Chapter 6
Integrity and Security

1. Introduction

- The term **integrity** refers to the accuracy or correctness of data in the database.
- **Integrity constraint** is a condition specified on a database schema which must hold on all of valid relation instances.
- **Integrity constraints** ensure that changes made to the database by authorized users do not result in loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.
- In general, an integrity constraint can be an arbitrary predicate pertaining to the database.
- All integrity constraints should be specified on the relational database schema if we want to enforce these constraints on the database states.
- In addition, we use **security mechanisms** to protect data stored in the database from unauthorized access and malicious destruction or alteration.

2. Domain Constraints

- Declaring an attribute to be a particular domain acts as a constraint on the values that it can take.
- Domain constraints are the most elementary form of integrity constraint.
- They are tested easily by the system whenever a new data item is entered into the database.
- It is possible for several attributes to have the same domain.
- A proper definition of domain constraints not only allows us to test values inserted in the database, but also permits us to test queries to ensure that the comparisons made make sense.
- The **create domain** clause can be used to define new domains. For example, the statements:

create domain *Dollars* **numeric**(12,2)

create domain *Pounds* **numeric**(12,2)

Domain Constraints (Cont.)

- An attempt to assign a value of type *Dollars* to a variable *Pounds* would result in a syntax error, although both are of the same numeric type.
- Values of one domain can be cast to another domain. If the attribute *A* on relation *r* is of type *Dollars*, we can convert it to *Pounds* by writing

  cast r.A **as** *Pounds*
- In a real application we would of course multiply *r.A* by a currency conversion factor before casting it to pounds.
- The **check** clause in SQL permits domains to be restricted in powerful ways.
- Specially, the **check** clause permits the schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain. For example,

Domain Constraints (Cont.)

create domain *HourlyWage* **numeric**(5,2)

constraint *wage-value-test* **check**(**value** >= 4.00)

- The clause **constraint** *wage-value-test* is optional, and is used to give the name *wage-value-test* to the constraint. The name is used to indicate to which constraint update violated.
- The **check** clause can also be used to restrict a domain to not contain any null values:

create domain *AccountNumber* **char**(10)

constraint *account-number-null-test* **check**(**value not null)**

- The domain can be restricted to contain only a specified set of values by using the **in** clause:

create domain *AccountType* **char**(10)

constraint *account-type-test*

check(**value in**('Checking', 'Saving')

Domain Constraints (Cont.)

- The check conditions can be more complex. For example,

  check(*branch-name* **in**(**select** *branch-name* **from** *branch*)
- SQL also provides **drop domain** and **alter domain** clauses to drop or modify domains that have been created earlier.

3. Referential Integrity

- We use **referential integrity** to ensure 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.
- For 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 name “Perryridge”. That is a foreign key should either be null or refer to an existing tuple.
- 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 is also called subset dependency since it can be written as $\prod_{\alpha} (r_2) \subseteq \prod_{K1} (r_1)$.

Referential Integrity (Cont.)

- **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).
- **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.

Referential Integrity (Cont.)

- 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

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

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

- **Referential Integrity in SQL:**

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 (Cont.)

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

create table *depositor*
(customer-name char(20)**,**
account-number char(10)**,**
primary key *(customer-name, account-number),*
foreign key *(account-number)* **references** *account*
on delete cascade
on update cascade*,*
foreign key *(customer-name)* **references** *customer*
on delete cascade
on update cascade*)*

Referential Integrity (Cont.)

- Alternatively, we can use **on delete set null** and **on update set null**.
- Also, we can use **on delete set default** and **on update set default**.

4. Assertions

- An assertion is a predicate expressing a condition that we wish the database always to satisfy.
- **Domain constraints** and **referential integrity constraints** are special form of assertions.
- However, there are many constraints that we cannot express by using only these special forms.
- 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.

Assertions (Cont.)

