数据库系统原理

Database System Principle

邵蓥侠

Email：shaoyx@bupt.edu.cn

北京邮电大学计算机学院

计算机应用技术中心

PART 2

RELATIONAL DATABASES

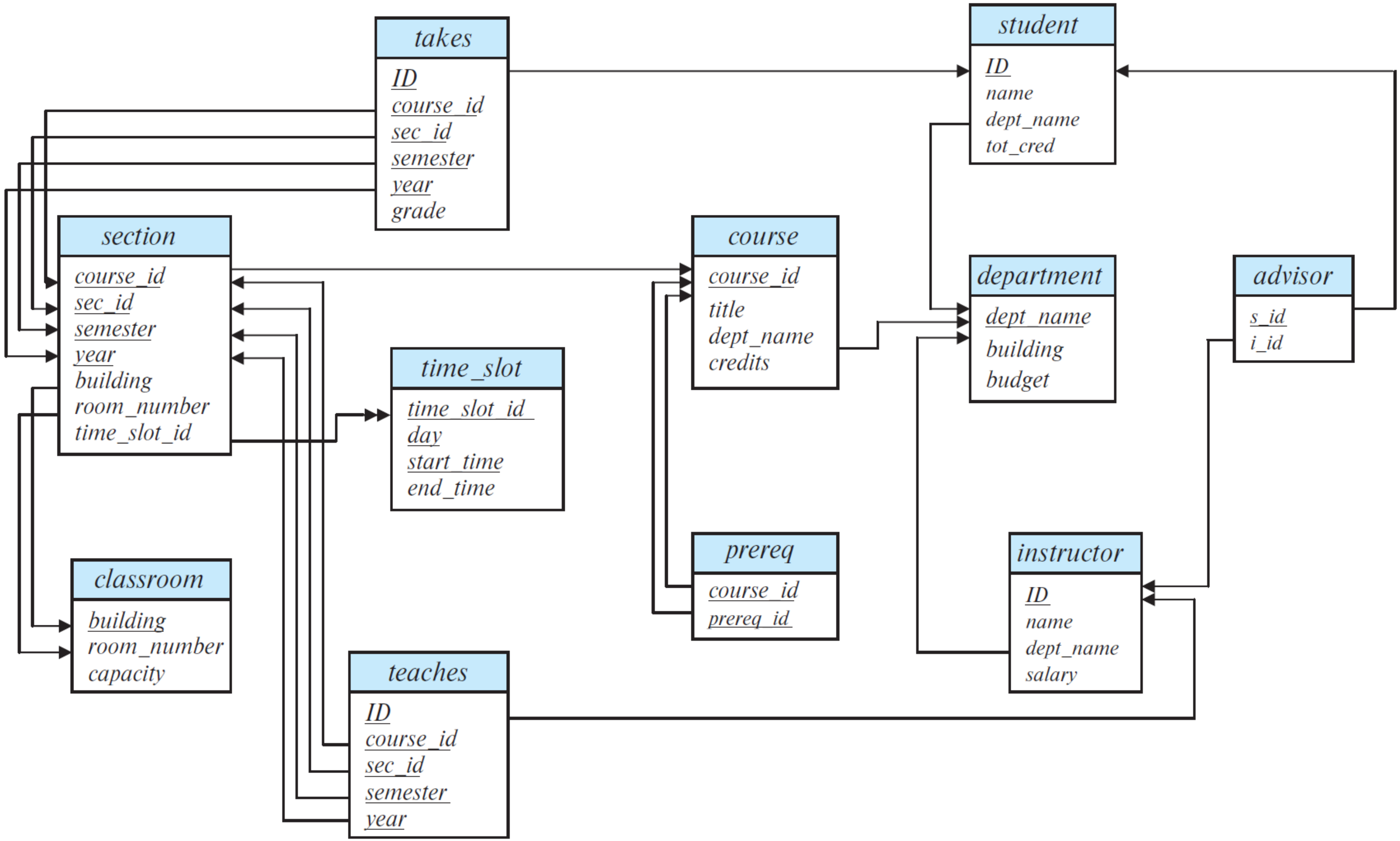
Chapter 4

Intermediate SQL

- Join Expressions（连接表达式）
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas

先讲4.5 扩展的数据类型

- Authorization
  - Security in SQL


takes
ID
course_id
sec_id
semester
year
grade
student
ID
name
dept_name
tot_cred
section
course_id
sec_id
semester
year
building
room_number
time_slot_id
course
course_id
title
dept_name
credits
department
dept_name
building
budget
advisor
s_id
i_id
time_slot
time_slot_id
day
start_time
end_time
classroom
building
room_number
capacity
prereq
course_id
prereq_id
instructor
ID
name
dept_name
salary
teaches
ID
course_id
sec_id
semester
year

§ 4.5 SQL Data Types and Schemas

4.5.1 Data and Time Types in SQL

- In addition to the basic data types, e.g. char(n), varchar(n), int, smallint, numeric(p, d), real, double precision, float(n), the other built-in data types include
  - date: Dates, containing a (4 digit) year, month and date
    - e.g. date '2017-7-27
  - time: Time of day, in hours, minutes and seconds
    - e.g. time '09:00:30.75
  - timestamp: date plus time of day
    - e.g. timestamp '2017-7-27 09:00:30.75'
  - interval: period of time
    - if *x* and *y* are of type date, then *x-y* is an interval whose value is the number of days from date *x* to date *y*
    - e.g. interval '1' day='2017-7-27' – '2017-7-28'

Built-in Data Types in SQL (cont.)

- cast *e* as *t*
  - convert a character string (or string valued expression) *e* to the type *t* , e.g. date/time/timestamp
  - e.g. **cast** ‘2017-07-20’ **as date**
  - e.g. **cast** ‘09:00:30.75’ **as time**
- For a ***day*** or ***time*** value *d*, its individual fields can be extracted
  - e.g. **extract** (*year* **from** 2017-07-20) =2017
  - e.g. **extract** (*timezone_hour* **from** 2017-7-27 09:00:30.75) =09

4.5.4 Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*
- **blob**: binary large object （二进制大型对象）
  - the 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
  - the 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 !

Large-Object Types (cont.)

- E.g.

  book_view **clob**(10KB)

  image **blob**(10MB)

  movie **blob**(2GB)

- Now, Oracle, DB2, and SQL Server also support the XML data type integrated in relational schemas, i.e. the attributes of XML data type in relational tables

e.g. Create table *book*

  book_id **int primary key**

  book_view **clob**(10KB)

  image **blob**(10MB)

  movie **blob**(2GB)

  catalog **XML**

4.5.5 User-defined Types

- The **create type** clause can be used to define new user-defined types
  - e.g. **create type** *Dollars* **as numeric (12,2) final**

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

- ***cast***, i.e. *convert*, values of one type to another domain
  - ***cast*** (*department.budget* to *numeric*(12,2))

  , then the expression

  (*department.budget* + 20)

  can be evaluated

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'));

