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”

    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
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, $\text{banner_id} \rightarrow \text{student_name}$ and $\text{banner_id} \rightarrow \text{student_birthdate}$
  - 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
   \[\text{if } Y \subseteq X \text{ then } X \rightarrow Y\]

2. augmentation
   \[\text{if } X \rightarrow Y \text{ then } WX \rightarrow WY\]

3. transitivity
   \[\text{if } X \rightarrow Y \text{ and } Y \rightarrow Z \text{ then } X \rightarrow Z\]

Derived axioms:

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

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

6. pseudotransitivity
   \[\text{if } X \rightarrow Y \text{ and } WY \rightarrow Z, \text{ then } WX \rightarrow Z\]
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

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

\( \{A, B\}^+ = \{A, B, C, D\} \)
\{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,

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
        \( B \) extraneous if (A \(\rightarrow\) B) \(\in\) \((F - \{A \rightarrow BC\} \cup \{A \rightarrow C\})^+\) = \(F^+\)
     b. Test if B extraneous in AB \(\rightarrow\) C
        \( B \) extraneous if (A \(\rightarrow\) C) \(\in\) \(F^+\) (this is an axiom)
Canonical Cover Example

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

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

combine A \to B and A \to BC, since A \to BC contains A \to B

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

A \to BC gives us A \to C, and so B is extraneous AB \to C

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

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

A \to BC gives us A \to C, and so A \to C is extraneous

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

A \to B with B \to C implies that A \to C, so C is extraneous in A \to 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: $R = (\text{ssn, name, address})$ can be broken down into:
  
  a) $R_1 = (\text{ssn, name})$ $R_2 = (\text{name, address})$
  
  b) $R_1 = (\text{ssn, name})$ $R_2 = (\text{ssn, address})$

- which is lossy (if either)?
- even though the result set has more tuples, this is lossy!
- why?
- lossless!
Dependency Preservation

- For a set of FDs $F$ over a relation $R$, if we decompose $R$ into $R_1$ and $R_2$
- and $R_1$ has a set of FDs $F_1$ contained in it, and $R_2$ has a set of FDs $F_2$, contained in it, where $F_1$ and $F_2$ are subsets of $F^+$
- then, if we can derive $F^+$ from just $F_1$ and $F_2$, then we say that the decomposition is dependency preserving.
Dependency Preservation

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

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

Note that \( \text{ssn} \rightarrow \text{can\_drink} \), by transitivity.

Decompose \( R \) into \( R_1 = (\text{ssn}, \text{name}, \text{age}) \), \( R_2 = (\text{age}, \text{can\_drink}) \)
then we can check \( \text{ssn} \rightarrow \{\text{name}, \text{age}\}, \text{age} \rightarrow \text{can\_drink} \)
from just checking \( R_1 \) and \( R_2 \), without having to do any joins.

And all other FDs can be derived from them, so it is dependency preserving.
Dependency Preservation

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

Suppose we decompose \( R \) into \( R_1 = (A, B), R_2 = (A, C, D) \),
Then we can only check \( A \rightarrow B, A \rightarrow C \) without joins, and not \( B \rightarrow C \) (for that we'd need a join)
So not DP.

However, if we decompose: \( R_1 = (A, B), R_2 = (B, C, D) \)
Then DP, since checking \( A \rightarrow B \) and \( B \rightarrow C \) can be done without joins, and they imply \( A \rightarrow C \), so
all dependencies are covered by just checking the dependencies inside of the relations
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, Not Dependency Preserving

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

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

\[ R_1 = (B, C) \quad R_2 = (C, A) \]
Boyce-Codd Normal Form

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