Relational Database Design

Relational Database Design

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies and Fourth Normal Form
- Overall Database Design Process

First Normal Form

- 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

First Normal Form (Contd.)

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

  For example:

  - 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

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

- **Consider the relation schema:**

  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

- **Redundancy:**
  - Data for *branch-name, branch-city, assets* are repeated for each loan
  - 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.

Redundancy – Drawbacks

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

- **Redundancy Storage:** Downtown repeated

?

- **Update Anomalies:** update assets where customer-name=Jones

?

- **Insertion Anomalies:** without loan-number cannot insert

?

- **Deletion Anomalies:** delete Smith record

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:** instances

  For all possible relations r on schema R

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

- **Dependency-Preservation:** constraints

Example of Non Lossless-Join Decomposition

- Decomposition of $R = (A, B)$

  $R_2 = (A) \qquad R_2 = (B)$

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$

B
1
2

$\Pi_{B(r)}$

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

A	B
α	1
α	2
β	1
β	2

Decomposition of $R = (A, B, C)$: $R_2 = (A, B)$ *and* $R_2 = (B, C)$

A	B	C

Goal — Devise a Theory for the Following

- 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 $\{R_1, R_2, \ldots, R_n\}$ such that
    - each relation is in good form
    - the decomposition is a lossless-join decomposition
- Our theory is based on:
    - functional dependencies
    - multivalued dependencies

Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.

Functional Dependencies (Cont.)

- Let R be a relation schema $\alpha \subseteq R$ *and* $\beta \subseteq R$
- Functional dependency $\alpha \rightarrow \beta$ holds on R **if and only if** for any legal relations *r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes β. That is, $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$
- Example: Consider *r(A,B)* with the following instance of *r.*

A	*B*
1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

Functional Dependencies

- Example: $R = (A, B, C, D)$

A	B	C	D

Functional Dependencies (Cont.)

- K is a super-key for a relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys.

  Consider the schema:

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

  We expect this set of functional dependencies to hold:

  loan-number $\rightarrow$ *amount*
  loan-number $\rightarrow$ *branch-name*

  but would not expect the following to hold:

  loan-number $\rightarrow$ *customer-name*

Use of Functional Dependencies

- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* satisfies *F*.
  - specify constraints on the set of legal relations
    - We say that *F* holds on *R* if all legal relations on *R* satisfy the set of functional dependencies *F*.
- **Note:** A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.

  For example, a specific instance of *Loan-schema* may, by chance, satisfy

  loan-number → *customer-name*

Functional Dependencies (Cont.)

- *A* functional dependency is trivial if it is satisfied by all instances of a relation
  - *E.g.*
    - *customer-name, loan-number* $\rightarrow$ *customer-name*
    - *customer-name* $\rightarrow$ *customer-name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

BREAK

Closure of a Set of Functional Dependencies

- Given a set *F*, set of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.
  - E.g. If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- Set of all functional dependencies logically implied by *F* is the *closure* of *F*.
- We denote the *closure* of *F* by F^+.
- We can find all of F^+ by applying Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (**reflexivity**)
  - if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$ (**augmentation**)
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (**transitivity**)
- These rules are
  - sound (generate only functional dependencies that actually hold) and
  - complete (generate all functional dependencies that hold).

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H \}$
- **some members of F^+**
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - from $CG \rightarrow H$ *and* $CG \rightarrow I$: "union rule" can be inferred from
      - definition of functional dependencies, or
      - Augmentation of $CG \rightarrow I$ to infer $CG \rightarrow CGI$, augmentation of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F:

- $F^+ = F$
  repeat
      for each functional dependency f in F^+
          apply reflexivity and augmentation rules on f
          add the resulting functional dependencies to F^+
      for each pair of functional dependencies f_1 and f_2 in F^+
          if f_1 and f_2 can be combined using transitivity
              then add the resulting functional dependency to F^+
  until F^+ does not change any further

NOTE: We will see an alternative procedure for this task later

Closure of Functional Dependencies (Cont.)

- We can further simplify manual computation of F^+ by using the following additional rules.

  - If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
  - If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

Closure of Attribute Sets

- Given a set of attributes α, define the *closure* of α under F (denoted by α^+) as the set of attributes that are functionally determined by α under F:

$$\alpha \rightarrow \beta \text{ is in } F^+ \;\Leftrightarrow\; \beta \subseteq \alpha^+$$

- Algorithm to compute α^+, the closure of α under F

$result := \alpha;$
while (changes to $result$) **do**
  for each $\beta \rightarrow \gamma$ **in** F **do**
   begin
    if $\beta \subseteq result$ **then** $result := result \cup \gamma$
   end

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H \}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$

- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$?
  2. Is any subset of AG a superkey?
     1. Does $A^+ \rightarrow R$?
     2. Does $G^+ \rightarrow R$?

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute α^{+}, and check if α^{+} contains all attributes of R.
- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^{+}), just check if $\beta \subseteq \alpha^{+}$.
  - That is, we compute α^{+} by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^{+}, and for each $S \subseteq \gamma^{+}$, we output a functional dependency $\gamma \rightarrow S$.

THANK YOU

Reference (Textbook):

- Silberschatz, H. Korth & S. Sudarshan, Database System Concepts, McGraw-Hill Education, 6th Edition, 2010