SQL Server2005
类型转换

源数据类型: \ 目标数据类型:	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml
binary		◎	◎	◎	◎	◎	◎	◎	◎	◎	○	○	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	○	○	◎	◎
varbinary	◎		◎	◎	◎	◎	◎	◎	◎	◎	○	○	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	○	○	◎	◎
char	●	●		◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	●	◎	◎	◎	◎	◎	◎
varchar	●	●	◎		◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	●	◎	◎	◎	◎	◎	◎
nchar	●	●	◎	◎		◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	●	◎	○	◎	◎	◎	◎
nvarchar	●	●	◎	◎	◎		◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	●	◎	○	◎	◎	◎	◎
datetime	●	●	◎	◎	◎	◎		◎	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	◎	○
smalldatetime	●	●	◎	◎	◎	◎	◎		●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	◎	○
decimal	◎	◎	◎	◎	◎	◎	◎	◎	*	*	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	○	○	○	○	◎	○
numeric	◎	◎	◎	◎	◎	◎	◎	◎	*	*	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	○	○	○	○	◎	○
float	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	◎	◎	◎	◎	◎	○	○	○	○	○	◎	○
real	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	◎	◎	◎	◎	○	○	○	○	○	◎	○
bigint	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	◎	◎	◎	◎	○	○	○	○	◎	○
bigint	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	◎	◎	◎	◎	○	○	○	○	◎	○
int(INT4)	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	◎	◎	◎	○	○	○	○	◎	○
smallint(INT2)	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	◎	◎	○	○	○	○	◎	○
tinyint(INT1)	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	◎	○	○	○	○	◎	○
money	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	◎	○	○	○	○	◎	○
smallmoney	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	◎	○	○	○	○	◎	○
bit	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎	◎		◎	○	○	○	○	◎	○
timestamp	◎	◎	◎	◎	○	○	◎	◎	◎	◎	○	○	◎	◎	◎	◎	◎	◎	◎		○	◎	○	○	○	○
uniqueidentifier	◎	◎	◎	◎	◎	◎	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○	○	○	◎	○
image	◎	◎	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◎	○		○	○	○	○
ntext	○	○	◎	◎	◎	◎	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		◎	○	◎
text	○	○	◎	◎	◎	◎	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◎		○	◎
sql_variant	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○	○	○		○
xml	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◪

● 显式转换

◎ 隐式转换

○ 不允许转换

* 要求使用显式转换，以防止使用隐式转换时降低了精度或减少了小数位数

◪ 仅当源或目标为非类型化的 xml 时，才支持 XML 数据类型之间的隐式转换；
否则，它们之间的转换必须为显式转换。

CAST 和 CONVERT (Transact-SQL)

⊟ 全部折叠 ⌄ 语言筛选器: 全部

将一种数据类型的表达式显式转换为另一种数据类型的表达式。CAST 和 CONVERT 提供相似的功能。

Transact-SQL 语法约定

⊟ 语法

```
Syntax for CAST:
CAST ( expression AS data_type [ (length ) ])

Syntax for CONVERT:
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

⊟ 备注

隐式转换指那些没有指定 CAST 或 CONVERT 函数的转换。显式转换指那些需要指定 CAST 或 CONVERT 函数的转换。以下图例显示了可对 SQL Server 2005 系统提供
数据类型转换。其中包括 xml、bigint 和 sql_variant。不存在对 sql_variant 数据类型的赋值进行的隐式转换，但是存在转换为 sql_variant 的隐式转换。

A. 同时使用 CAST 和 CONVERT

每个示例都检索列表价格的第一位是 3 的产品的名称，并将 ListPrice 转换为 int。

```
-- Use CAST
USE AdventureWorks;
GO
SELECT SUBSTRING(Name, 1, 30) AS ProductName, ListPrice
FROM Production.Product
WHERE CAST(ListPrice AS int) LIKE '3%';
GO
```

varchar, e.g. 350.00

```
-- Use CONVERT.
USE AdventureWorks;
GO
SELECT SUBSTRING(Name, 1, 30) AS ProductName, ListPrice
FROM Production.Product
WHERE CONVERT(int, ListPrice) LIKE '3%';
GO
```

B. 使用包含算术运算符的 CAST

以下示例将本年度截止到现在的全部销售额 (SalesYTD) 除以佣金百分比 (CommissionPCT)，从而得出单列计算结果 (Computed)。在舍入到最接近的整数后，将此结果转换为 int 数据类型。

```
USE AdventureWorks;
GO
SELECT CAST(ROUND(SalesYTD/CommissionPCT, 0) AS int) AS 'Computed'
FROM Sales.SalesPerson
WHERE CommissionPCT != 0;
GO
```

C. 使用 CAST 进行连接

以下示例使用 CAST 连接非字符型非二进制表达式。

