SQL – Other statements

Aims:

At the end of this group of three lectures, you should be able to use SQL to manage the database.

Read: Elmasri & Navathe, Chapters 4, 5, 24 & 25 (6th ed.)
or Chapters 6, 7, 10 & 30 (7th ed.)

Overview

1. Modifying data (INSERT,UPDATE,DELETE)
2. Views
3. Indexes
4. Integrity component
5. Programmed access to databases
6. Additional features of SQL
7. Database security in SQL

INSERT

- Used to add one or more tuples to a relation
- Attribute values should be listed in the same order as the attributes were specified in the CREATE TABLE command
- INSERT INTO table_name

  VALUES (list_of_values);

- **Example 1:** *Enter data about a new customer.*

INSERT (CONT.)

- An alternate form of INSERT specifies explicitly the attribute names that correspond to the values in the new tuple
- Attributes with NULL values can be left out
- INSERT INTO table_name(att_list)
  VALUES (list_of_values);

- **Example 2:** *Enter data about a new customer if only the customer's number and name are known.*

Entering multiple tuples

- Another variation of INSERT allows insertion of *multiple tuples* resulting from a query into a relation
- INSERT INTO table_name(att_list)

  select_statement;

- **Example 3:** *Create a table that holds the number, title, year and director's name for every movie.*

Notes on entering values

- Enclose values in single quotes for any attribute whose type is character or date
- If you need an apostrophe within the attribute value, type two single quote marks. e.g. ‘O’’Toole’

DELETE

- Removes tuples from a relation
- Includes a WHERE clause to select the tuples to be deleted
- Tuples are deleted from only *one table* at a time (unless CASCADE is specified on a referential integrity constraint)
- A missing WHERE clause specifies that *all tuples* in the relation are to be deleted; the table then becomes an empty table
- The number of tuples deleted depends on the number of tuples in the relation that satisfy the WHERE clause
- Referential integrity should be enforced

DELETE (CONT.)

- DELETE FROM table_name

  [WHERE condition];

- **Example 4:** *Delete information about the customer 127.*

- **Example 5:** *Delete the contents of the STARS table.*

UPDATE

- Used to modify attribute values of one or more selected tuples in a single table
- WHERE clause selects the tuples to be modified
- An additional SET clause specifies the attributes to be modified and their new values

- UPDATE table_name

  SET att_name=expression

  [WHERE condition];

UPDATE (CONT.)

- **Example 6:** *Increase the bonus of each customer by 5.*
- **Example 7:** *Change the last name of customer 303 to Simmons.*
- **Example 8:** *Change the number of Academy Awards of the movie no 503 to 3, and its rating to 4.*

VIEWS

- Creating views:

  Define a view called MOVDIR which consists of the movie number, movie title, director number and director name for all movies of type comedy.

- Dropping views

- Querying views

Updatable views

- A view with a single defining table is updatable if the view contains the primary key or a candidate key of the base relation.
- Views defined on multiple tables are generally not updatable.
- Views defined using grouping, aggregate functions, DISTINCT, IN, ANY, ALL, or EXISTS are not updatable.

Updatable views in Oracle

- single-table views without aggregate functions, GROUP BY, set operations and DISTINCT
- Only some of the multitable views are updatable. If the view contains the primary key of a table, it may be possible to modify one stored table. Information on updatable attributes can be obtained from USER_UPDATABLE_COLUMNS table.
- *select column_name, updatable*
  from user_updatable_columns
  where table_name = upper('view_name');

INDEXES

- Access structures, used to speed up retrieval
- CREATE INDEX index_name
  ON table_name (att_list);
- Example: *Create an index CNO on the NUMBER attribute of the CUSTOMER table.*

- Creating an index on more than one column
  Create an index CNAME on the LNAME and FNAME attributes of the CUSTOMER table.

- Specifying a key constraint (older versions of SQL)
  Specify an index on the key attribute NUMBER of the MOVIE table.

INDEXES (CONT.)

- Deleting an index

  DROP INDEX index_name;

- Create indexes before tables are populated
- Oracle creates indexes automatically for primary and secondary keys
- Created indexes on attributes used often in search and join conditions
- Do not create indexes on attributes that are frequently updated

Integrity Enhancement Feature

- Introduced in SQL 92
- Types of integrity constraints
  - Required data (NOT NULL)
  - Domain constraints
  - Entity integrity (PRIMARY KEY, UNIQUE)
  - Referential integrity (FOREIGN KEY, REFERENCES)
  - Referential integrity constraint violation
  - Enterprise constraints

Domain constraints

- CHECK condition
- Example: *sex char not null CHECK (sex IN ('M','F'))*
- CREATE DOMAIN domain_name [AS] data_type
  [DEFAULT default_option]
  [CHECK (search_option)]
- Example:

  CREATE DOMAIN sex_type AS CHAR
  CHECK (VALUE IN ('M','F'))
- DROP DOMAIN domain_name
  [RESTRICT | CASCADE]
- Not supported by Oracle!

