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 an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  ```sql
  select ID, name, dept_name
  from instructor
  ```

- 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**.

View Definition

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

  ```sql
  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

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

- Find all instructors in the Biology department

  ```sql
  create view biology_instructors
  as
  select name
  from faculty
  where dept_name = 'Biology'
  ```

- Create a view of department salary totals

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

Views Defined Using Other Views

- Create a view of physics courses in Fall 2009

  ```sql
  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 a view of physics courses in Fall 2009 at Watson Tower

  ```sql
  create view physics_fall_2009_watson
  as
  select course_id, room_number
  from physics_fall_2009
  where building = 'Watson';
  ```
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?
- 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.
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.

Transactions

- Unit of work
- 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 SQL:1999, can use: `begin atomic …. end`
    - Not supported on most databases

Transactions Example

- Example Atomicity (all-or-nothing)
  - Recall example from the introduction
  - Relation `accounts(accID, cust, type, balance)`
  - A user want 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 lose 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

- Transactions are also used to isolate concurrent actions of different users
- Recall from the introduction that if several users are modifying the database at the same time that can lead to inconsistencies
- More on that later once we talk about concurrency control

Integrity Constraints

modified from:
Database System Concepts, 6th Ed
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on reuse
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

Not Null and Unique Constraints

- not null
  - Declare name and budget to be not null
    - name varchar(20) not null
  - unique (A1, A2, …, Am)
    - The unique specification states that the attributes A1, A2, …, Am 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

- 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
Integrity Constraint Violation During Transactions

- E.g.
  ```
  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 (next slide)

Complex Check Clauses

- check (time_slot_id in (select time_slot_id from time_slot))
  - why not use a foreign key here?
  - Every section has at least one instructor teaching the section.
  - how to write this?
  - Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)
  - create assertion <assertion-name> check <predicate>;
  - Also not supported by anyone

Indexes and User-Defined Types (UDTs)

Modified from:
Database System Concepts, 6th Ed.
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on reuse

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

Index Creation

- 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
  - More on indices later

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)
  - `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.

**Access Control**

Forms of authorization on parts of the database:

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

Forms of authorization to modify the database schema:

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

**Authorization Specification in SQL**

- The `grant` statement is used to confer authorization
  
  ```
  grant <privilege list>
  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**

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U₁, U₂, and U₃ `select` authorization on the instructor relation:
    ```
    grant select on instructor to U₁, U₂, U₃
    ```
  - `insert`: the ability to insert tuples
  - `update`: the ability to update using the SQL update statement
  - `delete`: the ability to delete tuples.

- `all privileges`: used as a short form for all the allowable privileges
Revoking Authorization in SQL

- The `revoke` statement is used to revoke authorization.
  ```sql
  revoke <privilege list>
  on <relation name or view name> from <user list>
  ```
- Example:
  ```sql
  revoke select on branch from U1, U2, U3
  ```
- `<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
  ```sql
  (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?
- `create view` geo_instructor as
- `grant select on` instructor to instructor

Other Authorization Features

- `references` privilege to create foreign key
  - `grant reference` (dept_name) on department to Mariano;
  - why is this required?
- 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`;
- Etc. read text book Section 4.6 for more details we have omitted here.

Understanding RESTRICT/CASCADE

- Bob grants right X on Y to Alice with grant option
- Alice grants right X on Y to Peter
- Abandoned right
  - A right for which there is no justification anymore
- `revoke` X on Y from Alice restrict
  - With restrict fails if it would result in abandoned rights
- `revoke` X on Y from Alice cascade
  - Also revokes rights that would otherwise be abandoned

Bob grants right X on Y to Alice with grant option
- Alice grants right X on Y to Peter
- Bob grants right X on Y to Peter
- Abandoned privilege
  - A privilege for which there is no justification anymore
  - Indirect justifications count
- `revoke` X on Y from Alice restrict
  - Fails: even though there exists additional justification for the privilege.
- `revoke` X on Y from Alice cascade
  - Revokes that right from Peter.
  - Peter still has the right to do X on Y
Recap

- Views
  - Virtual
  - Materialized
  - Updates
- Integrity Constraints
  - Not null, unique, check
  - Foreign keys: referential integrity
- Access control
  - Users, roles
  - Privileges
  - GRANT / REVOKE
- Data types
  - Build-in types, Domains, Large Objects
  - UDTs
  - Indices

Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- SQL - Advanced
- Database Design
  - Transaction Processing, Recovery, and Concurrency Control
  - Storage and File Structures
  - Indexing and Hashing
  - Query Processing and Optimization

Figure 4.01

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept</th>
<th>grad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Zhang</td>
<td>Comp Sci.</td>
<td>2009 A</td>
</tr>
<tr>
<td>2</td>
<td>Zhan</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
<tr>
<td>3</td>
<td>Zhao</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
<tr>
<td>4</td>
<td>Zhang</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
<tr>
<td>5</td>
<td>Zhao</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
</tbody>
</table>

Figure 4.02

<table>
<thead>
<tr>
<th>ID</th>
<th>course_id</th>
<th>crs</th>
<th>semester</th>
<th>year</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CS-101</td>
<td>1721</td>
<td>Fall</td>
<td>2009 A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CS-101</td>
<td>1721</td>
<td>Spring</td>
<td>2009 A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CS-101</td>
<td>1721</td>
<td>Spring</td>
<td>2009 A</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CS-101</td>
<td>1721</td>
<td>Fall</td>
<td>2009 A</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CS-101</td>
<td>1721</td>
<td>Fall</td>
<td>2009 A</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.03

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept</th>
<th>grad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Zhang</td>
<td>Comp Sci.</td>
<td>2009 A</td>
</tr>
<tr>
<td>2</td>
<td>Zhan</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
<tr>
<td>3</td>
<td>Zhao</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
<tr>
<td>4</td>
<td>Zhang</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
<tr>
<td>5</td>
<td>Zhao</td>
<td>Comp Sci.</td>
<td>2009 C</td>
</tr>
</tbody>
</table>
Join types

inner join
left outer join
right outer join
full outer join

Join conditions

natural
on < predicate>
using (A₁, A₂, ..., Aₙ)