

Introduction to Databases

《数据库引论》



Lecture 3: Introduction to SQL

第3讲：结构化查询语言简介

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

Outline of the Course

- **Part 0: Overview**
 - Lect. 1 (Feb. 29) - Ch1: Introduction
- ☞ **Part 1 Relational Databases**
 - Lect. 2 (Mar. 7) - Ch2: Relational model (data model, relational algebra)
 - **Lect. 3 (Mar. 14) - Ch3: SQL (Introduction)**
 - Lect. 4 (Mar. 21) - Ch4/5: Intermediate and Advanced SQL
- **Part 2 Database Design**
 - Lect. 5 (Mar. 28) - Ch6: Database design based on E-R model
 - **Apr. 4 (Tomb-Sweeping Day): no course**
 - Lect. 6 (Apr. 11) - Ch7: Relational database design
- **Midterm exam: Apr. 18**
- **Part 3 Data Storage & Indexing**
 - Lect. 7 (Apr. 25) - Ch12/13: Storage systems & structures
 - Lect. 8 (May 3 -> Apr. 28) - Ch14: Indexing
- **Part 4 Query Processing & Optimization**
 - Lect. 9 (May 10) - Ch15: Query processing
 - Lect. 10 (May 17) - Ch16: Query optimization
- **Part 5 Transaction Management**
 - Lect. 11 (May 24) - Ch17: Transactions
 - Lect. 12 (May 31) - Ch18: Concurrency control
 - Lect. 13 (Jun. 7) - Ch19: Recovery system
- **Part 6 DB Systems & Course Review**
 - Lect. 14 (Jun. 14)

Final exam: 13:00-15:00, Jun. 26

University Database

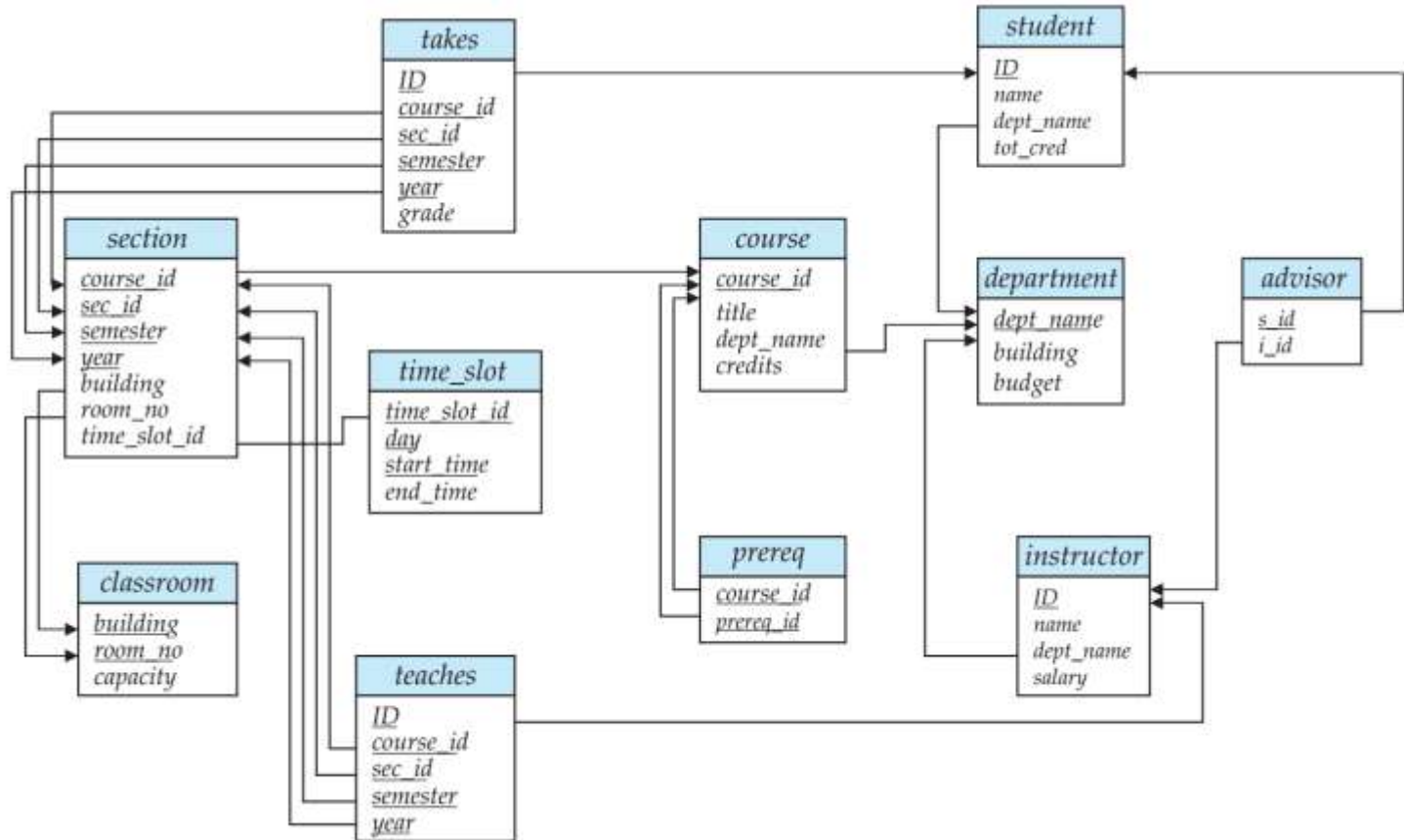
<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.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

