Introduction to Databases
《数据库引论》

Lecture 4/1: Intermediate SQL
第4/1讲：中级结构化查询语言

周水庚 / Shuigeng Zhou

邮件: sgzhou@fudan.edu.cn 网址：admis.fudan.edu.cn/sgzhou

复旦大学计算机科学技术学院

Content of the Course

- **Part 0: Overview**
  - Lect. 0/1 (Feb. 20) - Ch1: Introduction
- **Part 1 Relational Databases**
  - Lect. 2 (Feb. 27) - Ch2: Relational model (data model, relational algebra)
  - Lect. 3 (Mar. 6) - Ch3: SQL (Introduction)
  - Lect. 4 (Mar. 13) – Ch4 & 5: Intermediate & Advanced SQL
- **Part 2 Database Design**
  - Lect. 5 (Mar. 20) - Ch6: Database design based on E-R model
  - Lect. 6 (Mar. 27) - Ch7: Relational database design (Part I)
  - Lect. 7 (Apr. 3) - Ch7: Relational database design (Part II)
- **Midterm exam: Apr. 10**
- **Part 3 Data Storage & Indexing**
  - Lect. 7 (Apr. 17) - Ch12/13: Storage systems & structures
  - Lect. 8 (Apr. 24) - Ch14: Indexing
- **Part 4 Query Processing & Optimization**
  - May 1, holiday, no classes
  - Lect. 9 (May 8) - Ch15: Query processing
  - Lect. 10 (May 15) - Ch16: Query optimization
- **Part 5 Transaction Management**
  - Lect. 11 (May 22) - Ch17: Transactions
  - Lect. 12 (May 29) - Ch18: Concurrency control
  - Lect. 13 (Jun. 5) - Ch19: Recovery system
  - Lect. 14 (Jun. 5) – Course review

Final exam: 13:00-15:00, Jun. 18

Schema Diagrams

branch (*branch_name*, branch_city, assets)

customer (*customer_name*, customer_street, customer_city)

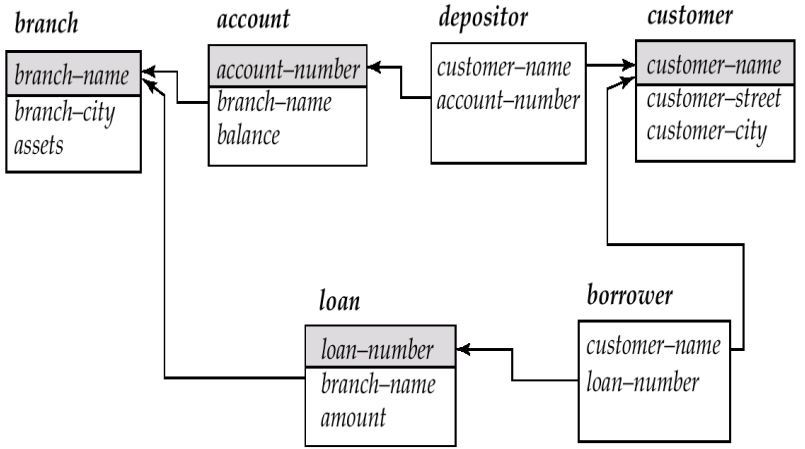
account (*account_number*, *branch_name*, balance)

loan (*loan_number*, *branch_name*, amount)