Referential integrity violation

- Qualifiers: ON DELETE, ON UPDATE
- Options: SET NULL, CASCADE, SET DEFAULT
- Example:

  create table MOVIE

  (MNUMBER integer not null primary key, …

  DIRECTOR integer,

  constraint DIRFK foreign key (DIRECTOR) references Director (dnumber)

  ON DELETE SET NULL

  ON UPDATE CASCADE);

- Oracle supports ON DELETE CASCADE and ON DELETE SET NULL

Enterprise constraints

- Table/attributes constraints (CHECK)
- In SQL2, the condition specified for CHECK may include nested SELECT statement.
- Oracle does not support nested queries in constraints
- **Assertions**: integrity constraints over several tables
- Table constraints are required to hold only if associated table is non-empty.
- Not supported by Oracle!
- CREATE ASSERTION assertion_name }
  [CHECK (search_option)]}

Assertions (cont)

- Example:

  The salary of an employee must not be greater than the salary of the manager of the department that the employee works for.

```
create assertion SALARY_CONSTRAINT
check (not exists
        (select *
         from employee e, employee m, department d
         where e.salary>m.salary and
                 e.dno=d.dnumber and
                 d.mgrssn=m.ssn));
```

Other Integrity Checking And Processing in Oracle

- Triggers (Chapter 24, pp. 869-873)
- Procedures

Triggers

- Triggers monitor a database and initiate actions when a condition occurs
- Triggers can be used to:
  - calculate or update values of derived attributes
  - enforce additional constraints
  - prevent invalid transactions
  - automatically perform actions
  - audit changes to data
  - maintain replicated tables

Parts of a trigger

- **Event** - a change to the db that activates the trigger:
  - DML event (UPDATE, INSERT or DELETE),
  - DDL event (e.g. CREATE ON db_name),
  - database event (e.g. SERVERERROR ON DATABASE)
- **Condition** - a test or a query which is run when the trigger is activated
- **Action** - one or more statements (or procedures) that are executed when the trigger is activated and its condition is satisfied

Triggers (cont)

- **Type**
  - Row-level triggers: execute once for each row in a transaction
  - Statement-level triggers: once for each transaction
- **Timing of execution**
  - BEFORE
  - AFTER
  - INSTEAD OF
- SQL 99 defines only BEFORE and AFTER triggers

Trigger types for DML events

- before insert row
- before insert statement
- after insert row
- after insert statement
- before update row
- before update statement
- after update row
- after update statement
- before delete row
- before delete statement
- after delete row
- after delete statement
- instead of row
- instead of statement

CREATE TRIGGER STATEMENT

```
create [or replace] trigger [schema.]trigger
   {before | after | instead of}
   {delete | insert | update [of column [, column] ...]}
   [or {delete | insert | update  [of column [, column]
   ...]}] ...
   on [schema.]{table-name | view-name}
   [ [referencing {old [as] old | new [as] new} ...]
   for each [row | statement]
   [when (condition)] ]
   PL/SQL block
/
```

Rules for triggers

- The name of each trigger must be unique within the DB
- The body of the trigger cannot contain DDL statements or transaction control statements
- The trigger cannot read from and modify the mutating table (generally; there are some exceptions)
- The INSTEAD OF triggers can only be defined on views
- The WHEN clause is only available for row-level triggers defined as before or after triggers. It specifies a condition to be checked after the trigger fires, but before the body of the trigger is executed. The condition specified in the WHEN clause cannot be a query.

