Functional Dependencies and Normalization for Relational Databases

Functional Dependencies

Semantics of the Relation Attributes

GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance.

- Attributes of different entities should not be mixed in the same relation.
- Only foreign keys should be used to refer to other entities.
- Entity and relationship attributes should be kept apart as much as possible.

Bottom Line: Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.
Redundant Information in Tuples and Update Anomalies

- Mixing attributes of multiple entities may cause problems
- Information is stored redundantly wasting storage
- Problems with update anomalies
  - Insertion anomalies
  - Deletion anomalies
  - Modification anomalies

Update Anomalies – Insertion Anomalies

- It may not be possible to store some information unless some other information is stored as well.

Update Anomalies – Deletion Anomalies

- It may not be possible to delete some information without losing some other information as well.
Update Anomalies – Modification Anomalies

- If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

Guideline to Redundant Information in Tuples and Update Anomalies

- **GUIDELINE 2**: Design a schema that does not suffer from update anomalies. If there are any present, then note them so that applications can be made to take them into account.

Null Values in Tuples

- **GUIDELINE 3**: Relations should be designed such that their tuples will have as few NULL values as possible.
  - Attributes that are NULL frequently could be placed in separate relations (with the primary key)
  - Reasons for NULL:
    - attribute not applicable or invalid
    - attribute value unknown (may exist)
    - value known to exist, but unavailable
Spurious Tuples

GUIDELINE 4: The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations.
- Bad designs for a relational database may result in erroneous results for certain JOIN operations
- The “lossless join” property is used to guarantee meaningful results for join operations

The Evils of Redundancy

- Redundancy is at the root of several problems associated with relational schemas:
  - redundant storage, update anomalies
- Integrity constraints, in particular functional dependencies, can be used to identify schemas with such problems and to suggest refinements.
- Main refinement technique: decomposition
- Decomposition should be used judiciously:
  - Is there a reason to decompose a relation?
  - What problems does decomposition cause?

Functional Dependencies

- Functional dependencies (FDs) are used to specify formal measures of the ‘goodness’ of relational designs
- FDs and keys are used to define normal forms for relations
- FDs are constraints that are derived from the meaning and interrelationships of the data attributes
- A set of attributes X functionally determines a set of attributes Y if the value of X determines a unique value for Y, denoted as X \( \rightarrow \) Y.
Functional Dependencies (Cont)

- $X \rightarrow Y$ holds if whenever two tuples have the same value for $X$, they must have the same value for $Y$ ($X$ and $Y$ are sets of attributes)
- $X \rightarrow Y$ specifies a constraint on all relation instances
  - Must be identified based on semantics of applications.
  - Given some allowable instance $tI$ of $R$, we can check if it violates some FD $f$, but we cannot tell if $f$ holds over $R$!

Examples of FD Constraints

- SSN determines employee name
  
  SSN $\rightarrow$ ENAME

- project number determines project name and location
  
  PNUMBER $\rightarrow$ {PNAME, PLOCATION}

- ssn and project number determines the hours per week that the employee works on the project
  
  {SSN, PNUMBER} $\rightarrow$ HOURS

Examples of FD Constraints (Cont)

- An FD is a property of the attributes in the schema
- The constraint must hold on every relation instance
- If $K$ is a key of $R$, then $K$ functionally determines all attributes in $R$ (although we never have two distinct tuples with $t1[K]=t2[K]$)

<table>
<thead>
<tr>
<th>S</th>
<th>N</th>
<th>L</th>
<th>R</th>
<th>W</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-22-3666</td>
<td>Attehoo</td>
<td>18</td>
<td>10</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>231-31-3988</td>
<td>Smith</td>
<td>22</td>
<td>10</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>131-34-3650</td>
<td>Smith</td>
<td>35</td>
<td>5</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>434-26-3571</td>
<td>Gudha</td>
<td>35</td>
<td>5</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>012-67-4124</td>
<td>Mayua</td>
<td>35</td>
<td>10</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>
Inference Rules for FDs

- Given some FDs, we can infer additional FDs.
- An FD \( f \) is implied by a set of FDs \( F \) if \( f \) holds whenever all the FDs in \( F \) hold.
  - \( F^+ = \text{closure of } F \) is the set of all FDs that are implied by \( F \).

Armstrong's inference rules: (\( X,Y,Z \) are sets of attributes)

- IR1. (Reflexivity) If \( Y \subseteq X \), then \( X \rightarrow Y \).
- IR2. (Augmentation) If \( X \rightarrow Y \), then \( XZ \rightarrow YZ \).
- IR3. (Transitivity) If \( X \rightarrow Y \) and \( Y \rightarrow Z \), then \( X \rightarrow Z \).

