CS 4092 Database Design and Development (DDD)

05: SQL - Intermediate

Seokki Lee

Slides are adapted from:

Database System Concepts, 6th & 7th Ed. ©Silberschatz, Korth and Sudarshan

Outline

- **Views**
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Access Control
- Accessing SQL From a Programming Language

Views

- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.
  - Not precomputed and stored
  - Computed by executing the query whenever it is used

Views

- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A list of all course sections offered by the Physics department in the Fall 2017 semester with the building and room number of each section

  select course.course id, sec id, building, room number
  from course, section
  where course.course id = section.course id
  and course.dept_name = 'Physics'
  and section.semester = 'Fall'
  and section.year = 2017

Views

- Possible to compute and store the results of the queries and make them available to users
  - What about the data in the relations?
  - Comparing to **with** clause

Views

- Why are views useful?

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
  - Security considerations
  - Personalized collection of “virtual” relations

- A view provides a mechanism to hide certain data from the view of certain users.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  - <query expression> is any legal SQL expression
  - *v* : the view name

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

- View definition is not the same as creating a new relation by evaluating the query expression.
  - Rather, a view definition causes the saving of an expression
  - The expression is substituted into queries using the view.

View Definition and Use

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

View Definition and Use

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Views Defined Using Other Views?

- One view may be used in the expression defining another view?

Views Defined Using Other Views

- One view may be used in the expression defining another view.
- A view relation v_2 is said to *depend directly* on a view relation v_1 if v_1 is used in the expression defining v_2
- A view relation v_2 is said to *depend on* view relation v_1 if either v_2 depends directly to v_1 or there is a path of dependencies from v_2 to v_1
- A view relation v is said to be *recursive* if it depends on itself.

Views Defined Using Other Views

- **create view** *physics_fall_2017* **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id* = *section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2017';

Views Defined Using Other Views

- **create view** *physics_fall_2017* **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id* = *section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2017';

- **create view** *physics_fall_2017_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2017*
  where *building*= 'Watson';

