Integrity Constraints and Functional Dependency
What are Integrity Constraints?

- Predicates that ensure the changes made to a DB do not result in loss of data consistency
- Does *semantic check*
- Can be declared by *Create Table* or *Alter Table* command
- While inserting a record, IC’s are checked and error is thrown in case it is violated
Types of Integrity Constraints

- There are primarily 4 kinds of IC’s:
  1. Key Constraints (1 table)
  2. Attribute Constraints (1 table)
  3. Referential Integrity Constraints (2 tables)
  4. Global Constraints (n tables)
Key Constraints

- Attributes declared as key columns should be unique to the table.

- This can be mainly executed through two ways:
  a. Primary Key (declared with the `primary key` keyword)
  b. Candidate Key (declared with the `unique` keyword)

- A relation can have only 1 primary key but multiple candidate keys.
Key Constraints (contd.)

- **Primary Key Example:**
  
  ```
  CREATE TABLE Persons(ID int PRIMARY KEY, 
    LastName varchar(255) NOT NULL, 
    FirstName varchar(255) NOT NULL, 
    Age int);
  OR
  CREATE TABLE Persons(ID int, 
    LastName varchar(255) NOT NULL, 
    FirstName varchar(255) NOT NULL, or 
    Age int, 
    PRIMARY KEY(ID));
  ```

- **Candidate Key Example:**
  
  ```
  CREATE TABLE customer(ID CHAR(19) PRIMARY KEY, 
    cname CHAR(15), 
    address CHAR(30), 
    city CHAR(10), 
    UNIQUE (cname, address, city));
  ```
Attribute Constraints

- Used to attach constraints to attributes in a relation
- It can be done in 3 ways:
  - a. Using *NOT NULL*:
    No null values can be inserted into the attribute that has this IC
    For eg: `CREATE TABLE Persons (ID int,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int);`

    First name and last name cannot have null values
b. Using `check<predicate>`:

- Used to check certain user-defined conditions specified in the predicate

Eg: `CREATE TABLE Persons (ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int,
    CHECK (Age>=18));`

This checks if the age of the person is at least 18
c. By using *Domains*:

- Allows user to define data types that associate constraints with attributes and hence avoid redundancy
- \textit{Domain Constraint} = \textit{data type} + \textit{Constraints (NOT NULL / UNIQUE / PRIMARY KEY / FOREIGN KEY / CHECK)}

Eg: You want to create a table “bank_account” with “account_type” field having value either “Checking” or “Saving”:

```
CREATE DOMAIN account_type char(12)
CONSTRAINT acc_type_test
    check(value in("Checking","Saving"));

CREATE TABLE bank_account( acc_no INT PRIMARY KEY,
    acc_holder_name VARCHAR(30),
    acc_type account_type);
```
Referential Integrity Constraints

- Allows values associated with certain attributes to appear for certain attributes in another relation
- Foreign key in the referencing (child) table should correspond to a Primary Key in the referenced (Parent) table
- The main idea behind this is to avoid dangling tuples. This affects update/delete operation in the following ways:
  a. Insertions/updates in the child relation: A tuple cannot be inserted/updated until the same operation has been carried out in the Parent table
  b. Delete/update in the parent relation: A tuple cannot be deleted from the parent relation unless the same is deleted in the child (if there is any present)
Referential Integrity Constraints (contd.)

Resolving Dangling Tuples:

There are 3 ways to deal with dangling tuples:

a. Reject the deletion/update: Leave blank

b. Set the value in the dangling tuple to null: ON DELETE SET NULL / ON UPDATE SET NULL  
   sets ti[c]= NULL, tj[c]= NULL

c. Cascade the operation to the dangling tuple: i.e delete/update the tuple in the child/parent table as well

   ON DELETE CASCADE
   ▶ delete ti, delete tj

   ON UPDATE CASCADE
   ▶ sets ti[c], tj[c]to new Key value
Global Constraints

- Global constraints span multiple relations. They can be executed in two ways:
  a. **Single relation** (constraint spans multiple columns)
     Eg: CHECK (total= svngs+ check)
     declared in CREATE TABLE for bank relation
  b. **Multiple relations**: Create Assertions
     Eg: *Every loan has a borrower with a savings account*
     CREATE ASSERTION loan-constraint
     CHECK (NOT EXISTS
     (SELECT * FROM loan AS l
     WHERE NOT EXISTS
     (SELECT *FROM borrower AS b, depositor AS d, account AS a,
     WHERE b.cname= d.cnameAND d.acct_no= a.acct_no
     AND l.lno= b.lno))))
Functional Dependencies (FD) are used to define constraints between two attributes of the given relation.

Given a relation $R$, a set of attributes $X$ in $R$ is said to functionally determine another set of attributes $Y$, also in $R$, (written $X \rightarrow Y$) if, and only if, each $X$ value in $R$ is associated with precisely one $Y$ value in $R$. 
Uses of functional dependency

- Take the following relation schema:

  \[\text{inst dept (ID, name, salary, dept name, building, budget)}\]

  Adding or updating a record in the given relation might lead to redundancy (multiple instructors may belong to the same department)/inconsistency (all the departments should agree on the same budget value) errors which is not desirable.

- FD’s break down the relation into smaller one’s that are normalized.
Decomposition using FD

When FD’s are used for decomposition, the decomposed relations should have the following characteristics:

a. **Lossless Joins**: No information should be lost i.e if R is decomposed into R1 and R2 then the natural join should result in R and no extra records should be added.