Triggers (cont)

- DROP TRIGGER [schema.]trigger_name;
- ALTER TRIGGER [schema.]trigger_name
  {ENABLE | DISABLE | COMPILE [DEBUG]}
- It is not possible to alter the body of a trigger - in that case, the *create or replace trigger* statement must be used. A trigger may invoke a procedure within its body, using the *execute* command.
- To see information about compilation errors, use:

  show errors [trigger trigger_name]

EXAMPLE

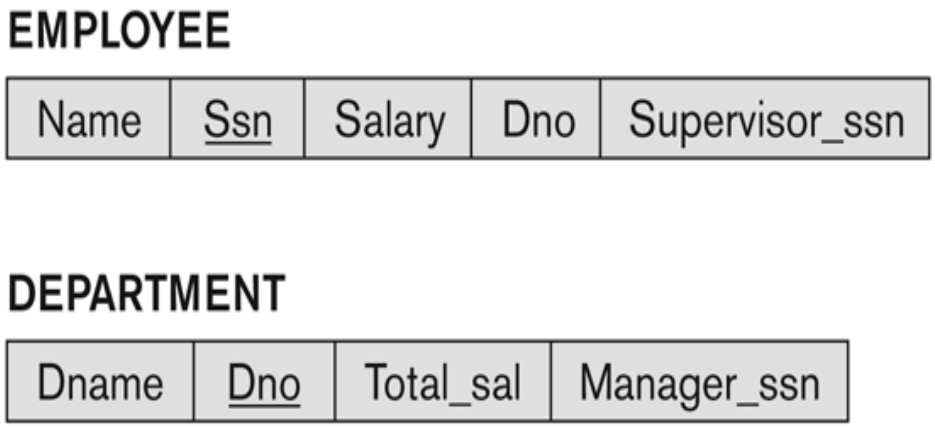

Figure 24.1
A simplified COMPANY database used for active rule examples.

- TOTAL_SAL is a derived attribute; its value must be changed whenever there is an INSERT, UPDATE or DELETE operation

Example (cont)

- Inserting new employees

```
create trigger totalsal1
after insert on employee
for each row
when (new.dno is not null)
        update department
        set total_sal = total_sal + :new.salary
        where dno = :new.dno
/
```

Example (cont)

- Salary is modified:

```
create trigger totalsal2
after update of salary on employee
for each row
   when (new.dno is not null)
   update department
   set total_sal = total_sal + :new.salary - :old.salary
   where dno=:new.dno
 /
```

Example (cont)

- Some tuples are deleted from EMPLOYEE

```
create trigger totalsal3
after delete on employee
for each row
when (old.dno is not null)
        update department
        set total_sal = total_sal - :old.salary
        where dno=:old.dno
 /
```

Example (cont)

- Employees change departments;

```
create trigger totalsal4
after update of dno on employee
for each row
begin
        update department
        set total_sal = total_sal + :new.salary
        where dno=:new.dno;
        update department
        set total_sal = total_sal - :old.salary
        where dno = :old.dno;
end;
/
```

An example of the audit trigger

```
CREATE TRIGGER audit_trigger BEFORE INSERT OR DELETE OR UPDATE
    ON classified_table
for each row
begin
  if INSERTING then
      insert into audit_table
      values (USER || ' is inserting' || ' new key: ' || :new.key);
  elsif DELETING then
      insert into audit_table
      values (USER || ' is deleting' || ' old key: ' || :old.key);
  elsif UPDATING('FORMULA') then
      insert into audit_table
      values (USER || ' is updating' || ' old formula: ' || :old.formula || ' new formula: '
    || :new.formula);
  elsif UPDATING then
     insert into audit_table
     values (USER || ' is updating' || ' old key: ' || :old.key || ' new key: ' || :new.key);
  end if;
end;
```

Another example (lab test 2008)

- Three tables given:

  BUYER (Id, Name, Credit)

  COMP (Code, Name, Manufacturer, Unit, Cost, Last_Ordered, Quantity)

  ORDERS (Onumber, Customer, Odate, Total)

- The task: create another table
  - ORDER_ITEM (Order, Product, Quantity, Product_Cost)

LAB TEST 2008 (CONT)

```
create table ORDER_ITEM
(order integer not null references orders,
 product integer not null references comp,
 quantity integer not null
      constraint check_quan check (quantity>0),
 product_cost float not null,
 primary key (order, product));
```

Problem: Cannot specify the constraint to check Product_cost!

