Week 12: Normal Forms

Database Design
Database Redundancies and Anomalies
Functional Dependencies
Entailment, Closure and Equivalence
Lossless Decompositions
The Third Normal Form (3NF)
The Boyce-Codd Normal Form (BCNF)
Normal Forms and Database Design

Logical Database Design

- We have seen how to design a relational schema by first designing an ER schema and then transforming it into a relational one.
- Now we focus on how to transform the generated relational schema into a "better" one.
- Goodness of relational schemas is defined in terms of the notion of normal form.

Normal Forms and Normalization

- A normal form is a property of a database schema.
- When a database schema is un-normalized (that is, does not satisfy the normal form), it allows redundancies of various types which can lead to anomalies and inconsistencies.
- Normal forms can serve as basis for evaluating the quality of a database schema and constitutes a useful tool for database design.
- Normalization is a procedure that transforms an un-normalized schema into a normalized one.

Examples of Redundancy

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
<th>Project</th>
<th>Budget</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>20</td>
<td>Mars</td>
<td>2</td>
<td>technician</td>
</tr>
<tr>
<td>Green</td>
<td>35</td>
<td>Jupiter</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Green</td>
<td>35</td>
<td>Venus</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
<td>Jupiter</td>
<td>15</td>
<td>manager</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
<td>Venus</td>
<td>15</td>
<td>consultant</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
<td>Mars</td>
<td>2</td>
<td>consultant</td>
</tr>
<tr>
<td>Moore</td>
<td>48</td>
<td>Mars</td>
<td>2</td>
<td>manager</td>
</tr>
<tr>
<td>Moore</td>
<td>48</td>
<td>Venus</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Kemp</td>
<td>48</td>
<td>Venus</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Kemp</td>
<td>48</td>
<td>Jupiter</td>
<td>15</td>
<td>manager</td>
</tr>
</tbody>
</table>
Anomalies
The value of the salary of an employee is repeated in every tuple where the employee is mentioned, leading to a redundancy. Redundancies lead to anomalies:
- If salary of an employee changes, we have to modify the value in all corresponding tuples (update anomaly.)
- If an employee ceases to work in projects, but stays with company, all corresponding tuples are deleted, leading to loss of information (deletion anomaly.)
- A new employee cannot be inserted in the relation until the employee is assigned to a project (insertion anomaly.)

What’s Wrong???
- We are using a single relation to represent data of very different types.
- In particular, we are using a single relation to store the following types of entities, relationships and attributes:
  - Employees and their salaries;
  - Projects and their budgets;
  - Participation of employees in projects, along with their functions.
- To set the problem on a formal footing, we introduce the notion of functional dependency (FD).

Functional Dependencies (FDs)
- Each employee has a unique salary. We represent this dependency as Employee → Salary
- and say "Salary functionally depends on Employee".
- This means that everywhere we have the same Employee attribute value, we also get the same Salary attribute value.
- Likewise, Project → Budget i.e., each project has a unique budget.

Functional Dependencies
- Given schema \( R(X) \) and non-empty subsets \( Y \) and \( Z \) of the attributes \( X \), we say that there is a functional dependency between \( Y \) and \( Z \) (\( Y → Z \)), iff for every relation instance \( r \) of \( R(X) \) and every pair of tuples \( t_1, t_2 \) of \( r \), if \( t_1.Y = t_2.Y \), then \( t_1.Z = t_2.Z \).
- A functional dependency is a statement about all allowable relations for a given schema.
- Functional dependencies have to be identified by understanding the semantics of the application.
- Given a particular relation \( r_0 \) of \( R(X) \), we can tell if a dependency holds or not; but just because it holds for \( r_0 \), doesn’t mean that it also holds for \( R(X) \)!
Looking for FDs

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
<th>Project</th>
<th>Budget</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>20</td>
<td>Mars</td>
<td>2</td>
<td>technician</td>
</tr>
<tr>
<td>Green</td>
<td>35</td>
<td>Jupiter</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Green</td>
<td>35</td>
<td>Venus</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
<td>Venus</td>
<td>15</td>
<td>manager</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
<td>Jupiter</td>
<td>15</td>
<td>consultant</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
<td>Mars</td>
<td>2</td>
<td>consultant</td>
</tr>
<tr>
<td>Moore</td>
<td>48</td>
<td>Mars</td>
<td>2</td>
<td>manager</td>
</tr>
<tr>
<td>Moore</td>
<td>48</td>
<td>Venus</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Kemp</td>
<td>48</td>
<td>Venus</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Kemp</td>
<td>48</td>
<td>Jupiter</td>
<td>15</td>
<td>manager</td>
</tr>
</tbody>
</table>

Non-Trivial Dependencies

- A functional dependency \( Y \rightarrow Z \) is **non-trivial** if no attribute in \( Z \) appears among attributes of \( Y \), e.g.,
  - Employee \( \rightarrow \) Salary is non-trivial;
  - Employee, Project \( \rightarrow \) Project is trivial.