```
USE AdventureWorks;
GO
SELECT 'The list price is ' + CAST(ListPrice AS varchar(12)) AS ListPrice
FROM Production.Product
WHERE ListPrice BETWEEN 350.00 AND 400.00;
GO
```

float

D. 使用 CAST 生成可读性更高的文本

以下示例使用选择列表中的 CAST 将 Name 列转换为 char(10) 列。

```
USE AdventureWorks;
GO
SELECT DISTINCT CAST(p.Name AS char(10)) AS Name, s.UnitPrice
FROM Sales.SalesOrderDetail s JOIN Production.Product p on s.ProductID = p.ProductID
WHERE Name LIKE 'Long-Sleeve Logo Jersey, M';
GO
```

varchar

4.5.2 Default Values

- **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*))

- **insert into** *student*(*ID*, *name*, *dept_name*)
  values (‘12789’, ‘Newman’, ‘Comp.Sci’)

4.5.3 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 in Chapter 14

4.5.5 Schemas, Catalogs, and Environments

- How to name the relations in DB, or how to specify the names of the relations in DB
- The three-level hierarchy for naming relations
  - ***catalogs*（全文目录）**, each of which can contain ***schemas***;

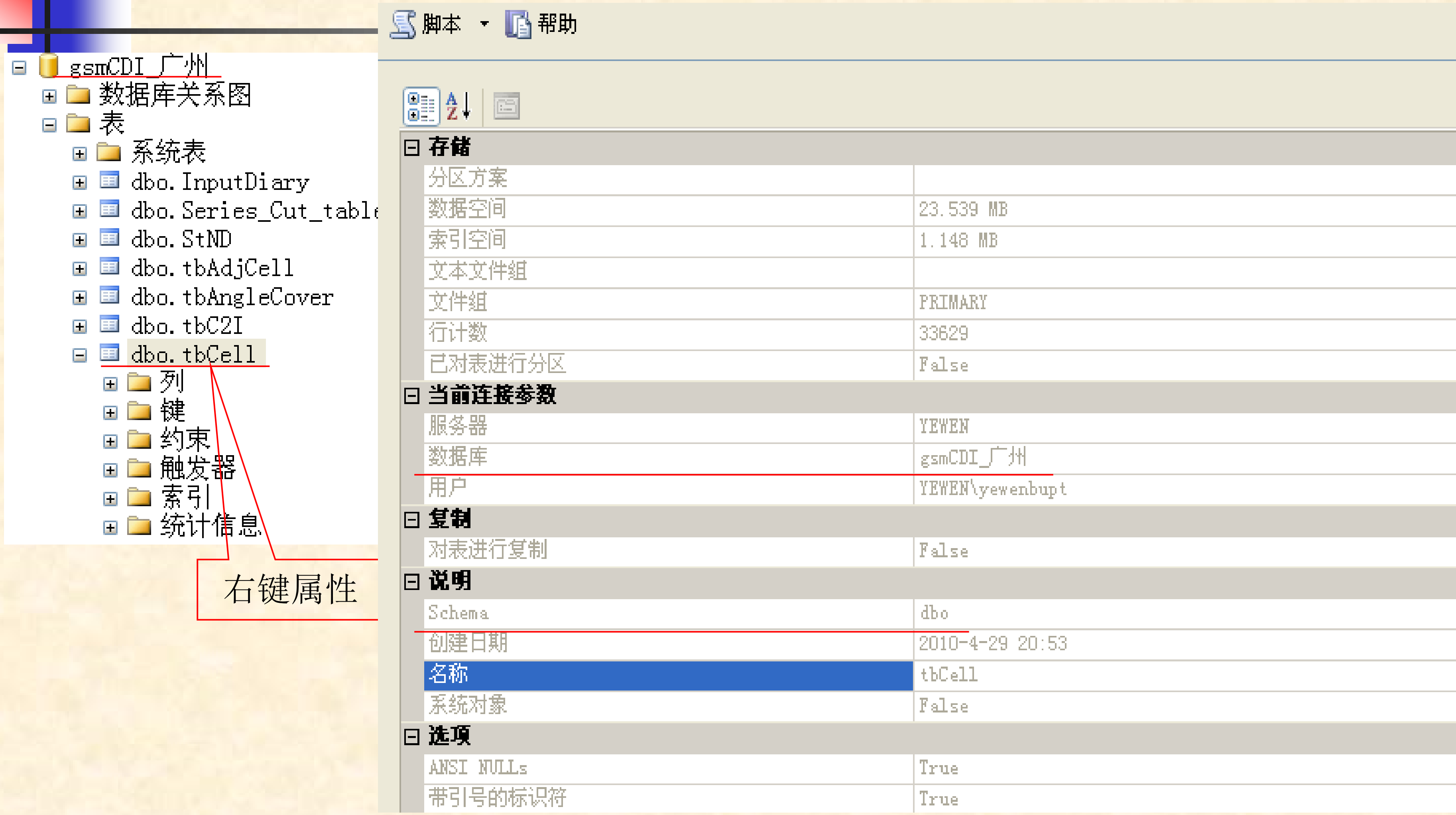
    The ***catalog*** is also called ***database***
  - ***schemas*（模式）**, in which the relations and views are stored

    the **Schema** object represents an ownership context for a Microsoft SQL Server database object.
  - relations and views

    e.g. a three-part name for the relation

    catalog5.bank_schema.account

Schemas, Catalogs, and Environments



- SQL environments, including
  - connection, catalog, schema, user identifier

4.1 Joined Expressions

- **Join operations** take two relations and return as a result another relation.
- A ***join*** operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The ***join*** operations are typically used as subquery expressions in the ***from*** clause

Joined Relations

- **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 $(A_1, A_1, \ldots, A_n)$

Natural Join operations

For the following two relations

- *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

prereq

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

Natural Join operations

- **select** *
  from *course* **natural join** *prereq*
- **select** *
  from *course* **join** *prereq* **on** *course.course_id=prereq. 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

Observe that, *course* **natural join** *prereq* **on** *course_id*
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*

- **select** *

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

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

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

- **select** *

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

