introduces its own problems, in particular the fact that queries over the decomposed schemas now require joins; if such queries are very common then the deterioration in performance may be more severe than the original problems due to redundancy.

To make informed decisions about whether to decompose or not requires a formal understanding about the types of redundancy and which can be resolved through decomposition — this is the theory of *functional dependencies*.

Functional dependencies

A *functional dependency* (an FD) is a generalization of the concept of a *key* in a relation.

Suppose that X and Y are two *subsets of the attributes* of a relation with the following property:

"No two tuples can be identical *on X, but* different *on Y"*

In this situation we say that X *determines* Y and write

$$X \to Y.$$

Note that an FD arises from the "business logic" underlying the database and not from *its contents* at any one time.

Keys

The obvious functional dependencies come from the *keys* of a relation.

For example, in the student-number / name relation SN we have the obvious functional dependency

$$S \rightarrow N$$

meaning that the student number determines the name of the student.

Obviously S determines S and so

$$S \rightarrow SN$$

which is just another way of saying that the student-number is a key for the whole relation.

Superkeys

A key is a minimal set of attributes that determines all of the remaining attributes of a relation.

For example, in the SNUM relation above, the pair SU is a key because the student number and unit code determine both the name and the mark, or in symbols

$$SU \to SNUM.$$

(It is clear that no legal instance of the relation can have two tuples with the same student number and unit code, but different names or marks.)

Any superset of a key is called a superkey — it determines all of the remaining attributes, but is not minimal.

Reasoning about FDs

Often some functional dependencies will be immediately obvious from the semantics[1] of a relation, while others may follow as a consequence of these initial ones.

For example, if R is a relation with FDs $A \rightarrow B$ and $B \rightarrow C$, then it follows that

$$A \rightarrow C$$

as well.

(Take two tuples with the same values for attribute A, then they must have the same values for attribute B because of the first FD, and so they must have the same values for C by the second FD.)

[1] i.e. the *meaning* of the attributes

Armstrong's Axioms

Armstrong's Axioms is a set of three rules that can be repeatedly applied to a set of FDs:

- Reflexivity: If $Y \subseteq X$ then $X \to Y$.
- Augmentation: If $X \to Y$ then $XZ \to YZ$ for any Z.
- Transitivity: If $X \to Y$ and $Y \to Z$ then $X \to Z$.

In addition there are a couple of obvious rules:

- Union: If $X \to Y$ and $X \to Z$ then $X \to YZ$.
- Decomposition: If $X \to YZ$, then $X \to Y$ and $X \to Z$.

Sound and complete

The key point about Armstrong's axioms is that they are both sound and complete. That is, if we start with a set F of FDs then:

- Repeated application of Armstrong's axioms to F generates only FDs that are consequences of F
- Any FD that is a consequence of F be obtained by repeated application of Armstrong's axioms to F.

Example

Consider a relation with attributes ABC and let

$$F = \{A \to B, B \to C\}$$

Then from transitivity we get $A \to C$, by augmentation we get $AC \to BC$ and by union we get $A \to BC$.

FDs that arise from reflexivity such as

$$AB \to B$$

are known as *trivial* dependencies.

Closure

Given a set X of attributes from some relation R, the *closure* X^+ is the set of all attributes that are determined by X. In symbols

$$X^+ = \{A : X \to A\}$$

The following properties hold:

- $X \subseteq X^+$
- $X^+ = R$ if and only if X is a superkey
- $X \to X^+$ is a sort of "maximal" FD

Example

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BDE^+? (We use BDE as shorthand for $\{B, D, E\}$.)

Example

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BDE^+? (We use BDE as shorthand for $\{B, D, E\}$.)

Example

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BDE^+? (We use BDE as shorthand for $\{B, D, E\}$.)

- So far we know that BDE^+ contains BDE

Example

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BDE^+? (We use BDE as shorthand for $\{B, D, E\}$.)

- So far we know that BDE^+ contains BDE
- From the FD $D \to C$, BDE^+ contains $BCDE$

Example

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BDE^+? (We use BDE as shorthand for $\{B, D, E\}$.)

- So far we know that BDE^+ contains BDE
- From the FD $D \to C$, BDE^+ contains $BCDE$
- From the FD $CE \to A$, BDE^+ contains $ABCDE$

Therefore BDE^+ is the whole relation and BDE is a superkey for R.