Instructor table

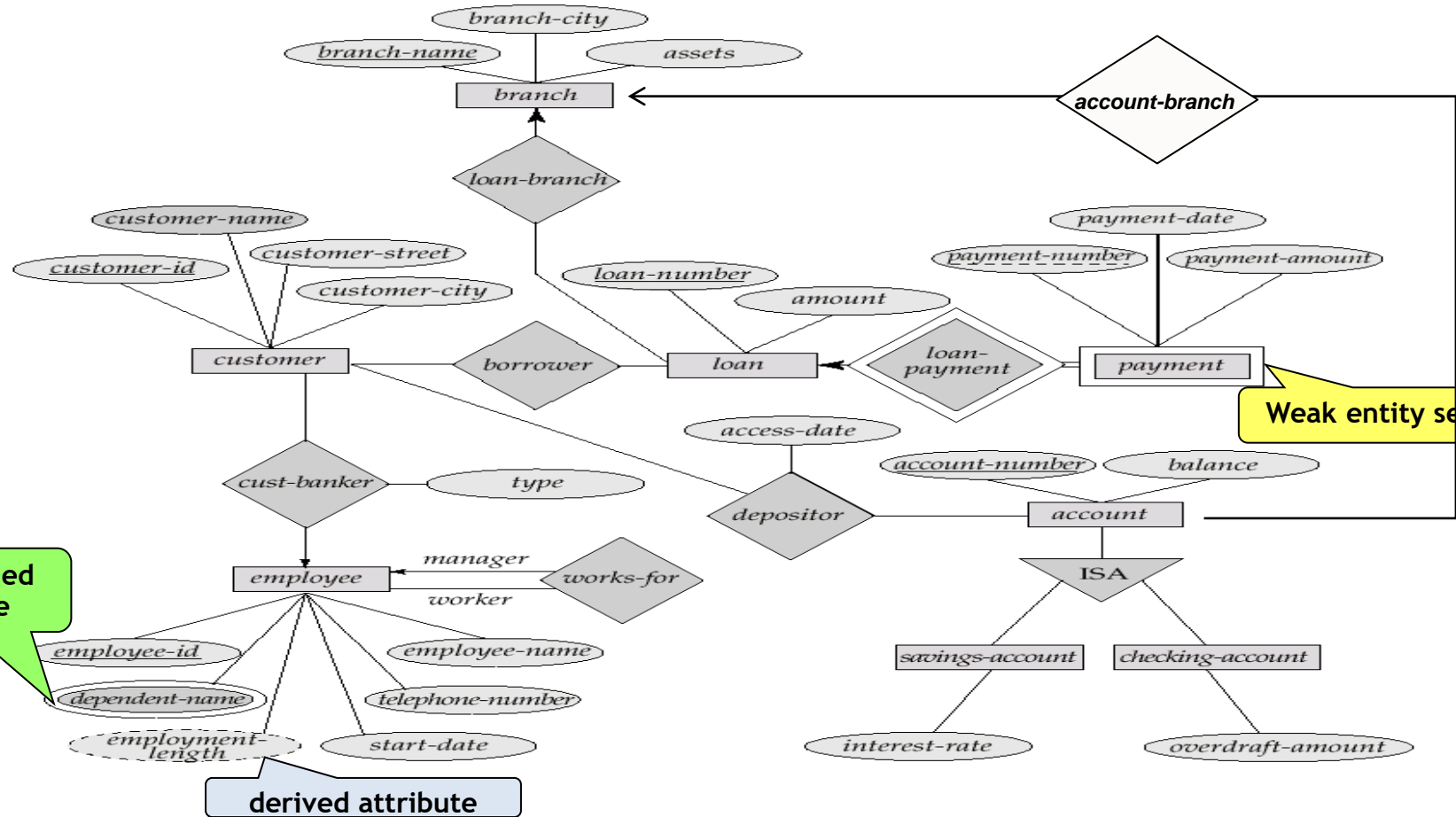
<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Student table

