Chapter 4: Intermediate SQL

Chapter 4: Intermediate SQL

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

Joined Relations

- A join operation
  - is a Cartesian product
  - requires that tuples in the two relations match (under some condition)
  - specifies the attributes that are present in the result of the join

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- 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
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

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-347

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from 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

Full Outer Join

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

(Note: Check Latest MYSQL)

course_id	*title*	*dept_name*	*credits*	*prere_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 – Example 2

- *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*)

(Note: Check Latest MYSQL)

course_id	*title*	*dept_name*	*credits*	*prere_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

Difference between on and where

- **select** *
  from *course* **left outer join** *prereq* **on**
  course.course_id = prereq.course_id

course_id	*title*	*dept_name*	*credits*	*prere_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*

- **select** *
  from *course* **left outer join** *prereq* **on** *true*
  where *course.course_id = prereq.course_id*

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

Difference between on and where contd.

- Every tuple satisfies **"on** *true"*
- Therefore there are no dangling tuples
- **on** condition is a part of **join** clause, but **where** is not
- **where** clause is evaluated after **from** clause

Views

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.)
- Consider a person who needs to know 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*

Views contd.

- A **view** provides a mechanism to hide certain data from the view of certain users.
- 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 pre-computed and stored

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 contd.

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

Example Views

- 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

- **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

  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?
  - what if no department is in Taylor?
  - if nulls are inserted for unknown values, the view *instructor_info* still does not include (’69987’, ’White’, ’Taylor’)
- Therefore, modifications are generally not permitted on view relations, except in limited cases

Updates on views in SQL

- 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.

Tuples not satisfying where

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= ’History’;
- What happens if we insert (’25566’, ’Brown’, ’Biology’, 100000) into *history_instructors?*
  - SQL will allow this
- **with check option** clause at the end of view definition
  - ensures tuples not satisfying the where clause are not inserted

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.
- SQL does not provide a standard way of specifying that a view is materialized

Transactions

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

Transactions contd.

- 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 SQL:1999, can use: **begin atomic** …. **end**
    - Not supported on most databases

Integrity Constraints on a Single Relation

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

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**

  Primary keys do not need to be explicitly declared **not null**
- **unique** ($A_1, A_2, \ldots, A_m$)
  - The unique specification states that the attributes *A*1, *A*2, … *A*m
    form a 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

- 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.
  - if for any values of A appearing in R these values also appear in S, A is said to be a
  - **foreign key** of R
- In general, a referential integrity constraint does not require A to be a primary key of S

Referential integrity in SQL: direct support

- By default, a foreign key references a primary key attribute of another table

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

- A list of attributes *A* of the referenced relation can be specified explicitly
  - *A* must be declared a candidate key using a **unique** constraint or a **primary key** constraint

Cascading Actions in Referential Integrity

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

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

- **on delete cascade** will work when you delete a tuple in *department*
- alternative actions to cascade: **set null**, **set default**

Integrity Constraint Violation

- E.g.

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

- 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 (not supported in many databases)

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database to satisfy always.
- Domain and referential integrity constraints are special forms of assertions
- The following constraint can be expressed using assertions, but not using domain / referential integrity constraints:
  - 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 assertion is tested for validity – often time consuming
- Therefore not supported in many RDBMs

Assertions contd.

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

create assertion *credits_earned_constraint* **check**
not exists (select *ID*
from *student S*
where *tot_cred* <> (**select sum**(*credits*)
from *takes, course*
where *takes.course_id = course.course_id*
and
*S.ID= takes.ID.***and**
takes.grade <> 'F' **and**
takes.grade **is not null**));

Index Creation

- Many queries reference only a small proportion of the records in a table.
- Inefficient for the system to read every record
- 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 contd.

- **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*)

  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*

  More on indices later

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- uninterpreted binary data
  - **clob**: character large object -- 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

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

create type *Rupees* **as integer final;**
create type *Dollars* **as integer final;**

/* Check latest MySQL */

- 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 **/* Check latest MySQL */**

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

- Example for a constraint:

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

- Types and domains - differences
  - Domains can have constraints, such as **not null**
  - Domains can have **default** values for variables

Authorization

- We may assign a user several forms of authorizations on parts of the database (related to data).
  - **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.