View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017')
    where building= 'Watson';
```

View Expansion (Cont.)

- A way to define the meaning of views defined in terms of other views.

- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
  - View expansion of an expression repeats the following replacement step:

    repeat
    Find any view relation v_i in e_1
    Replace the view relation v_i by the expression defining v_i
    until no more view relations are present in e_1
  - As long as the view definitions are not recursive, this loop will terminate

Materialized Views

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called Materialized view
  - In which case is it useful?

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called Materialized view
  - In which case is it useful?
    - Using a view frequently
    - Demanding fast response to certain queries
      - E.g., aggregates over large relations

Materialized Views

- (Materialized) View maintenance
  - If relations used in the query are updated, the materialized view result becomes out of date.
  - Need to maintain the view, by updating the view whenever the underlying relations are updated
  - Lazy or Periodic

Materialized Views

- (Materialized) View maintenance
  - If relations used in the query are updated, the materialized view result becomes out of date.
  - Need to maintain the view, by updating the view whenever the underlying relations are updated
  - Lazy or Periodic
    - Lazy: update the view when accessed
    - Periodic: update the view periodically
    - It should be carefully considered with storage costs and overhead for updates.
    - Some systems permit DBA to control which method to be used

Update of a View

- Useful but serious problems may occur with modification
  - Updates, insertions, or deletions

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty* **values** ('30765', 'Green', 'Music');

Update of a View

- Useful but serious problems may occur with modification
  - Updates, insertions, or deletions

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty* **values** ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation.
  - instructor(ID, name, dept name, salary)
  - Should have a value for salary

Update of a View

- Two approaches
  - Reject the insert
  - Insert the tuple

    ('30765', 'Green', 'Music', null)

    into the *instructor* relation

Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor.dept_name= department.dept_name*;
- **insert into** *instructor_info* **values** ('69987', 'White', 'Taylor');

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

department

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor.dept_name= department.dept_name*;
- **insert into** *instructor_info* **values** ('69987', 'White', 'Taylor');

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

department

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

Some Updates Cannot be Translated Uniquely

- Any problem?

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

department

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

Some Updates Cannot be Translated Uniquely

- Which department, if multiple departments in Taylor?
  - Unknown department is at Taylor
- What if no department is in Taylor?
  - Violation on primary key constraint
  - ('69987', 'White','Taylor') will not appear in the view.

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

department

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

Some Updates Cannot be Translated Uniquely

- In general, modifications are not permitted on view relations.

- Most SQL implementations allow updates only on simple views.
  - The **from** clause has only **one** database **relation**.
  - The **select** clause contains **only attribute names** of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null.
  - The query does not have a **group** by or **having** clause.

And Some Not at All

- **create view** *history_instructors* **as** (
  select *
  from *instructor*
  where *dept_name*= 'History');

- What happens if we insert

  ('25566', 'Brown', 'Biology', 100000)

  into *history_instructors?*

And Some Not at All

- **create view** *history_instructors* **as** (
  select *
  from *instructor*
  where *dept_name*= 'History')
  with check option;

- What happens if we insert
  ('25566', 'Brown', 'Biology', 100000)
  into *history_instructors?*

- By default, it would be allowed.
  - Views can be defined with a **with check option**.
  - If the inserted tuple doesn't satisfy the condition, the insertion is rejected.
  - Updates are similarly rejected.

Outline

- Views
- **Transactions**
- Integrity Constraints
- SQL Data Types and Schemas
- Access Control
- Accessing SQL From a Programming Language

Transactions

- Unit of work
  - Consists of a sequence of query and/or update statements

Transactions

- Unit of work
  - Consists of a sequence of query and/or update statements
- Atomic transaction
  - Either fully executed or rolled back as if it never occurred
  - Isolation from concurrent transactions

Transactions

- Unit of work
  - Consists of a sequence of query and/or update statements
- Atomic transaction
  - Either fully executed or rolled back as if it never occurred
  - Isolation from concurrent transactions
- Transactions **begin** implicitly when an SQL statement is executed
  - Ended by **commit** (**work**) or **rollback** (**work**)
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)

Transactions Example

- Atomicity (all-or-nothing)
  - Relation **accounts(accID, cust, type, balance)**
  - A customer wants to transfer $100 from his savings (accID = *100*) to his checking account (*accID= 101*)

    UPDATE accounts **SET** balance = balance – 100 **WHERE** accID = 100;

    UPDATE accounts **SET** balance = balance + 100 **WHERE** accID = 101;

Transactions Example

- Atomicity (all-or-nothing)
  - Relation **accounts(accID, cust, type, balance)**
  - A customer wants to transfer $100 from his savings (accID = *100*) to his checking account (*accID= 101*)

    UPDATE accounts **SET** balance = balance – 100 **WHERE** accID = 100;

    UPDATE accounts **SET** balance = balance + 100 **WHERE** accID = 101;

- This can cause inconsistencies if the system crashes after the first update (user would loose money)
  - Using a transaction either both or none of the statements are executed

    BEGIN

      UPDATE accounts **SET** balance = balance – 100 **WHERE** accID = 100;

      UPDATE accounts **SET** balance = balance + 100 **WHERE** accID = 101;

    COMMIT

Transactions and Concurrency

- Also used to isolate concurrent actions of different users
  - Recall that if several users are modifying the database at the same time that can lead to inconsistencies

Outline

- Views
- Transactions
- **Integrity Constraints**
- SQL Data Types and Schemas
- Access Control
- Accessing SQL From a Programming Language

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.
  - A checking account must have a balance greater than $10,000.
  - A salary of a bank employee must be at least $40.00 an hour.
  - A customer must have a (non-null) phone number.

- Usually defined as part of the database schema design process
  - Can be also added to an existing relation

Constraints on a Single Relation

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

Not Null Constraints

- **not null**
  - Prohibits the insertion of a null value for the attribute
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**

Unique Constraints

- **unique** (A_1, A_2, …, A_m)

Unique Constraints

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a superkey.
  - Null permitted?