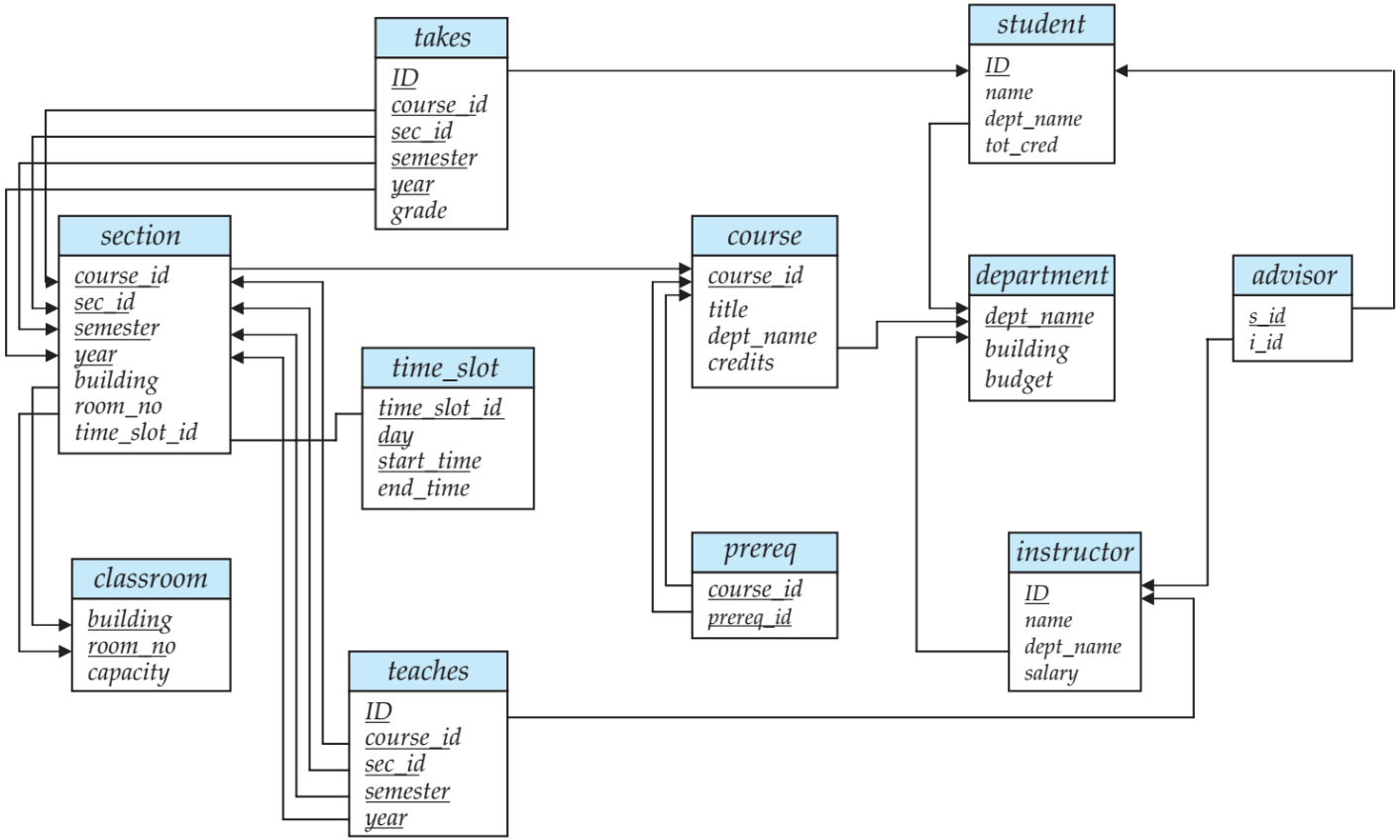
depositor (*customer_name*, *account_number*)

borrower (*customer_name*, *loan_number*)

Banking database



Schema Diagram (模式图)
for the banking enterprise



Schema diagram for the university database

Outline

- ☞ **Join Expressions**
- Views
- Transactions
- Integrity Constraints
- Data Types in SQL
- Index Definition in SQL
- Authorization

The Natural Join

select $A_1, A_2, \ldots, A_n$
from r_1 ***natural join*** r_2 ***natural join*** ***…natural join*** r_m
where *P;*

select *name, course_id*
from *instructor* ***natural join*** *teaches;*

select *name, title*
from *instructor* ***natural join*** *teaches, course*
where *teaches.course_id = course.course_id;*

Join Expressions

- **Join operations**
  - Take two relations and return another relation as the results
- **Join type**
  - Define how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated
- **Join condition**
  - Define which tuples in the two relations match, and what attributes are present in the result of the join

Join Types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using (A_1, A_2, ..., A_n)

Relations for Examples

Loan_number	*Branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Relation *loan*

Customer_name	*Loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

Relation *borrower*

Note: borrower information is missing for L-260 and loan information is missing for L-155

Joined Relations – Examples

*loan **inner join** borrower **on***
loan.loan_number = borrower.loan_number

Loan_number	*Branch_name*	*amount*	*Customer_name*	*Loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

*loan **left outer join** borrower **on***
loan.loan_number = borrower.loan_number

Loan_number	*Branch_name*	*amount*	*Customer_name*	*Loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

loan ***natural inner join*** *borrower*

Loan_number	*Branch_name*	*amount*	*Customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

loan ***natural right outer join*** *borrower*

Loan_number	*Branch_name*	*amount*	*Customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Joined Relations – Examples

loan full outer join borrower using (loan_number)

Loan_number	*Branch_name*	*amount*	*Customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

join-using = natural join

Find all customers who have either an account or a loan (but not both) at the bank:

```
select customer_name
from (depositor natural full outer join borrower)
where account_number is null or loan_number is null
```

