Databasesystemer, forår 2006
IT Universitetet i København

Forelæsning 6: Normalisering

9. marts 2006

Forelæser: Rasmus Pagh
Today’s lecture

- Anomalies in relations.
- Functional dependencies.
- Normal forms.
- Boyce-Codd normal form (BCNF).
- 3rd normal form.
- A little bit on 4th normal form.
- Attribute value redundancy.
In this lecture I will assume that you remember:

- Key concepts of the relational data model:
  - Relation
  - Attributes
  - Relation schema
  - Relation instance

- Key concepts in SQL
  - Projection
  - Join
Next: Anomalies in relations
Redundant (i.e., “unnecessary”) information occurs in a relation if the same fact is repeated in several different tuples.

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.
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.)

- **Insertion anomalies.** The “dual” of deletion anomalies.

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.
As we will see, anomalies are a sign that we tried to encode several, unrelated types of facts into a single relation.

Thus, normalization can help improving the design such that the unrelatedness of these facts are captured by the database schema (and E-R diagram).
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)
The relation instances for Movies1 and Movies2 were found by **projection** of Movies onto their attributes. In SQL, Movies2 could be computed as follows:

```sql
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.
We need the decomposed relations to contain the same information as the original relation. In particular, we must be able to recombine them to recover the original relation.

If decomposition is done properly, recombining can be done by **joining** the relations on attributes of the same name (this is called a **natural join**).

**Example:** In SQL we can compute Movies as follows:

```sql
SELECT *
FROM Movies1, Movies2
WHERE Movies1.title = Movies2.title AND Movies1.year = Movies2.year
```
**Candidate keys of a relation**

A **candidate 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:**

- \{Title, year, starName\} is a candidate key for the Movies relation.
- \{Title, year, starName, length\} is a superkey, but not a candidate key, for the Movies relation.
- \{Title, year\} does not satisfy uniqueness for Movies.
Candidate keys vs primary keys

Note that the concept of a candidate key is defined with respect to the relation (schema), and not with respect to any particular instance of the relation.

The primary key of a relation in a DBMS should be a candidate key, but there could be several candidate keys to choose from. When talking about normalization, it is irrelevant which key is chosen as primary key.
Next: Functional dependencies
When values of attribute $B$ can be derived from the attributes $A_1, \ldots, A_n$ we say that $B$ is **functionally dependent** on $A_1, \ldots, A_n$. This is written as follows:

$$A_1 A_2 \ldots A_n \rightarrow B$$

**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 candidate key

If the attributes of some candidate key is included in \( \{A_1, \ldots, A_n\} \), these attributes 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

\[
A_1 A_2 \ldots A_n \rightarrow B
\]

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

\[
A_1 A_2 \ldots A_n \rightarrow A_i
\]
Next: Normal Forms
A normal form is a criterion on a relation schema. A relation schema is in first normal form (1NF) if it specifies that all tuples contain the same number of atomic attribute values.

In pure relational DBMSs it is only possible to have 1NF relations schemas. A relation schema is in second normal form (2NF) if it is in 1NF and there are no functional dependencies with a proper subset of a candidate key on the left hand side.

Example: Movies has the functional dependency

\[ title \quad year \rightarrow length \]

where \{title, year\} is a subset of the candidate key \{title, year, starnname\}. Thus, Movies is not in 2NF.
A relation schema is in **Boyce-Codd normal form (BCNF)** if there are only unavoidable functional dependencies among its attributes.

**Example:** Movies has the functional dependency

\[
\text{title \ year} \rightarrow \text{length}
\]

which is *not* unavoidable because it is nontrivial and \{\text{title, year}\} is not a superkey. 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):

\[
\begin{align*}
title \text{ year} & \rightarrow length \\
title \text{ year} & \rightarrow \text{filmType} \\
title \text{ year} & \rightarrow \text{studioName}
\end{align*}
\]

and they are unavoidable since \{title, year\} is a candidate 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

\[ \text{title year filmType } \rightarrow \text{ length} \]
\[ \text{title year studioName } \rightarrow \text{ length} \]

which can both be reduced to

\[ \text{title year } \rightarrow \text{ length} \]

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:

\[ \text{title year } \rightarrow \text{ length filmType studioName} \]
Suppose we have a relation R which is not in BCNF. Then there is an FD

\[ A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m \]

which is not unavoidable.

To eliminate the FD we split R into two relations:

- One with all attributes of R except \( B_1, B_2, \ldots, B_m \).
- One with attributes \( A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m \).

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

**Note:** \( A_1, A_2, \ldots, A_n \) is a superkey for the second relation – therefore we can recover R as the natural join of the two relations.
Recall the relation Movies with schema

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

It has the following FD, which is not unavoidable:

\[
\text{title \ year} \rightarrow \text{length \ filmType \ studioName}
\]

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

Movies1(title, year, length, filmType, studioName)
Movies2(title, year, starName)
Movies(title, year, length, filmType, studioName, starName) could also have been decomposed by using the following FDs, one by one:

\[
\begin{align*}
\text{title} & \quad \text{year} \quad \rightarrow \quad \text{length} \\
\text{title} & \quad \text{year} \quad \rightarrow \quad \text{filmType} \\
\text{title} & \quad \text{year} \quad \rightarrow \quad \text{studioName}
\end{align*}
\]

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.
Consider a relation containing an inventory record:

\text{Inventory}(\text{part, warehouse, quantity, warehouse-address})

- What are the candidate keys of the relation?
- What are the avoidable functional dependencies?
- Perform a decomposition into BCNF.
Next: 3rd normal form
Consider the relation with schema \text{Bookings}(title, theater, city)

Under certain assumptions, it has the FD \text{theater} \rightarrow \text{city}, but \text{theater} is not a candidate key. The BCNF decomposition yields relation schemas \text{Bookings}1(\text{theater, city}) and \text{Bookings}2(\text{theater, title}).

These schemas and their FDs allow, e.g., the relation instances:

<table>
<thead>
<tr>
<th>theater</th>
<th>city</th>
<th>theater</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guild</td>
<td>Menlo Park</td>
<td>Guild</td>
<td>The net</td>
</tr>
<tr>
<td>Park</td>
<td>Menlo Park</td>
<td>Park</td>
<td>The net</td>
</tr>
</tbody>
</table>

which violate the presumed FD \text{title city} \rightarrow \text{theater}.

Thus, there are implicit dependencies between values in different relations. 
\textit{We cannot check FDs separately in each relation to see such a dependency.}
As we just saw, decomposition can result in a relational database schema where a functional dependency “disappeared”.

The problem in the previous example arose because we decomposed according to the FD $theater \rightarrow city$, where $city$ is part of a candidate key for the Bookings relation. Thus we ended up splitting the candidate key \{city, theater\}.

This problem of FDs that are not preserved never arises if we do not decompose in this case.
**Third normal form**

We have motivated the following normal form which never splits a candidate keys of the original relation:

A relation schema is in **3rd normal form** (3NF) if any functional dependency among its attributes is either unavoidable, or has a member of some candidate key on the right hand side.

In words: A relation is in 3NF if there are no unavoidable functional dependencies among non-candidate key attributes.

In the terminology of MDM, 3NF eliminates all *partial* and *transitive* functional dependencies.
Whether it is a good idea to stop decomposition when third normal form is reached depends on the specific scenario.

- Mostly, 3NF and BCNF coincide, so there is nothing to consider.
- If not, the redundancy in tuples in 3NF should be weighed against the fact that some FD is difficult to check/maintain in BCNF.

**Example:**
In the Bookings example, we might want to make the DBMS check that to every title and city, there is at most one theater. For the BCNF decomposed relations, this would involve a query on Bookings1 for every change of Bookings2, and vice versa.
Next: A little bit on 4th normal form.
Redundancy in BCNF relations

Boyce-Codd normal form eliminates redundancy in each tuple, but may leave redundancy among tuples in a relation.

This typically happens if two many-many relationships (or in general: a combination of two types of facts) are represented in one relation.

Example:

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>title</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>C. Fisher</td>
<td>123 Maple St.</td>
<td>Hollywood</td>
<td>Star Wars</td>
<td>1977</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>123 Maple St.</td>
<td>Hollywood</td>
<td>Empire Strikes Back</td>
<td>1980</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>123 Maple St.</td>
<td>Hollywood</td>
<td>Return of the Jedi</td>
<td>1983</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>5 Locust Ln.</td>
<td>Malibu</td>
<td>Star Wars</td>
<td>1977</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>5 Locust Ln.</td>
<td>Malibu</td>
<td>Empire Strikes Back</td>
<td>1980</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>5 Locust Ln.</td>
<td>Malibu</td>
<td>Return of the Jedi</td>
<td>1983</td>
</tr>
</tbody>
</table>
--- Curing it with NULL values? ---

Then what about something like one of these:

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>title</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>C. Fisher</td>
<td>123 Maple St.</td>
<td>Hollywood</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>5 Locust Ln.</td>
<td>Malibu</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>NULL</td>
<td>NULL</td>
<td>Star Wars</td>
<td>1977</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>NULL</td>
<td>NULL</td>
<td>Empire Strikes Back</td>
<td>1980</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>NULL</td>
<td>NULL</td>
<td>Return of the Jedi</td>
<td>1983</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>title</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>C. Fisher</td>
<td>123 Maple St.</td>
<td>Hollywood</td>
<td>Star Wars</td>
<td>1977</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>5 Locust Ln.</td>
<td>Malibu</td>
<td>Empire Strikes Back</td>
<td>1980</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>NULL</td>
<td>NULL</td>
<td>Return of the Jedi</td>
<td>1983</td>
</tr>
</tbody>
</table>
A better idea is to eliminate redundancy by decomposing StarsIn as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>C. Fisher</td>
<td>123 Maple St.</td>
<td>Hollywood</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>5 Locust Ln.</td>
<td>Malibu</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>title</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>C. Fisher</td>
<td>Star Wars</td>
<td>1977</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>Empire Strikes Back</td>
<td>1980</td>
</tr>
<tr>
<td>C. Fisher</td>
<td>Return of the Jedi</td>
<td>1983</td>
</tr>
</tbody>
</table>
Roughly speaking, a relation is in 4th normal form if it cannot be meaningfully decomposed into two relations.

**Example:** StarsIn is not in 4th normal form, since it can be decomposed, as we have just shown.
Next: Some observations on normalization
Inclusion among normal forms:

Any relation in 4NF is also in BCNF.
Any relation in BCNF is also in 3NF.

Properties of normal forms:

A “higher” normal form has less redundancy, but may not preserve functional and multivalued dependencies.
How should normal forms be used?

The various normal forms may be seen as *guidelines* for designing a good relation schema. Some complexities that arise are:

- Should we split candidate keys, introducing dependencies between relations (in 3NF we do not)?
- What is the effect of decomposition on performance? (More on this later.)

If one chooses a relation schema with avoidable functional dependencies, one should be aware of the extra effort that is needed to avoid anomalies.
Normalization does not remove all kinds of redundancy. An example of this 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*. 

---

**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.
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 or 3rd 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.
- Understand in what situations multivalued dependencies arise, and what should be done about them.