Databases - Normalization I
This lecture introduces normal forms, decomposition and normalization.

We will explore problems that arise from poorly designed schema, and introduce "decomposition".
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

A number of 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 the marks are kept in the following format:

<table>
<thead>
<tr>
<th>Student Number</th>
<th>Name</th>
<th>Unit Code</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>14058428</td>
<td>John Smith</td>
<td>CITS 3240</td>
<td>72</td>
</tr>
<tr>
<td>14058428</td>
<td>John Smith</td>
<td>CITS 1200</td>
<td>68</td>
</tr>
<tr>
<td>14058428</td>
<td>John Smith</td>
<td>CITS 2200</td>
<td>77</td>
</tr>
<tr>
<td>15712381</td>
<td>Jill Tan</td>
<td>CITS 1200</td>
<td>88</td>
</tr>
<tr>
<td>15712381</td>
<td>Jill Tan</td>
<td>CITS 2200</td>
<td>82</td>
</tr>
</tbody>
</table>

Then this table contains *redundant data*, because the student’s name is needlessly repeated.

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Level</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>10</td>
<td>25.00</td>
</tr>
<tr>
<td>Jones</td>
<td>8</td>
<td>20.50</td>
</tr>
<tr>
<td>Tan</td>
<td>10</td>
<td>25.00</td>
</tr>
<tr>
<td>White</td>
<td>9</td>
<td>22.00</td>
</tr>
</tbody>
</table>

This table contains the employee data, but also the association between level of an employee and the rate they are paid.
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 $20.50* is also lost.

In this situation the right approach is to keep a *separate table* that relates levels with hourly rates, and to remove the “rate” information from the employee table.

<table>
<thead>
<tr>
<th>Level</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>20.50</td>
</tr>
<tr>
<td>9</td>
<td>22.00</td>
</tr>
<tr>
<td>10</td>
<td>25.00</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Student Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>14058428</td>
<td>John Smith</td>
</tr>
<tr>
<td>15712381</td>
<td>Jill Tan</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student Number</th>
<th>Unit Code</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>14058428</td>
<td>CITS 3240</td>
<td>72</td>
</tr>
<tr>
<td>14058428</td>
<td>CITS 1200</td>
<td>68</td>
</tr>
<tr>
<td>14058428</td>
<td>CITS 2200</td>
<td>77</td>
</tr>
<tr>
<td>15712381</td>
<td>CITS 1200</td>
<td>88</td>
</tr>
<tr>
<td>15712381</td>
<td>CITS 2200</td>
<td>82</td>
</tr>
</tbody>
</table>

What is the algorithm for deciding how to split a relation?
Decomposition

Both of these examples were solved by replacing the original redundant table by two tables each containing a subset of the original fields. This leads to the following definition:

A decomposition of a relational schema $R$ consists of replacing the relational schema by two (or more) relational schemas each containing a subset of the attributes of $R$ and that together contain all of the attributes of $R$.

Data stored under the original schema is projected onto each of the schemas in the decomposition.
Decomposition - formally

Given a relation $R(A_1, A_2, \ldots, A_n)$ we may decompose $R$ into the relations $S(B_1, B_2, \ldots, B_m)$ and $T(C_1, C_2, \ldots, C_k)$ such that:

1. $\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots, C_k\}$

2. $S = \pi_{B_1, B_2, \ldots, B_m}(R)$

3. $T = \pi_{C_1, C_2, \ldots, C_k}(R)$
Example

Suppose that $R$ is the original “Student Number / Name / Unit Code / Mark” schema above — we’ll abbreviate this to

$$R = SNUM$$

($S =$ Student Number, $N =$ Name, $U =$ Unit Code, $M =$ Mark).

Then the decomposition suggested above would decompose $R$ into

$$R_1 = SUM \quad R_2 = SN$$
Storing and Recovering Data

Any data stored in the original schema is stored in the decomposed schema(s) by storing its projections onto $R_1$ and $R_2$.

We can then recover the original data by performing a join of the decomposed relations. In this situation we have the property that for every legal instance $r$ of $R$

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

In other words, the original relation is the join of the decomposed relations.
Lossless-join decomposition

The property that any instance of the original relation can be recovered as the join of the decomposed relations is called lossless-join decomposition.

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

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

\[
\begin{array}{ccc}
A & B & C \\
a_1 & b_1 & c_1 \\
a_2 & b_2 & c_2 \\
a_3 & b_1 & c_3 \\
\end{array}
\]

Instance \( r \)

\[
\begin{array}{cc}
A & B \\
a_1 & b_1 \\
a_2 & b_2 \\
a_3 & b_1 \\
\end{array}
\]

Instance \( R_1 = \pi_{AB}(r) \)

\[
\begin{array}{cc}
B & C \\
b_1 & c_1 \\
b_2 & c_2 \\
b_1 & c_3 \\
\end{array}
\]

Instance \( R_2 = \pi_{BC}(r) \)

The attributes in \( R_1 \cap R_2 \) is \( B \) which is a key for neither \( AB \) nor \( BC \), so the condition for lossless join is **not** met.
Lossy join

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_1$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
</tbody>
</table>

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 such that whenever two tuples have the same values on $X$, then they also have the same values on $Y$. In this situation we say that $X$ determines $Y$ and write

$$X \rightarrow Y.$$ 

Note that this condition must hold for *any* legal instance of the relation.
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

$$SU \rightarrow SNUM.$$  

(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.
Example

Consider a relation $R$ with four attributes $ABCD$, and suppose that the following functional dependencies hold:

$$ABC \rightarrow D \quad D \rightarrow A$$

What are the keys for $R$?

The first FD shows us that $ABC$ is a key for this relation, but this is not the only key for this relation.

The second FD shows us that $D$ determines $A$, and so $BCD$ determines all the attributes and hence is a (candidate) key.
Reasoning about FDs

Often some functional dependencies will be immediately obvious from the semantics of a relation, but 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.

The set of all FDs implied by a set $F$ of FDs is called the closure of $F$ and is denoted $F^+$. In order to consider the various normal forms it is important to be able to calculate $F^+$ given an initial set $F$ of FDs.
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 $X \rightarrow Y$.
- Augmentation: If $X \rightarrow Y$ then $XZ \rightarrow YZ$ for any $Z$.
- Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$.

We can augment these with two rules that follow directly from Armstrong’s Axioms.

- Union: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$.
- Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$. 
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 in $F^+$.
- Any FD in $F^+$ can be obtained by repeated application of Armstrong’s axioms to $F$. 
Example

Consider a relation with attributes $ABC$ and let

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

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

FDs that arise from reflexivity such as $AB \rightarrow B$ are known as *trivial* dependencies.