Differences

- **join on/join using**
  - on is predicate
  - using is to specify attributes for natural join

```
select name, title
from (instructor natural join teaches) join course using (course_id);
```

- **join on/where**

```
select *
from student left outer join takes on student.ID=takes.ID
```

```
select *
from student left outer join takes
on true
where student.ID=takes.ID
```

- **Join on**：结果中会出现ID也为NULL情况，如学生没有选课，就没有课的ID；on为join的一部分，一旦连接不成自动生成NULL
- **Where**：因为join on true，断言的结论一直为真（没有空集），实际产生的是笛卡尔集合，不会出现NULL连接的情况，无法产生outer join的效果

Outline

- Join Expressions
- ☞ **Views**
- Transactions
- Integrity Constraints
- Data Types in SQL
- Authorization

View and View Updates

- Create a view of all loan data in relation **loan**, hiding the amount attribute

```
create view branch_loan as
select branch_name, loan_number
from loan
```

- Add a new tuple to relation **branch_loan**

```
insert into branch_loan
values ('Perryridge', 'L-307')
```

This insertion should be represented by the insertion of the tuple

('L-307', 'Perryridge', null)

into the relation **loan**, ***loan*** (loan_number, branch_name, **amount**)

Update of a View (Cont.)

- Updates on complex views are difficult or impossible to translate, and hence are disallowed
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

- In general, an SQL view is updatable if:
  - The select clause contains only attribute names of the relation, and does not have any **expressions**, **aggregates**, or **distinct specification**
  - Any attributes not listed in the select clause can be set to null
  - The from clause has only one relation
  - The query does not have a **group by** or **having** clause

Materialized Views (物化视图)

- The relation of a view is stored
- Will change if the actual relations used in the view definition change. The view is kept up-to-date
- The aggregated result is likely to be much smaller than the large relations on which the view is defined; as a result, the materialized view can be used to answer the query very quickly, avoiding reading the large underlying relations. Of course, the benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.
- Materialized view maintenance
  - Real-time updates vs. periodic updates
  - Chapter 4 (Version 7)

Outline

- Join Expressions
- Views
- ☞ **Transactions**
- Integrity Constraints
- Data Types in SQL
- Authorization

Transactions（事务）

- A transaction is a sequence of queries and update statements executed as a single unit (atomicity, 原子性)
- Transactions are started implicitly and terminated by one of
  - **commit [work]**: makes all updates of the transaction permanent in the database （提交）
  - **rollback [work]**: undoes all updates performed by the transaction
  - **[work]** means ``work'' is optional （回滚）

Transactions (Cont.)

- E.g., transfer of money from one account to another involves two steps: deduct from one account and credit to another
  - If one step succeeds and the other fails, database is in an inconsistent state
  - Either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by rollback work
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed
  - Each transaction consists of only a single statement
  - Automatic commit can be turned off, allowing multi-statement transactions, but depends on the database system
  - Another option in SQL:1999: enclose statements within

    begin atomic

    …

    end

Outline

- Join Expressions
- Views
- Transactions
- ☞ **Integrity Constraints**
- Data Types in SQL
- Index Definition in SQL
- Authorization

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
- Types
  - Domain constraints
  - Not null constraint
  - Unique constraint
  - Referential integrity
  - …

Domain Constraints (域约束)

- Domain constraints are the most elementary form of integrity constraint
- New domains can be created from existing data types, e.g.,

  create domain Dollars numeric(12, 2)
  create domain Pounds numeric(12, 2)

  - We cannot assign or compare a value of type Dollars to a value of type Pounds
- The check clause in SQL-92 permits domains to be restricted

  create domain hourly_wage numeric(5,2)
  constraint value_test check(value >= 6.00)

  - The domain has a constraint to ensure that the hourly_wage is greater than 6.00
  - The clause constraint value_test is optional but useful to indicate which constraint an update violates

Not Null Constraint

- Declare branch_name for relation branch to be not null

  branch_name char(15) ***not null***

- Declare the domain Dollars to be not null

  create domain Dollars numeric(12,2) ***not null***

Unique Constraint

- **unique $(A_1, A_2, \ldots, A_m)$**
  - The unique specification states that the attributes $A_1, A_2, \ldots, A_m$ form a **candidate key**
  - Candidate keys are **permitted to be null** (in contrast to primary keys)
  - However, candidate key attributes are permitted to be *null* unless they have explicitly been declared to be ***not null***. Recall that a null value does not equal any other value

The check Clause

- **check (P)**, where P is a predicate
  - E.g., declare branch_name as the **primary key** for relation branch and ensure that the values of assets are non-negative

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

Referential Integrity (参照完整性)