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 ***inner join*** mentioned 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 **natural 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

4.2 Views

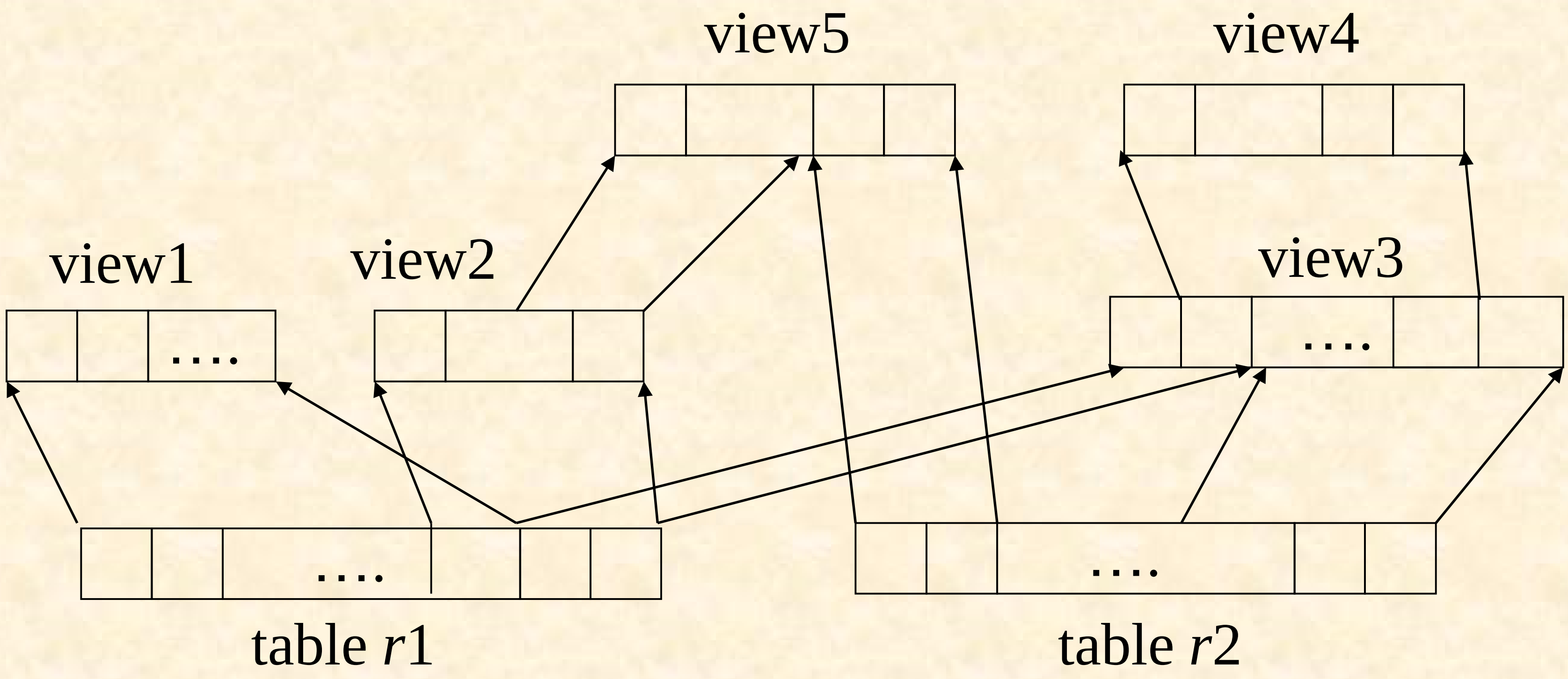
- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  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**.

Views (cont.)

- E.g. Views definition



- Views = ***projection*** on *one or more relations*

Views (cont.)

- The view is also known as the *virtual relation/table*
  - only the definition of the view itself is stored in DBS, the *tuples* of the view is stored in the relations on which the view is defined
  - evaluation of a view is reduced to evaluation of the relation-algebra expression that define the view

- Note:

  differences between the *with* clause and the *create view* clause

- Materialized views (实体化/物化视图)
  - both the definition and data of the views are stored in DBS

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 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’;

Views Defined Using Other Views

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

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

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

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= ’History’;
- What happens if we insert (’25566’, ’Brown’, ’Biology’, 100000) into *history_instructors?*

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.

§ 4.4 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
  - An instructor name can not be *null*
  - No two instructors can have the same instructor *ID*
  - Every department name in the *course* relation must have a matching department name in the *department* relation

	attribute-level	*tuple-level*	*relation-level*
static	▪ data type ▪ data format, e.g. YY.MM.DD ▪ domain constraints, e.g.1 ▪ null value, e.g. not null	constraints among attributes values ➢ e.g.2 ➢ mapping cardinality constraints	▪ **entity integrity** ➢ e.g.3 ▪ **referential integrity** ▪ functional dependency (§ 7) ▪ statistical constraints ➢ e.g.6
dynamic	constraints on updating of attribute values or attribute definition ➢ e.g.4	constraints among attributes values ➢ e.g.5	transaction constraint: atomy, consistency, isolation, durability (§ 17, 18, 19)

Some Examples of Integrity Constraints

- E.g.1 " the *salary of manager* should not be lower than $1000" in *Employee*
- E.g.2 table T (x, y, z), z =x+y, z is a ***derived*** attributes from x and y.
- E.g.3 "the *student#* for table *student* should not be null"
- E.g.4 "the *age of students* should only be added"
- E.g.5 *when employee tuples is modified*, new.sal > old.sal + 0.5*age
