# Chapter 4: Intermediate SQL

# Outline

Joined Relation

SQL Data Types and Schemas

Integrity Constraints

Views

Indexes

Transactions

Authorization

# Joined Relations

**Join operations** take two relations and return as a result another relation.

Join operations are typically used as subquery expressions in the **from** clause

**Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.

**Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

| *Join types* |
| --- |
| **inner join** |
| **left outer join** |
| **right outer join** |
| **full outer join** |

| *Join Conditions* | |
| --- | --- |
| **natural** | 自然连接 |
| **on** <predicate> | 条件连接 |
| **using** $(A_1, A_1, \ldots, A_n)$ | 等值连接 |

# Join operations – Example

Relation *course*

| *course_id* | *title* | *dept_name* | *credits* |
|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 |
| CS-190 | Game Design | Comp. Sci. | 4 |
| CS-315 | Robotics | Comp. Sci. | 3 |

Relation *prereq*

| *course_id* | *prereq_id* |
|---|---|
| BIO-301 | BIO-101 |
| CS-190 | CS-101 |
| CS-347 | CS-101 |

Observe that

prereq information is missing for CS-315 and

course information is missing for CS-437

# Outer Join

An extension of the join operation that avoids loss of information.

Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.

Uses *null* values.

# Left Outer Join

*course* **natural left outer join** *prereq*

| *course_id* | *title* | *dept_name* | *credits* | *prereq_id* |
|---|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 | BIO-101 |
| CS-190 | Game Design | Comp. Sci. | 4 | CS-101 |
| CS-315 | Robotics | Comp. Sci. | 3 | *null* |

# Right Outer Join

*course* **natural right outer join** *prereq*

| *course_id* | *title* | *dept_name* | *credits* | *prereq_id* |
|---|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 | BIO-101 |
| CS-190 | Game Design | Comp. Sci. | 4 | CS-101 |
| CS-347 | *null* | *null* | *null* | CS-101 |

select count(*)

from *course* **natural right outer join** *prereq*

where *prereq_id* is null;

# Full Outer Join

*course* **natural full outer join** *prereq*

| *course_id* | *title* | *dept_name* | *credits* | *prereq_id* |
|---|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 | BIO-101 |
| CS-190 | Game Design | Comp. Sci. | 4 | CS-101 |
| CS-315 | Robotics | Comp. Sci. | 3 | *null* |
| CS-347 | *null* | *null* | *null* | CS-101 |

# Joined Relations – Examples

*course* **inner join** *prereq* **on**
*course.course_id = prereq.course_id*

| *course_id* | *title* | *dept_name* | *credits* | *prereq_id* | *course_id* |
|---|---|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 | BIO-101 | BIO-301 |
| CS-190 | Game Design | Comp. Sci. | 4 | CS-101 | CS-190 |

What is the difference between the above, and a natural join?

*course* **left outer join** *prereq* **on**
*course.course_id = prereq.course_id*

| *course_id* | *title* | *dept_name* | *credits* | *prereq_id* | *course_id* |
|---|---|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 | BIO-101 | BIO-301 |
| CS-190 | Game Design | Comp. Sci. | 4 | CS-101 | CS-190 |
| CS-315 | Robotics | Comp. Sci. | 3 | *null* | *null* |

# Joined Relations – Examples

*course* **natural right outer join** *prereq*

| *course_id* | *title* | *dept_name* | *credits* | *prereq_id* |
|---|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 | BIO-101 |
| CS-190 | Game Design | Comp. Sci. | 4 | CS-101 |
| CS-347 | *null* | *null* | *null* | CS-101 |

*course* **full outer join** *prereq* **using** (*course_id*)

| *course_id* | *title* | *dept_name* | *credits* | *prereq_id* |
|---|---|---|---|---|
| BIO-301 | Genetics | Biology | 4 | BIO-101 |
| CS-190 | Game Design | Comp. Sci. | 4 | CS-101 |
| CS-315 | Robotics | Comp. Sci. | 3 | *null* |
| CS-347 | *null* | *null* | *null* | CS-101 |

# Built-in Data Types in SQL

**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' **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

**date, time functions:**

current_date(), current_time()

year(x), month(x), day(x), hour(x), minute(x), second(x)

# User-Defined Types

**create type** construct in SQL creates user-defined type

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

**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.

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

# 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)
- MySQL BLOB datatypes:
  - **TinyBlob** : 0 ~ 255 bytes.
  - **Blob**: 0 ~ 64K bytes.
  - **MediumBlob** : 0 ~ 16M bytes.
  - **LargeBlob** : 0 ~ 4G bytes.
- **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.