LAB TEST 2008 (CONT)

```
create or replace trigger CHECK_PRICE
BEFORE INSERT on ORDER_ITEM
for each row
    declare
    product_cost float;

begin
    select cost into product_cost
    from comp
    where code=:new.product;

    if :new.cost <> (product_cost * :new.quantity) then
            raise_application_error (num => -20000,
                msg => 'Cannot add this tuple: the price is incorrect!');
    end if;
end;
/
```

Preventing invalid transactions

- BEFORE triggers
- Check for an appropriate situation
- Prevent the statement from being executed by raising an error
- *RAISE_APPLICATION_ERROR*

  (num=> -20107,

  msg=> 'Duplicate customer or order ID');

LAB TEST 2011 - BANK

```
create table ACCOUNT
(Acc_No integer not null check (ACC_No>0),
 Type varchar(7) not null check (Type in ('savings','cheque')),
 Branch varchar(7) not null,
 Customer char(9) not null references CUST,
 Open_Date date not null,
 Status varchar(7) not null check (Status in ('active','closed','frozen')),
 Closed_Date date,
 constraint pk_account primary key (Acc_No),
 constraint check_dates check (Closed_Date > Open_Date));
```

LAB TEST 2011 – BANK (CONT.)

- Constraint: a customer can have only one account of each type.
- This can be done with a constraint or with a trigger

```
ALTER TABLE ACCOUNT
add constraint one_account
unique(type,branch,customer);
```

LAB TEST 2011 – BANK (CONT.)

```
create or replace trigger INSERT_ACCOUNT
before insert on ACCOUNT
for each row
declare
  ac_num integer;
begin
select count(*) into ac_num
from ACCOUNT
where Customer=:new.Customer and Type = :new.Type;
if ac_num = 1 then
   Raise_application_error(-20201, 'An account of this type already
exists');
end if;
end;
/
```

Procedures

- Procedure: a database object made up of a collection of statements.
- Procedures can be used with triggers to enforce integrity constraints, and can also be used to specify complex business rules.

- create [or replace] procedure [schema.]proc_name
  [(argument [IN | OUT | IN OUT] datatype
  [, argument [IN | OUT | IN OUT] datatype]...)]
  {IS | AS}
  {block | external program};

PROCEDURES (CONT)

- Example: a procedure that inserts a new tuple into a table.

```
create procedure NEW_EMPLOYEE
   (Person_Name varchar(50))
AS
begin
insert into employee (Name, Age, Address)
values (Person_Name, null, null);
end;
```

Procedures (cont)

- PL/SQL blocks within procedures can include any DML statement, but DDL statements are not allowed.
- DROP PROCEDURE [schema.]procedure_name;
- ALTER PROCEDURE [schema.]procedure_name COMPILE
- EXECUTE procedure_name [arguments];
- *execute NEW_EMPLOYEE('Adam Brown');*

Programmed access to databases

- SQL can also be used for programmed access to databases, via
  - PL/SQL
  - Embedded SQL
  - ODBC/JDBC
  - tools (reports, forms, WWW interfaces)

PL/SQL

- Oracle's procedural language extension to SQL
- PL/SQL block has the following structure:

```
[DECLARE
  -- declarations]
BEGIN
  -- statements
[EXCEPTION
  -- handlers]
END;
```

- Control statements: *if-then-else, for-loop, while-loop, exit-when, go-to*
- The statement list must not contain DDL statements, nor any commands involving other procedures. SELECT command inside a procedure must assign their results to local variables, the names of which are preceded with a colon

Additional features of SQL

- Security and authorization
- Object-relational support
  - row type
    CREATE ROW TYPE row_type_name (declarations);
    CREATE TABLE table_name OF TYPE row_type_name;
  - inheritance, overloading, ...
- Transaction support
- Support for temporal data

Security and authorization

- Chapter 25

- Database security
  - Legal and ethical issues
  - Policy issues
  - System-related issues
  - The need to identify multiple security levels

- Threats to databases
  - Loss of **integrity**
  - Loss of **availability**
  - Loss of **confidentiality**

Security and authorization (cont)

- Security and authorization subsystem is responsible for ensuring security
- Two types of database security mechanisms:
  - **Discretionary** security mechanisms
  - **Mandatory** security mechanisms
- Control measures
  - Access control
  - Inference control
  - Flow control
  - Encryption

Database Security and the DBA

- DBA is responsible for the overall security of DB
- The DBA's responsibilities include
  - granting privileges to users who need to use the system
  - classifying users and data in accordance with the policy of the organization

- DBA account (system or superuser)
  - Account creation (access control)
  - Privilege granting (discretionary)
  - Privilege revocation (discretionary)
  - Security level assignment (mandatory)

Types of Discretionary Privileges

- The **account level**:
  - At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database
  - Examples: create table, create schema, create trigger, alter, drop, modify, select, …
- The **relation level** (or **table level**):
  - At this level, the DBA can control the privilege to access each individual relation or view in the database