- Anomalies arise precisely for the attributes which are involved in (non-trivial) functional dependencies:
  - Employee \( \rightarrow \) Salary;
  - Project \( \rightarrow \) Budget.

- Moreover, note that our example includes another functional dependency:
  - Employee, Project \( \rightarrow \) Function.

Dependencies Cause Anomalies, ...Sometimes!

- The first two dependencies cause undesirable redundancies and anomalies.
- The third dependency, however, does not cause redundancies because \{Employee, Project\} constitutes a key of the relation (...and a relation cannot contain two tuples with the same values for the key attributes.)

**Dependencies on keys are OK, other dependencies are not!**

Another Example

<table>
<thead>
<tr>
<th>SI#</th>
<th>Name</th>
<th>Address</th>
<th>Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>{biking, hiking}</td>
</tr>
</tbody>
</table>

Redundancy

This is NOT a relation

ER Model

Relational Model

Redundancy
How Do We Eliminate Redundancy?

- **Decomposition**: Use two relations to store Person information:
  - Person1 (SI#, Name, Address)
  - Hobbies (SI#, Hobby)
- The decomposition is more general: people with hobbies can now be described independently of their name and address.
- **No update anomalies**:
  - Name and address stored once;
  - A hobby can be separately supplied or deleted;
  - We can represent persons with no hobbies.

Superkey Constraints

- A *superkey constraint* is a special functional dependency: Let K be a set of attributes of R, and U the set of all attributes of R. Then K is a superkey iff the functional dependency K → U is satisfied in R.
  - E.g., SI# → SI#, Name, Address (for a Person relation)
- A *key* is a minimal superkey, i.e., for each X ⊂ K, X is not a superkey
  - SI#, Hobby → SI#, Name, Address, Hobby but
  - SI# → SI#, Name, Address, Hobby
  - Hobby → SI#, Name, Address, Hobby
- A *key attribute* is an attribute that is part of a key.

More Examples

- **Address → PostalCode**
  - DCS’s postal code is M5S 3H5
- **Author, Title, Edition → PublicationDate**
  - Ramakrishnan, et al., Database Management Systems, 3rd publication date is 2003
- **CourseID → ExamDate, ExamTime**
  - CSC343’s exam date is December 18, starting at 7pm

When are FDs "Equivalent"?

- Sometimes functional dependencies (FDs) seem to be saying the same thing, e.g., Addr → PostalCode,Str# vs Addr → PostalCode, Addr → Str#
- Another example
  - Addr → PostalCode, PostalCode → Province vs Addr → PostalCode, PostalCode → Province
  - Addr → Province
- When are two sets of FDs equivalent? How do we "infer" new FDs from given FDs?
Entailment, Closure, Equivalence

- If $F$ is a set of FDs on schema $R$ and $f$ is another FD on $R$, then $F$ entails $f$ (written $F \models f$) if every instance $r$ of $R$ that satisfies every FD in $F$ also satisfies $f$.
  - Example: $F = \{A \rightarrow B, B \rightarrow C\}$ and $f$ is $A \rightarrow C$
    - If $\text{Phone#} \rightarrow \text{Address}$ and $\text{Address} \rightarrow \text{ZipCode}$, then $\text{Phone#} \rightarrow \text{ZipCode}$
- The closure of $F$, denoted $F^+$, is the set of all FDs entailed by $F$.
- $F$ and $G$ are equivalent if $F$ entails $G$ and $G$ entails $F$.

Armstrong’s Axioms for FDs

- This is the syntactic way of computing/testing semantic properties of FDs
  - Reflexivity: $Y \subseteq X \models X \rightarrow Y$ (trivial FD)
    - e.g., $\models \text{Name}, \text{Address} \rightarrow \text{Name}$
  - Augmentation: $X \rightarrow Y \models XZ \rightarrow YZ$
    - e.g., $\text{Address} \rightarrow \text{ZipCode} \models \text{Address,Name} \rightarrow \text{ZipCode, Name}$
  - Transitivity: $X \rightarrow Y, Y \rightarrow Z \models X \rightarrow Z$
    - e.g., $\text{Phone#} \rightarrow \text{Address}, \text{Address} \rightarrow \text{ZipCode} \models \text{Phone#} \rightarrow \text{ZipCode}$

How Do We Compute Entailment?