```
create table products (
    product_no   integer,
    name    text,
    price    numeric,
    unique (product_no)
);
```

Unique Constraints

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a superkey.
  - The keys are permitted to be null (in contrast to primary keys).

```
create table products (
    product_no   integer,
    name    text,
    price    numeric,
    unique (product_no)
);
```

The Check Clause

- The **check**(P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
  - Permits attribute domains to be restricted in powerful ways

- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
    (course_id varchar (8),
     sec_id varchar (8),
     semester varchar (6),
     year numeric (4,0),
     building varchar (15),
     room_number varchar (7),
     time slot id varchar (4),
     primary key (course_id, sec_id, semester, year),
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

The Check Clause

- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
    (course_id varchar (8),
     sec_id varchar (8),
     semester varchar (6),
     year numeric (4,0),
     building varchar (15),
     room_number varchar (7),
     time slot id varchar (4),
     primary key (course_id, sec_id, semester, year),
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

- What about semester has null values?

The Check Clause

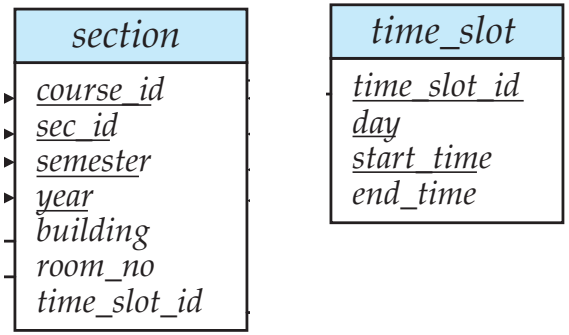
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
    (course_id varchar (8),
     sec_id varchar (8),
     semester varchar (6),
     year numeric (4,0),
     building varchar (15),
     room_number varchar (7),
     time slot id varchar (4),
     primary key (course_id, sec_id, semester, year),
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

- What about semester has null values?
  - A check clause is satisfied if not false.

Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.




```
create table section
        (course_id varchar (8),
         sec_id varchar (8),
         semester varchar (6),
         year numeric (4,0),
         building varchar (15),
         room_number varchar (7),
         time_slot_id varchar (4),
         primary key (course_id, sec_id, semester, year),
         check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
         check (time_slot_id  in (select time_slot_id from time_slot))
```

Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

  check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

  - The check condition states that the *time_slot_id* in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation.
  - The condition has to be checked not only when a tuple is inserted or modified in *section*, but also when the relation *time_slot* changes

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 “Biology” is a department name appearing in one of the tuples in the ***instructor*** relation, then there exists a tuple in the ***department*** relation for “Biology”.

- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Referential Integrity (Cont.)

- Foreign keys can be specified as part of the SQL **create table** statement

  foreign key (*dept_name*) **references** *department*

- By default, a foreign key *references* the primary-key attributes of the *referenced* table.

- SQL allows a list of attributes of the referenced relation to be specified explicitly.

  foreign key (*dept_name*) **references** *department* (*dept_name*)

Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

- An alternative, in case of delete or update is to cascade

```
create table course (
        (…
         dept_name varchar(20),
         foreign key (dept_name) references department
             on delete cascade
             on update cascade,
          . . .)
```

- Instead of cascade we can use :
  - **set null** (set the field to null)
  - **set default** (set the field to the default value for the domain)

When to Verify Integrity Constraint Violation

- After a single database modification
  - Insert, update or delete
- At the end of a transaction
  - After a sequence of modifications

Integrity Constraint Violation During Transactions

- Consider:

```
create table person (
        ID  char(10),
        name char(40),
        mother char(10),
        father  char(10),
        primary key ID,
        foreign key father references person,
        foreign key mother references person)
```

- How to insert a person who does not have father/mother without causing constraint violation?

Integrity Constraint Violation During Transactions

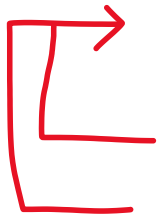
- Consider:

