
Introduction to Databases, Fall 2005
IT University of Copenhagen

Lecture 4: Normalization

September 19, 2005

Lecturer: Rasmus Pagh

— Today's lecture —

- Motivation: Anomalies in relations.
- Decomposing relations.
- Keys and functional dependencies (FDs).
- Boyce-Codd normal form (BCNF).
- Working with FDs.
- Attribute value redundancy.

Next: Anomalies and decomposition

— Redundancy in a relation —

Redundant (i.e., “unnecessary”) information occurs in a relation if the same fact is repeated in several different tuples.

[Figure 3.21 shown on slide]. Problem: Particular values for *title* and *year* can be repeated in several tuples, and each time we will record the same *length*, *filmtype*, and *studio name*.

One obvious problem with redundant information is that we use more memory than is necessary. Redundancy is an example of an **anomaly** of the relation schema.

— Other kinds of anomalies —

The other principal kinds of unwanted anomalies are:

- **Update anomalies.** Occur when it is possible to change a fact in one tuple but leave the same fact unchanged in another. (E.g., the length of Star Wars in the `Movies` relation.)
- **Deletion anomalies.** Occur when deleting a tuple (recording some fact) may delete another fact from the database. (E.g., information on a movie in the `Movies` relation.)

[Figure 3.21 shown on slide]

Ideally, we would like relation schemas that do not allow anomalies.

Normalization is a process that can often be used to arrive at such schemas.

— Decomposing relations —

The anomalies in the example we saw can be eliminated by splitting (or **decomposing**) the relation schema

```
Movies(title, year, length, filmType, studioName, starName)
```

into two relation schemas

```
Movies1(title, year, length, filmType, studioName)
```

```
Movies2(title, year, starName)
```

[Figure 3.22 and 3.23 shown on slide]

— Decomposition and projection —

The relation instances for Movies1 and Movies2 were found by **projection** of Movies onto their attributes. In SQL, Movies2 could be computed as follows:

```
SELECT title, year, starName
FROM Movies
```

This is a *general rule* when decomposing: The decomposed relation instances are found by projection of the original relation instance.

— Recombining relations —

We would like the decomposed relations to contain the same information as the original relation. In particular, we should be able to recombine them to recover the original relation.

Recombining can be done by **joining** the relations on attributes of the same name (this is called a **natural join**). [Figure 3.28 shown on slide]

Example: In SQL we can compute Movies as follows:

```
SELECT *  
FROM Movies1, Movies2  
WHERE Movies1.title = Movies2.title AND  
Movies1.year = Movies2.year
```

— Problem session (5 minutes) —

Consider these two attempts at decomposing Movies into relations

MoviesA(length, filmType, studioName)

MoviesB(title, year, starName)

MoviesX(title, year, length, filmType)

MoviesY(title, year, starName, studioName)

What are the problems with these attempts?

Next: Keys and functional dependencies

— Keys of a relation —

A **key** for a relation is a set of its attributes that satisfy:

- **Uniqueness.** The values of the attributes uniquely identify a tuple.
- **Minimality.** No proper subset of the attributes has the uniqueness property.

If uniqueness is satisfied (but not necessarily minimality) the attributes are said to form a **superkey**.

Examples: [Figure 3.21 shown on slide]

- {Title, year, starName} is a key for the Movies relation.
- {Title, year, starName, length} is a superkey, but not a key, for the Movies relation.
- {Title, year} is not a superkey (or key) for the Movies relation.

— Key terminology —

Confusingly, what we call a superkey in the context of relations corresponds to what we called a key in the context of E/R models.

The key/superkey terminology is *not* well-established, i.e., other terms may be used in other texts/contexts.

Keys consisting of more than one attribute are sometimes called **composite**.

— A superkey suffices for recombining —

Suppose we decompose a relation R into two relations R_1 and R_2 with common attributes B_1, B_2, \dots, B_m . What is required to be able to recover R ?

[Figure 3.28 shown on slide]

Recombining works when (but not only when):

$\{B_1, B_2, \dots, B_m\}$ is a superkey for R_1 or R_2 .