- Satisfaction, entailment, and equivalence are semantic concepts – defined in terms of the "meaning" of relations in the "real world."
- How to check if $F$ entails $f$, $F$ and $G$ are equivalent?
  - Apply the respective definitions for all possible relation instances for a schema $R$ ...
  - Find algorithmic, syntactic ways to compute these notions.
- Note: The syntactic solution must be "correct" with respect to the semantic definitions.
- Correctness has two aspects: soundness and completeness – see later.

Soundness

- Theorem: $F \models f$ implies $F \models f$
- In words: If FD $f: X \rightarrow Y$ can be derived from a set of FDs $F$ using the axioms, then $f$ holds in every relation that satisfies every FD in $F$.
- Example: Given $X \rightarrow Y$ and $X \rightarrow Z$ then
  - $X \rightarrow XY$ Augmentation by $X$
  - $YX \rightarrow YZ$ Augmentation by $Y$
  - $X \rightarrow YZ$ Transitivity
- Thus, $X \rightarrow YZ$ is satisfied in every relation where both $X \rightarrow Y$ and $X \rightarrow Z$ are satisfied. We have derived the union rule for FDs.
Completeness

- Theorem: $F \models f$ implies $F \vdash f$
- In words: If $F$ entails $f$, then $f$ can be derived from $F$ using Armstrong’s axioms.
- A consequence of completeness is the following (naïve) algorithm to determining if $F$ entails $f$:

**Algorithm:** Use the axioms in all possible ways to generate $F^+$ (the set of possible FD’s is finite so this can be done) and see if $f$ is in $F^+$

---

Correctness

- The notions of **soundness** and **completeness** link the syntax (Armstrong’s axioms) with semantics, i.e., entailment defined in terms of relational instances.
- This is a precise way of saying that the algorithm for entailment based on the axioms is “correct” with respect to the definitions.

---

Decomposition Rule

- Another example of a derivation rule we can use in generating $F^+$:
- $X \rightarrow AB, AB \rightarrow A$ (refl), $X \rightarrow A$ (trans)
- So, whenever we have $X \rightarrow AB$, we can "decompose" this functional dependency to two functional dependencies $X \rightarrow A, X \rightarrow B$

---

Generating $F^+$

Thus, $AB \rightarrow BD, AB \rightarrow BCD, AB \rightarrow BCDE, AB \rightarrow E$ and $AB \rightarrow E$ are all elements of $F^+$.
Attribute Closure

- Calculating *attribute closure* leads to a more efficient way of checking entailment.
- The *attribute closure* of a set of attributes $X$ with respect to a set of FDs $F$, denoted $X^+_F$, is the set of all attributes $A$ such that $X \rightarrow A$. $X^+_F$ is not necessarily same as $X^+_G$ if $F \neq G$.
- Attribute closure and entailment:

  Algorithm: Given a set of FDs, $F$, then $X \rightarrow Y$ if and only if $Y \subseteq X^+_F$.

Computing Attribute Closure: An Example

<table>
<thead>
<tr>
<th>$F$:</th>
<th>$AB \rightarrow C$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A$</td>
</tr>
<tr>
<td></td>
<td>$AB$</td>
</tr>
<tr>
<td></td>
<td>$AC$</td>
</tr>
<tr>
<td></td>
<td>$B$</td>
</tr>
<tr>
<td></td>
<td>$D$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$X$</th>
<th>$X^+_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>${A, D, E}$</td>
</tr>
<tr>
<td>$AB$</td>
<td>${A, B, C, D, E}$</td>
</tr>
<tr>
<td>$AC$</td>
<td>${A, C, B, D, E}$</td>
</tr>
<tr>
<td>$B$</td>
<td>${B}$</td>
</tr>
<tr>
<td>$D$</td>
<td>${D, E}$</td>
</tr>
</tbody>
</table>

Is $AB \rightarrow E$ entailed by $F$? Yes
Is $D \rightarrow C$ entailed by $F$? No

Result: $X^+_F$ allows us to determine all FDs of the form $X \rightarrow Y$ entailed by $F$.

Computing the Attribute Closure $X^+_F$

```
closure := X; // since $X \subseteq X^+_F$
repeat
  old := closure;
  if there is an FD $Z \rightarrow V$ in $F$ such that $Z \subseteq closure$ and $V \subseteq closure$
  then closure := closure $\cup$ $V$
until old = closure
```

- If $T \subseteq closure$ then $X \rightarrow T$ is entailed by $F$.