```
create table person (
    ID  char(10),
    name char(40),
    mother char(10),
    father  char(10),
    primary key ID,
    foreign key father references person,
    foreign key mother references person)
```

- How to insert a person who does not have father/mother without causing constraint violation?
  - Insert father and mother of a person before inserting person

Integrity Constraint Violation During Transactions

- Consider:

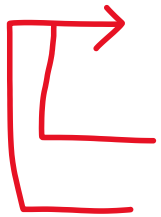


```
create table person (
        ID  char(10),
        name char(40),
        mother char(10),
        father  char(10),
        primary key ID,
        foreign key father references person,
        foreign key mother references person)
```

- How to insert a person who does not have father/mother without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - Set father and mother to null initially, update after inserting all persons
    - not possible if father and mother attributes declared to be **not null**

Integrity Constraint Violation During Transactions

- Consider:



```
create table person (
        ID  char(10),
        name char(40),
        mother char(10),
        father  char(10),
        primary key ID,
        foreign key father references person,
        foreign key mother references person)
```

- How to insert a person who does not have father/mother without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - Set father and mother to null initially, update after inserting all persons
    - not possible if father and mother attributes declared to be **not null**
  - Defer constraint checking

Assertions

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

- The following constraints, can be expressed using assertions:
  - For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
  - An instructor cannot teach in two different classrooms in a semester in the same time slot.

Assertions

- An assertion in SQL takes the form:

  create assertion <assertion-name> **check** (<predicate>);

  - For each tuple in the student relation, the value of the attribute tot_cred must equal the sum of credits of courses that the student has completed successfully.

