Chapter 6: Integrity and Security

- Domain Constraints
- Referential Integrity
- Assertions
- Triggers
- Security
- Authorization
- Authorization in SQL

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.
- New domains can be created from existing data types
  - E.g. **create domain** *Dollars* **numeric**(12, 2)
    create domain *Pounds* **numeric**(12,2)
- We cannot assign or compare a value of type Dollars to a value of type Pounds.
  - However, we can convert type as below
    (**cast** *r.A* **as** *Pounds*)
    (Should also multiply by the dollar-to-pound conversion-rate)

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 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.
- Can have complex conditions in domain check
  - **create domain** *AccountType* **char**(10)
    constraint *account-type-test*
    check (**value in** ('Checking', 'Saving'))
  - **check** (*branch-name* **in** (**select** *branch-name* **from** *branch*))

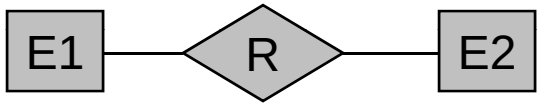
Referential Integrity

- 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 also called subset dependency since its can be written as

$$\Pi_{\alpha}(r_2) \subseteq \Pi_{K1}(r_1)$$

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 attributes of the entity set on which it depends

Checking Referential Integrity on 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:

$$\sigma_{\alpha = t1[K]}(r_2)$$

  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 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:
    1. The system must compute

$$\sigma_{\alpha = t1[K]}(r_2)$$

using the old value of t_1 (the value before the update is applied).

    2. If this set is not empty
       1. the update may be rejected as an error, or
       2. the update may be cascaded to the tuples in the set, or
       3. 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 lists attributes that comprise the primary key.
  - The **unique key** clause lists attributes that comprise a candidate key.
  - The **foreign key** clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key.
- By default, a foreign key references the primary key attributes of the referenced table

  foreign key (*account-number*) **references** *account*
- Short form for specifying a single column as foreign key

  account-number **char** (10) **references** *account*
- Reference columns in the referenced table can be explicitly specified
  - but must be declared as primary/candidate keys

  foreign key (*account-number*) **references** *account*(*account-number*)

Referential Integrity in SQL – Example

create table *customer*
(*customer-name* char(20)**,**
customer-street char(30),
customer-city char(30),
primary key (*customer-name))*

create table *branch*
(branch-name char(15)**,**
branch-city char(30),
assets integer,
primary key *(branch-name))*

Referential Integrity in SQL – Example (Cont.)

create table *account*
(account-number char(10)**,**
branch-name char(15),
balance integer,
primary key (*account-number),*
foreign key (*branch-name)* **references** *branch)*

create table *depositor*
(customer-name char(20)**,**
account-number char(10)**,**
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 to 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.
- Referential integrity is only checked at the end of a transaction
  - Intermediate steps are allowed to violate referential integrity provided later steps remove the violation
  - Otherwise it would be impossible to create some database states, e.g. insert two tuples whose foreign keys point to each other
    - E.g. *spouse* attribute of relation
      marriedperson(name, address, spouse)

Referential Integrity in SQL (Cont.)

- Alternative to cascading:
  - **on delete set null**
  - **on delete set default**
- Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using **not null**
  - if any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint!

Assertions

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

  create assertion <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting

  for all X, P(X)

  is achieved in a round-about fashion using

  not exists X such that not P(X)

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 or $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.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

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.

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
BEFORE | AFTER

DELETE | [OR] INSERT | [OR] UPDATE [OF <column> [, <column>...]]
 ON <table-name>          } triggering event

[REFERENCING [OLD [AS] <old>] [NEW [AS] <new>]]

[FOR EACH ROW [WHEN <condition>]]   } Trigger constraint


BEGIN

        PL/SQL block                    } Trigger body (trigger action)

END;
```

Trigger Example in SQL:1999

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
        insert into borrower
           (select customer-name, account-number
            from depositor
            where nrow.account-number =
                        depositor.account-number);
        insert into loan values
           (n.row.account-number, nrow.branch-name,
                                              – nrow.balance);
        update account set balance = 0
        where account.account-number = nrow.account-number
end
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - **E.g. create trigger** *overdraft-trigger* **after update of** *balance* **on** *account*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as :** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blanks to null.

```
create trigger setnull-trigger before update on r
referencing new row as nrow
for each row
    when nrow.phone-number = ' '
    set nrow.phone-number = null
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

Triggers

- Triggers can be disabled or enabled; by default they are enabled when they are created.

  alter trigger trigger_name **disable**

  alter trigger trigger_name **enable**

  drop trigger trigger_name

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

End of Chapter