— Functional dependencies cause anomalies —

When values of attribute B can be derived from the attributes A_1, \dots, A_n we say that B is **functionally dependent** on A_1, \dots, A_n . This is written as follows:

$$A_1 A_2 \dots A_n \rightarrow B$$

[Figure 3.16 shown on slide]

Example: Movies has the functional dependency (FD)

$$title \ year \rightarrow \ length$$

but *not* the FD

$$title \ year \rightarrow \ starName$$

This is in fact *the very reason* for the anomalies we saw!

— Unavoidable functional dependency —

Functional dependency on a superkey

Whenever we see the attribute values of some (super)key $\{A_1, \dots, A_n\}$, we can uniquely identify the tuple from which the values come.

In particular, we can determine the value of any other attribute B in the relation, so we unavoidably have the FD

$$A_1 A_2 \dots A_n \rightarrow B$$

Trivial functional dependency

Also, we can always determine the value of attribute A_i from the value of attribute A_i . So we unavoidably have the FD

$$A_1 A_2 \dots A_n \rightarrow A_i$$

— Problem session (10 minutes) —

Consider a relation containing an inventory record:

Inventory(part, warehouse, quantity, warehouse-address)

- What are the keys of the relation?
- What are the avoidable functional dependencies?
- Can you suggest a way of decomposing the relation to eliminate the avoidable functional dependencies?

Next: Boyce-Codd normal form (BCNF)

— Boyce-Codd normal form (BCNF) —

A **normal form** is a criterion on a relation schema.

A relation is in **Boyce-Codd normal form** (BCNF) if there are only unavoidable functional dependencies among its attributes.

Example: `Movies` has the functional dependency

$$title \quad year \quad \rightarrow \quad length$$

which is *not* unavoidable because it is nontrivial and $\{title, year\}$ is not a (super)key. Thus, `Movies` is not in BCNF.

— Examples of relations in BCNF —

The relations of our decomposition:

Movies1(title, year, length, filmType, studioName)

Movies2(title, year, starName)

are in BCNF. The only nontrivial nonreducible FDs are (all in Movies1):

$$title \ year \ \rightarrow \ length$$
$$title \ year \ \rightarrow \ filmType$$
$$title \ year \ \rightarrow \ studioName$$

and they are unavoidable since $\{title, year\}$ is a key for both relations.

— Writing functional dependencies —

Reducing FDs: Whenever we can reduce the number of attributes when writing an FD we do so. For example, `Movies1` has the FDs

$$title \ year \ filmType \ \rightarrow \ length$$
$$title \ year \ studioName \ \rightarrow \ length$$

which can both be reduced to

$$title \ year \ \rightarrow \ length$$

Combining FDs: Whenever several FDs have the same left hand side we combine them. For example, the three FDs we saw for `Movies` can be written succinctly as:

$$title \ year \ \rightarrow \ length \ filmType \ studioName$$

— Decomposing a relation into BCNF —

Suppose we have a relation R which is not in BCNF. Then there is an FD

$$A_1A_2 \dots A_n \rightarrow B_1B_2 \dots B_m$$

which is not unavoidable.

To eliminate the FD we split R into two relations: [Figure 3.24]

- One with all attributes of R except B_1, B_2, \dots, B_m .
- One with attributes $A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m$.

If any of the resulting relations is not in BCNF, the process is repeated.

Note: A_1, A_2, \dots, A_n is a superkey for the second relation – therefore we can recover R as the natural join of the two relations.

— BCNF decomposition example —

Recall the relation `Movies` with schema

`Movies(title, year, length, filmType, studioName, starName)`

It has the following FD, which is not unavoidable:

$$\textit{title year} \rightarrow \textit{length filmType studioName}$$

Thus the decomposition yields the following relations (both in BCNF):

`Movies1(title, year, length, filmType, studioName)`

`Movies2(title, year, starName)`

— BCNF decomposition example 2 —

Movies(title, year, length, filmType, studioName, starName)
could also have been decomposed by using the following FDs, one by one:

$$\textit{title year} \rightarrow \textit{length}$$

$$\textit{title year} \rightarrow \textit{filmType}$$

$$\textit{title year} \rightarrow \textit{studioName}$$

Then the decomposition yields the following relations (all in BCNF):

Movies1(title, year, length), Movies2(title, year, filmType),
Movies3(title, year, studioName), Movies4(title, year, starName)

To avoid too many relations, as in this example, you should generally use *maximal* FDs where it is not possible to add attributes on the right side.