```
create assertion credits_earned_constraint check
(not exists (select ID
             from student
             where tot_cred <> (select coalesce(sum(credits), 0)
                                from takes natural join course
                                where student.ID= takes.ID
                                      and grade is not null and grade<> 'F' )))
```

Outline

- Views
- Transactions
- Integrity Constraints
- **SQL Data Types and Schemas**
- Access Control
- Accessing SQL From a Programming Language

Built-in Data Types in SQL

- **date:** Calendar 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' **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - Used to specify the number of fractional digits for seconds
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - x and y are **date** → x - y is an interval (number of days from x to y)
  - Adding or subtracting an interval from a date/time gives back a date/time, respectively.

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

book_review **clob**(10KB)
image **blob**(10MB)
movie **blob**(2GB)

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

$$\textit{book\_review}\ \textbf{clob}(10\text{KB})$$
$$\textit{image}\ \textbf{blob}(10\text{MB})$$
$$\textit{movie}\ \textbf{blob}(2\text{GB})$$

- 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)**

- Example:

  create table *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Domains

- **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.

- Example:

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Index Creation

- Many queries reference only a small proportion of the records in a table.
  - Find the salary value of the instructor with ID 22201.

- It is inefficient for the system to read every record to find a record with particular value.

Index Creation

- Many queries reference only a small proportion of the records in a table.
  - Find the salary value of the instructor with ID 22201.

- It is inefficient for the system to read every record to find a record with particular value.

- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.

- We create an index with the **create index** command.

  create index <name> **on** <relation-name> (attribute);

Index Creation Example

- **create table** *student* (

  ID **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))

- **create index** *studentID_index* **on** *student*(*ID*)

- The query:

  select *
  from *student*
  where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

Outline

- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- **Access Control**
- Accessing SQL From a Programming Language

Authorization

- We may assign a user several forms of authorizations 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.

- Each of these types of authorizations is called a **privilege**.
  - We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

- When a user submits a query or an update, the SQL implementation first checks if they are authorized.

Authorization (Cont.)

- 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 in SQL

- The **grant** statement is used to confer authorization.

  grant <privilege list> **on** <relation or view > **to** <user list>

- <privilege list>: **select**, **insert**, **update**, and **delete**

- <user list>
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - a role (more on this later)

Privileges in SQL

- **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 *instructor* relation:

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

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Authorization Specification in SQL

- Privilege can be given for all attributes or only for some.
  - **grant select on** *department* **to** Amit, Satoshi
  - **grant update**(budget) **on** *department* **to** Amit, Satoshi

- 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).

Revoking Authorization in SQL

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

  revoke <privilege list> **on** <relation or view> **from** <user list>

- Example:

  revoke select on *student* **from** U_1, U_2, U_3

  revoke select on department **from** Amit, Satoshi;

  revoke update (budget) **on** department **from** Amit, Satoshi;

Revoking Authorization in SQL

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

  revoke <privilege list> **on** <relation or view> **from** <user list>

  - <privilege list> may be **all** to revoke all privileges the revokee may hold.
  - If <user 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.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.

- To create a role we use: **create role** <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign "users" to the role using:
  - **grant** <role> **to** <users>

- What about creating an instructor user_id (permitting each instructor to use the id to connect to the database)?
  - Not possible to identify which instructor carried out an update
  - If left or moved → need to create a new password and distributed it in a secure manner

Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;

- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant.*

- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

- Consider a staff member who needs to know the salaries of all faculty in Geology department.
  - Not authorized to see them in other departments

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
    - The staff must not have direct access to the instructor table.
    - The result of the query is visible to the staff.

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
    - The staff must not have direct access to the instructor table.
    - The result of the query is visible to the staff.
  - Creator of view did not have some permissions on *instructor?*

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
    - The staff must not have direct access to the instructor table.
    - The result of the query is visible to the staff.
  - Creator of view did not have some permissions on *instructor?*
    - The creator must have **select** authorization on the *instructor* relation.

Authorizations on Schema

- SQL standard specifies a primitive authorization mechanism for the database shema.
  - Only the owner of the schema can carry out any modification.

- **references** privilege permits to create foreign key.
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;

Authorizations on Schema

- SQL standard specifies a primitive authorization mechanism for the database shema.
  - Only the owner of the schema can carry out any modification.

- **references** privilege permits to create foreign key.
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why does Mariano need previlege on department?

Authorizations on Schema

- SQL standard specifies a primitive authorization mechanism for the database shema.
  - Only the owner of the schema can carry out any modification.

- **references** privilege permits to create foreign key.
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why does Mariano need previlege on department?
    - Suppose Mariano creates a foreign key on *dept_name* in table *r*.
      - Insert a tuple with Geology department into r
    - He should have an authorization on *dept_name* of *department*.
    - It is no longer possible to delete the Geology department from the *department* relation without modifying the table *r.*
  - It is also required to create **check** constraint referencing *department* on *r.*

Other Authorization Features

- A user granted may be allowed to pass on the authorizations to others.
- Transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - And more!

Outline

- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Access Control
- **Accessing SQL From a Programming Language**

Accessing SQL from a Programming Language

- A database programmer must have access to a general-purpose programming language for at least two reasons.
  - Not all queries can be expressed in SQL
    - SQL does not provide the full expressive power of a general-purpose language.
  - Non-declarative actions
    - Printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

Accessing SQL from a Programming Language

- **Dynamic SQL** -- can connect to and communicate with a database server using a collection of functions or methods
  - Allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time.
  - JDBC & ODBC
    - Extended to C++, Python, PHP, Go, etc
- **Embedded SQL** -- provides a means by which a program can interact with a database server.
  - The SQL statements are translated at compile time into function calls.
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

Accessing SQL from a Programming Language

- A Major challenge is the **mismatch** in the ways that manipulate data
  - SQL operates on relations and returns a relation as a result.
  - Programming language operates on a variable.
    - Variables correspond to roughly the value of an attribute of a tuple.
- Providing mechanism to return the result of a query

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
 {
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(  // connect to server
              "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
       Statement stmt = conn.createStatement();  // create Statement object
          … Do Actual Work ….
       stmt.close();  // close Statement and release resources
       conn.close(); // close Connection and release resources
    }
    catch (SQLException sqle) {
       System.out.println("SQLException : " + sqle);  // handle exceptions
    }
}
```