Normal Forms

- Each normal form is a set of conditions on a schema that together guarantee certain properties (relating to redundancy and update anomalies).
- First normal form (1NF) is the same as the definition of relational model (relations = sets of tuples; each tuple = sequence of atomic values).
- Second normal form (2NF) - has no practical or theoretical value – won’t discuss.
- The two most used are *third normal form* (3NF) and *Boyce-Codd normal form* (BCNF).
- We will discuss in detail the 3NF.
The Third Normal Form

- A relation \( R(X) \) is in \textit{third normal form} (\( 3NF \)) if, for each (non-trivial) functional dependency \( Y \rightarrow Z \), at least one of the following is true:
  - \( Y \) contains a key \( K \) of \( R(X) \);
  - Each attribute in \( Z \) is contained in at least one key of \( R(X) \).

- \( 3NF \) does not remove all redundancies.
- \( 3NF \) decompositions founded on the notion of \textit{minimal cover}.

Decomposition into \( 3NF \): Basic Idea

- Decomposition into \( 3NF \) can proceed as follows.
  - For each functional dependency of the form \( Y \rightarrow Z \), where \( Y \) contains a subset of a key \( K \) of \( R(X) \), create a projection on all the attributes \( Y, Z \) (\( 2NF \)).
  - For each dependency of the form \( Y \rightarrow Z \), where \( Y \) doesn’t contain any key, and not all attributes of \( Z \) are key attributes, create a projection on all the attributes \( Y, Z \) (\( 3NF \)).

- The new relations only include dependencies \( Y \rightarrow Z \), where \( Y \) contains a key \( K \) of \( R(X) \), or \( Z \) contains only key attributes.

Basic Idea

- \( R(ABCD), A --> D \)
- Projection:
  - \( R1(AD), A --> D \)
  - \( R2(ABC) \)

Normalization Through Decomposition

- A relation that is not in \( 3NF \), can be replaced with one or more normalized relations using \textit{normalization}.
- We can eliminate redundancies and anomalies for the example relation

\[ \text{Emp(Employee,Salary,Project,Budget,Function)} \]

if we replace it with the three relations obtained by projections on the sets of attributes corresponding to the three functional dependencies:

- \( \text{Employee} \rightarrow \text{Salary}; \)
- \( \text{Project} \rightarrow \text{Budget}. \)
- \( \text{Employee,Project} \rightarrow \text{Function}. \)
...Start with...

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
<th>Project</th>
<th>Budget</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>20</td>
<td>Mars</td>
<td>2</td>
<td>technician</td>
</tr>
<tr>
<td>Green</td>
<td>35</td>
<td>Jupiter</td>
<td>15</td>
<td>designer</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
<td>Venus</td>
<td>15</td>
<td>manager</td>
</tr>
<tr>
<td>Kemp</td>
<td>48</td>
<td>Mars</td>
<td>2</td>
<td>manager</td>
</tr>
</tbody>
</table>

Another Example

<table>
<thead>
<tr>
<th>Employee</th>
<th>Project</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Mars</td>
<td>Chicago</td>
</tr>
<tr>
<td>Green</td>
<td>Jupiter</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Venus</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Kemp</td>
<td>Saturn</td>
<td>Birmingham</td>
</tr>
</tbody>
</table>

This relation satisfies the functional dependencies:

- Employee → Branch
- Project → Branch

Result of Normalization

<table>
<thead>
<tr>
<th>Employee</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>20</td>
</tr>
<tr>
<td>Green</td>
<td>35</td>
</tr>
<tr>
<td>Hoskins</td>
<td>55</td>
</tr>
<tr>
<td>Kemp</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mars</td>
<td>2</td>
</tr>
<tr>
<td>Jupiter</td>
<td>15</td>
</tr>
<tr>
<td>Venus</td>
<td>15</td>
</tr>
</tbody>
</table>

A Possible Decomposition

<table>
<thead>
<tr>
<th>Employee</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Chicago</td>
</tr>
<tr>
<td>Green</td>
<td>Jupiter</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Birmingham</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mars</td>
<td>Chicago</td>
</tr>
<tr>
<td>Jupiter</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Saturn</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Venus</td>
<td>Birmingham</td>
</tr>
</tbody>
</table>

The keys of new relations are lefthand sides of functional dependencies; satisfaction of 3NF is therefore guaranteed for the new relations.

...but now we don’t know each employee’s projects!
The Join of the Projections

<table>
<thead>
<tr>
<th>Employee</th>
<th>Project</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Mars</td>
<td>Chicago</td>
</tr>
<tr>
<td>Green</td>
<td>Jupiter</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Green</td>
<td>Venus</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Saturn</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Venus</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Green</td>
<td>Saturn</td>
<td>Birmingham</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Jupiter</td>
<td>Birmingham</td>
</tr>
</tbody>
</table>

The result of the join is different from the original relation.  

