Agenda

- Check-in
- Design Project ERD Presentations
- Database Design Principles
  - Decomposition
  - Functional Dependencies
  - Closures
  - Canonical Cover
- Homework 3
Check-in
The Entities and Relationships of the Psalms
Psalm 30, 31, and 32 (NIV)
Some Assertions about Psalm Data

• Each psalm is identified by a number, has a text, and may have a type, recipient, and zero or more instruments.

• An author is identified by a name and has a position. An author may write multiple psalms, but every author must be associated with at least one psalm.

• An occasion is identified by a name and has a location. A single psalm can be used for multiple occasions, but it doesn’t make sense to have an occasion without a psalm. (How boring would that be!)

• A psalm can describe one or more acts of God, and multiple psalms can describe a single act of God.
Design Project ERD Presentations

Milestone II
Database Design Principles
Introduction

- Terminology review
  - Relation scheme – set of attributes for some relation (R, R₁, R₂)
  - Relation – the actual data stored in some relation scheme (r, r₁, r₂)
  - Tuple – a single actual row in the relation (t, t₁, t₂)

- Changes to the library database schema
  - We make the following updates for this discussion
    - Add the following attributes to the book relation
      - copy_number – a library can have multiple copies of a book
      - accession_number – unique number (ID) assigned to a copy of a book when the library acquires it
  - New book and checked_out relation scheme
    - Book( call_number, copy_number, accession_number, title, author )
    - Checked_out( borrower_id, call_number, copy_number, date_due )
The Art of Database Design

• Designing a database is a balancing act

• On the one extreme, you can have a *universal relation* (in which all attributes reside within a single relation scheme)
  • Everything(
    borrower_id, last_name, first_name,  // from borrower
    call_number, copy_number,
    accession_number, title, author     // from book
    date_due                             // from checked_out
  )

• Leads to numerous anomalies with changing data in the database
Break Up Relations with Decomposition

- *Decomposition* is the process of breaking up an original scheme into two or more schemes
  - Each attribute of the original scheme appears in at least one of the new schemes
- But this can be taken too far
  - Borrower( borrower_id, last_name, first_name )
  - Book( call_number, copy_number, accession_number, title, author )
  - Checked_out( date_due )
- Leads to *lossy-join* problems
We Want Lossless-Join Decompositions

• Part of the middle ground in the balancing act
  • Allows decomposition of the Everything relation
  • Preserves connections between the tuples of the participating relations
  • So that the natural join of the new relations = the original Everything relation

• Formal definition
  • For some relation scheme $R$ decomposed into two or more schemes ($R_1, R_2, \ldots R_n$)
  • Where $R = R_1 \cup R_2 \cup \ldots \cup R_n$
  • A lossless-join decomposition means that for every legal instance $r$ of $R$ decomposed into $r_1, r_2, \ldots r_n$ of $R_1, R_2, \text{ and } R_n$
  • $r = r_1 |X| r_2 |X| \ldots |X| r_n$
Database Design Goal: Create “Good” Relations

• We want to be able to determine whether a particular relation $R$ is in “good” form.
  • We’ll talk about how to do this shortly

• 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 Dependency (FD)

• When the value of a certain set of attributes uniquely determines the value for another set of attributes
  • Generalization of the notion of a key
  • A way to find “good” relations
  • A → B (read: A determines B)

• Formal definition
  • For some relation scheme R and attribute sets A (A ⊆ R) and B (B ⊆ R)
  • A → B if for any legal relation on R
    • If there are two tuples t₁ and t₂ such that t₁(A) = t₂(A)
    • It must be the case that t₂(B) = t₂(B)
Finding Functional Dependencies

- From keys of an entity
- From relationships between entities
- Implied functional dependencies
FDs from Entity Keys

A → BC
FDs from One to Many / Many to One Relationships

A → BC
W → XY
A → BCMWXY
FDs from One to One Relationships

A → BC
W → XY
A → BCMWXY
W → XYMABC
FDs from Many to Many Relationships

A → BC
W → XY
AW → M
Implied Functional Dependencies

- Initial set of FDs *logically implies* other FDs
  - If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$

- Closure
  - If $F$ is the set of functional dependencies we develop from the logic of the underlying reality
  - Then $F^+$ (the transitive closure of $F$) is the set consisting of all the dependencies of $F$, plus all the dependencies they imply
Rules for Computing F+

- We can find $F^+$, the closure of $F$, by repeatedly applying Armstrong’s Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (reflexivity)
  - Trivial dependency
  - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ (augmentation)
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (transitivity)

- Additional rules (inferred from Armstrong’s Axioms)
  - If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta \gamma$ (union)
  - If $\alpha \rightarrow \beta \gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ (decomposition)
  - If $\alpha \rightarrow \beta$ and $\gamma \beta \rightarrow \delta$, then $\alpha \gamma \rightarrow \delta$ (pseudotransitivity)
Applying the Axioms

\[ R = (A, B, C, G, H, I) \]
\[ F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H \} \]

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 \( CG \rightarrow I \)
- \( CG \rightarrow HI \)
  - by augmenting \( CG \rightarrow I \) to infer \( CG \rightarrow CGI \),
  - and augmenting of \( CG \rightarrow H \) to infer \( CGI \rightarrow HI \),
  - and then transitivity
- or by the union rule
Algorithm to Compute 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
Algorithm to Compute the Closure of Attribute Sets

