IF3111 Basis Data - Normalisasi

Tricya Widagdo
Departemen Teknik Informatika
Institut Teknologi Bandung

First Normal Form (1NF)

- Domain is atomic if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in first normal form if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - E.g. Set of accounts stored with each customer, and set of owners stored with each account

Atomicity is actually a property of how the elements of the domain are used.

•E.g. Strings would normally be considered indivisible

•Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*

•If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.

•Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

All relations are assumed to be in first normal form.

Pitfalls in Relational Database Design

- Relational database design requires that we find a "good" collection of relation schemas. A bad design may lead to
  - Repetition of Information.
  - Inability to represent certain information.
- Design Goals:
  - Avoid redundant data
  - Ensure that relationships among attributes are represented
  - Facilitate the checking of updates for violation of database integrity constraints.
- Example:
  Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount)*

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500

Problems with the lending instance:

•Redundancy:

•Data for *branch-name*, *branch-city*, assets are repeated for each loan that a branch makes

•Wastes space

•Complicates updating, introducing possibility of inconsistency of *assets* value

•Null values

•Cannot store information about a branch if no loans exist

•Can use null values, but they are difficult to handle.

Decomposition

- Decompose the relation schema *Lending-schema* into:

Branch-schema = (branch-name, branch-city,assets)

Loan-info-schema = (customer-name, loan-number, branch-name, amount)

- All attributes of an original schema (R) must appear in the decomposition (R_1, R_2):

$$R = R_1 \cup R_2$$

- Lossless-join decomposition.
  For all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

Example non lossless-join decomposition of $R = (A, B)$ to $R_1 = (A)$ and $R_2 = (B)$

r

A	B
a	1
a	2
b	1

$\Pi_A(r)$

A
a
b

$\Pi_B(r)$

B
1
2

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B
a	1
a	2
b	1
b	2

Goals of Normalization

- Decide whether a particular relation *R* is in "good" form.
- In the case that a relation *R* is not in "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - functional dependencies
  - multivalued dependencies

Multivalued dependencies would not be discussed in this course

Lossless-join Decomposition

- Lossless-join decomposition.
  For all possible relations *r* on schema *R*
  $$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$
- A decomposition of R into R_1 and R_2 is lossless join if and only if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$

  Otherwise, the decomposition will be lossy

Normalization Using Functional Dependencies

- When we decompose a relation schema *R* with a set of functional dependencies *F* into R_1, R_2,.., R_n we want
  - Lossless-join decomposition: Otherwise decomposition would result in information loss.
  - No redundancy: The relations R_i preferably should be in either Boyce-Codd Normal Form or Third Normal Form.
  - Dependency preservation: Let F_i be the set of dependencies F^+ that include only attributes in R_i.
    - Preferably the decomposition should be dependency preserving, that is, $(F_1$ È F_2 È ... È $F_n)^+ = F^+$
    - Otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
  – Can be decomposed in two different ways
- $R_1 = (A, B), \quad R_2 = (B, C)$
  – Lossless-join decomposition:
  $$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$
  – Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  – Lossless-join decomposition:
  $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$
  – Not dependency preserving
  (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into R_1, R_2, ..., R_n we apply the following simplified test (with attribute closure done w.r.t. F)
  - *result* = α
    while (changes to *result*) do
    for each R_i in the decomposition
    $t = (result \cap R_i)^+ \cap R_i$
    $result = result \cup t$
  - If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved.
- We apply the test on all dependencies in F to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and
  (F_1 È F_2 È ... È F_n)$^+$

Second Normal Form (2NF)

- A relation is in 2NF if and only if it is in 1NF and every nonkey attribute is irreducibly dependent on the primary key
- The 2NF still allows transitive dependencies
- Transitive dependencies lead to update anomalies

The above definition assuming only one candidate key, which we assume is the primary key

Third Normal Form (3NF)

- A relation is in 3NF if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.
- The 3NF assumes that the relation has only one candidate key.
- 3NF does not adequately deals with the case of a relation that:
  - Had two or more candidate keys, such that
  - The candidate keys were composite, and
  - The overlapped, i.e. have at least one attribute in common

Third Normal Form (Formal Definition)

- A relation schema R is in third normal form (3NF) if for all:

  $$\alpha \rightarrow b \text{ in } F^+$$

  at least one of the following holds:
  - $\alpha \rightarrow b$ is trivial (i.e., $b \in \alpha$)
  - α is a superkey for R
  - Each attribute A in $b - \alpha$ is contained in a candidate key for R.
    (NOTE: each attribute may be in a different candidate key)
- Example
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$
  - Two candidate keys: JK and JL
  - R is in 3NF