*We lost some information during the decomposition!*

A Condition for Lossless Decomposition

- Let \( R(X) \) be a relation schema and let \( X_1 \) and \( X_2 \) be two subsets of \( X \) such that \( X_1 \cup X_2 = X \). Also, let \( X_0 = X_1 \cap X_2 \).
- If \( R(X) \) satisfies the functional dependency \( X_0 \rightarrow X_1 \) or \( X_0 \rightarrow X_2 \), then the decomposition of \( R(X) \) on \( X_1 \) and \( X_2 \) is lossless.
- In other words, \( R(X) \) has a lossless decomposition on two relations if the set of attributes common to the relations is a superkey for at least one of the decomposed relations.

Lossless Decomposition

- The decomposition of a relation \( R(X) \) on \( X_1 \) and \( X_2 \) is lossless if for every instance \( r \) of \( R(X) \) the join of the projections of \( R \) on \( X_1 \) and \( X_2 \) is equal to \( r \) itself (that is, does not contain spurious tuples).
- Of course, it is clearly desirable to allow only lossless decompositions during normalization.

Intuition Behind the Test for Losslessness

- Suppose \( R_1 \cap R_2 \rightarrow R_2 \). Then a row of \( r_1 \) can combine with exactly one row of \( r_2 \) in the natural join (since in \( r_2 \) a particular set of values for the attributes in \( R_1 \cap R_2 \) defines a unique row)

\[
\begin{array}{c}
R_1 \cap R_2 \\
\bullet \bullet \bullet a \\
\bullet \bullet \bullet b \\
\bullet \bullet \bullet c \\
\end{array}
\quad
\begin{array}{c}
R_1 \cap R_2 \\
\bullet \bullet \bullet a \\
\bullet \bullet \bullet b \\
\bullet \bullet \bullet c \\
\end{array}
\]
A Lossless Decomposition

Notation

- Instead of saying that we have relation schema \( R(X) \) with functional dependencies \( F \), we will say that we have schema \( R = (R, F) \), where \( R \) is a set of attributes and \( F \) is a set of functional dependencies.
- The 3NF normalization problem is then to generate a set of relation schemas \( \mathcal{R} = (R_1, F_1), \ldots, (R_n, F_n) \), such that \( \mathcal{R} \) is in 3NF.

Another Example

- Schema \((R, F)\) where
  \[
  R = \{SI\#, Name, Address, Hobby\}
  \]
  \[
  F = \{SI\# \rightarrow Name, Address\}
  \]
  can be decomposed into
  \[
  R_1 = \{SI\#, Name, Address\}
  \]
  \[
  F_1 = \{SI\# \rightarrow Name, Address\}
  \]
  and
  \[
  R_2 = \{SI\#, Hobby\}
  \]
  \[
  F_2 = \{\}
  \]
since \( R_1 \cap R_2 = SI\# \) and \( SI\# \rightarrow R_1 \) the decomposition is lossless.

Another Problem...

- Assume we wish to insert a new tuple that specifies that employee Armstrong works in the Birmingham branch and participates in project Mars.
- In the original relation, this update would be identified as illegal, because it would cause a violation of the \( Project \rightarrow Branch \) dependency.
- For the decomposed relations, however, this is not possible because the two attributes \( Project \) and \( Branch \) have been moved to different relations.
Preserving Dependencies (Intuition)

- A decomposition preserves dependencies if each of the functional dependencies of the original relation schema involves attributes that appear together in one of the decomposed relation schemas.
- It is clearly desirable that a decomposition preserves dependencies because then it is possible to ensure that the decomposed schema satisfies the same constraints as the original schema.

Example

- Schema \((R, F)\) where
  \[R = \{SI\#, Name, Address, Hobby\}\]
  \[F = \{SI\# \rightarrow Name, Address\}\]
  can be decomposed into
  \[R_1 = \{SI\#, Name, Address\}\]
  \[F_1 = \{SI\# \rightarrow Name, Address\}\]
  and
  \[R_2 = \{SI\#, Hobby\}\]
  \[F_2 = \{\}\]\n- Since \(F = F_1 \cup F_2\) the decomposition is dependency preserving.

