Database System Concepts

Chapter 4(+8): Advanced SQL

Departamento de Engenharia Informática
Instituto Superior Técnico

1st Semester
2009/2010

Slides (fortemente) baseados nos slides oficiais do livro "Database System Concepts"

Outline

Outline

Time and Dates

- date: Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- time: Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' or **time** '09:00:30.75'
- timestamp: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- interval: period of time
  - Example: **interval 1 day**
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Time and Dates (cont.)

- Can extract values of individual fields from date/time/timestamp
  - Example: **extract (year from** *r.starttime*)
- Can cast string types to date/time/timestamp
  - **cast** <string-valued-expression> **as date**
    - **cast** '2005-7-27' **as date**
  - **cast** <string-valued-expression> **as time**
    - **cast** '09:00:30' **as time**

Large-Object Types

Large objects (photos, videos, CAD files, etc.) are stored as a large object:

- blob: binary large object — object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
- clob: character large object — object is a large collection of character data

When a query returns a large object, a pointer is returned rather than the large object itself.

User-Defined Types

- create type construct in SQL creates user-defined type

create type *Dollars* **as numeric(12,2) final**

- create domain construct in SQL-92 creates user-defined domain types

create domain *person_name* **char(20) not null**

- Types and domains are similar.
  - Domains can have constraints, such as *not null*, specified on them.

Outline

Domain Constraints

- 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
  - Example:

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

  - We cannot assign or compare a value of type Dollars to a value of type Euros. However, we can convert types:

cast r.A as Euros

(Note: In this case, we should also multiply by the dollar-to-euro conversion-rate)

Integrity 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.

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

Integrity Constraints — Unique

unique($A_1, A_2, \ldots, A_m$)

- The unique specification states that the attributes $A_1, A_2, \ldots, Am$ form a candidate key
- Example:

```
create table branch (
    branch_name char(15),
    branch_city char(30),
    assets integer not null,
    primary key (branch_name)
    unique(branch_city)
)
```

Integrity Constraints - Check

check (P)

- Example: Ensure that the values of assets are non-negative.

```
create table branch (
    branch_name char(15),
    branch_city char(30),
    assets integer,
    primary key (branch_name),
    check (assets >= 0)
)
```

Integrity Constraints - Check (cont.)

- The **check** clause in permits domains to be restricted
  - Example: 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* $>= 4.0$**)**

- The domain has a constraint that ensures that the hourly_wage is greater than 4.0
- 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'.
- Foreign keys can be specified as part of the SQL create table statement:
  - The primary key lists attributes that comprise the primary key
  - The unique 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 Example

create table *customer* **(**
customer_name **char(20),**
customer_street **char(30),**
customer_city **char(30),**
primary key (*customer_name***))**

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***)**

Deleting and Updating Foreign Keys

- Actions can be associated with foreign keys, when the target row is deleted or updated
  - use **on delete** <action> or **on update** <action>
- The action can be on of
  - cascade: deletes/updates the rows that point to the deleted row
  - set null: sets to *null* the rows that point to the deleted/updated row
  - set default: sets to the default value the rows that point to the deleted/updated row
  - no action: stops the deletion/update from occurring

foreign key (*customer_name*) **references** *customer*
on delete cascade

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.

Assertions (cont.)

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

Assertions (cont.)