University Database



E-R Diagram for a Banking Enterprise



The Banking Schema

- *branch* = (*branch_name*, *branch_city*, *assets*)
- *customer* = (*customer_id*, *customer_name*, *customer_street*, *customer_city*)
- *loan* = (*loan_number*, *amount*)
- *account* = (*account_number*, *balance*)
- *employee* = (*employee_id*, *employee_name*, *telephone_number*, *start_date*)

- *dependent_name* = (*employee_id*, *dname*) (derived from a multivalued attribute)

- *account_branch* = (*account_number*, *branch_name*)
- *loan_branch* = (*loan_number*, *branch_name*)
- *borrower* = (*customer_id*, *loan_number*)
- *depositor* = (*customer_id*, *account_number*, *access_date*)
- *cust_banker* = (*customer_id*, *employee_id*, *type*)
- *works_for* = (*worker_employee_id*, *manager_employee_id*)

- *payment* = (*loan_number*, *payment_number*, *payment_date*, *payment_amount*)

- *savings_account* = (*account_number*, *interest_rate*)
- *checking_account* = (*account_number*, *overdraft_amount*)

Outline

👉 Overview of SQL

- SQL Data Definition
- Basic Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database


Overview of the SQL Query Language

- IBM SEQUEL (**S**tructured **E**nglish **Q**UERY Language) language developed as part of System R project at the IBM San Jose Research Laboratory in the early 1970s
- Later SEQUEL was renamed to **Structured Query Language (SQL)** because it was already trademarked by an airplane company
- ANSI (美国国家标准学会) and ISO (国际标准化组织) standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999, 2003, 2006, 2008
- Commercial systems offer most, if not all, **SQL-92** features, plus varying feature sets from later standards and special proprietary features
 - Not all examples here may work on the particular system

Structured Query Language (SQL)

- **SQL语言包含**
 - **数据定义语言 (Data definition language, DDL)**
 - Relation schemas
 - Integrity constraints
 - View definition
 - Authorization
 - **数据操纵语言 (Data manipulation language, DML)**
 - Queries
 - Insertion, Deletion, Updates
 - Transaction processing

Outline

- Overview of SQL
-  **SQL Data Definition**
- Basic Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

Data Definition Language (DDL)

- Allows the specification of not only a set of relations but also information about each relation, including:
 - The **schema** for each relation
 - The **domain of values** associated with each attribute
 - Integrity **constraints**
 - The set of **indices** to be maintained for each relations
 - **Security and authorization** information for each relation
 - The **physical storage structure** of each relation on disk

Domain Types in SQL

- **char(n)**
 - Fixed length character string, with user-specified length n
- **varchar(n)**
 - Variable length character strings, with user-specified maximum length n
- **int**
 - Integer (a finite subset of the integers that is machine-dependent)
- **smallint**
 - Small integer (a machine-dependent subset of the integer domain type)
- **numeric(p, d)**
 - Fixed point number (定点数), with user-specified precision of p digits, with d digits to the right of decimal point
 - Numeric(3,1) allows 44.5 to be stored exactly, but neither 444.5 nor 0.32 can be stored exactly in a field of this type

Domain Types in SQL (Cont.)