- 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
- Formal definition
  - Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2 respectively
  - If the subset α of R_2 is a foreign key referencing K_1 in relation r_1, for every t_2 in r_2, there must be a tuple t_1 in r_1 such that $t_1[K_1] = t_2[\alpha]$
  - Referential integrity constraint also called **subset dependency** since its can be written as

$$\Pi_{\alpha}(r_2) \subseteq \Pi_{K_1}(r_1)$$

Integrity Constraint Violation in Transactions

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

- That is, the constraint says that the *spouse* attribute must contain a name that is present in the *person* table.
- Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples, one for John and one for Mary, in the above relation, with the spouse attributes set to Mary and John, respectively. The insertion of the first tuple would violate the foreign-key constraint, regardless of which of the two tuples is inserted first. After the second tuple is inserted the foreign-key constraint would hold again.
- **How to handle such situations?**

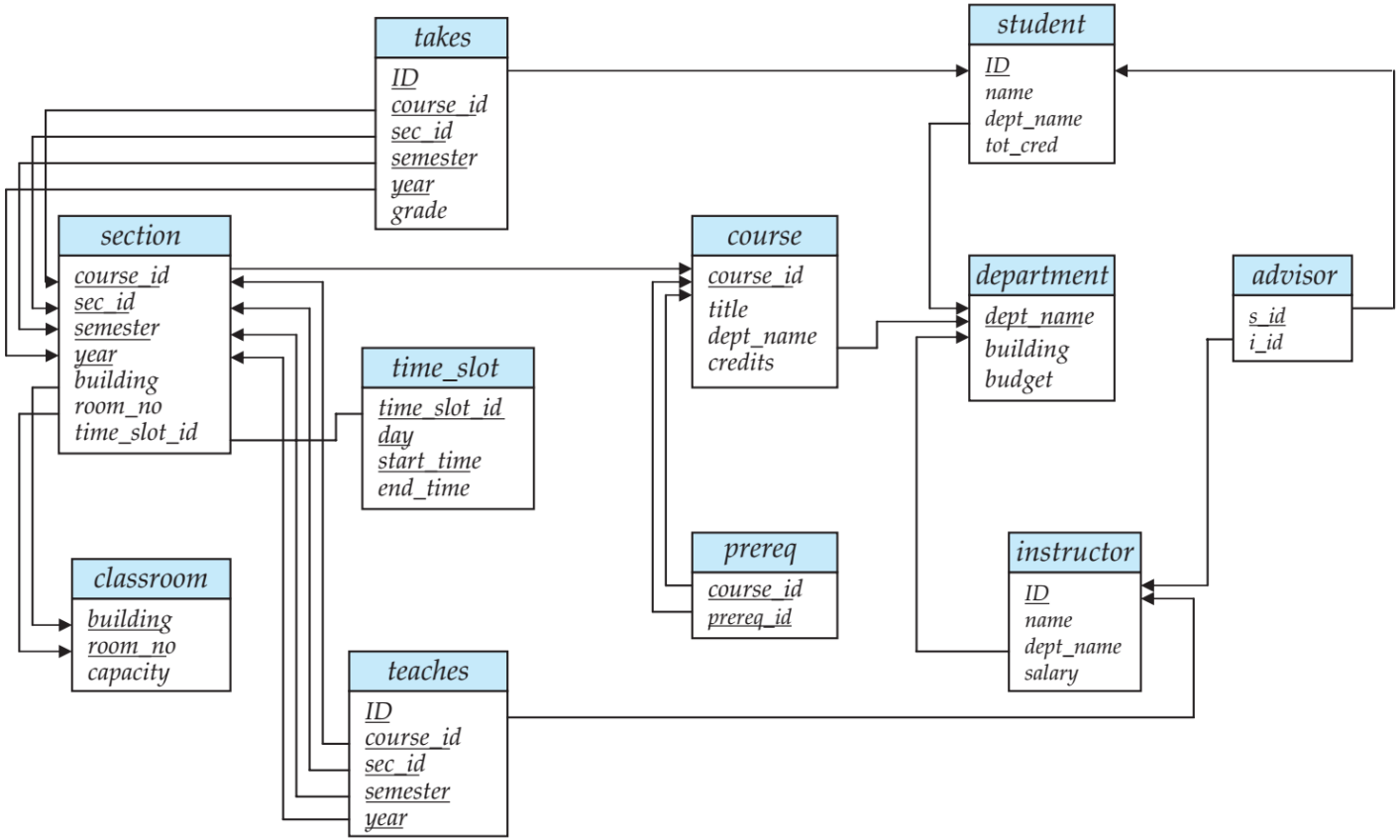
Integrity Constraint Violation in Transactions