IR1, IR2, IR3 form a sound and complete set of inference rules.

Some additional inference rules that are useful:

- IR4. (Decomposition) If \( X \rightarrow YZ \), then \( X \rightarrow Y \) and \( X \rightarrow Z \).
- IR5. (Union) If \( X \rightarrow Y \) and \( X \rightarrow Z \), then \( X \rightarrow YZ \).
- IR6. (Pseudotransitivity) If \( X \rightarrow Y \) and \( WY \rightarrow Z \), then \( WX \rightarrow Z \).

The last three inference rules, as well as any other inference rules, can be deduced from IR1, IR2, and IR3 (completeness property).

Inference Rules for FDs (Cont)

Closure of a set \( F \) of FDs is the set \( F^+ \) of all FDs that can be inferred from \( F \).
- Computing \( F^+ \) is expensive.
- Size of \( F^+ \) is exponential in number of attributes.

Typically, check if a given FD \( X \rightarrow Y \) is in \( F^+ \) (if \( X \rightarrow Y \) can be inferred from \( F \)). An efficient check:
  - Compute attribute closure of \( X \) (denoted as \( X^+ \)) wrt \( F \):
    - \( X^+ \): Set of all attributes \( A \) such that \( X \rightarrow A \) can be inferred.
    - \( X^+ \) can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in \( F \) (linear time algorithm).
  - Check if \( Y \) is in \( X^+ \):
    - If yes, then \( X \rightarrow Y \) is in \( F^+ \).
Compute Attribute Closure

Comp_Attr_closure(X)
{
  closure=X;
  repeat until there is no change:
  if there is an FD U → V in F such that U closure
  then set closure = closure ∪ V
  return(closure)
}

Equivalence of Sets of FDs

- Two sets of FDs $F$ and $G$ are equivalent if:
  - every FD in $F$ can be inferred from $G$, and
  - every FD in $G$ can be inferred from $F$
- $F$ covers $G$ if every FD in $G$ can be inferred from $F$
  (i.e., if $G^+ F^+$)
  - calculate $X^+$ wrt $F$ for each $X → Y$ in $G$
  - check whether $X^+$ includes the attributes in $Y$
  - if yes for every FD in $G$, then $F$ covers $G$
- $F$ and $G$ are equivalent if $F$ covers $G$ and $G$ covers $F$

Minimal Cover of FDs

- A set of FDs $F$ is minimal if it satisfies the following conditions:
  (1) Every dependency in $F$ has a single attribute for its right-hand side.
  (2) We cannot remove any attribute from an FD in $F$ and still have a set of FDs that is equivalent to $F$.
  (3) We cannot remove any FD in $F$ and still have a set of FDs that is equivalent to $F$. 
Minimal Cover of FDs (Cont)

- $F$ is minimal in two respects:
  - Every dependency is as small as possible
  - Every dependency in $F$ is required in order for the closure to be equal to $F^+$
- Every set of FDs has an equivalent minimal set
- There can be several equivalent minimal sets

Compute Minimal Set

Method:
1. Rewrite every FD so that every right side is a single attribute
2. Remove redundant attribute from left side for every FD
   e.g., $XY \rightarrow A$, check if $Y$ is redundant
   a. compute $X^+$ wrt $F$
   b. if $X^+$ contains $A$, than $Y$ is redundant
3. Remove redundant FDs
   e.g., check if $X \rightarrow Y$ is redundant,
   a. compute $X^+$ wrt $F' = F - \{X \rightarrow Y\}$,
   b. if $X^+$ contains $Y$, $X \rightarrow Y$ is redundant

Normalization for Relational Databases
Normalization of Relations

- **Normalization**: The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations.
- **Normal form**: Conditions using keys and FDs of a relation to certify whether a relation schema is in a particular normal form.
- **Role of FDs in detecting redundancy**:
  - Consider a relation R with 3 attributes, ABC.
  - No FDs hold: no redundancy.
  - Given $A \rightarrow B$: several tuples could have the same A value, and if so, they'll all have the same B value!

First Normal Form

- A relation is in **first normal form (1NF)** if every field contains only atomic values.
- Disallows composite attributes, multivalued attributes; attributes whose values for an individual tuple are non-atomic.
- Considered to be part of the definition of relation.