Connecting to the Database

- To open a connection to the database
  - Select which database to use
- Using **getConnection**() method
  - Connection conn = DriverManager.**getConnection**(// connect to server
    "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
  - Specifying the URL
    - Protocol: jdbc:oracle:thin:
    - Port: 2000
    - Database server: rhode.uc.edu
    - Database: univdb
  - User identifier
  - Password

Shipping SQL Statements to Database System

- Once the connection is open, the program can use an instance of the class **Statement** to send SQL statements to the database system.
- A **Statement** object allows the Java program to invoke methods.
  - Statement stmt = conn.**createStatement**(); // create Statement object
- To execute a statement
  - **executeQuery**(): SQL statement is a query
  - **executeUpdate**(): nonquery statement
    - Update, insert, delete or create

JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
} catch (SQLException sqle)
{
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rs = stmt.executeQuery(
                        "select dept_name, avg (salary)
                         from instructor
                         group by dept_name");
while (rs.next()) {
    System.out.println(rs.getString("dept_name") + " " +
                                rs.getFloat(2));
}
```

JDBC Code Details

- Result stores the current row position in the result
  - Pointing before the first row after executing the statement
  - **.next**() moves to the next tuple
    - Returns false if no more tuples

- Getting result fields:
  - **rs.getString**("dept_name") and **rs.getString**(1) equivalent if dept_name is the first argument of select result.

- Dealing with Null values
  - int a = rs.getInt("a");

    if (rs.**wasNull**()) Systems.out.println("Got null value");

Exceptions and Resource Management

- To catch any exceptions (error conditions) that arise
  - **SQLException** for SQL specific exception
  - **Exception** for any Java exception
- Example
  - catch (SQLException sqle) {

    System.out.println("SQLException : " + sqle); // handle exceptions

    }

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
 {
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(  // connect to server
              "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
      Statement stmt = conn.createStatement();  // create Statement object
         … Do Actual Work ….
      stmt.close();          // close Statement and release resources
      conn.close();          // close Connection and release resources
    }
    catch (SQLException sqle) {
      System.out.println("SQLException : " + sqle);          // handle exceptions
    }
}
```

JDBC Code

```
public static void JDBCexample(String userid, String passwd)
{
    try (
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
    ) {
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987','Kim','Physics',98000)");
        }
        catch (SQLException sqle) {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select dept_name, avg (salary) "+
                " from instructor "+
                " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                    rset.getFloat(2));
        }
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

Automatically closed at the end of the try block

Prepared Statement

- We can create a prepared statement in which some values are replaced by “?”.
  - Specifying that actual values will be provided later.
- Each time the query is executed (with new values to replace the “?”s), the database system can **reuse** the previously compiled form of the query and apply the new values.

Prepared Statement

- We can create a prepared statement in which some values are replaced by “?”.
  - Specifying that actual values will be provided later.
- Each time the query is executed (with new values to replace the “?”s), the database system can **reuse** the previously compiled form of the query and apply the new values.
- **PreparedStatement** pStmt = conn.**prepareStatement**(
  "insert into instructor values(?,?,?,?)");

```
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

Prepared Statement

- **WARNING**: always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - ”insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ')“
  - What if name is “D'Souza”?

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ' " + name + " ' "
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = ‘Y

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ' " + name + " ' "
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = ‘Y
- The resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = ‘Y’
  - The where clause is always true.

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ‘ " + name + " ‘ "
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = ‘Y
- The resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = ‘Y’
  - The where clause is always true.
  - Clever malicious users could have even used
    - X'; update instructor set salary = salary + 10000; --

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ‘ " + name + " ‘ "
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = ‘Y
- The resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = ‘Y’
  - The where clause is always true.
  - Clever malicious users could have even used
    - X'; update instructor set salary = salary + 10000; --
- Prepared statement internally uses
  - "select * from instructor where name = 'X\' or \'Y\' = \'Y’
  - **Internally escapes the special characters → Empty result**

Metadata Features

- A Java program does not include declarations for data stored in the database.
  - Part of the SQL DDL statements
- The java program that uses JDBC must determine the information directly from the database system at runtime.