- To handle such situations, the SQL standard allows a clause **initially deferred** to be added to a constraint specification; the constraint would then be checked at the end of a transaction, and not at intermediate steps.
- A constraint can alternatively be specified as **deferrable**, which means it is checked immediately by default, but can be deferred when desired. For constraints declared as deferrable, executing a statement **set constraints** *constraint-list* **deferred** as part of a transaction causes the checking of the specified constraints to be deferred to the end of that transaction
- How to insert a tuple without causing constraint violation?
  - Set spouse to null initially, update after inserting all persons (not possible if spouse attributes declared to be not null)
  - OR defer constraint checking

    set constraints *constraint_list* ***deferred***

Complex Check Clauses

- Every section must assigned a time_slot
  - *check (time_slot_id in (**select** time_slot_id **from** time_slot))*
  - Can we use a foreign key here?
- Every section has at least one instructor teaching the section
  - how to write this?
  - *check ((course_id, sec_id, semester, year) in (**select** course_id, sec_id, semester, year **from** takes))*
  - Can we use a foreign key here?
- Unfortunately: subquery in check clause or create assertion (断言) is not supported by many database systems
  - Alternative: **triggers** (later)



Schema diagram for the university database

Database Modification

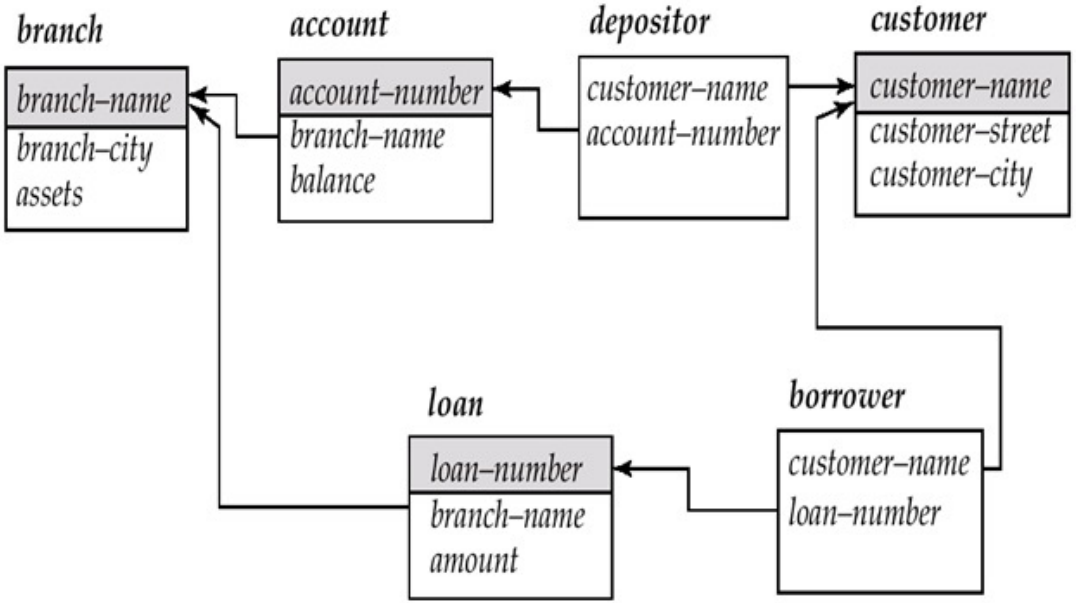
- r_2's attribute set α reference r_1 on attributes K
- Insert
  - If a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$. That is $t_2[\alpha] \in \Pi_K(r_1)$
- Delete
  - If a tuple t_1 is deleted from r_1, the database system must compute the set of tuples in r_2 that reference t_1: $\sigma_{\alpha=t_1[K]}(r_2)$
  - If this set is not empty
    - either the delete command is rejected as an error, or
    - the tuples that reference t_1 must be deleted (cascading deletions are possible)
    - Alternatively, set the values of attributes α to null in r_2

Database Modification (Cont.)

- Update
  - If a tuple t_2 is updated in relation r_2 and the update modifies values for foreign key α, then a test similar to the insert case is made
  - If a tuple t_1 is updated in r_1, and the update modifies values for the primary key(K), then a test similar to the delete case is made:
    - The system must compute $\sigma_{\alpha=t_1[K]}(r_2)$ using the old value of t_1
    - If this set is not empty
      - the update may be rejected as an error, or
      - the update may be cascaded to the tuples in the set, or
      - the tuples in the set may be deleted.