- E.g.6 statistical constraints: in table *employee* , "the *salary* of manager should be four times more than that of workers"

4.4.1-4 Constraints on a Single Relation

- An SQL relation is defined using the **create table** command

  create table ***r*** (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *r* is the name of the relation
  - each A_i is an attribute name in the schema of relation *r*
  - D_i is the data type of values in the domain of attribute A_i

Constraints on a Single Relation (cont.)

- The allowed integrity constraints include
  - primary key
  - not null
  - unique
  - check (*P*), where *P* is a predicate

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

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

Constraints on a Single Relation (cont.)

- The Unique Constraint is as follows

  unique (A_1, A_2, …, A_m)

  - it 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)
  - e.g. **create table** *customer*

    (*customer-id* char(15)
    customer-name char(15)
    customer-city char(30),
    primary key (*customer-id*)
    unique (*customer-name*)

Constraints on a Single Relation (cont.)

- The check clause is applied to relation declaration as well as to domain declaration
  - **check** (*P*), where *P* is a predicate
  - e.g.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'))
);

4.4.5-6 Referential Integrity

- **Referential Integrity (**关联/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

  - E.g. 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*” (*next slide*)

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

instructor

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

department

where is ‘**Math**’?

Referential Integrity (cont.)

- Definition of ***referential integrity/subset dependecies***
  - Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2 respectively, refer to Fig. 4.0.2
    - e.g. r_1: ***department***(*dept_name*, *building*, *budgt*)

      r_2: ***course***(*course_id*, *title*, *dept_name*, *credits*),

  - the subset α of R_2 (e.g. *dept_name*) is a ***foreign key*** (from r_2, e.g. *course*) referencing K_1 in relation r_1 (e.g. *dept_name* in *department*) , if for every t_2 in r_2 there must be a tuple t_1 in r_1 such that

$$t_1[K_1] = t_2[\alpha]$$

    - note: α = K1⊆ R2

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
[illegible]	[illegible]	[illegible]	[illegible]

r_2

Figure A.5 The *course* relation.

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

r_1

Figure A.4 The *department* relation.

4.2.5 Referential Integrity (cont.)