Metadata Features

- ResultSet has a method for metadata, **getMetaData**().
- After executing query to get a ResultSet rs:
  - ResultSetMetaData rsmd = **rs.getMetaData**();

```
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

Metadata (Cont)

- **DatabaseMetaData** interface provides a way to find metadata about the database.
- DatabaseMetaData dbmd = conn.getMetaData();

  ResultSet rs = dbmd.**getColumns**(null, "univdb", "department", "%");

  while(rs.next()) {

  System.out.println(rs.getString("COLUMN_NAME"),

  rs.getString("TYPE_NAME");

  }

  - getColumns(Catalog, Schema, Table, and Column)
  - Returns: One row for each column; row has a number of attributes such as COLUMN_NAME, TYPE_NAME

Metadata (Cont)

- **DatabaseMetaData** interface provides a way to find metadata about the database.
- DatabaseMetaData dbmd = conn.getMetaData();

  ResultSet rs = dbmd.**getTables** (“”, "", “%", new String[] {“TABLES”});

  while(rs.next()) {

  System.out.println(rs.getString(“TABLE_NAME“));

  }

  - The first three parameters are same as **getColumns**().
  - The fourth parameter can be used to restrict the types of tables returned
    - If null → all tables including system internal tables
    - Set to restrict the tables returned to only user-created tables
  - Returns: One row for each table; row has a number of attributes such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..

Finding Primary Keys

- DatabaseMetaData dmd = connection.getMetaData();

```
// Arguments below are:  Catalog, Schema, and Table
// The value “”  for Catalog/Schema indicates current catalog/schema
//  The value null indicates all catalogs/schemas
ResultSet rs = dmd.getPrimaryKeys(“”, “”, tableName);

while(rs.next()){
    // KEY_SEQ indicates the position of the attribute in
    // the primary key, which is required if a primary key has multiple
    // attributes
    System.out.println(rs.getString(“KEY_SEQ”),
                                    rs.getString("COLUMN_NAME");
}
```

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
  - conn.setAutoCommit(true) to turn it on
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();

Other JDBC Features

- Calling functions and procedures
  - CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");
- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively
  - get data from these objects by getBytes() and getSubString()
  - associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream).

JDBC Resources

- JDBC Basics Tutorial
  - Oracle
    - https://docs.oracle.com/javase/tutorial/jdbc/index.html
  - Postgres
    - https://jdbc.postgresql.org/index.html
  - …

ODBC

- Open DataBase Connectivity (ODBC) standard
  - Standard for application program to communicate with a database server
  - Application program interface (API) to
    - open a connection with a database
    - send queries and updates
    - get back results
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Defined originally for Basic and C, versions available for many languages

ODBC (Cont.)

- Each database system supporting ODBC provides a "**driver**" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- Opens database connection using **SQLConnect**(). Parameters for SQLConnect:
  - connection handle
  - the server to which to connect
  - the user identifier
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.

ODBC Code

- int ODBCexample()

```
{
    RETCODE error;
    HENV    env;     /* environment */
    HDBC    conn;  /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, “db.yale.edu", SQL_NTS, "avi", SQL_NTS, "avipasswd",
      SQL_NTS);
    { …. Do actual work … }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

ODBC Code (Cont.)

- Program sends SQL commands to database by using **SQLExecDirect**
- Result tuples are fetched using **SQLFetch**()
- SQLBindCol() binds C language variables to attributes of the query result
  - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
  - Arguments to **SQLBindCol**()
    - ODBC stmt variable, attribute position in query result
    - The type conversion from SQL to C.
    - The address of the variable.
    - For variable-length types like character arrays,
      - The maximum length of the variable
      - Location to store actual length when a tuple is fetched.
      - Note: A negative value returned for the length field indicates null value

ODBC Code (Cont.)

