Managing Functional Dependencies and Redundancy

General functional constraints (Review)

General form for relational model:
- Let \( \alpha(R) \) denote the set of names of attributes in the schema for relation \( R \)
- Let \( X \) and \( Y \) be subsets of \( \alpha(R) \)

The functional dependency \( X \rightarrow Y \) holds if for any instance \( I \) of \( R \) and for any pair of tuples \( t_1 \) and \( t_2 \) of \( R_i \),

\[
\pi_X(t_1) = \pi_X(t_2) \Rightarrow \pi_Y(t_1) = \pi_Y(t_2)
\]

- special cases: candidate keys, superkeys

Functional Constraint in SQL

CREATE TABLE Student
(
  sid CHAR(10),
  street CHAR(40),
  city CHAR(40),
  state CHAR(40),
  zipcode CHAR(10)
)

PRIMARY KEY (sid)

CHECK (not exist (
  SELECT *
  FROM Student S
  WHERE S.zipcode=zipcode
  AND S.state<>state
)
)

Why store state?

Redundancy
- Functional dependencies capture redundancy in a relation e.g. zipcode→state: why store state?
- Redundancy good for reliability
- Redundancy bad for
  - space to store: repetitions
  - must maintain on changes
  - representation of one relationship embedded in another

Example relation for a city elementary school system:

school_child: (name, st_addr, apt., birthday, school)

st_addr→school

consider a large apt. building

Solution: decompose

Example:

child: (name, st_addr, apt., birthday)
placement: (st_addr, school)

- child \( \bowtie \) placement gives school_child because of functional dependency
- space gain larger than space cost
- functional dependency now primary key constraint
- st_addr, school correspondence implicitly maintained
- computation cost?

General Form:
- for \( X, Y \subseteq \alpha(R) \) and \( X \rightarrow Y \)
- decompose \( R \) into
  \[
  R_1: \alpha(R) \setminus (Y \setminus X)
  R_2: X \cup Y
  \]

Constraint (school, stuID) → (st_addr, apt., birthday)
- was primary key constraint
- now split constraint
to check requires \( \bowtie \) - expensive
- primary key for stu?
Propose: 
**decompose to eliminate redundancy**

Two examples
1. school_child: (name, st_addr, apt., birthday, school)
   st_addr → school
   becomes       stu: (stuID, st_addr, apt., birthday)
   placement: (st_addr, school)

   primary key for stu?
   (stuID, st_addr) → (stuID, st_addr, school)
   (stuID, st_addr, school) → (stuID, st_addr, apt., birthday)
   so              stu: (stuID, st_addr, apt., birthday)

    new primary key constraint does not imply
   old primary key constraint:
   (school, stuID ) → (st_addr, apt., birthday)

Decomposition: Formal Properties

- Let \( \Phi \) be a set of functional dependencies (FDs) for a relational scheme \( R \) with attribute set \( \alpha(R) \)
- Let \( \Phi^+ \) denote the set of all FDs implied by \( \Phi \)
  the closure of \( \Phi \)
- Let \( X, Y \subseteq \alpha(R) \), where \( X \cap Y \) is not necessarily empty
- Let \( \Phi_2 \) denote set of FDs \( V \rightarrow W \) in \( \Phi^+ \) with \( Y \subseteq X \) and \( W \subseteq X \)
- Decomposition of \( R \) into \( R_1 \) and \( R_2: Y \) is
  - **lossless** if for every instance \( I \) of \( R \) that satisfies \( \Phi \)
    \( r_1(I) \bowtie r_2(I) = I \)
    • guaranteed to get back \( R \)
  - **dependency preserving** if \( (\Phi_2 \cup \Phi_2)^* = \Phi^+ \)
    • can check all FDs for \( R \) by checking all for \( X \) and all for \( Y \) without doing JOIN

Implied functional dependencies

- **Definition:** a functional dependency \( X \rightarrow Y \) is **implied by** \( \Phi \) if \( X \rightarrow Y \) holds whenever all functional dependencies in \( \Phi \) hold
- **Armstrong’s Axioms** for attribute sets \( X, Y, Z \)
  1. if \( X \subseteq Y \) then \( Y \rightarrow X \) (reflexivity)
  2. if \( X \rightarrow Y \) then \( YZ \rightarrow XZ \) (augmentation)
  3. if \( X \rightarrow Y \) and \( Y \rightarrow Z \) then \( X \rightarrow Z \) (transitivity)
- **Theorem:** The set of all functional dependencies obtained from \( \Phi \) by repeated application of Armstrong’s Axioms gives \( \Phi^+ \)