- **real, double precision**
 - Floating point and double-precision floating point numbers, with machine-dependent precision
- **float(*n*)**
 - Floating point number, with user-specified precision of at least *n* digits
- **null value**
 - Allowed in all domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
- Create domain construct in SQL-92 creates user-defined domain types
 - *create domain person_name char(20) not null*

Date/Time Types in SQL (Cont.)

- **date**
 - Dates, containing a (4 digit) year, month and date
 - E.g., date '2020-9-30'
- **time**
 - Time of day, in hours, minutes and seconds.
 - E.g., time '09:25:30' time '09:25:30.75'
- **Timestamp**
 - date plus time of day
 - E.g., timestamp '2020-9-30 09:25:30.75'

Date/Time Types in SQL (Cont.)

- **Interval**: period of time
 - Subtracting a date/time/timestamp value from another gives an interval value, e.g., Interval '1' day
 - Interval values can be added to date/time/timestamp values
- **Extract** values of individual fields from date/time/timestamp
 - E.g., **extract** (year from r.starttime)
- **Cast** string types to date/time/timestamp
 - E.g., **cast** <string-valued-expression> as date

Basic Schema Definition

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

```
create table r(A1 D1, A2 D2, ..., An Dn,  
              (integrity_constraint1), ..., (integrity_constraintk))
```

- *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*}
- Example:

```
create table branch  
  (branch_name  char(15) not null,  
   branch_city  char(30),  
   assets       integer)
```


Integrity Constraints in Creating Tables

- **not null**
- **primary key** (A_1, \dots, A_n)
- **foreign key** $(A_{k1}, A_{k2}, \dots, A_{kn})$ references s
- **check** (P) , where P is a predicate

create table instructor

```
(ID          varchar(5),  
 name       varchar(20) not null,  
 dept_name  varchar(20),  
 salary     numeric(8, 2),  
 primary key (ID),  
 check (salary >= 0) )
```

Note: **Primary key** declaration on an attribute automatically ensures **not null** and **unique** in SQL-92 onwards, needs to be explicitly stated in SQL-89

Basic Insertion and Deletion of Tuples

- Newly created table is empty
- Add a new tuple to table instructor

insert into instructor values ('10211', 'Smith', 'Computer Science', 66000)

- Insertion fails if any integrity constraint is violated

- Delete **all** tuples from table instructor

delete from instructor

Drop and Alter Table Constructs

- The **drop table** command deletes all information (**both schema and tuples**) about the dropped relation from the database
- The **alter table** command is used to add attributes to an existing relation