- Main body of program

```
char deptname[80];
float salary;
int lenOut1, lenOut2;
HSTMT stmt;
char * sqlquery = "select dept_name, sum (salary)
                   from instructor
                   group by dept_name";
SQLAllocStmt(conn, &stmt);
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR, deptname , 80, &lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
    while (SQLFetch(stmt) == SQL_SUCCESS) {
        printf (" %s %g\n", deptname, salary);
    }
}
SQLFreeStmt(stmt, SQL_DROP);
```

ODBC Prepared Statements

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?,?,?)
  - Repeatedly executed with actual values for the placeholders
- To prepare a statement
  SQLPrepare(stmt, <SQL String>);
- To bind parameters
  SQLBindParameter(stmt, <parameter#>,
  … type information and value omitted for simplicity..)
- To execute the statement
  SQLExecute(stmt);
- To avoid SQL injection security risk, do not create SQL strings directly using user input; instead use prepared statements to bind user inputs

More ODBC Features

- **Metadata features**
  - finding all the relations in the database and
  - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
  - Transactions must then be committed or rolled back explicitly by
    - SQLTransact(conn, SQL_COMMIT) or
    - SQLTransact(conn, SQL_ROLLBACK)

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/I
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement >;

  Note: this varies by language:
  - In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - In Java embedding uses # SQL { …. };

Embedded SQL (Cont.)

- Before executing any SQL statements, the program must first connect to the database.

  EXEC SQL **connect to** *server* **user** *user-name* **using** *password*;

- Variables of the host language can be used within embedded SQL statements. They are preceded by a **colon** (:) to distinguish from SQL variables (e.g., :*credit_amount)*
- **Variables** used as above must be **declared** within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

```
EXEC SQL BEGIN DECLARE SECTION;
    int  credit-amount ;
EXEC SQL END DECLARE SECTION;
```

Embedded SQL (Cont.)

- To write an embedded SQL query, we use

  declare *c* **cursor for** **<SQL query>**

  - The variable *c* is used to identify the query
- Example:
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable **credit_amount** in the host language
  - Specify the query in SQL as follows:

```
EXEC SQL
    declare c cursor for
    select ID, name
    from student
    where tot_cred > :credit_amount;
```

Embedded SQL (Cont.)

- The **open** statement for our example is as follows:

  EXEC SQL **open** *c* ;

  - This statement causes the database system to **execute** the query and to **save** the results within a temporary relation.
  - The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.
- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** :*si, :sn;*

  - Single fetch returns only one tuple.
  - Repeated calls to fetch get successive tuples in the query result

Embedded SQL (Cont.)

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* ;

Updates Through Embedded SQL

- Embedded SQL expressions for database modification
  - **update**, **insert**, and **delete**
- Can update tuples fetched by cursor by declaring that the cursor is for update

```
EXEC SQL
  declare c cursor for
    select *
    from instructor
    where dept_name = 'Music'
    for update;
```

- We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier).

```
EXEC SQL
  update instructor
  set salary = salary + 100
  where current of c;
```

Embedded SQL - Example

```
/* define a cursor */
   EXEC SQL DECLARE emp_cursor CURSOR FOR
      SELECT ename, job
         FROM emp
         WHERE empno = :emp_number
         FOR UPDATE OF job;

/* open the cursor and identify the active set */
   EXEC SQL OPEN emp_cursor;

/* break if the last row was already fetched */
   EXEC SQL WHENEVER NOT FOUND DO break;

/* fetch and process data in a loop */
   for (;;)
   {
      EXEC SQL FETCH emp_cursor INTO :emp_name, :job_title;

/* optional host-language statements that operate on
   the FETCHed data */

      EXEC SQL UPDATE emp
         SET job = :new_job_title
         WHERE CURRENT OF emp_cursor;
   }
```

SQLJ

- SQLJ: embedded SQL in Java
  - **#sql** iterator deptInfoIter (String dept name, int avgSal);

```
deptInfoIter iter = null;
#sql iter = { select dept_name, avg(salary) from instructor
                    group by dept name };
while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}
iter.close();
```

Corresponding Reading Materials

- Database System Concepts 6th & 7th Edition
  - Chapter 4 & 5