- referential integrity constraint also called *subset dependency* since it can be written as

$$\Pi_{\alpha}(r_2) \subseteq \Pi_{K1}(r_1)$$

  - e.g. $\Pi_{dept\_name}(course) \subseteq \Pi_{dept\_name}(department)$

- a foreign key α of table r_2 references the primary key attributes K_1 of the *referenced table* r_1

- A ***good*** DB design should ensures that any relation schema R_2 (and its any tuples) can only reference other relation schema R_1 through its foreign key

Referential Integrity (cont.)

- The primary key, candidate keys and foreign keys can be specified as parts of the SQL **create table** statement:
  - the primary key clause lists attributes that comprise the primary key.
  - the unique key clause lists attributes that comprise a candidate key.
  - the foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key.

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

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**

Cascading Actions in Referential Integrity

- The *delete*/*update* operations on the *referenced* relation *department* will result in the *delete/update* on the *referencing* relation *course*

DBMS

Bio

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

r_1

Figure A.4 The *department* relation.

Bio

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure A.5 The *course* relation.

r_2

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)

（知乎话题）
数据库系统设计时，是否使用外键？！

- 外键反映了不同数据项间的逻辑关联和约束关系，是一种客观存在
  - e.g. *dept_name* in the table *department*, *course*

- 对外键的处理方式反映了对这种数据相互间关系的处理态度，需要根据实际情况而定，考虑数据完整性、处理效率等多种因素
  - 并非简单的“需要外键”，“不设外键”（e.g. 部分MySQL课程、MySQL社区的论点）

（知乎话题）数据库系统设计时，是否使用外键？！

- **方式1：** 设置数据库表的外键
  - 优点：

    DBMS自动维护数据间的关联、约束关系
  - 缺点：

    数据导入、增删改时，多个表间的相互关联，保证了数据一致性、完整性，但影响到整体数据处理效率（额外的DBMS处理开销）

    ——e.g.插入一条*dept_name*不存在的*course*元组，失败，并影响前面已经成功插入的数据——局部错误影响整体

（**知乎**话题）

数据库系统设计时是否使用外键？！

- **方式2：** 不设置外键，由应用程序在业务逻辑层维护数据一致性、完整性
  - 优点

    1）避免局部错误影响整体进度；

    2）业务逻辑层的数据一致性处理的速度可以比较快，e.g. 将数据提至内存中批量处理
  - 缺点：对应用层业务处理程序设计的额外要求
- 上述2种方式用在对数据一致性完整性要求高的场景下，e.g. 银行
- **方式3：** 不设置外键，业务层应用逻辑也不处理数据间一致性
  - 牺牲数据的正确性，换取数据处理的效率
  - e.g. 大数据应用场景下，对数据精度、正确性要求不高

4.4.7 Complex Check Conditions and Assertions

- In the SQL standard, the some construct such as ***check*** and ***assertion*** are defined to specify the complex integrity constraints.
  - **However, they not currently supported by most database systems (?)**
- E.g. For the relation *section*, the time_slot_id in each tuple should be is actually the identifier of a time_slot in the *time_slot* relation. This constraint can be defined as

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

- *check* in *create table*