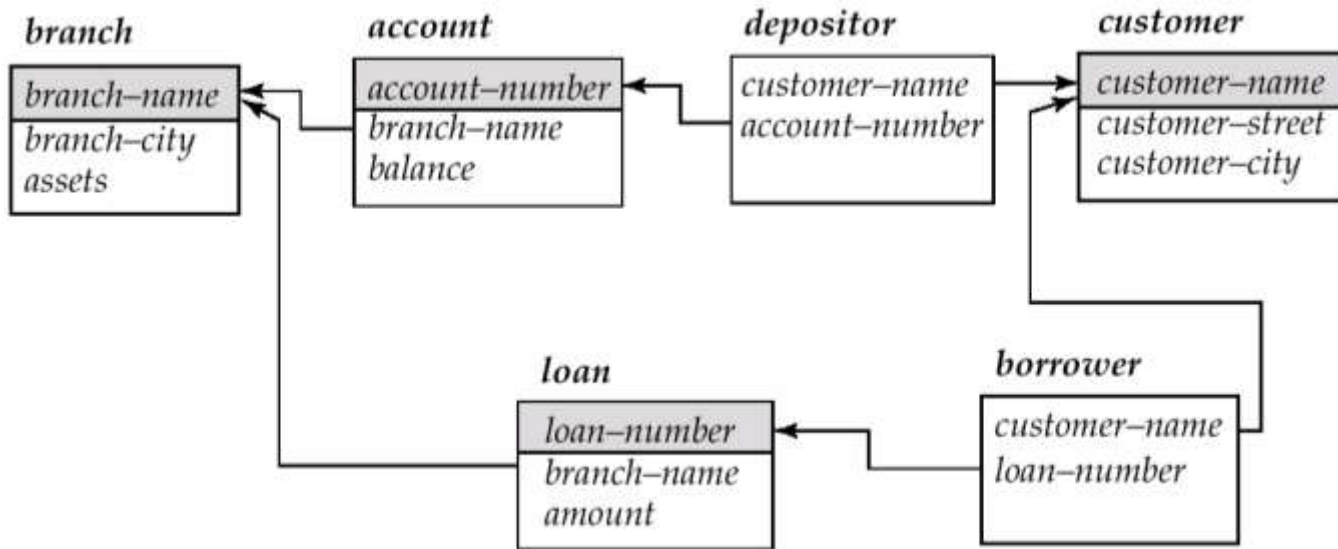
alter table r add A D

- All tuples in the relation are assigned **null** as the value for the new attribute.
- The alter table command can also be used to drop attributes of a relation

