Chapter 6: Integrity Constraints

- Domain Constraints
- Referential Integrity
- Assertions
- Triggers
- Functional Dependencies

Domain Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Domain constraints are the most elementary form of integrity constraint.
- They test values inserted in the database, and test queries to ensure that the comparisons make sense.

Domain Constraints (Cont.)

- The **check** clause in SQL-92 permits domains to be restricted:
  - Use **check** clause to ensure that an hourly-wage domain allows only values greater than a specified value.

    create domain *hourly-wage* **numeric(5,2)**
            constraint *value-test* **check**(*value* $>$= 4.00)
  - The domain *hourly-wage* is declared to be a decimal number with 5 digits, 2 of which are after the decimal point
  - The domain has a constraint that ensures that the hourly-wage is greater than 4.00.
  - The clause **constraint** *value-test* is optional; useful to indicate which constraint an update violated.

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: if “Perryridge” is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch “Perryridge”.
- Formal Definition
  - Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2 respectively.
  - The subset α of R_2 is a *foreign key* referencing K_1 in relation r_1, if for every t_2 in r_2 there must be a tuple t_1 in r_1 such that $t_1[K_1] = t_2[\alpha]$.
  - Referential integrity constraint: $\Pi_\alpha (r_2) \subseteq \Pi_{K_1} (r_1)$

Referential Integrity in the E-R Model

- Consider relationship set R between entity sets E_1 and E_2. The relational schema for R includes the primary keys K_1 of E_1 and K_2 of E_2.

  Then K_1 and K_2 form foreign keys on the relational schemas for E_1 and E_2 respectively.
- Weak entity sets are also a source of referential integrity constraints. For, the relation schema for a weak entity set must include the primary key of the entity set on which it depends.

Database Modification

- The following tests must be made in order to preserve the following referential integrity constraint:

$$\Pi_\alpha (r_2) \subseteq \Pi_K (r_1)$$

- **Insert.** If a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$. That is

$$t_2[\alpha] \in \Pi_K (r_1)$$

- **Delete.** If a tuple t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference t_1:

$$\sigma_{\alpha = t_1[K]} (r_2)$$

  If this set is not empty, either the delete command is rejected as an error, or the tuples that reference t_1 must themselves be deleted (cascading deletions are possible).

Database Modification (Cont.)

- **Update.** There are two cases:
  - If a tuple t_2 is updated in relation r_2 and the update modifies values for the foreign key α, then a test similar to the insert case is made. Let $t_2{}'$ denote the new value of tuple t_2. The system must ensure that

    $$t_2{}'[\alpha] \in \Pi_K(r_1)$$
  - If a tuple t_1 is updated in r_1, and the update modifies values for the primary key (K), then a test similar to the delete case is made. The system must compute

    $$\sigma_{\alpha = t_1[K]}(r_2)$$

    using the old value of t_1 (the value before the update is applied). If this set is not empty, the update may be rejected as an error, or the update may be cascaded to the tuples in the set, or the tuples in the set may be deleted.

Referential Integrity in SQL

- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
  - The **primary key** clause of the **create table** statement includes a list of the attributes that comprise the primary key.
  - The **unique key** clause of the **create table** statement includes a list of the attributes that comprise a candidate key.
  - The **foreign key** clause of the **create table** statement includes both a list of the attributes that comprise the foreign key and the name of the relation referenced by the foreign key.

Referential Integrity in SQL - Example

```
create table customer
    (customer-name    char(20)  not null,
     customer-street  char(30),
     customer-city    char(30),
     primary key (customer-name))

create table branch
    (branch-name      char(15)  not null,
     branch-city      char(30),
     assets           integer,
     primary key (branch-name))
```

Referential Integrity in SQL - Example (Cont.)

```
create table account
    (branch-name      char(15),
     account-number   char(10)  not null,
     balance          integer,
     primary key (account-number),
     foreign key (branch-name) references branch)

create table depositor
    (customer-name    char(20)  not null,
     account-number   char(10)  not null,
     primary key (customer-name, account-number),
     foreign key (account-number) references account,
     foreign key (customer-name) references customer )
```

Cascading Actions in SQL

create table *account*
    . . .
    foreign key (*branch-name*) **references** *branch*
                              on delete cascade
                              on update cascade,
    . . .)