```
create table student (
    ID          varchar(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    tot_cred    numeric(3,0),
    primary key (ID),
    foreign key (dept_name) references department)
    check(tot_cred>0)
```

Complex Check Conditions and Assertions (cont.)

- An ***assertion*** is a predicate expressing a condition that we wish the database always to satisfy
  - e.g. domain constraints, referential-integrity constraint
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>

  - **Also not supported by anyone, e.g. SQL Server等数据库系统似乎不支持创建assertion !!!**
- **Why？**

  When an assertion is made, the DBMS tests it for validity. Any modification to DB is allowed only if it does not cause that assertion to be violated

  - this testing may introduce a significant amount of overhead, hence assertions should be used with great care

Complex Check Conditions and Assertions (cont.)

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

- **create assertion** *credits_earned_constraints* **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’)

)

§ 4.6 Authorization

- Forms of authorization on parts of the ***database*** include
  - **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***
  - **resources** - allows creation of new relations
  - **alteration** - allows addition or deletion of attributes in a relation
  - **drop** - allows deletion of relations
  - **index**

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
- 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 U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

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

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*
- 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

- **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 Section 4.7 for more details we have omitted here.

Conclusion

- SQL Data Types and Schemas
- Join Expressions（连接表达式）
  - Left, Right, Nature, Full
- Views
- Integrity Constraints
- Authorization
  - Security in SQL

Homework

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)

Figure 3.19 Employee database.

3.16 Consider the employee database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find ID and name of each employee who lives in the same city as the location of the company for which the employee works.

b. Find ID and name of each employee who lives in the same city and on the same street as does her or his manager.

c. Find ID and name of each employee who earns more than the average salary of all employees of her or his company.

d. Find the company that has the smallest payroll.

Homework

4.14 Consider the query

select *course_id*, *semester*, *year*, *sec_id*, **avg** (*tot_cred*)
from *takes* **natural join** *student*
where *year* = 2017
group by *course_id*, *semester*, *year*, *sec_id*
having count (*ID*) >= 2;

Explain why appending **natural join** *section* in the **from** clause would not change the result.

Homework

employee (<u>*ID*</u>, *person_name*, *street*, *city*)
works (<u>*ID*</u>, *company_name*, *salary*)
company (<u>*company_name*</u>, *city*)
manages (<u>*ID*</u>, *manager_id*)

Figure 4.12 Employee database.

4.18 For the database of Figure 4.12, write a query to find the ID of each employee with no manager. Note that an employee may simply have no manager listed or may have a *null* manager. Write your query using an outer join and then write it again using no outer join at all.