# 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 \$10,000.00
- A salary of a bank employee must be at least \$4.00 an hour
- A customer must have a (non-null) phone number

# Integrity Constraints on a Single Relation

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

# Not Null and Unique Constraints

**not null**

Declare *name* and *budget* to be **not null**

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

**unique** ( $A_1$, $A_2$, …, $A_m$)

The unique specification states that the attributes *A*1, *A*2, … *A*m form a super key ( × candidate key) .

Candidate keys are permitted to be null (in contrast to primary keys).

# The check clause

**check** (P)

where P is a predicate

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'))
);

# 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.

# Cascading Actions in Referential Integrity

```
create table course (
    course_id  char(5) primary key,
    title      varchar(20),
    dept_name  varchar(20) references department
)
```

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

alternative actions to cascade: **set null**, **set default,** **<u>restricted</u>**

# Integrity Constraint Violation During Transactions

```
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 tuple without causing constraint violation ?

insert father and mother of a person before inserting person

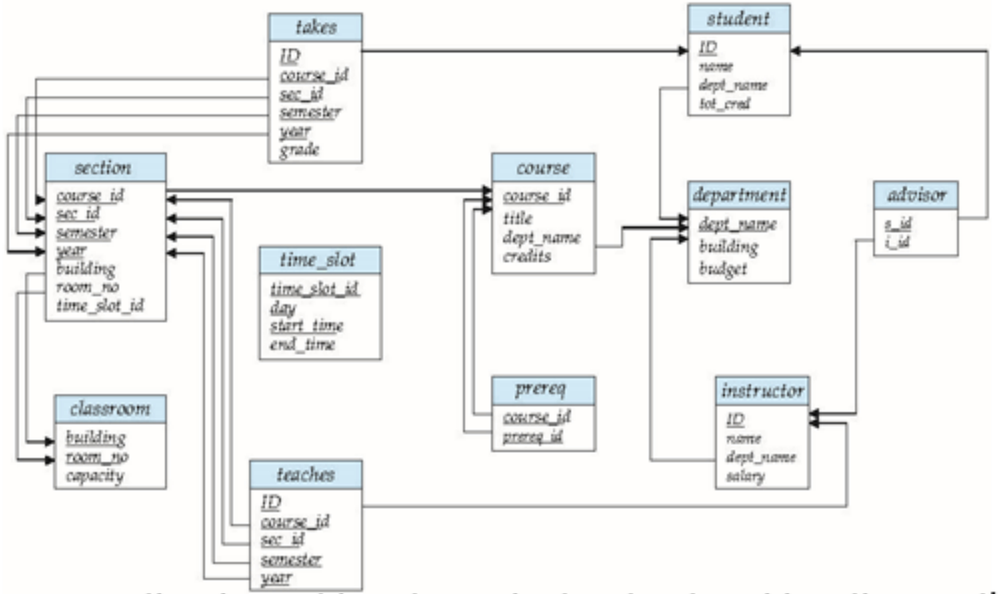
OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)

OR defer constraint checking to transaction end.

# Complex Check Clauses

**check** (*time_slot_id* **in**

(**select** *time_slot_id* **from** *time_slot*))




Every section has at least one instructor teaching the section.

**check** ((*course_id, sec_id, semester, year*) **in**

(**select** *course_id, sec_id, semester, year* **from** *teaches*))

# Complex Check Clauses

Unfortunately: subquery in check clause not supported by pretty much any database

Alternative: triggers

# assertion

```
create assertion <assertion-name> check
<predicate>;

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

# Views

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

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*

Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

# View Definition

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

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

where <query expression> is any legal SQL expression. The view name is represented by *v*.

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.