- Due to the **on delete cascade** clauses, if a delete of a tuple in *branch* results in referential-integrity constraint violation, the delete “cascades” to the *account* relation, deleting the tuple that refers to the branch that was deleted.
- Cascading updates are similar.

Cascading Actions in SQL (Cont.)

- If there is a chain of foreign-key dependencies across multiple relations, with **on delete cascade** specified for each dependency, a deletion or update at one end of the chain can propagate across the entire chain.
- If a cascading update or delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.

Assertions

- An *assertion* is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL-92 takes the form

  create assertion <assertion-name> **check** <predicate>

- When an assertion is made, the system tests it for validity. This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
create assertion sum-constraint check
   (not exists (select * from branch
      where (select sum(amount) from loan
            where loan.branch-name = branch.branch-name)
         >= (select sum(amount) from account
            where loan.branch-name = branch.branch-name)))
```

Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance of $1000.00.

create assertion *balance-constraint* **check**
  (**not exists** (**select** * **from** *loan*
    where not exists (**select** *
      from *borrower*, *depositor*, *account*
      where *loan.loan-number* = *borrower.loan-number*
        and *borrower.customer-name* = *depositor.customer-name*
        and *depositor.account-number* = *account.account-number*
        and *account.balance* >= 1000)))

Triggers

- A *trigger* is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- The SQL-92 standard does not include triggers, but many implementations support triggers.

Trigger Example

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - setting the account balance to zero
  - creating a loan in the amount of the overdraft
  - giving this loan a loan number identical to the account number of the overdrawn account
- The condition for executing the trigger is an update to the *account* relation that results in a negative *balance* value.

Trigger Example (Cont.)

```
define trigger overdraft on update of account T
   (if new T.balance < 0
    then (insert into loan values
               (T.branch-name, T.account-number, − new T.balance)
           insert into borrower
               (select customer-name, account-number
                from depositor
                where T.account-number = depositor.account-number)
           update account S
           set S.balance = 0
           where S.account-number = T.account-number))
```

The keyword **new** used before *T.balance* indicates that the value of *T.balance* after the update should be used; if it is omitted, the value before the update is used.

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
$$\alpha \subseteq R, \ \beta \subseteq R$$
- The 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,
$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$
- K is a superkey for 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 (Cont.)

- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

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

  We expect this set of functional dependencies to hold:

  $$\textit{loan-number} \rightarrow \textit{amount}$$
  $$\textit{loan-number} \rightarrow \textit{branch-name}$$

  but would not expect the following to hold:

  $$\textit{loan-number} \rightarrow \textit{customer-name}$$

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* $\rightarrow$ *customer-name*.

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.
- The 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 and complete.

Closure (Cont.)

- We can further simplify computation of F^+ by using the following additional rules.
  - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ 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 and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds (**pseudotransitivity**)

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

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^+
  - $A \rightarrow H$
  - $AG \rightarrow I$
  - $CG \rightarrow HI$

Closure of Attribute Sets

- 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

- 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 $A \subseteq AGB$)
  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. $AG \rightarrow R$
  2. does $A^+ \rightarrow R$?
  3. does $G^+ \rightarrow R$?

Canonical Cover

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.
- A *canonical cover* F_c for F is a set of dependencies such that F logically implies all dependencies in F_c and F_c logically implies all dependencies in F, and further
  - No functional dependency in F_c contains an extraneous attribute.
  - Each left side of a functional dependency in F_c is unique.

Canonical Cover (Cont.)

- 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 $\beta$
    If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until F does not change
```

Example of Computing a Canonical Cover

- $R = (A, B, C)$

  $F = \{A \rightarrow BC$

  $B \rightarrow C$

  $A \rightarrow B$

  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
- A is extraneous in $AB \rightarrow C$ because $B \rightarrow C$ logically implies $AB \rightarrow C$.
- C is extraneous in $A \rightarrow BC$ since $A \rightarrow BC$ is logically implied by $A \rightarrow B$ and $B \rightarrow C$.
- The canonical cover is:

  $A \rightarrow B$

  $B \rightarrow C$