Constraints and Normalization

Kyle Cui
Constraints

- conditions that must be met for the relation to be valid
- four types of constraints
  - key constraints
  - attribute (and domain) constraints
  - referential integrity constraints
  - global constraints
Key Constraints

- primary & candidate keys
  - defined with PRIMARY KEY, UNIQUE respectively
  - triggers SQL checks when inserting a tuple with conflicting primary/candidate key

CREATE college (  
  student_id integer PRIMARY KEY,  
  student_name VARCHAR(128),  
  student_grad_date integer  
);
Attribute Constraints

- **Attribute constraints**: constraints on a single column/attribute
  - conditions such as **NOT NULL**, numeric ranges like `attr > 0`, etc.

```sql
CREATE college (  
  student_id integer PRIMARY KEY,
  student_name NOT NULL VARCHAR(128),
  student_grad_date NOT NULL integer CHECK(student_grad_date >= 2020 AND student_grad_date <= 2024)
);```
Domain Constraints

- extension of attribute constraints
- domain = user-defined data type (with one or more constraints!)
- example: “bank_account” with “account_type” field; field can only be “checking” or “saving”

```sql
CREATE DOMAIN account_type VARCHAR(12) (
    CONSTRAINT is_not_null
    CHECK (value NOT NULL),
    CONSTRAINT valid_account_type
    CHECK (value in("checking", "saving"))
);