- For example, to ensure that the sum of all loan amounts for each branch must be less than the sum of all account balances at the branch, we write`

```
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)))
```

5. Triggers

- A **trigge**r is a statement that the system executes automatically as a side effect of a modification to the database.
- To design a trigger mechanism, we must meet two requirements:
  - Specify when a trigger is to be executed. This is broken up into an **event** that causes the trigger to be checked and a **condition** that must be satisfied for trigger execution to proceed.
  - Specify the **actions** to be taken when the trigger executes.
- Triggers are useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met.
- For 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

Triggers (Cont.)

- The condition for executing the trigger is an update to the *account* relation that results in a negative *balance* value.

```
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
```

- Here in above syntax,
- Trigger definition specifies that trigger is initiated **after any update** on the relation account is executed.
- < **referencing new row as** *nrow* > clause creates a variable nrow which stores the value of an updated row after the updates.
- < **for each row** > clause would explicitly iterate over each updated row .
- <**when**> statement specifies condition namely nrow.balance < 0.
- **<begin atomic >…….<end >** contains trigger body.

Triggers (Cont.)

- Triggering event can be **insert**, **delete** instead of **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
```

Triggers (Cont.)

- 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

6. Security

- **Security** - protection from malicious attempts to steal or modify data.
  - Database system level
    - *Authentication* and *authorization* mechanisms to allow specific users access only to required data
    - We concentrate on authorization in the rest of this chapter
  - Operating system level
    - Operating system super-users can do anything they want to the database! Good operating system level security is required.
  - Network level: must use encryption to prevent
    - Eavesdropping (unauthorized reading of messages)
    - Masquerading (pretending to be an authorized user or sending messages supposedly from authorized users)

Security (Cont.)

- Physical level
  - Physical access to computers allows destruction of data by intruders; traditional lock-and-key security is needed
  - Computers must also be protected from floods, fire, etc.
- Human level
  - Users must be screened to ensure that an authorized users do not give access to intruders
  - Users should be trained on password selection and secrecy

7. Authorization

Forms of authorization on parts of the database:

- **Read authorization** - allows reading, but not modification of data.
- **Insert authorization** - allows insertion of new data, but not modification of existing data.
- **Update authorization** - allows modification, but not deletion of data.
- **Delete authorization** - allows deletion of data

Forms of authorization to modify the database schema:

- **Index authorization** - allows creation and deletion of indices.
- **Resources authorization** - allows creation of new relations.
- **Alteration authorization** - allows addition or deletion of attributes in a relation.
- **Drop authorization** - allows deletion of relations.

8. Authorization and Views

- Users can be given authorization on views, without being given any authorization on the relations used in the view definition
- Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job
- A combination or relational-level security and view-level security can be used to limit a user's access to precisely the data that user needs.
- Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.
  - Approach: Deny direct access to the *loan* relation, but grant access to the view *cust-loan*, which consists only of the names of customers and the branches at which they have a loan.
  - The *cust-loan* view is defined in SQL as follows:

```
create view cust-loan as
    select branchname, customer-name
    from   borrower, loan
    where borrower.loan-number = loan.loan-number
```

Authorization and Views (Cont.)

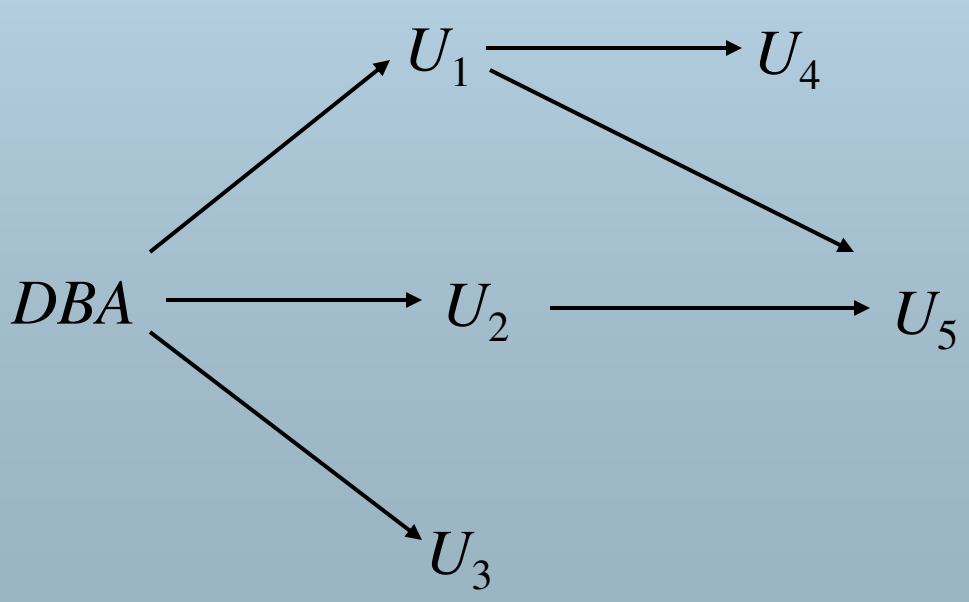
- The clerk is authorized to see the result of the query:

  select *
  from *cust-loan*

- When the query processor translates the result into a query on the actual relations in the database, we obtain a query on *borrower* and *loan*.
- Authorization must be checked on the clerk's query before query processing replaces a view by the definition of the view.
- Creation of view does not require **resources** authorization since no real relation is being created
- The creator of a view gets only those privileges that provide no additional authorization beyond that he already had.
- E.g. if creator of view *cust-loan* had only **read** authorization on *borrower* and *loan*, he gets only **read** authorization on *cust-loan*

9. Granting of Privileges

- The passage of authorization from one user to another may be represented by an **authorization graph**.
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- Consider graph for update authorization on loan.
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j.



Granting of Privileges

- *Requirement*: All edges in an authorization graph must be part of some path originating with the database administrator
- If DBA revokes grant from U_1:
  - Grant must be revoked from U_4 since U_1 no longer has authorization
  - Grant must not be revoked from U_5 since U_5 has another authorization path from DBA through U_2
- Must prevent cycles of grants with no path from the root:
  - DBA grants authorization to U_7
  - U7 grants authorization to U_8
  - U8 grants authorization to U_7
  - DBA revokes authorization from U_7
- Must revoke grant U_7 to U_8 and from U_8 to U_7 since there is no path from DBA to U_7 or to U_8 anymore.

10. Authorization in SQL

- The grant statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> to <user list>

- <user list> is:
  - a user-id
  - *public*, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).
- **select:** allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *branch* relation:

    grant select on *branch* **to** U_1, U_2, U_3

Authorization in SQL (Cont.)

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **references**: ability to declare foreign keys when creating relations.
- **usage**: In SQL-92; authorizes a user to use a specified domain
- **all privileges**: used as a short form for all the allowable privileges
- **with grant option**: allows a user who is granted a privilege to pass the privilege on to other users.
  - Example:

    grant select on *branch* **to** U_1 **with grant option**

    gives U_1 the **select** privileges on branch and allows U_1 to grant this privilege to others

11. Roles

- Roles permit common privileges for a class of users can be specified just once by creating a corresponding “role”
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles
- SQL:1999 supports roles

create role *teller*
create role *manager*

grant select on *branch* **to** *teller*
grant update (*balance***) on** *account* **to** *teller*
grant all privileges on *account* **to** *manager*

grant *teller* **to** *manager*

grant *teller* **to** *alice, bob*
grant *manager* **to** *avi*

12. Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke<privilege list>

  on <relation name or view name> **from** <user list>
  [**restrict**|**cascade**]

- Example:

  revoke select on *branch* **from** U_1, U_2, U_3 **cascade**

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the **revoke**.

- We can prevent cascading by specifying **restrict**:

  revoke select on *branch* **from** U_1, U_2, U_3 **restrict**

  With **restrict**, the **revoke** command fails if cascading revokes are required.

Revoking Authorization in SQL (Cont.)

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.