Normalization into 1NF

<table>
<thead>
<tr>
<th>Student ID</th>
<th>Name</th>
<th>Email</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456</td>
<td>John Doe</td>
<td><a href="mailto:john@example.com">john@example.com</a></td>
<td>555-1234</td>
</tr>
<tr>
<td>789012</td>
<td>Jane Smith</td>
<td><a href="mailto:jane@example.com">jane@example.com</a></td>
<td>555-5678</td>
</tr>
<tr>
<td>345678</td>
<td>Michael Brown</td>
<td><a href="mailto:mike@example.com">mike@example.com</a></td>
<td>555-9012</td>
</tr>
</tbody>
</table>
Normalizing nested relations into 1NF

Second Normal Form - Preliminaries

- A superkey of a relation schema \( R \) is a set of attributes \( S \) with the property that no two tuples \( t_1 \) and \( t_2 \) in any legal relation state have \( t_1[S] = t_2[S] \).
- A key \( K \) is a minimal superkey: the removal of any attribute from \( K \) will cause \( K \) not to be a superkey any more.
- If a relation has more than one key, each is called a candidate key.
- One of the candidate keys is arbitrarily designated to be the primary key.

Second Normal Form – Preliminaries (Cont)

- Prime attribute: an attribute that is a member of some candidate key.
- Full functional dependency: \( Y \rightarrow Z \) is a full FD if removal of any attribute from \( Y \) means the FD does not hold any more.
- Partial functional dependency: \( Y \rightarrow Z \) is a partial FD if some attribute can be removed from \( Y \) and the FD still holds.
Second Normal Form (Cont)

A relation schema $R$ is in **second normal form (2NF)** if every non-prime attribute $A$ in $R$ is fully functionally dependent on the primary key.

Normalizing into 2NF

Third Normal Form

**Transitive functional dependency**: a FD $X \rightarrow Z$ that can be derived from two FDs $X \rightarrow Y$ and $Y \rightarrow Z$, also $Y$ is neither a candidate key nor a subset of any key of $R$.

A relation schema $R$ is in **third normal form (3NF)** if it is in 2NF and no non-prime attribute $A$ in $R$ is transitively dependent on the primary key.

- **NOTE**: In $X \rightarrow Y$ and $Y \rightarrow Z$, with $X$ as the primary key, we consider this a problem only if $Y$ is not a candidate key. When $Y$ is a candidate key, there is no problem with the transitive dependency.
Normalizing into 3NF

General Normal Form Definitions - For Multiple Keys

- The following more general definitions take into account relations with multiple candidate keys
- A relation schema R is in \(2NF\) if every non-prime attribute \(A\) in \(R\) is fully functionally dependent on every key of \(R\).
- A relation schema \(R\) is in \(3NF\) if whenever a FD \(X \rightarrow A\) holds in \(R\), then either:
  - (a) \(A \in X\) (a trivial FD), or
  - (b) \(X\) is a superkey of \(R\), or
  - (c) \(A\) is a prime attribute of \(R\)

Boyce-Codd Normal Form

- A relation schema \(R\) is in Boyce-Codd Normal Form (BCNF) if whenever an FD \(X \rightarrow A\) holds in \(R\), then:
  - (a) \(A \in X\) (a trivial FD), or
  - (b) \(X\) is a superkey of \(R\)
- Each normal form is strictly stronger than the previous one
  - Every 2NF relation is in 1NF
  - Every 3NF relation is in 2NF
  - Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF
- The goal is to have each relation in BCNF or 3NF
Normalizing into BCNF

A Relation TEACH that is in 3NF but not in BCNF

3NF vs. BCNF

• If 3NF violated by \( X \rightarrow A \), one of the following holds:
  - \( X \) is a subset of some key \( K \)
    • We store \( (X, A) \) pairs redundantly.
  - \( X \) is not a proper subset of any key.
    • There is a chain of FDs \( K \rightarrow X \rightarrow A \), which means that we cannot associate an \( X \) value with a \( K \) value unless we also associate an \( A \) value with an \( X \) value.
  - But: even if relation is in 3NF, these problems could arise.
    • e.g., \( \text{Reserves(SBDC)} \), \( S \rightarrow C \), \( C \rightarrow S \) is in 3NF, but for each reservation of sailor \( S \), same \( (S, C) \) pair is stored.
• Thus, 3NF is indeed a compromise relative to BCNF.
Decomposition of a Relation

- A decomposition of $R$ consists of replacing $R$ by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of $R$ (and no attributes that do not appear in $R$).
  - Every attribute of $R$ appears as an attribute of one of the new relations.

- Intuitively, decomposing $R$ means we will store instances of the relation schemes produced by the decomposition, instead of instances of $R$.