Another Example

- Schema: \((ABC; F)\), \(F = \{A \rightarrow B, B \rightarrow C, C \rightarrow B\}\)
- Decomposition:
  - \((AC, F_1), F_1 = \{A \rightarrow C\}\)
    - [Note: \(A \rightarrow C \not\in F\), but in \(F^+\)]
  - \((BC, F_2), F_2 = \{B \rightarrow C, C \rightarrow B\}\)

- \(A \rightarrow B \not\in (F_1 \cup F_2)\), but \(A \rightarrow B \in (F_1 \cup F_2)^+\).
  - So \(F^+ = (F_1 \cup F_2)^+\) and thus the decomposition is still dependency preserving.

Dependency Preservation

- If \(f\) is a FD in \(F\), but \(f\) is not in \(F_1 \cup F_2\), there are two possibilities:
  - \(f \in (F_1 \cup F_2)^+\)
    - If the constraints in \(F_1\) and \(F_2\) are maintained, \(f\) will be maintained automatically.
    - \(f \in (F_1 \cup F_2)^+\)
    - \(f\) can be checked only by first taking the join of \(r_1\) and \(r_2\). ...This is costly...
Desirable Qualities for Decompositions

Decompositions should always satisfy the properties of lossless decomposition and dependency preservation:

- **Lossless decomposition** ensures that the information in the original relation can be accurately reconstructed based on the information represented in the decomposed relations.
- **Dependency preservation** ensures that the decomposed relations have the same capacity to represent the integrity constraints as the original relations and therefore to reveal illegal updates.

Minimal Cover

- A **minimal cover** for a set of dependencies $F$ is a set of dependencies $U$ such that:
  - $U$ is equivalent to $F$ (i.e., $F^+ = U^+$)
  - All FDs in $U$ have the form $X \rightarrow A$ where $A$ is a single attribute
  - It is not possible to make $U$ smaller (while preserving equivalence) by deleting an FD
  - Deleting an attribute from an FD (its LHS)
- FDs and attributes that can be deleted in this way are called redundant.

Computing the Minimal Cover

Example: $F = \{ABH \rightarrow CK, A \rightarrow D, C \rightarrow E, BGH \rightarrow L, L \rightarrow AD, E \rightarrow L, BH \rightarrow E\}$

- **Step 1**: Make RHS of each FD into a single attribute: Use decomposition rule for FDs.
  - Example: $L \rightarrow AD$ replaced by $L \rightarrow A, L \rightarrow D$;
  - $ABH \rightarrow CK$ by $ABH \rightarrow C, ABH \rightarrow K$.
- **Step 2**: Eliminate redundant attributes from LHS: If B is a single attribute and FD $XB \rightarrow A \in F$, $X \rightarrow A$ is entailed by $F$, then $B$ is unnecessary.
  - e.g., Can an attribute be deleted from $ABH \rightarrow C$? Compute $ABH^+_F, AH^+_F, BH^+_F$; Since $C \in (BH)^+_F$, $BH \rightarrow C$ is entailed by $F$ and $A$ is redundant in $ABH \rightarrow C$.

Computing the Minimal Cover (cont’d)

- **Step 3**: Delete redundant FDs from $F$: If $F - \{f\}$ entails $f$, then $f$ is redundant; if $f$ is $X \rightarrow A$ then check if $A \in X^+_F - \{f\}$.
  - e.g., $BGH \rightarrow L$ is entailed by $E \rightarrow L, BH \rightarrow E$; so it is redundant.
- Note: The order of steps 2, 3 can’t be interchanged!! See textbook for a counterexample.

$F_1 = \{ABH \rightarrow C, ABH \rightarrow K, A \rightarrow D, C \rightarrow E, BGH \rightarrow L, L \rightarrow A, L \rightarrow D, E \rightarrow L, BH \rightarrow E\}$

$F_2 = \{BH \rightarrow C, BH \rightarrow K, A \rightarrow D, C \rightarrow E, BH \rightarrow L, L \rightarrow A, L \rightarrow D, E \rightarrow L, BH \rightarrow E\}$

$F_3 = \{BH \rightarrow C, BH \rightarrow K, A \rightarrow D, C \rightarrow E, L \rightarrow A, E \rightarrow L\}$
Synthesizing a 3NF Schema

Starting with a schema \( R = (R, F) \):

**Step 1:** Compute minimal cover \( U \) of \( F \). The decomposition is based on \( U \), but since \( U^+ = F^+ \) the same functional dependencies will hold.