# Example Views

A view of instructors without their salary

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

Find the names of 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

```
create view physics_fall_2009 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 = ’2009’;

create view physics_fall_2009_watson as
   select course_id, room_number
   from physics_fall_2009
   where building= ’Watson’;
```

# View Expansion

Expand use of a view in a query/another view

```
create view physics_fall_2009_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 = '2009')
 where building= 'Watson';
```

# Update of a View

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

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

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

This insertion must be represented by the insertion of 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');

- which department, if multiple departments in Taylor?

Most SQL implementations allow updates only on simple views(updatable 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.

# *Materialized Views

Materializing a view: create a physical table containing all the tuples in the result of the query defining the view

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.

# * View and Logical Data Indepencence

If relation **S(a, b, c)** is split into two sub relations **S1(a,b)** and **S2(a,c)**

How to realize the logical data independence?

1) create table S1 …;

   create table S2 …;

2) insert into S1 select a, b from S;

   insert into S2 select a, c from S;

3) drop table S;

4) create view S(a,b,c) as select a,b,c from S1 natural join S2;

select * from S where … → select * from S1 natural join S2 where …

insert into S values (1 ,2,3) → insert into S1 values (1, 2);

insert into S2 values (1 ,3);

# Indexes

```
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)
```

Indices are data structures used to speed up access to records with specified values for index attributes

e.g.
```
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*

# Transactions

Unit of work (NONE or ALL)

Atomic transaction

- either fully executed or rolled back as if it never occurred

Isolation from concurrent transactions

Transactions begin implicitly

- 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)
- In MySQL:
- **>SET AUTOCOMMIT=0;**
- In SQL:1999, can use: **begin atomic** .... **end**
  - Not supported on most databases

# Transactions

Transaction example :

**SET AUTOCOMMIT=0**

**UPDATE** account **SET** balance=balance-100 **WHERE** ano=‘1001’;

**UPDATE** account **SET**balance=balance+100 **WHERE** ano=‘1002’;

**COMMIT;**

**UPDATE** account **SET** balance=balance -200 **WHERE** ano=‘1003’;

**UPDATE** account **SET** balance=balance+200 **WHERE** ano=‘1004’;
**COMMIT;**

**UPDATE** account **SET** balance=balance+balance*2.5%;

**COMMIT;**

# ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

**Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.

**Consistency.** Execution of a transaction in isolation preserves the consistency of the database.

**Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

> That is, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$, finished execution before $T_i$ started, or $T_j$ started execution after $T_i$ finished.

**Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

# Authorization(授权)

Forms of authorization on parts of the database:

- **Select** - 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 in SQL

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

**grant** <privilege list> // privilege：权限

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

<user list> is:

a user-id

**public**, which allows all valid users the privilege granted

A role (more on this later)

Granting a privilege on a view does not imply granting any privileges on the underlying relations.

The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

# Privileges in SQL

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

**grant select on** *department* **to** *public*

**grant update** *(budget)* **on** *department* **to** *U1,U2*

**grant all privileges on department to** $U_1$

# Revoking Authorization in SQL

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

**revoke** <privilege list>

**on** <relation name or view name>

**from** <user list>

Example:

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

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

If <revokee-list> includes **public**, all users lose the privilege except those granted it explicitly.

If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

All privileges that depend on the privilege being revoked are also revoked.

# Roles

**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

```
create view geo_instructor as
(select *
 from instructor
 where dept_name = 'Geology');
```

```
grant select on geo_instructor  to  geo_staff
```

# Other Authorization Features

**references** privilege to create foreign key

- **grant reference** (*dept_name*) **on** *department* **to** Mariano;
- why is this required?

**instructor** | **department**

**Computer Science** → **Computer Science**

# Other Authorization Features

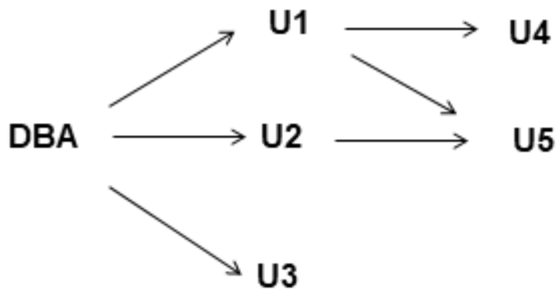
transfer of privileges

**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**;

**revoke grant option for select on** *department* **from** Amit;



# End of Chapter 4