Problems with Decompositions

- There are three potential problems to consider:
  - P1: Some queries become more expensive.
    - E.g., How much did sailor Joe earn? (salary = W*H)
  - P2: Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
  - P3: Checking some dependencies may require joining the instances of the decomposed relations.

- **Tradeoff:** Must consider these issues vs. redundancy.

Lossless Join Decomposition

- Decomposition of $R$ into $X$ and $Y$ is lossless-join wrt a set of FDs $F$ if, for every instance $r$ that satisfies $F$:
  - $X(Y(r)) \rightarrow A(r) = r$

- It is always true that $X(Y(r)) \rightarrow A(r)$
  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.

- Definition extended to decomposition into 3 or more relations in a straightforward way.

- It is essential that all decompositions used to deal with redundancy be lossless! (Avoids P2)
Testing Binary Lossless Join Decomposition

- The decomposition of R into X and Y is lossless-join wrt F if and only if F+ contains:
  - X \cap Y \rightarrow X, or
  - X \cap Y \rightarrow Y
- In particular, the decomposition of R into XY and R-Y is lossless-join if X\rightarrow Y holds over R.

- Observation1: If X\rightarrow Y holds in R and X\cap Y is empty, the decomposition of R into R-Y and XY is lossless.

- Observation2: R is lossless-join decomposed into R1 and R2, and R1 is lossless-join decomposed into R11 and R12, then the decomposition of R into R11, R12 and R2 is lossless.

Testing Lossless Join Decomposition

Input: R(A1, ..., An), FDs F, decomposition D={R1, ..., Rm}
1. Create a m×n matrix S with one row i for each Ri and one column j for each attribute Aj.
2. If Aj\in Ri, set S(i,j)=aj; otherwise, S(i,j)=bij.
3. Repeat the following loop until a complete loop execution results in no changes to S
   a. For each X\rightarrow Y in F:
   b. For all rows in S that have the same symbols in the columns corresponding to attributes in X:
      {Make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows:
       If any of the rows has an ‘a’ for the column, set the other rows to that same ‘a’ in the column
       If no ‘a’ exists of the attribute in any of the rows, choose one of the ‘b’ that appears in one of the rows for the attribute and set the other rows to that same ‘b’ in the column}
4. If a row is made up entirely of ‘a’, then D is lossless; otherwise, it does not.

Dependency Preserving Decomposition

- Dependency preserving decomposition:
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. (Avoids P3)

- Projection of set of FDs F: If R is decomposed into X, ..., Y, then projection of F onto X (denoted F\rightarrow \theta) is the set of FDs U\rightarrow V in F\theta such that U, V are in X.
Dependency Preserving Decomposition (Cont)

- Decomposition of $R$ into $X$ and $Y$ is dependency preserving if $(F_X \cup F_Y)'' = F''$
  
- **Important** to consider $F''$, not $F$
  
  - ABC, A $\rightarrow$ B, B $\rightarrow$ C, C $\rightarrow$ A, decomposed into AB and BC.
  
  - Is this dependency preserving? Is C $\rightarrow$ A preserved?

- Dependency preserving does not imply lossless join.
  
  - ABC, A $\rightarrow$ B, decomposed into AB and BC.
  
  - And vice-versa!

Normalization into BCNF and 3NF

- Converting relations to BCNF
  
  - Possible to obtain a lossless-join decomposition
  
  - May be no dependency-preserving decomposition

- Converting relations to 3NF
  
  - There is always a dependency-preserving and lossless-join decomposition

Decomposition into BCNF

- Consider relation $R$ with FDs $F$. If $X \rightarrow Y$ violates BCNF, decompose $R$ into $XY$ and $R - Y$.
  
  - Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.

- In general, several dependencies may cause violation of BCNF. The order in which we “deal with” them could lead to very different sets of relations!
BCNF and Dependency Preserving

- In general, there may not be a dependency preserving decomposition into BCNF.
  - CSZ, CS → Z, Z → C
  - Can’t decompose while preserving CS → Z; not in BCNF.
- Similarly, decomposition of CSJDQV into SDP, JS and CIDQV is not dependency preserving (wrt the JP → C, SD → P and J → S).
  - However, it is a lossless join decomposition.
  - In this case, adding IPC to the collection of relations gives us a dependency preserving decomposition.
  - IPC tuples stored only for checking FD (Redundancy)

Decomposition into 3NF

- Obviously, the algorithm for lossless join decomposed into BCNF can be used to obtain a lossless join decomposed into 3NF (typically, can stop earlier).
- To ensure dependency preservation, one idea:
  - If X → Y is not preserved, add relation XY.
- Refinement: Instead of the given set of FDs F, use a minimal cover for F.