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(balance)
            from account
            where account.branch_name = branch.branch_name)))
```

Outline

Forms of Authorization

Forms of authorization on parts of the database:

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

Forms of authorization to modify the database schema:

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

Authorization Specification

- The grant statement is used to confer authorization

  grant <privilege list>
  on <relation/view name> **to** <user list>

- <user list> is:
  - user id
  - public, which allows all valid users the privilege granted
  - a *role*
- 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)

Privileges in SQL

- select: allows read access to relation,or the ability to query using the view
  - Example: grant users U1, U2, and U3 select authorization on the branch relation:

grant select on *branch* **to** $U1, U2, U3$

- 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: authorizes a user to use a specified domain
- all privileges: used as a short form for all the allowable privileges

Privilege To Grant 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** $U1$ **with grant option**

gives $U1$ the select privileges on branch and allows $U1$ to grant this privilege to others

Revoking Authorization

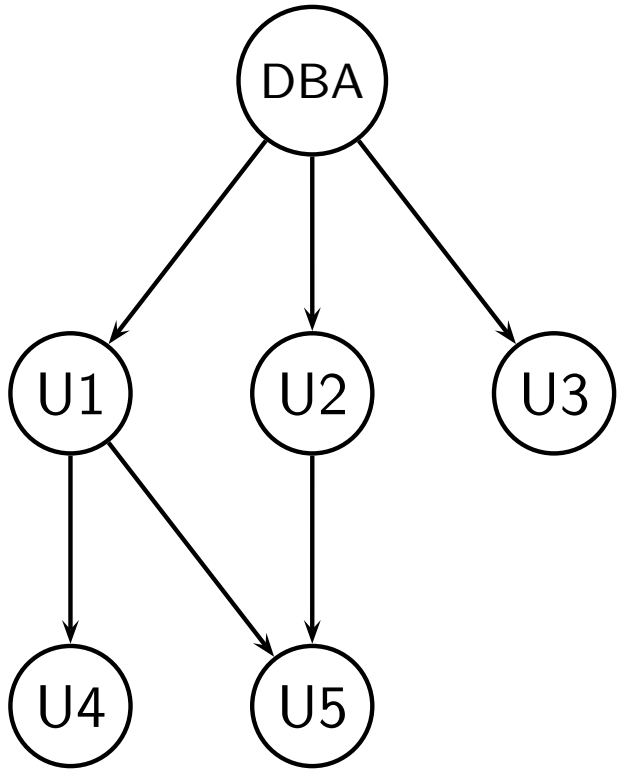
- The revoke statement is used to revoke authorization
  revoke <privilege list> **on** <relation/view name>
  from <user list> **[restrict|cascade]**
- Example:

revoke select on *branch* **from** *U*1, *U*2, *U*3 **cascade**

  - <privilege-list> may be all to revoke all privileges
  - If <user-list> includes public, all users lose the privilege except those granted it explicitly
- Revocation of a privilege from a user may cause other users also to lose that privilege (cascading)
  - With *restrict*, the revoke command fails if cascading revokes are required
- All privileges that depend on the privilege being revoked are also revoked

Granting of Privileges

- The passage of authorization from one user to another may be represented by an authorization graph



- Requirement: All edges in an authorization graph must be part of some path originating with the database administrator
- Must prevent cycles of grants with no path from the root

Authorization and Views

- Users can be given authorization on views, without being given any authorization on the relations used in the view definition
- Example: 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 branch_name, 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
```

- 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.
  - I.e. if creator of view *cust_loan* had only read authorization on *borrower* and *loan*, he gets only read authorization on *cust_loan*

Roles

- Roles permit common privileges for a class of users
- Privileges can be granted to or revoked from roles, just like users
- Roles can be assigned to users, and even to other 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** *john*

Limitations of SQL Authorization

- SQL does not support authorization at a tuple level
  - E.g. we cannot restrict students to see only (the tuples storing) their own grades
- All end-users of an application (such as a web application) may be mapped to a single database user
- The task of authorization in above cases falls on the application program, with no support from SQL
  - Benefit: fine grained authorizations, such as to individual tuples, can be implemented by the application.
  - Drawback: Authorization must be done in application code, and may be dispersed all over an application
  - Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code

Outline

Functions and Procedures

- SQL supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions are particularly useful with specialized data types such as images and geometric objects
  - Example: functions to check if polygons overlap, or to compare images for similarity
  - Some database systems support table-valued functions, which can return a relation as a result
- SQL also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from standard SQL

SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```
create function account_count(customer_name varchar(20))
    returns integer
begin
    declare a_count integer;
    select count(*) into a_count
    from depositor
    where depositor.customer_name = customer_name;
    return a_count
end
```

- Find each customer that has more than one account.

```
select customer_name, customer_street, customer_city
from customer
where account_count(customer_name) > 1
```

Table Functions

- SQL:2003 added functions that return a relation as a result
  - Example: Return all accounts owned by a given customer

```
create function accounts_of (customer_name char(20))
    returns table(account_number char(10), branch_name char(15),
        balance numeric(12,2))
return table (
    select account_number, branch_name, balance
    from account, depositor
    where depositor.customer_name = accounts_of.customer_name
        and depositor.account_number = account.account_number)
```

- Usage:

```
select *
from table(accounts_of ('Smith'))
```

SQL Procedures

- The *account_count* function could instead be written as procedure:

```
create procedure account_count(in customer_name char(20),
        out a_count integer)
begin
    select count(*) into a_count
    from depositor
    where depositor.customer_name = account_count.customer_name
end
```

- Procedures can be invoked using the call statement:

```
declare a_count integer;
call account_count('Smith', a_count);
```

Procedural Constructs

- Compound statements:

begin
<SQL statements>
end

  - May contain multiple SQL statements
  - Local variables can be declared

- Declaring variables

declare <variable> <type> **[default** <value>**]**

  - The scope of a local variable is within the block where it is declared

Procedural Constructs (cont.)

- Assigning values

set <variable> **=** **expression**

- The select into statement

select <columns> **into** <variable>
<table expression>

Conditional statements

- The if-then-else statement

```
if r.balance < 1000 then
    set l = l + r.balance
elseif r.balance < 5000 then
    set m = m + r.balance
else
    set h = h + r.balance
end if
```

- The case statement

```
case
    when r.balance < 1000 then
        set l = l + r.balance
    else set h = h + r.balance
end case
```

  - There is also a *case* statement similar to the one in *C*

Exceptions

- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition;
declare exit handler for out_of_stock
begin
   ...
   signal out_of_stock
end;
```

- The handler here is exit— causes enclosing block to be exited
- Other actions possible on exception (continue, undo, ...)