— Problem session (5 minutes) —

Consider the following relation with information on movie studios:

MovieStudio(title, year, length, filmType, studioName, studioAddr)

Argue that the relation is not in BCNF, and find a decomposition into BCNF.

Next: Working with FDs

Finding all FDs

The following systematic method (not described in GUW) can be used to find all FDs in a relation.

Suppose we have a relation with attributes A_1, A_2, \dots, A_n and B . To find all FDs with B on the right hand side, first determine whether

$$A_1 A_2 \dots A_n \rightarrow B$$

is an FD:

- If not, there are no FDs with (a subset of) A_1, A_2, \dots, A_n on the left hand side and B on the right hand side, and we may stop.
- Otherwise repeat the same procedure with the n candidate FDs we get by taking one attribute away from the left hand side of the above FD.

— Reducing the number of cases —

To find all *nonreducible* FDs (which suffices for normalization), we may use the following method, which often looks at much fewer candidate FDs:

First determine whether

$$A_1A_2 \dots A_n \rightarrow B$$

is an FD:

- If not, we may stop.
- Otherwise, if we can spot an FD from a subset of A_1, A_2, \dots, A_n to B , repeat the procedure with each of the attributes of the subset removed from the left hand side of the above FD.
- Otherwise repeat the procedure with all n candidate FDs we get by taking one attribute away from the left hand side of the above FD.

— Example of finding all nonreducible FDs —

Movies(title, year, length, filmType, studioName, starName)

To find all nonreducible FDs with studioName on the right hand side, we consider the following candidate FDs:

• title year length filmType starName → studioName (FD)

• title year starName → studioName (spotted FD)

• year length filmType starName → studioName (not FD)

• title length filmType starName → studioName (not FD)

• title year length filmType → studioName (FD)

• title year → studioName (spotted FD)

• year length filmType → studioName (not FD)

• title length filmType → studioName (not FD)

— Implied FDs —

Another way of working with FDs, described in the book, is to start with a number of known FDs, and use them to derive others.

Example: If $A \rightarrow B$ and $B \rightarrow C$ then we can deduce that $A \rightarrow C$ (*transitive rule*).

The book presents a systematic way of finding all FDs implied by a set of FDs. This method is called the *closure algorithm*. (You do not need to know about this algorithm beyond what is mentioned on this slide.)

Notation: For a set of attributes S , we denote by S^+ the largest set T of attributes such that $S \rightarrow T$ is an FD (possibly an implied FD found by the closure algorithm).

Next: Attribute value redundancy (not in book)

— Attribute value redundancy —

A kind of redundancy, which is different from functional dependence, is redundancy in attribute values.

Example: The string `Star Wars` was repeated many times to designate the movie. Longer strings would make the problem even more obvious.

This kind of redundancy could also cause update anomalies.

Example: Suppose the working title `Lucky Luke Skywalker` had to be changed in the whole database to `Star Wars`.

— Reducing attribute value redundancy —

A way of reducing this redundancy is to use (or introduce) a short key value for each string, and put strings in a separate relation like

```
MovieNames(key,name)
```

Whether this is a good idea depends on the number of occurrences and other factors such as the need for efficiency.

— Most important points in this lecture —

As a minimum, you should after this week:

- Understand the significance of normalization.
- Be able to determine whether a relation is in Boyce Codd normal form.
- Be able to split a relation in several relations to achieve any of the normal forms.
- Know how to recombine normalized relations in SQL.

Next time

Next week we finish the study of normalization:

- 4th normal form
- Some observations on normalization.

Then we will go through one or two cases of database design, including

- E/R design
- Conversion to relation schemas
- Normalization