Discretionary Privileges in SQL

- To control the granting and revoking of relation privileges, each relation R in a database is assigned an **owner account**, which is typically the account that was used when the relation was created in the first place
- The owner of a relation is given all privileges on that relation
- In SQL2, the DBA can assign an owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the **CREATE SCHEMA** command
- The owner account holder can **pass privileges** on any of the owned relation to other users by **granting** privileges to their accounts

Specifying Privileges Using Views

- The mechanism of **views** is an important discretionary authorization mechanism
- If the owner A of a relation R wants another account B to be able to retrieve only some fields of R, then A can create a view V of R that includes only those attributes and then grant SELECT on V to B
- The same applies to limiting B to retrieving only certain tuples of R; a view V' can be created by defining a query that selects only those tuples from R that A wants to allow B to access

Granting and Revoking Privileges

- GRANT {account_priv | obj_priv | ALL [PRIVILEGES]}
  [ON {object}]
  TO {user | role | PUBLIC}
  {WITH GRANT OPTION};

- REVOKE {system_priv | obj_priv | ALL [PRIVILEGES]}
  [ON {object}]
  FROM {user | role | PUBLIC};

Table privileges

- SELECT
- INSERT, UPDATE
  can specify that only certain attributes can be updated
- ALTER
- DELETE
- INDEX
- DEBUG
- REFERENCES (can be **restricted** to specific attributes)

Propagation of Privileges

- Whenever the owner A of a relation R grants a privilege on R to another account B, privilege can be given to B with or without the **GRANT OPTION**.
- B can also grant that privilege on R to other accounts.
  - Suppose that B is given the **GRANT OPTION** by A and that B then grants the privilege on R to a third account C, also with **GRANT OPTION**. In this way, privileges on R can **propagate** to other accounts without the knowledge of the owner of R.
  - If the owner account A now revokes the privilege granted to B, all the privileges that B propagated based on that privilege should automatically be revoked by the system.

Example

- Suppose that the DBA creates four accounts
  - A1, A2, A3, A4
- and wants only A1 to be able to create base relations

  GRANT CREATE TABLE TO A1;
- In SQL2 the same effect can be accomplished by having the DBA issue a **CREATE SCHEMA** command as follows:

  CREATE SCHEMA EXAMPLE

  AUTHORIZATION A1;

EXAMPLE (2)

- User account A1 can create tables under the schema called **EXAMPLE**
- Suppose that A1 **creates** the two base relations **EMPLOYEE** and **DEPARTMENT**
  - A1 is then **owner** of these two relations and hence all the relation privileges on each of them.
- Suppose that A1 wants to grant A2 the privilege to insert and delete tuples in both of these relations, but A1 does not want A2 to be able to propagate these privileges to additional accounts:

GRANT INSERT, DELETE ON

EMPLOYEE, DEPARTMENT **TO A2;**

EXAMPLE (3)

EMPLOYEE

Name	Ssn	Bdate	Address	Sex	Salary	Dno

DEPARTMENT

Dnumber	Dname	Mgr_ssn

Figure 24.1
Schemas for the two relations EMPLOYEE and DEPARTMENT.

EXAMPLE (4)

- Suppose that A1 wants to allow A3 to retrieve information from either table and also to be able to propagate the SELECT privilege to other accounts.

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT
    TO A3 WITH GRANT OPTION;
```

- A3 can grant the **SELECT** privilege on the **EMPLOYEE** relation to A4 by issuing:

```
GRANT SELECT ON EMPLOYEE TO A4;
```

  - Notice that A4 can't propagate the SELECT privilege because GRANT OPTION was not given to A4

Example (5)

- Suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3:

  REVOKE SELECT ON EMPLOYEE **FROM** A3**;**

- The DBMS must now automatically revoke the SELECT privilege on EMPLOYEE from A4, too, because A3 granted that privilege to A4 and A3 does not have the privilege any more.

EXAMPLE (6)

- Suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege.
  - The limitation is to retrieve only the NAME, BDATE, and ADDRESS attributes and only for the tuples with DNO=5.
- A1 then create the view:

```
CREATE VIEW A3EMPLOYEE AS
    SELECT NAME, BDATE, ADDRESS
    FROM EMPLOYEE
    WHERE DNO = 5;
```

- After the view is created, A1 can grant **SELECT** on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3
        WITH GRANT OPTION;
```

Example (7)

- Finally, suppose that A1 wants to allow A4 to update only the SALARY attribute of EMPLOYEE;

GRANT UPDATE ON EMPLOYEE **(**SALARY**)**
TO A4**;**