CREATE TABLE bank_account(acc_no INT PRIMARY KEY,
    acc_holder_name VARCHAR(30),
    acc_type account_type);
```
Referential 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
- purpose: to avoid dangling tuples.
  - triggers SQL checks upon:
    a. insertions/updates in the child relation
    b. delete/update in the parent relation
Referential Constraints

CREATE TABLE cities (  
city varchar(80) primary key,  
location point
);

CREATE TABLE weather (  
city varchar(80) references cities(city),  
temp_lo int,  
temp_hi int,  
date date
);

- cities parent, weather child
- upon insertion or update into weather, makes sure that the city field exists in cities
- upon deletion or update in cities, updates or deletes corresponding fields in weather (cascade delete)
Global Constraints

- Constraints that the database enforces across one or more (even all) relations
- Can be very expensive!
- Single table: `CHECK(savings + expenses > 0)`
  - Enforced at a single table level and may use multiple columns
- Multiple relations:
  - Enforced on any database change/update
  - `CREATE ASSERTION constraint1 CHECK (NOT EXIST (SELECT ... ))`
  - Can select from multiple tables
Functional Dependencies

- used to define a set of constraints between two attributes of some given relation
- given distinct sets of attributes $X$ and $Y$ in some relation $R$, $X$ functionally determines $Y$ (notation: $X \rightarrow Y$) iff each $X$ value in $R$ is mapped to exactly one $Y$ value in $R$.
- example: attributes banner_id, student_name, student_birthdate
  - since each banner_id is associated with exactly one student and each student has only one birthday, 
    \texttt{banner_id} \rightarrow \texttt{student_name} and \texttt{banner_id} \rightarrow \texttt{student_birthdate}
  - but student_name does not functionally determine banner_id!
Closure

- for any set of functional dependencies (FDs) $F$, $F^+$ is called the closure
- or, the set of all functional dependencies implied by $F$
- simple examples
  - attributes banner_id, student_name, student_birthdate
    - $\text{banner_id} \rightarrow \text{student_name}$ and $\text{banner_id} \rightarrow \text{student_birthdate}$
    - thus, $\text{banner_id} \rightarrow \{\text{student_name}, \text{student_birthdate}\}$
  - attributes course_id, course_time, course_room
    - $\{\text{course_time}, \text{course_room}\} \rightarrow \text{course_id}$
    - (assuming you can’t hold two courses simultaneously in the same place!)
    - note $\text{course_time}$ or $\text{course_room}$ alone do not functionally determine $\text{course_id}$
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
   \[ if \ X \rightarrow Y \ and \ Y \rightarrow Z \ then \ X \rightarrow Z \]

derived axioms:

4. union
   \[ if \ X \rightarrow Y \ and \ X \rightarrow Z, \ then \ X \rightarrow YZ \]
   \[ \text{- important note!} \]
   \[ \text{- } A \rightarrow B \ \text{and} \ A \rightarrow C \ \text{guarantees that} \ A \rightarrow BC; \ \text{but} \]
   \[ \text{- } AB \rightarrow C \ \text{doesn’t guarantee} \ A \rightarrow B \ \text{and} \ A \rightarrow C \]

5. decomposition
   \[ if \ X \rightarrow YZ \ then \ X \rightarrow Y \ \text{and} \ X \rightarrow Z \]

6. pseudotransitivity
   \[ if \ X \rightarrow Y \ and \ WY \rightarrow Z, \ then \ WX \rightarrow Z \]
Computing the Closure

let $F$ be the set of functional dependencies; initialize $F+$ to be $\{}$

let $S$ be the set of possible attribute combinations in $R$

for each $s$ in $S$:
    compute the attribute closure $s+$ on $F$
    for each attribute $A$ in $s+$:
        add $s \rightarrow A$ to $F+$

return $F+$
Example

\[ R = (A, B, C, D) \]

\[ F = \{ A \rightarrow BC, C \rightarrow D \} \]
Attribute Closure

- set of all attributes which can be determined from an attribute set
- example: compute \( \{A, B\}^+ \) given the previous \( F = \{A \rightarrow BC, C \rightarrow D\} \)
  - use Armstrong’s axioms!
  - start by setting \( \{A, B\}^+ = \{\} \), then update the set

\[
A \rightarrow A \text{ and } B \rightarrow B \text{ from reflexivity: update } \{A, B\}^+ = \{A, B\}
\]
\[
A \rightarrow BC \text{ gives } A \rightarrow B \text{ and } A \rightarrow C: \text{ update } \{A, B\}^+ = \{A, B, C\}
\]
\[
C \rightarrow D \text{ combined with } A \rightarrow C \text{ gives } A \rightarrow D: \text{ update } \{A, B\}^+ = \{A, B, C, D\}
\]

\[\{A, B\}^+ = \{A, B, C, D\}\]
\{A\}^+ = \{A, B, C, D\} \quad \leftarrow \text{minimum candidate key}
\{B\}^+ = \{B\}
\{C\}^+ = \{C, D\}
\{D\}^+ = \{D\}
\{A, B\}^+ = \{A, B, C, D\} \quad \leftarrow \text{superkey}
\{A, C\}^+ = \{A, B, C, D\} \quad \leftarrow \text{superkey}
\{A, D\}^+ = \{A, B, C, D\} \quad \leftarrow \text{superkey}
\{B, C\}^+ = \{B, C, D\}
\{B, D\}^+ = \{B, D\}
\{C, D\}^+ = \{C, D\}
\{A, B, C\}^+ = \{A, B, C, D\} \quad \leftarrow \text{superkey}
\{A, B, D\}^+ = \{A, B, C, D\} \quad \leftarrow \text{superkey}
\{A, C, D\}^+ = \{A, B, C, D\} \quad \leftarrow \text{superkey}
\{B, C, D\}^+ = \{B, C, D\}
\{A, B, C, D\}^+ = \{A, B, C, D\} \quad \leftarrow \text{superkey}
Canonical Cover

- a minimal set of functional dependencies C which imply every FD defined in the closure of F
- in other words, remove all redundant dependencies in F+ (a set of FDs)

canonical-cover(X: FD Set)
 REPEAT UNTIL STABLE
  1. apply UNION rule whenever possible (X \rightarrow Y and X \rightarrow Z means X \rightarrow YZ)