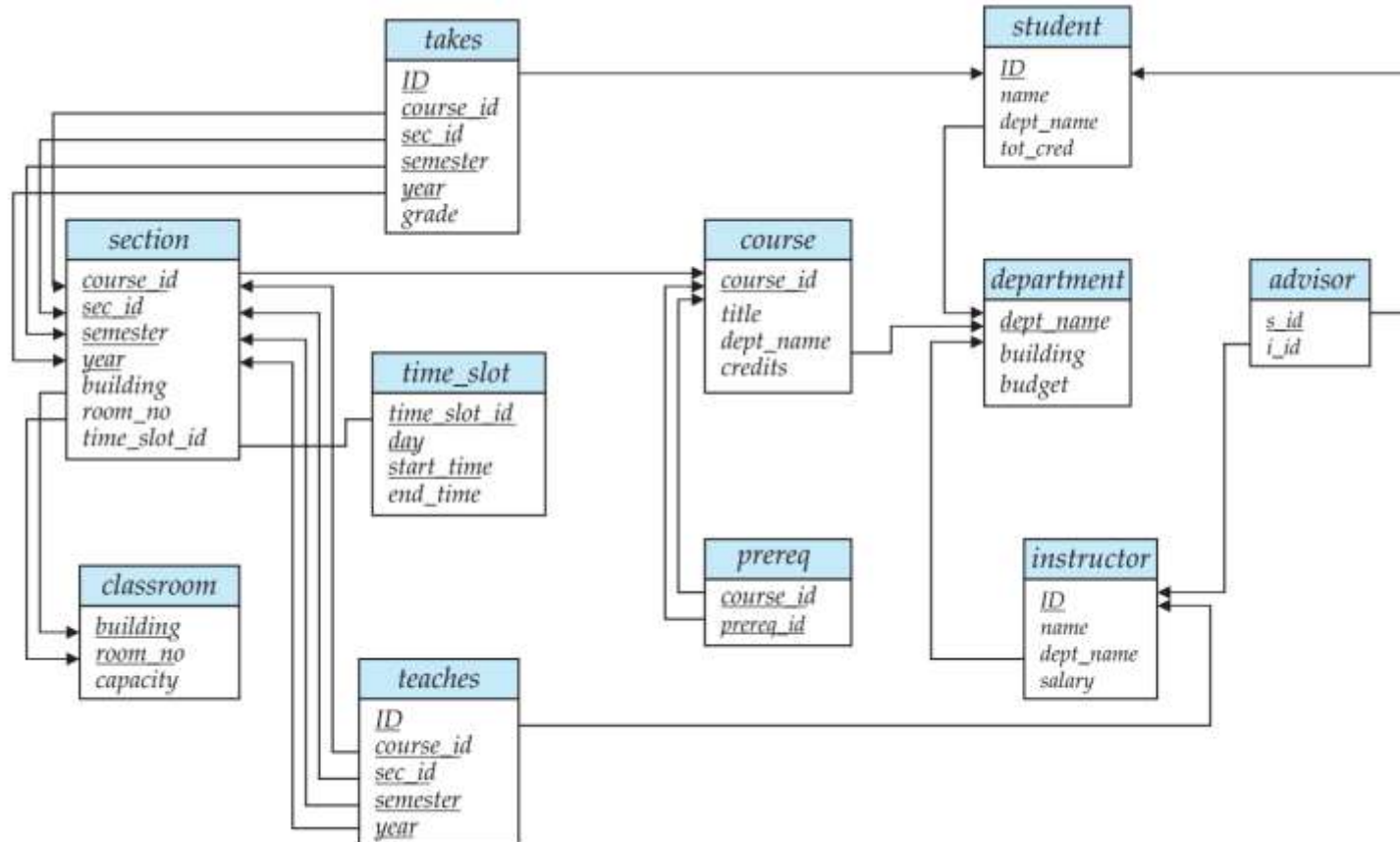
alter table r drop A

- Dropping of attributes not supported by many databases

Schema Used in Examples



Schema Used in Examples



Outline

- Overview of SQL
- SQL Data Definition
- **👉 Basic Structure of SQL Queries**
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

Basic Structure of SQL Queries

- SQL is based on **set** and **relational operations** with certain modifications and enhancements
- A typical SQL query has the form:

select A_1, A_2, \dots, A_n
from r_1, r_2, \dots, r_m
where P

- This query is equivalent to the relational algebra expression:

$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation

The Select Clause

- The **select clause** lists the attributes desired in the result of a query
 - corresponds to the **projection** operation of the RA (relational algebra)
- E.g., find the names of all departments in the instructor relation

```
select dept_name  
from instructor
```

- In the “pure” RA syntax, the query would be:

$\Pi_{dept_name}(instructor)$

- **NOTE:** SQL names are **case insensitive**, i.e. you can use capital or small letters

The select Clause (Cont.)

- SQL allows **duplicates** in relations. To eliminate duplicates, insert the keyword **distinct** after select
- Find the names of all departments in the instructor relation, and **remove duplicates**

```
select distinct dept_name  
from instructor
```

- The keyword **all** specifies that **duplicates should not be removed** by default

```
select all dept_name  
from instructor
```

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

```
select *  
from instructor
```

- The select clause can contain **arithmetic expressions** involving the operation, **+**, **-**, *****, and **/**, and operating on constants or attributes of tuples
- Example:

```
select ID, dept_name, salary * 1.1  
from instructor
```

The where Clause

- The **where clause** specifies conditions that the result must satisfy
 - correspond to the selection predicate of the RA (relational algebra)
 - **E.g.**, to find all loan numbers for loans made at the Perryridge branch with loan amount greater than \$1200.

```
select loan_number  
from loan  
where branch_name = 'Perryridge' and amount > 1200
```
 - Comparison results can be combined using the logical connectives **and**, **or**, and **not**
 - Comparison can be applied to results of arithmetic expressions

The where Clause (Cont.)

- SQL includes a **between** comparison operator
 - **E.g.** find the loan numbers of those loans with loan amount between \$90,000 and \$100,000

```
select loan_number
```

```
from loan
```

```
where amount between 90000 and 100000
```

The from Clause

- The **from clause** lists the relations involved in the query
 - corresponds to the **Cartesian product** operation of the RA
- E.g., find the Cartesian product $borrower \times loan$

```
select *  
from borrower, loan
```
- E.g., find the name, loan number and loan amount of all customers having a loan at the Perryridge branch