- A minimal cover for \( F = \{ABH \rightarrow CK, A \rightarrow D, C \rightarrow E, BGH \rightarrow L, L \rightarrow AD, E \rightarrow L, BH \rightarrow E\} \)
  - Therefore \( U = \{BH \rightarrow C, BH \rightarrow K, A \rightarrow D, C \rightarrow E, L \rightarrow A, E \rightarrow L\} \)

**Step 2:** Partition \( U \) into sets \( U_1, U_2, \ldots U_n \) such that the LHS of all elements of \( U_i \) are the same:

- \( U_1 = \{BH \rightarrow C, BH \rightarrow K\}, U_2 = \{A \rightarrow D\}, U_3 = \{C \rightarrow E\}, U_4 = \{L \rightarrow A\}, U_5 = \{E \rightarrow L\} \)

**Step 3:** For each \( U_i \) form schema \( R_i = (R_i, U_i) \), where \( R_i \) is the set of all attributes mentioned in \( U_i \).

- Each FD of \( U \) will be in some \( R_i \). Hence the decomposition is *dependency preserving*:
  - \( R_1 = (BHCK; BH \rightarrow C, BH \rightarrow K) \),
  - \( R_2 = (AD; A \rightarrow D) \),
  - \( R_3 = (CE; C \rightarrow E) \),
  - \( R_4 = (AL; L \rightarrow A) \),
  - \( R_5 = (EL; E \rightarrow L) \)

**Step 4:** If no \( R_i \) is a superkey of \( R \), add schema \( R_0 = (R_0, \{\}) \) where \( R_0 \) is a key of \( R \).

- \( R_0 = (BGH, \{\}); R_0 \) might be needed when not all attributes are contained in \( R_1 \cup R_2 \cup \ldots \cup R_n \).
  - A missing attribute \( A \) must be part of all keys (since it’s not in any FD of \( U \), deriving a key constraint from \( U \) involves the augmentation axiom);
  - \( R_0 \) might be needed even if all attributes are accounted for in \( R_1 \cup R_2 \cup \ldots \cup R_n \).
Synthesizing ... Step 4 (cont'd)

- Example: \((ABCD; \{A \rightarrow B, C \rightarrow D\})\), with step 3 decomposition: \(R_1 = (AB; \{A \rightarrow B\}), R_2 = (CD; \{C \rightarrow D\})\).

**Lossy! Need to add \((AC; \{\})\), for losslessness**

- Step 4 guarantees lossless decomposition:
  \(ABCD \rightarrow decomp --> AB, ACD \rightarrow decomp --> AB, AC, CD\)

Boyce–Codd Normal Form (BCNF)

- A relation \(R(X)\) is in **Boyce–Codd Normal Form** if for every non-trivial functional dependency \(Y \rightarrow Z\) defined on it, \(Y\) contains a key \(K\) of \(R(X)\). That is, \(Y\) is a superkey for \(R(X)\).

- Example: \(Person1(SI\#, Name, Address)\)
  - The only FD is \(SI\# \rightarrow Name, Address\)
  - Since \(SI\#\) is a key, \(Person1\) is in BCNF

- Anomalies and redundancies, as discussed earlier, do not occur in databases with relations in BCNF.

Non-BCNF Examples

- **Person(SI\#, Name, Address, Hobby)**
  - The FD \(SI\# \rightarrow Name, Address\) does **not** satisfy conditions for BCNF since the key is \((SSN, Hobby)\)

- **HasAccount(AcctNum, ClientId, OfficeId)**
  - The FD \(AcctNum \rightarrow OfficeId\) does **not** satisfy BCNF conditions if we assume that keys for HasAccount are \((ClientId, OfficeId)\) and \((AcctNum, ClientId)\); rather than \(AcctNum\).

Normalization Drawbacks

- By limiting redundancy, normalization helps maintain consistency and saves space.

- **But** performance of querying can suffer because related information that was stored in a single relation is now distributed among several.

- Example: A join is required to get the names and grades of all students taking CS343 in 2006F.

  ```sql
  SELECT S.Name, T.Grade
  FROM Student S, Transcript T
  WHERE S.Id = T.StudId AND
  T.Crscode = 'CS343' AND T.Semester = '2006F'
  ```
Denormalization

- Tradeoff: *Judiciously* introduce redundancy to improve performance of certain queries
- Example: Add attribute Name to Transcript' 

\[
\text{SELECT T.Name, T.Grade}
\text{FROM Transcript' T}
\text{WHERE T.CrsCode = 'CS305' AND T.Semester = 'S2002'}
\]

- Join is avoided;
- If queries are asked more frequently than Transcript is modified, added redundancy might improve average performance;
- But, Transcript' is no longer in BCNF since key is (StudId,CrsCode,Semester) and StudId \(\rightarrow\) Name.