Database Modification

- r_2=$depositor$, α=customer-name
- r_1=customer, K=$custom-name$



Referential Integrity in SQL

- Primary, candidate, foreign keys can be specified as part of the SQL create table statement:
  - The primary key clause
  - The unique key clause
  - The foreign key clause
- By default, a foreign key references the primary key attributes of the referenced table

  foreign key (account_number) references account
- Short form for specifying a single column as foreign key

  account_number char (10) references account

Referential Integrity in SQL – Example

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

create table branch
    (branch_name    char(15),
    branch_city     char(30),
    assets          integer,
    primary key (branch_name));
```

```
create table account
    (account_number  char(10),
    branch_name      char(15),
    balance          integer,
    primary key (account_number),
    foreign key (branch_name) references branch);


create table depositor
    (customer_name   char(20),
    account_number   char(10),
    primary key (customer_name, account_number),
    foreign key (account_number) references account,
    foreign key (customer_name) references customer);
```

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,
     …
);
```

- Due to the **on delete cascade** clauses, if a delete of a tuple in department results in referential-integrity constraint violation, the delete "cascades" to the course relation, the tuples that refer to the department that should be deleted
- Cascading updates are similar

Cascading Actions in SQL (Cont.)

- If there is a chain of foreign-key dependencies across multiple relations, with on delete cascade specified for each dependency, a deletion or update at one end of the chain can propagate across the entire chain

- Referential integrity is only checked at the end of a transaction
  - Intermediate steps are allowed to violate referential integrity provided later steps remove the violation
  - Otherwise it would be impossible to create some database states, e.g. insert two tuples whose foreign keys point to each other
    - E.g. spouse attribute of relation married_person(name, address, spouse)

Referential Integrity in SQL (Cont.)

- Alternative to cascading
  - on delete set null
  - on delete set default

- Null values in foreign key attributes complicate SQL referential integrity semantics
  - if any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint

Assertions (断言)

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy
- An assertion in SQL takes the form

  create assertion <assertion-name> check <predicate>

- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used **with great care**

Example

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

```
create assertion sum_constraint check
   (not exists
      (select *
     from branch
       where (select sum(amount)
            from   loan
                  where loan.branch_name = branch.branch_name)
                >= (select sum(balance)
                  from   account
                        where loan.branch_name = branch.branch_name)))
```

Example

- Every loan has at least one borrower who maintains an account with a minimum balance at least $1000

```
create assertion balance_constraint check
(not exists (
   select *
  from   loan
   where not exists (
       select *
       from  borrower, depositor, account
       where loan.loan_number = borrower.loan_number
          and  borrower.customer_name = depositor.customer_name
          and  depositor.account_number = account.account_number
          and  account.balance >= 1000)))
```

Note: SQL has no (**for all**) predicate, so $(\forall x)P \equiv \neg(\exists x(\neg P))$

Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- ☞ **Data Types in SQL**
- Index Definition in SQL
- Authorization

Built-in Data Types in SQL

- **date:** dates, containing a (4 digit) year, month and day
  - E.g., date '2005-07-27'
- **time:** time of day, in hours, minutes and seconds
  - E.g., time '09:00:30' time '09:00:30.75'
- **timestamp:** date plus time of day
  - E.g., timestamp '2005-07-27 09:00:30.75'
  - timestamp (p) specifies the number of digits after the decimal point
- **interval:** period of time
  - E.g., 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

Built-in Data Types in SQL (Cont.)

- Can extract values of individual fields from date/time/timestamp, e.g.,
  - *extract (year from current_date)*
- Can cast string types to date/time/timestamp, e.g.,
  - *cast <string-valued-expression> as date*
  - *cast <string-valued-expression> as time*

Default Values

- A default value to be specified for an attribute

```
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));
```

- How an insertion can omit the value for the tot_cred attribute?

```
insert into student(ID, name, dept_name)
        values ('12789', 'Newman', 'Comp. Sci.');
```

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. The interpretation is left to an application outside of the database system
  - **clob**: character large object - object is a large collection of character data
  - E.g. book_review **clob**(10KB)
    image **blob** (10MB)
    movie **blob**(2GB)
  - When a query returns a large object, **a locator (pointer)** is returned rather than the large object itself

User-Defined Types

- Create type construct in SQL creates user-defined type
  - *create type Dollars as numeric (12,2) [final]*
- Create domain construct in SQL-92 creates user-defined domain types
  - *create domain person_name char(20) not null*
- Types and domains are similar. Domains can have constraints, such as not null/default values, specified on them

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

Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- Data Types in SQL
- ☞Index Definition in SQL
- Authorization