    $JK \rightarrow L$ JK is a superkey
    $L \rightarrow K$ K is contained in a candidate key
- There is some redundancy in this schema

The above example is equivalent to the example in Silberschatz:

Banker-schema = (branch-name, customer-name, banker-name)

banker-name $\rightarrow$ branch name

branch name customer-name $\rightarrow$ banker-name

Testing for 3NF

- Optimization: Need to check only FDs in *F*, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of *R*
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form can be done in polynomial time

Boyce-Codd Normal Form

- A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form $\alpha \rightarrow b$, where $\alpha \subseteq R$ and $b \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow b$ is trivial (i.e., $b \subseteq \alpha$)
  - α is a superkey for R
- Example:

  $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  Key $= \{A\}$
  - R is not in BCNF
  - Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
    - R_1 and R_2 in BCNF
    - Lossless-join decomposition
    - Dependency preserving

If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).

Third condition is a minimal relaxation of BCNF to ensure dependency preservation.

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow b$ causes a violation of BCNF
  1. compute α^+ (the attribute closure of α), and
  2. verify that it includes all attributes of *R*, that is, it is a superkey of *R*.
- Simplified test: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in F^+.
  - If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.
- However, using only F is incorrect when testing a relation in a decomposition of R
  - E.g. Consider *R* (*A, B, C, D*), with *F* = { *A* ® *B, B* ® *C*}
    - Decompose *R* into $R_1(A,B)$ and $R_2(A,C,D)$
    - Neither of the dependencies in *F* contain only attributes from (*A,C,D*) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $A \rightarrow C$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

- To check if a relation R_i in a decomposition of R is in BCNF,
  - Either test R_i for BCNF with respect to the restriction of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
  - or use the original set of dependencies F that hold on R, but with the following test:
    - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.
  - If the condition is violated by some $\alpha \rightarrow b$ in F, the dependency
    $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$
    can be shown to hold on R_i, and R_i violates BCNF.
  - We use above dependency to decompose R_i

BCNF and Dependency Preservation

- It is not always possible to get a BCNF decomposition that is dependency preserving
- $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$
  Two candidate keys = JK and JL
- R is not in BCNF
- Any decomposition of R will fail to preserve

$$JK \rightarrow L$$

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into relations in 3NF and
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into relations in BCNF and
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.

Comparison of BCNF and 3NF (Cont.)

- Example of problems due to redundancy in 3NF
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

A schema that is in 3NF but not in BCNF has the problems of

- repetition of information (e.g., the relationship l_1, k_1)
- need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J).

Design Goals

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

  Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Testing for FDs Across Relations

- If decomposition is not dependency preserving, we can have an extra **materialized view** for each dependency $\alpha \rightarrow \beta$ in F_c that is not preserved in the decomposition
- The materialized view is defined as a projection on $\alpha\ \beta$ of the join of the relations in the decomposition
- Many newer database systems support materialized views and database system maintains the view when the relations are updated.
  - No extra coding effort for programmer.
- The functional dependency $\alpha \rightarrow \beta$ is expressed by declaring α as a candidate key on the materialized view.
- Checking for candidate key cheaper than checking $\alpha \rightarrow \beta$
- BUT:
  - Space overhead: for storing the materialized view
  - Time overhead: Need to keep materialized view up to date when relations are updated
  - Database system may not support key declarations on materialized views

Overall Database Design Process

- We have assumed schema *R* is given
  - *R* could have been generated when converting E-R diagram to a set of tables.
  - *R* could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
  - Normalization breaks *R* into smaller relations.
  - *R* could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design there can be FDs from non-key attributes of an entity to other attributes of the entity
- E.g. *employee* entity with attributes *department-number* and *department-address*, and an FD *department-number* ® *department-address*
  - Good design would have made department an entity
- FDs from non-key attributes of a relationship set possible, but rare --- most relationships are binary

Denormalization for Performance

- May want to use non-normalized schema for performance
- E.g. displaying *customer-name* along with *account-number* and *balance* requires join of *account* with *depositor*
- Alternative 1: Use denormalized relation containing attributes of *account* as well as *depositor* with all above attributes
  - faster lookup
  - Extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as
  account $\bowtie$ depositor
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:

  Instead of *earnings*(*company-id, year, amount*), use
  - *earnings-2000, earnings-2001, earnings-2002*, etc., all on the schema (*company-id, earnings*).
    - Above are in BCNF, but make querying across years difficult and needs new table each year
  - *company-year*(*company-id, earnings-2000, earnings-2001,*

    earnings-2002)
    - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
    - Is an example of a **crosstab**, where values for one attribute become column names
    - Used in spreadsheets, and in data analysis tools