Normal Forms

- How do we find “good” (“best”?) decomposition?
- Identify **normal forms** with desirable properties
  - must be lossless – can’t lose anything
  - should be dependency preserving - avoid need for joins to check dependencies
- Decompose so resulting relations are in normal form

Boyce-Codd Normal Form (BCNF)

- Let \( R \) denote a relational scheme with attribute set \( \alpha(R) \)
- \( R \) is in BCNF with respect to a set \( \Phi \) of FDs if for all FDs in \( \Phi^+ \) of the form \( X \rightarrow Y \) with \( X, Y \subseteq \alpha(R) \), at least one of
  - \( Y \subseteq X \) (trivial func. dep.)
  - \( X \) is a superkey for \( R \)
- very strong normal form
- can’t always get dependency preserving decomposition into set of BCNF relations
Third Normal Form (3NF)

- Let \( R \) denote a relational scheme with attribute set \( \alpha(R) \).
- \( R \) is in 3NF with respect to a set \( \Phi \) of FDs if for all FDs in \( \Phi^+ \) of the form \( X \rightarrow Y \) with \( X, Y \subseteq \alpha(R) \), at least one of:
  - \( Y \subseteq X \) (trivial func. dep.)
  - \( X \) is a superkey for \( R \)
  - each attribute \( A \) in \( Y \) is contained in a candidate key for \( R \)

- can always get lossless, dependency preserving decomposition into 3NF relations
- cannot always remove all functional dependencies

Why allow right hand side part of some candidate key?

- consider decomposing \( R \) using \( X \rightarrow A \)
  - \( A \) an attribute
  - \( X \) not superkey
  - \( A \) not in \( X \)
- get \( R_1: \alpha(R) - (A) \) and \( R_2: X \cup \{A\} \)
- if \( A \) not part of a candidate key then for any candidate key \( K \subseteq \alpha(R) \):
  check \( K \rightarrow \alpha(R) - \{A\} \) in \( R_1 \)
  including \( K \rightarrow X \)
  check \( X \rightarrow A \) in \( R_2 \)
  conclude \( K \rightarrow A \)

- if \( A \) is part of a candidate key \( K \):
  splitting key: \( K \cup \{A\} \in \alpha(R_1) \)
  \( K \cap (X \cup \{A\}) \in \alpha(R_2) \)
  to check \( K \) is a candidate key need \( R_1 \bowtie R_2 \) avoiding

Revisit example

<table>
<thead>
<tr>
<th>Lossless-join decomposition?</th>
<th>Dependency preserving decomposition?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal forms?</td>
<td></td>
</tr>
</tbody>
</table>

- school_child: (school, stuID, st_addr, apt., birthday)
  - becomes: school_child (st_addr \rightarrow school)
  - stu: (stuID, st_addr, apt., birthday)
  - placement: (st_addr, school)

Constraint (school, stuID) \( \rightarrow \) (st_addr, apt., birthday)
  - was primary key constraint
  - now split constraint
  to check requires \( \bowtie \) expensive

Decomposition to achieve 3NF

- Is polynomial-time algorithm for 3NF lossless-join, dependency-preserving decomposition
- Can require adding “extra” relation.
- Get at expense of redundancy

Example
- \( R \) with attributes \( ABCD \); \( AB \) primary key;
  - other functional dependencies \( A \rightarrow C \); \( B \rightarrow C \)
  - decompose \( R_1: ABD; R_2: BC \)
  - lossless? dependency-preserving?

Discussion

- Consider normal forms when designing relations.
- Using 3NF minimizes problems of general functional dependencies
  - does not eliminate
- Use BCNF if can get it
  - decomposition algorithm simpler too!

Example
- \( R \) with attributes \( ABCD \); \( AB \) primary key;
  - decompose \( R_1: ABD; R_2: BC \)
  - add \( R_3: AC \)
    - redundant because can get from \( R_1 \bowtie R_2 \)