• Given a set of attributes \( \alpha \), define the closure of \( \alpha \) under \( F \) (denoted by \( \alpha^+ \)) as the set of attributes that are functionally determined by \( \alpha \) under \( F \)

• Algorithm to compute \( \alpha^+ \), the closure of \( \alpha \) under \( F \)

\[
\text{result} := \alpha; \\
\text{while (changes to result) do} \\
\quad \text{for each } \beta \rightarrow \gamma \text{ in } F \text{ do} \\
\quad \quad \text{begin} \\
\quad \quad \quad \text{if } \beta \subseteq \text{result} \text{ then } \text{result} := \text{result} \cup \gamma \\
\quad \quad \text{end}
\]
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. \( result = ABCG \) \((A \rightarrow C \text{ and } A \rightarrow B)\)
  3. \( result = ABCGH \) \((CG \rightarrow H \text{ and } CG \subseteq AGBC)\)
  4. \( result = ABCGHI \) \((CG \rightarrow I \text{ and } CG \subseteq AGBCH)\)

- Is \( AG \) a candidate key?
  1. Is \( AG \) a super key?
     1. Does \( AG \rightarrow R \) \(\Rightarrow\) \( (AG)^+ \subseteq R \)
     2. Is any subset of \( AG \) a superkey?
        1. Does \( A \rightarrow R \) \(\Rightarrow\) \( (A)^+ \subseteq R \)
        2. Does \( G \rightarrow R \) \(\Rightarrow\) \( (G)^+ \subseteq R \)
Canonical Cover

• Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  • For example: \( A \rightarrow C \) is redundant in: \( \{ A \rightarrow B, \ B \rightarrow C, \ A \rightarrow C \} \)
  • Parts of a functional dependency may be redundant
    • E.g.: on RHS: \( \{ A \rightarrow B, \ B \rightarrow C, \ A \rightarrow CD \} \) can be simplified to \( \{ A \rightarrow B, \ B \rightarrow C, \ A \rightarrow D \} \)
    • E.g.: on LHS: \( \{ A \rightarrow B, \ B \rightarrow C, \ AC \rightarrow D \} \) can be simplified to \( \{ A \rightarrow B, \ B \rightarrow C, \ A \rightarrow D \} \)

• Intuitively, a canonical cover of \( F \) is a “minimal” set of functional dependencies equivalent to \( F \), having no redundant dependencies or redundant parts of dependencies
Definition of Canonical Cover

• A **canonical cover** for $F$ is a set of dependencies $F_c$ such that
  • $F$ logically implies all dependencies in $F_c$, and
  • $F_c$ logically implies all dependencies in $F$, and
  • No functional dependency in $F_c$ contains an extraneous attribute, and
  • Each left side of functional dependency in $F_c$ is unique.

• To compute a canonical cover for $F$:
  repeat
    Use the union rule to replace any dependencies in $F$
    $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
    Find a functional dependency $\alpha \rightarrow \beta$ with an
    extraneous attribute either in $\alpha$ or in $\beta$
    /* Note: test for extraneous attributes done using $F_c$, not $F$*/
    If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
  until $F$ does not change

• Note: Union rule may become applicable after some extraneous attributes
  have been deleted, so it has to be re-applied
How to Find a Canonical Cover

• Another algorithm
  • Write F as a set of dependencies where each has a single attribute on the right hand side
  • Eliminate trivial dependencies
    • In which $\alpha \rightarrow \beta$ and $\beta \subseteq \alpha$ (reflexivity)
  • Eliminate redundant dependencies (implied by other dependencies)
  • Combine dependencies with the same left hand side

• For any given set of FDs, the canonical cover is not necessarily unique
Uses of Functional Dependencies

- Testing for lossless-join decomposition
- Testing for dependency preserving decompositions
- Defining keys
Testing for Lossless-Join Decomposition

• The closure of a set of FDs can be used to test if a decomposition is lossless-join

• For the case of $R = (R_1, R_2)$, we require that for all possible relations $r$ on schema $R$

$$r = \Pi_{R_1}(r) \cdot \Pi_{R_2}(r)$$

• A decomposition of $R$ into $R_1$ and $R_2$ is lossless join if at least one of the following dependencies is in $F^+$:
  • $R_1 \cap R_2 \rightarrow R_1$
  • $R_1 \cap R_2 \rightarrow R_2$

• Does the intersection of the decomposition satisfy at least one FD?
Testing for Dependency Preserving Decompositions

• The closure of a set of FDs allows us to test a new tuple being inserted into a table to see if it satisfies all relevant FDs without having to do a join
  • This is desirable because joins are expensive

• Let $F_i$ be the set of dependencies $F^+$ that include only attributes in $R_i$.
  • A decomposition is dependency preserving, if
    $$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$
  • If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

• The closure of a dependency preserving decomposition equals the closure of the original set

• Can all FDs be tested (either directly or by implication) without doing a join?
Keys and Functional Dependencies

- Given a relation scheme R with attribute set $K \subseteq R$
  - $K$ is a superkey if $K \rightarrow R$
  - $K$ is a candidate key if there is no subset $L$ of $K$ such that $L \rightarrow R$
    - A superkey with one attribute is always a candidate key
    - Primary key is the candidate key $K$ chosen by the designer
- Every relation must have a superkey (possibly the entire set of attributes)
- *Key attribute* – an attribute that is or is part of a candidate key
Homework 3