Authorization Specification in SQL - data

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

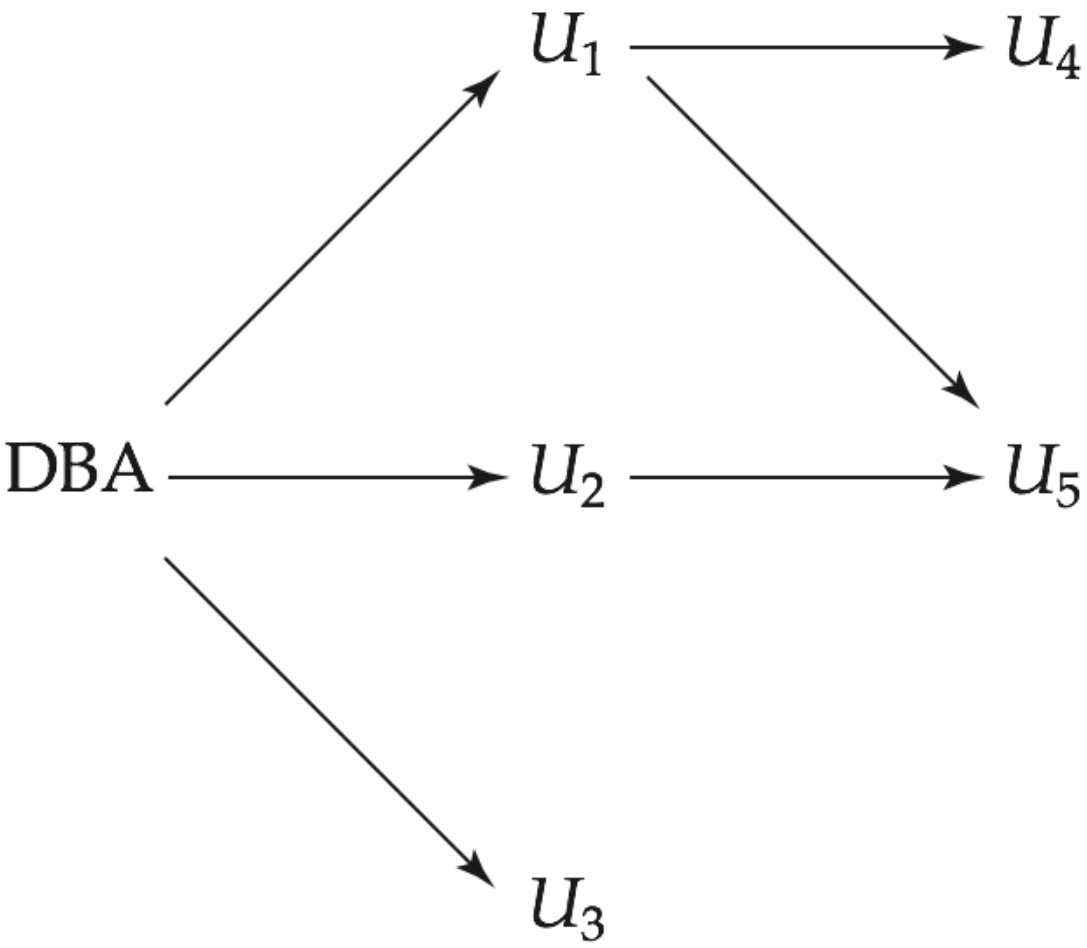
  grant <privilege list> **on** <relation or view > **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:
  - **grant select on** *department* **to** Amit, Satoshi
- Granting a privilege on a view 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).

Privileges in SQL - data

- **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 all or some attributes of tuples
- **update**: the ability to update all or some attributes
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Authorization-Grant Graph



Transfer of privileges:
grant select on *department* **to** *U1* **with grant option**;

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

- <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 – view and underlying relations

Roles

- A **role** is a way to distinguish among various users
  - denotes what these users can access/update in the database.
  - No need to give individual authorizations
- 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>

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

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

  --geo_staff is a role
- Suppose that a *geo_staff* member issues
  - **select** * **from** *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

Other Authorization Features

- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
    - Revocation will cascade (default behavior)
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
    - An error if there are cascading revocations

Authorizations on schema

- Primitive: Only the owner can carry out any modification to the schema
- **references** privilege to declare foreign keys while creating a relation
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Now Mariano can create a foreign key in r referencing *department*
  - why is this a privileged operation ?

End of Chapter 4