Example 2

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BD^+?

Example 2

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BD^+?

Example 2

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BD^+?

- So far we know that BD^+ contains BD

Example 2

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

What is BD^+?

- So far we know that BD^+ contains BD
- From the FD $D \to C$, BD^+ contains BCD

Example 2

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \rightarrow C, CE \rightarrow A, D \rightarrow A, AE \rightarrow D$$

What is BD^+?

- So far we know that BD^+ contains BD
- From the FD $D \rightarrow C$, BD^+ contains BCD
- From the FD $D \rightarrow A$, BD^+ contains $ABCD$

Therefore BD^+ is not a superkey because E is not determined by it.
(In fact, E is not on the right-hand side of *any* FD and so it must be in every key and super key.)

Example 3

Let $R = (A, B, C, D, E, F)$ be a relation schema with the following FDS:

$$C \to F, E \to A, EC \to D, A \to B$$

Which of the following is a key for R?

- CD
- EC
- AE
- AC

Example 3

Let $R = (A, B, C, D, E, F)$ be a relation schema with the following FDS:

$$C \to F, E \to A, EC \to D, A \to B$$

Which of the following is a key for R?

- CD
- EC
- AE
- AC

The answer is EC, using $C \to F$, $E \to A$, then $A \to B$ and finally $EC \to D$.

Normal forms

There is a *hierarchy* of *normal forms*

- First normal form
  Entries in the table are *scalar values* — sets of values not allowed
- Second normal form
  1NF *plus* every non-key attribute depends on the whole key (only relevant where there are composite keys)
- Third normal form
  2NF *plus* no transitive dependencies
- Boyce-Codd normal form
  3NF *plus* conditions to be discussed

BCNF

A relational schema is in *Boyce-Codd normal form* if for every functional dependency $X \to A$ (where X is a subset of the attributes and A is a single attribute) either

- $A \in X$ (that is, $X \to A$ is a trivial FD), or
- X is a superkey.

In other words, the only functional dependencies are either the trivial ones (which always hold) or ones based on the keys of the relation.

If a relational schema is in BCNF, then there is no redundancy within the relations.

No redundancy in BCNF

Loosely speaking, a relational schema in BCNF is already in its "leanest" possible form — each attribute is determined by the key(s) alone so nothing that is stored can be deduced from a smaller amount of information.

The student number / name / unit code / mark relation *SNUM* from last lecture is *not* in BCNF because there is a functional dependency

$$S \to N$$

but S is *not* a superkey.

BCNF decomposition

Suppose a relation R is *not* in BCNF. Then there must be some functional dependency

$$X \to Y$$

where X is not a superkey; this is a *Boyce-Codd violation*.

We can assume that $Y \cap X = \emptyset$ so Y only contains some "extra" attributes determined by X, not the ones in X itself.

Then the relation can be decomposed into the two relations

$$R_1 = R - Y \qquad R_2 = XY$$

As X is a key for R_2, this is a lossless-join decomposition.

Example

Suppose $R(A, B, C, D, E)$ has the following FDs

$$D \to C, CE \to A, D \to A, AE \to D$$

Now $D^+ = ACD$ and so

$$D \to ACD$$

is a BCNF violation.

So by putting $X = D$ and $Y = AC$ the decomposition rule says to decompose into

$$R_1 = BDE \quad R_2 = ACD$$

Back to SNUM

In our SNUM example,

$$S \to N$$

is a BCNF violation.

So the rule says to decompose into

$$R_1 = SUM \qquad R_2 = SN$$

which is exactly the decomposition we found earlier.

BCNF decomposition cont.

If either R_1 or R_2 is not in BCNF then the process can be continued, by decomposing them in the same fashion.

By continually decomposing any relation not in BCNF into smaller relations, we must eventually end up with a collection of relations that are in BCNF.

Therefore any initial schema can be decomposed into BCNF.

Is BCNF the ultimate answer?

Definitely not!

There are various problems associated with decomposing into BCNF

- While it reduces redundancy, queries may take considerably longer, as they now involve possibly complicated joins
- Some FDs that hold on the original relation can no longer be enforced using the decomposed relations

There are numerous other "normal forms" each of which has an associated decomposition theory, and choosing whether and how to decompose is an important task for the database designer.