Eg: Consider the following relation

\[ R = (\text{SSN}, \text{Name}, \text{Address}) \]

If we decompose this into

\[ R1(\text{SSN}, \text{Name}) \]

\[ R2(\text{Name}, \text{Address}) \]

and now do a natural join of R1 and R2, we get

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Joe</td>
<td>1 Pine</td>
</tr>
<tr>
<td>222</td>
<td>Alice</td>
<td>2 Oak</td>
</tr>
<tr>
<td>333</td>
<td>Alice</td>
<td>3 Pine</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Joe</td>
</tr>
<tr>
<td>222</td>
<td>Alice</td>
</tr>
<tr>
<td>333</td>
<td>Alice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>1 Pine</td>
</tr>
<tr>
<td>Alice</td>
<td>2 Oak</td>
</tr>
<tr>
<td>Alice</td>
<td>3 Pine</td>
</tr>
</tbody>
</table>

We are *losing* the information for the person with SSN value 222 and 333 as we no longer know where they reside.

To fix this we can break them into

\[ R1(\text{SSN}, \text{Name}) \]

\[ R2(\text{SSN}, \text{Address}) \]

which would be "**lossless**" decomposition.
b. **Dependency Preservation**: Ensure that a single table is checked for each FD and it should not span multiple relations

- Reduces the cost of checking FD

Eg:

\[ F = \{A \rightarrow B, AB \rightarrow D, C \rightarrow D\} \]

\( R_1 = (A, B, C); R_2 = (C, D) \) is not DP as checking \( AB \rightarrow D \) would span multiple relations

Alternatively,

\( R_1 = (A, B, D); R_2 = (C, D) \) is DP as all FD checks would span only one relation at any time

c. **Redundancy avoidance**: Avoid deletion, update anomalies by ensuring the relation is in BCNF
Closures for Functional Dependencies

- For FD $F$, $F^+$ is the set of all FD’s implied by $F$
- Can be calculated in two ways:
  - a. Attribute Closures
  - b. Armstrong’s Axioms
Attribute Closures

Set of attributes that can be derived from the given attribute using FD’s

*Algorithm attr-closure (X: set of attributes)*

1. \( \text{result} \leftarrow X \)
2. repeat until stable
   - for each FD in \( F \), \( Y \rightarrow Z \), do
     - if \( Y \subseteq \text{result} \) then
       - \( \text{result} \leftarrow \text{result} \cap Z \)

Eg: Consider the following set of FD’s: \( \text{name} \rightarrow \text{color} \)
\( \text{category} \rightarrow \text{department} \)
\( \text{color, category} \rightarrow \text{price} \)

The attribute closure for \{name, category\} would be calculated as follows:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{name, category}</td>
</tr>
<tr>
<td>1</td>
<td>{name, category, color}</td>
</tr>
<tr>
<td>2</td>
<td>{name, category, color, department}</td>
</tr>
<tr>
<td>3</td>
<td>{name, category, color, department, price}</td>
</tr>
</tbody>
</table>
Armstrong’s Axioms

1. Reflexivity
   If $Y \subseteq X$ then $X \rightarrow Y$

2. Augmentation
   If $X \rightarrow Y$ then $WX \rightarrow WY$

3. Transitivity
   If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

4. Union
   If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

5. Decomposition
   If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

6. Pseudotransitivity
   If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$
Canonical Cover Algorithm (CCA)

- Given a FD set $F$, CCA finds the minimal FD set equivalent of $F$
- All the implied FD’s are removed from the $F$

**ALGORITHM canonical-cover**($X$: FD Set)

**BEGIN**

**REPEAT UNTIL STABLE**

1. Where possible, apply UNION rule ($A$’s Axioms)
2. Remove all extraneous attributes:
   a. Test if $B$ extraneous in $A \rightarrow BC$
      
      $(B$ extraneous if $(A \rightarrow B) \subseteq (F -\{A \rightarrow BC\} \cup \{A \rightarrow C\}^+) = F^+$
   b. Test if $B$ extraneous in $AB \rightarrow C$
      
      $(B$ extraneous if $(A \rightarrow C) \subseteq F^+)$

**END**
Eg: Consider the following set of FD’s:

\[ F = \{ A \rightarrow BC; B \rightarrow C; A \rightarrow B; AB \rightarrow C \} \]

**Step 1:** Take \( A \rightarrow BC \) and \( A \rightarrow B \). Can be combined to get \( A \rightarrow BC \)

Hence, \( F = \{ A \rightarrow BC; B \rightarrow C; AB \rightarrow C \} \)

**Step 2:** There is an extraneous attribute in \( AB \rightarrow C \) because even after removing it, we get the same closures. This is because \( B \rightarrow C \) is already a part of \( F \).

Hence, \( F = \{ A \rightarrow BC; B \rightarrow C \} \)

Note: In Step 2, we cannot get rid of \( A \rightarrow BC \) as that would not imply \( B \rightarrow C \) anymore.

**Step 3:** \( C \) is extraneous in \( A \rightarrow BC \) as it can be derived transitively from \( A \rightarrow B \) and \( B \rightarrow C \).

Hence, \( F = \{ A \rightarrow B; B \rightarrow C \} \)

\( F \) does not change anymore so the final canonical cover of \( F \) is

\[ F = \{ A \rightarrow B; B \rightarrow C \} \]