Index Creation

- Many queries reference only a small proportion of the records in a table.
- 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

```
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

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

  - Can be executed by using the index to find the required record, without looking at all records of relation student
  - More details on index in Chapter 14 (Version 7)(Binary tree, B+ tree, B tree, Hash…)

Outline

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- Data Types in SQL
- Index Definition in SQL
- ☞ **Authorization**

Security

- Security - protection from malicious attempts to steal or modify data
  - Database system level
    - Authentication and Authorization mechanisms to allow specific users access only to required data
  - Operating system level
    - Operating system super-users can do anything they want to the database
  - Network level: must use encryption to prevent
    - Eavesdropping (窃听，unauthorized reading of messages)
    - Masquerading (伪装、冒充，pretending to be an authorized user or sending messages supposedly from authorized users)

Security (Cont.)

- Security protection from malicious attempts to steal or modify data
  - Physical level
    - Physical access to computers allows destruction of data by intruders, and traditional lock-and-key security is needed
    - Computers must also be protected from floods, fire, etc. (cf. Chapter 19 Recovery)
  - Human level
    - Users must be screened to ensure that an authorized users do not give access to intruders
    - Users should be trained on password selection and secrecy

Authorization

- Forms of authorization on parts of the database
  - Read authorization - allows reading, but not modification of data.
  - Insert authorization - allows insertion of new data, but not modification of existing data.
  - Update authorization - allows modification, but not deletion of data.
  - Delete authorization - allows deletion of data

Authorization (Cont.)

- Forms of authorization to modify the database schema
  - Index authorization - allows creation and deletion of indices
  - Resources authorization - allows creation of new relations
  - Alteration authorization - allows addition or deletion of attributes in a relation
  - Drop authorization - allows deletion of relations

Authorization on Views

- Users can have the authorization on **views** without any authorization on the relations used in the view definition
  - Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job
- A combination of **relational-level security** and **view-level security** can be used to limit a users' precisely access to the data that they need

View Example

- Suppose a bank clerk Deny needs to know the names of the customers of each branch, but is not authorized to see specific loan information
  - Deny does not have permit of direct access to the loan relation, but grant access to the view cust_loan, which consists only of the names of customers and the branches at which they have a loan
  - The cust_loan view is defined in SQL as follows:

```
create view cust_loan as
    select branch_name, customer_name
    from   borrower, loan
    where borrower.loan_number = loan.loan_number
```

View Example (Cont.)

- The clerk is authorized to see the result of the query:

  *select * from cust_loan*

- When the query processor translates the result into a query on the actual relations in the database, we obtain a query on borrower and loan

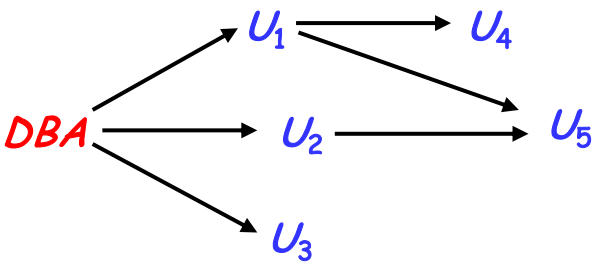
- Authorization must be checked on the clerk's query before query processing replaces a view by the definition of the view

Authorization on Views

- Creation of a view does not require resources authorization since no real relation is being created
- The creator of a view gets only those privileges that provide no additional authorization beyond that he already had
  - E.g., if creator of view cust_loan had only read authorization on borrower and loan, he gets only read authorization on cust_loan

Transfer/Granting of Privileges

- The passage of authorization from one user to another may be represented by an **authorization graph**
- The **nodes** of this graph are the users
- The **root** of the graph is the database administrator
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j
- E.g., consider graph for update authorization on loan



Authorization Grant Graph

- Requirement: All edges in an authorization graph must be part of some path originating with the root
- If DBA revokes grant from U_1:
  - Grant must be revoked from U_4 since U_1 no longer has authorization
  - Grant must not be revoked from U_5 since U_5 has another authorization path from DBA through U_2
- Must prevent cycles of grants with no path from the root:
  - DBA grants authorization to U_7
  - U_7 grants authorization to U_8
  - U_8 grants authorization to U_7
  - DBA revokes authorization from U_7
  - Must revoke grant U_7 to U_8 and from U_8 to U_7 since there is no path from DBA to U_7 or to U_8 anymore

Security 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

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - E.g., *grant select on branch 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
- **references**: ability to declare foreign keys when creating relations
- **usage**: In SQL-92, authorizes a user to use a specified domain
- **all privileges**: used as a short form for all the allowable privileges

Privilege to Grant Privileges

- **with grant option**
  - Allow a user who is granted a privilege to pass the privilege on to other users
  - E.g., give U_1 the select privilege on branch and allows U_1 to grant this privilege to others

    grant select on branch to U_1 with grant option

Roles (角色)

- **Roles** permit common privileges for a class of users can be specified just once by creating a corresponding "role"
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles

```
create role teller
create role manager

grant select on branch to teller
grant update (balance) on account to teller
grant all privileges on account to manager

grant teller to manager
grant teller to alice, bob
grant manager to avi
```

Revoking Authorization in SQL

- The revoke statement is used to revoke authorization.

  Revoke *<privilege list>*

  on *<relation name or view name>* ***from*** *<user list> [restrict|cascade]*

- Example:

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

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the revoke
- We can prevent cascading by specifying restrict:

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

  - with restrict, the revoke command fails if cascading revokes are required

Revoking Authorization in SQL (Cont.)

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

Limitations of SQL Authorization

- SQL does not support authorization at a tuple level
  - E.g., we cannot restrict students to see only their own grades
- With the growth in Web access to databases, database accesses come primarily from application servers
  - End users don't have database user ids, they are all mapped to the same database user id
- The task of authorization in such cases falls on the application program, with no support from SQL
  - **Benefit:** fine-grained authorizations, such as to individual tuples, can be implemented by the application.
  - **Drawbacks**
    - Authorization must be done in application code, and may be dispersed all over the application
    - Checking for the authorization loopholes (漏洞) becomes very difficult since it requires reading large amounts of application code

Audit Trails (审计追踪)

- An audit trail is a log of all changes (inserts / deletes / updates) to the database along with information such as
  - which user performed the change
  - when the change was performed

- Used to track erroneous/fraudulent(欺骗性的) updates

- Can be implemented using triggers, but many database systems provide direct support

Encryption

- Data may be encrypted when database authorization provisions do not offer sufficient protection

- Properties of good encryption technique:
  - Relatively simple for authorized users to **encrypt** and **decrypt** data
  - Encryption scheme depends **not on** the secrecy of the algorithm but **on** the secrecy of a parameter of the algorithm called the **encryption key** (密钥)
  - Extremely difficult for an intruder to determine the encryption key

Encryption (Cont.)

- Data Encryption Standard (DES)
  - Substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorized users via a secure mechanism
  - Scheme is no more secure than the key transmission mechanism since the key has to be shared

- Advanced Encryption Standard (AES)
  - a new standard replacing DES, and is based on the Rijndael algorithm, but is also dependent on shared secret keys

Encryption (Cont.)

- Public-key encryption
  - each user has two keys:
    - public key – used to encrypt data, but cannot be used to decrypt data
    - private key -- used to decrypt data
  - Encryption scheme is impossible or extremely hard to decrypt data given only the public key
  - The RSA public-key encryption scheme is based on the hardness of factoring a very large number (100's of digits) into its prime components.

Authentication

- **Password-based authentication** is widely used, but is susceptible to sniffing on a network

- *Challenge-response systems* avoid transmission of passwords
  - DB sends a (randomly generated) challenge string to user
  - User encrypts string and returns result
  - DB verifies identity by decrypting result
  - Can use public-key encryption system by DB sending a message encrypted using user's public key, and user decrypting and sending the message back

Authentication (cont.)

- **Digital signatures** are used to verify authenticity of data
  - E.g. use private key (in reverse) to encrypt data, and anyone can verify authenticity by using public key (in reverse) to decrypt data. Only holder of private key could have created the encrypted data.
  - Digital signatures also help ensure nonrepudiation (不可否认): sender cannot later claim to have not created the data

- Blockchain techniques

Review Terms

- Join types
  - Inner and outer join
  - Left, right and full outer join
  - Natural, using, and on
- View definition
- Materialized views
- View update
- Transactions
  - Commit work
  - Rollback work
  - Atomic transaction
- Integrity constraints
- Domain constraints
- Unique constraint
- Check clause
- Referential integrity
  - Cascading deletes
  - Cascading updates
- Assertions
- Date and time types
- Default values
- Indices
- Large objects
- User-defined types
- Domains
- Schemas
- Authorization
- Privileges
  - select
  - insert
  - update
  - All privileges
  - Granting of privileges
  - Revoking of privileges
  - Privilege to grant privileges
  - Grant option
- Roles
- Authorization on views
- Execute authorization
- Invoker privileges
- Row-level authorization

Homework

- **Further Reading**
  - Chapter 4
- **Exercises**
  - 4.7, 4.16, 4.18
- **Submission**
  - Deadline: 12:00pm, March 19, 2025

End of Lecture 4/1