BCNF and 3NF

- The Project-Branch-Manager schema is not in BCNF, but it is in 3NF.
- In particular, the Project,Branch \(\rightarrow\) Manager dependency has as its left hand side a key, while Manager \(\rightarrow\) Branch has a unique attribute for the right hand side, which is part of the \{Project,Branch\} key.
- The 3NF is less restrictive than the BCNF and for this reason does not offer the same guarantees of quality for a relation; it has the advantage however, of always being achievable.

### 3NF Tolerates Some Redundancies!

A Revised Example

<table>
<thead>
<tr>
<th>Manager</th>
<th>Project</th>
<th>Branch</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Mars</td>
<td>Chicago</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>Jupiter</td>
<td>Birmingham</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>Mars</td>
<td>Birmingham</td>
<td>1</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Saturn</td>
<td>Birmingham</td>
<td>2</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Venus</td>
<td>Birmingham</td>
<td>2</td>
</tr>
</tbody>
</table>

Functional dependencies:
- Manager \(\rightarrow\) Branch,Division -- each manager works at one branch and manages one division;
- Branch,Division \(\rightarrow\) Manager -- for each branch and division there is a single manager;
- Project,Branch \(\rightarrow\) Division,Manager -- for each branch, a project is allocated to a single division and has a sole manager responsible.
A Good Decomposition

<table>
<thead>
<tr>
<th>Manager</th>
<th>Branch</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>Chicago</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>Birmingham</td>
<td>1</td>
</tr>
<tr>
<td>Hoskins</td>
<td>Birmingham</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project</th>
<th>Branch</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mars</td>
<td>Chicago</td>
<td>1</td>
</tr>
<tr>
<td>Jupiter</td>
<td>Birmingham</td>
<td>1</td>
</tr>
<tr>
<td>Mars</td>
<td>Birmingham</td>
<td>1</td>
</tr>
<tr>
<td>Saturn</td>
<td>Birmingham</td>
<td>2</td>
</tr>
<tr>
<td>Venus</td>
<td>Birmingham</td>
<td>2</td>
</tr>
</tbody>
</table>

- Note: The first relation has a second key {Branch, Division}.
- The decomposition is in 3NF but not in BCNF; moreover, it is lossless and dependencies are preserved.
- This example demonstrates that BCNF is too strong a condition to impose on a relational schema.

Database Design and Normalization

- The theory of normalization can be used as a basis for quality control operations on schemas, during both conceptual and logical design.
- Analysis of the relations obtained during the logical design phase can identify places where the conceptual design was inaccurate: such a validation of the design is usually relatively easy.
- Normalization can also be used during conceptual design for quality control of each element of a conceptual schema (entity or relationship).

Analysis of an Entity

- The functional dependency SupplierCode \(\rightarrow\) Supplier, Address holds here: all properties of a supplier are identified by its SupplierCode.
- The entity violates 3NF since this dependency has a left hand side that does not contain the identifier and a right hand side made up of attributes that are not part of the key.

Decomposing Product

- Supplier is (or should be) an independent entity, with its own attributes (code, surname and address).
- If Product and Supplier are distinct entities, they should be linked through a relationship.
- Since there is a functional dependency from Code to SupplierCode, we are sure that each product has at most one supplier (maximum cardinality 1).
- Since there is no dependency from SupplierCode to Code, we have an unrestricted maximum cardinality (N) for Supplier in the relationship.
Decomposing Product

This decomposition satisfies fundamental properties:

- It is a lossless decomposition, because of one-to-many relationship that allows us to reconstruct the values of the attributes of the original entity;
- Moreover, it preserves dependencies because each dependency is embedded in one of the entities or can be reconstructed from them.

Analysis of a Relationship

Now we show how to analyze n-ary relationships for n≥3, in order to determine whether they should be decomposed.

Consider

Some Functional Dependencies

- Student → DegreeProgramme (each student is enrolled in one degree programme)
- Student → Professor (each student writes a thesis under the supervision of a single professor)
- Professor → Department (each professor is associated with a single department and the students under her supervision are students in that department)

The (unique) key of the relationship is Student (given a student, the degree programme, the professor and the department are identified uniquely)

The third FD causes a violation of 3NF.

Decomposing Thesis

The following is a decomposition of Thesis where the two decomposed relationships are both in 3NF(also in BCNF)
More Observations...

- The relationship Thesis is in 3NF, because its key is made up of the Student entity, and its dependencies all have this entity on the left hand side.
- However, not all students write theses, therefore not all students have supervisors.
- From a normal form point of view, this is not a problem.
- However, our conceptual schema should reflect the fact that being in a degree programme and having a supervisor are independent facts.

Another Decomposition

![Diagram of database schema showing relationships between entities such as Professor, Thesis, Student, Department, and Degree Programme.](image)