Loops

- While and repeat statements:

```
declare n integer default 0;
while n < 10 do
    set n = n + 1
end while
```

```
declare n integer default 10;
repeat
    set n = n − 1
until n = 0
end repeat
```

Loops (cont.)

- The for loop
  - Permits iteration over all results of a query
  - Example: find total of all balances at the Perryridge branch

```
declare n integer default 0;
for r as
    select balance
    from account
    where branch_name = 'Perryridge'
do
    set n = n + r.balance
end for
```

Cursors


```
declare n integer default 0;
declare r integer default 0;
declare done integer default 0;
declare c cursor for
    select balance from account
    where branch_name = 'Perryridge';
declare continue handler for sqlstate '02000' set done = 1;

open c;
repeat
    fetch c into r;
    if not done then
        set n = n + r
    end if
until done
end repeat;
close c;
```

Outline

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
  - creating a loan in the amount of the overdraft
  - giving this loan a loan number identical to the account number of the overdrawn account
  - setting the account balance to zero
- The condition for executing the trigger is an update to the account relation that results in a negative balance value

Trigger Example


```
create trigger overdraft after update on account
for each row
when new.balance < 0
begin
   insert into borrower
      (select customer_name, account_number
      from depositor
      where new.account_number = depositor.account_number);
   insert into loan values (new.account_number,
         new.branch_name, −new.balance);
   update account
   set balance = 0
   where account.account_number = new.account_number
end
```

Triggering Events and Actions

- Triggering event can be insert, delete or update
- Triggers on update can be restricted to specific attributes
  - **create trigger** *overdraft* **after update of** *balance* **on** *account*
- Values of attributes before and after an update can be referenced
  - **referencing new row as** <var> : for inserts and updates
  - **referencing old row as** <var> : for deletes and updates
- Triggering times can be before or after
- Trigger can be at row level or statement level
  - use **for each row** or **for each statement**
  - At statement level, **old** and **new** refer to temporary tables (called transition tables) containing the affected rows
  - Statement level can be more efficient when dealing with SQL statements that update a large number of rows
- If the trigger fails, the triggering statement fails

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

Outline

Example Function

Implement an SQL stored function that, given a branch name, returns the percentage of assets that branch represents to the bank.

Example Function


```
create function percent(branch_name varchar(50))
        returns numeric(5,2)
begin
    declare total double;
    declare branch_assets double;

    select sum(assets) into total
    from branch b;

    select assets into branch_assets
    from branch b
    where b.branch_name = branch_name;

    return branch_assets/total * 100;
end
```

Example Procedure

Implement an SQL stored procedure that takes as input the name of a city and returns the names of the two customers with accounts in the given city with the highest amount of money deposited.

Example Procedure


```
create procedure top_customer(in city varchar(50), out top1 varchar(50),
                              out top2 varchar(50))
begin
   declare done int default 0;
   declare c cursor for
      select d.customer_name from depositor d, account a, branch b
      where d.account_number = a.account_number and
         a.branch_name = b.branch_name and b.branch_city = city
      group by d.customer_name
      order by sum(a.balance) desc;
   declare continue handler for sqlstate '02000' set done = 1;

   open c;
   if not done then
      fetch c into top1;
   end if;
   if not done then
      fetch c into top2;
   end if;
   close c;
end
```

Example Trigger

Implement an SQL trigger to assure that every client with a loan also has a bank account at the same branch. If the bank account does not exist, it should be created with balance zero and account number equal to the loan number.

Example Trigger


```
create trigger loan_account_trigger
before insert on borrower
for each row
begin
   declare b_name varchar(50);
   declare accounts int;

   select branch_name into b_name
   from loan where loan_number = new.loan_number;

   select count(*) into accounts
   from depositor d, account a
   where d.account_number = a.account_number and
      d.customer_name = new.customer_name and
      a.branch_name = b_name;

   if accounts = 0 then
      insert into account values(new.loan_number, b_name, 0);
      insert into depositor values(new.customer_name, new.loan_number);
   end if;
end
```

End of Chapter 4(+8)