  2. remove all extraneous attributes:
     a. Test if B extraneous in A \rightarrow BC
        \[ B \text{ extraneous if } (A \rightarrow B) \subseteq (F - \{A \rightarrow BC\} U \{A \rightarrow C\})^+ = F^+ \]
     b. Test if B extraneous in AB \rightarrow C
        \[ B \text{ extraneous if } (A \rightarrow C) \subseteq F^+ \text{ (this is an axiom)} \]
Canonical Cover Example

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

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

\text{combine } A \rightarrow B \text{ and } A \rightarrow BC, \text{ since } A \rightarrow BC \text{ contains } A \rightarrow B

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

A \rightarrow BC \text{ gives } A \rightarrow C, \text{ and } A \rightarrow C \text{ is extraneous } AB \rightarrow C

F = \{A \rightarrow BC; B \rightarrow C\}

A \rightarrow BC \text{ gives } A \rightarrow C, \text{ and } A \rightarrow C \text{ is extraneous}

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

A \rightarrow B \text{ with } B \rightarrow C \text{ implies that } A \rightarrow C, \text{ so } C \text{ is extraneous in } A \rightarrow BC
Questions?
Schema Decomposition

- breaking down a relation with 2 or more smaller relations
- motivation?
  - easier to express data constraints
  - avoid excessively large relations that can have data redundancy leading to inconsistencies
- desired properties of (good) decompositions
  - lossless joins
  - dependency preservation
  - redundancy avoidance
Joins & Lossless Joins

- breaking down a relation into smaller ones should not cause data to be lost
  - if any sort of information is lost, it is considered lossy
- if $R$ is broken down into $R_1, R_2$, then $R = R_1 \bowtie R_2$
- ex: $R = (\text{ssn}, \text{name}, \text{address})$ can be broken down into:
  
  a) $R_1 = (\text{ssn}, \text{name})$ $R_2 = (\text{name}, \text{address})$

  b) $R_1 = (\text{ssn}, \text{name})$ $R_2 = (\text{ssn}, \text{address})$

- which is lossy (if either)?
- even though the result set has more tuples, this is lossy!
- why?
<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>adam</td>
<td>add1</td>
</tr>
<tr>
<td>02</td>
<td>bob</td>
<td>add2</td>
</tr>
<tr>
<td>03</td>
<td>bob</td>
<td>add3</td>
</tr>
</tbody>
</table>

- lossless!
Dependency Preservation

- every dependency must be satisfied by at least one of the broken-up relations
- more formally, for a set of FDs $F$ over a relation $R$, assume we decompose $R$ into $R_1$ and $R_2$
  - assume $R_1$ has a set of FDs $F_1$ and $R_2$ has a set of FDs $F_2$, where $F_1$ and $F_2$ are derived from $F$
  - in a dependency preserving decomposition, if $F_1$ is satisfied in $R_1$ and $F_2$ is satisfied in $R_2$, then $F$ is satisfied in $R$
Dependency Preservation

\[ R = (\text{ssn, name, age, can\_drink}) \]
\[ F = \{\text{ssn} \rightarrow \{\text{name, age}\}, \text{age} \rightarrow \text{can\_drink}\} \]

decompose \( R \) into \( R_1 = (\text{ssn, name, age}) \), \( R_2 = (\text{age, can\_drink}) \)
then we can derive \( F_1 = \{\text{ssn} \rightarrow \{\text{name, age}\}\}, F_2 = \{\text{age} \rightarrow \text{can\_drink}\}\)
then if \( F_1 \) is true and \( F_2 \) is true, \( F \) is true

good dependency-preserving decomposition!
Dependency Preservation

\[ R = (A, B, C, D) \]
\[ F = \{ A \rightarrow B, B \rightarrow C \} \]

decompose \( R \) into \( R1 = (A, B) \), \( R2 = (A, C) \), \( R2 = (A, D) \)
then we can only derive \( F1 = \{ A \rightarrow B \} \), \( F2 = \{ A \rightarrow C \} \), \( F3 = {} \)

not dependency preserving!
Boyce-Codd Normal Form

- A relation $R$ is in Boyce-Codd Normal Form (BCNF) if $F^+$ has no $FD$ $X \rightarrow A$ such that
  - attribute $A$ and all the attributes of set $X$ appear in $R$ (all attributes from both sides of the $FD$ are in $R$)
  - $A$ not in $X$ (the $FD$ is not trivial)
  - $X$ (the left side) does not contain any candidate key of $R$
- If we can find a $FD$ that satisfies all of the above, then it is not in BCNF
Boyce-Codd Normal Form

- assume 4 attributes A, B, C, D and F = \{A \rightarrow B, B \rightarrow C\}
- is R = (A, B, C) in BCNF?
  - B \rightarrow C involves R, since B and C are both in R
  - not trivial
  - left side (B) does not contain a candidate key of R (A)
- since there exists an FD in R that satisfies all three conditions, it is not BCNF
BCNF Algorithm

1. split $R$ on some FD $X \rightarrow Y$ in $F$ into $R_1(X_1,Y_1)$
2. update $R$ by setting $R = R - \{Y\}$ (remove $Y$ from the pool of original attributes)
3. split $R$ on another FD $X_2 \rightarrow Y_2$ in $F$ into $R_2(X_2,Y_2)$
4. repeat 2-3 until every $R_j$ is in BCNF
BCNF Example Dependency Preserving

\[ R = (A, B, C, D) \]

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

\[ R_1 = (A, B, C) \quad R_2 = (A, D) \quad R_3 = (A, B) \]

trivial, so remove
BCNF Example Not DP

\[ R = (A, B, C, D) \]

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

\[ R_1 = (A, B, C) \quad R_2 = (C, A) \quad R_3 = (B, D) \]

uh-oh, codependent!
BCNF Example Not DP

\[ R = (A, B, C, D) \]

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

\[ R_1 = (\underline{A}, \underline{B}, C) \quad R_2 = (B, D) \]  

in BCNF, but missing \( C \rightarrow A \)
Boyce-Codd Normal Form

- useful because:
  - guarantees no redundancies and lossless joins!
  - but is *not* dependency preserving