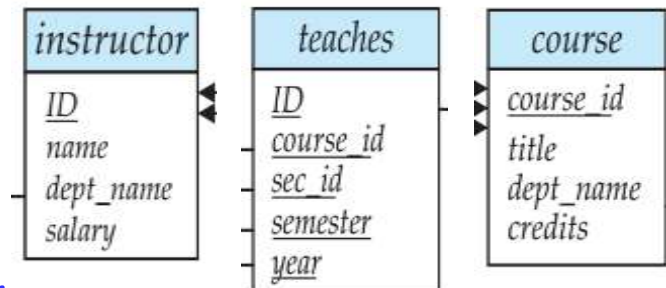
```
select customer_name, borrower.loan_number, amount  
from borrower, loan  
where borrower.loan_number = loan.loan_number and  
branch_name='Perryridge'
```

The Natural Join

```
select  $A_1, A_2, \dots, A_n$   
from  $r_1$  natural join  $r_2$  natural join ... natural join  $r_m$   
where  $P$ ;
```

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

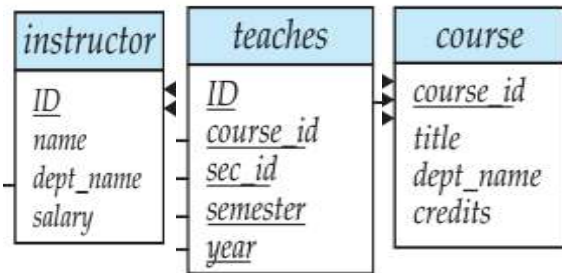
```
select name, title  
from instructor natural join teaches, course  
where teaches.course_id = course.course_id;
```



Notice that we **do not repeat** those attributes that appear in the schemas of both relations; rather they appear only once. **Notice** also the **order** in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

join ... using(...)

- natural join of instructor and teaches
 - (ID, name, dept_name, salary, course_id, sec_id, semester, year)
- course
 - (course_id, title, dept_name, credits)



结果不等

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

```
select name, title
from instructor natural join teaches natural join course;
```

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

结果相等

Outline

- Overview of SQL
- SQL Data Definition
- Basic Structure of SQL Queries
- **Additional Basic Operations**
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

The Rename Operation

- The SQL allows renaming relations and attributes using the **as clause**:
old_name as new_name
- Find the name, loan_number and loan_amount of all customers; rename the column name loan_number as loan_id:

```
select customer_name, borrower.loan_number as loan_id, amount  
from borrower, loan  
where borrower.loan_number = loan.loan_number
```

Tuple Variables

- **Tuple variables** are defined in the from clause via the use of the as clause
- Find the customer names and their loan numbers for all customers having a loan at some branch

```
select customer_name, T.loan_number, S.amount  
from borrower as T, loan as S  
where T.loan_number = S.loan_number
```

- Find the names of all branches that have greater assets than **some** branch located in Brooklyn.

```
select distinct T.branch_name  
from branch as T, branch as S  
where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

String Operations

- SQL includes a string-matching operator for comparisons on character strings.
 - **percent (%)**: The % character matches any substring
 - **underscore (_)**: The _ character matches any character
- **like/not like**: Find the names of all customers whose street includes (or not) the substring "Main"

```
select customer_name
from customer
where customer_street like '%Main%'
```
- Match the name "Main%" 要匹配的字符中有百分号的情况, 需要转义

```
like 'Main\%'; escape '\'
```
- "*" denote "all attributes" : `select instructor.*`
- SQL supports a variety of string operations such as
 - Concatenation (串联) (using "||")
 - converting from upper to lower case (and vice versa)
 - finding string length, extracting substrings, etc.

Order the Display of Tuples

- List in **alphabetic order** the names of all customers having a loan in Perryridge branch

```
select distinct customer_name  
from borrower, loan  
where borrower.loan_number = loan.loan_number and  
branch_name = 'Perryridge'  
order by customer_name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; **ascending order is the default**.

```
select *  
from loan  
order by amount desc, loan-number asc
```

Where Clause Predicates

- SQL includes a **between***not between* comparison operator
 - Example: find the names of all instructors with salary between \$90,000 and \$100,000

```
select name  
from instructor  
where salary between 90000 and 100000
```

- Tuple comparison

```
select name, course_id  
from instructor, teaches  
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

等价

```
select name, course_id  
from instructor, teaches  
where instructor.ID=teaches.ID and dept_name='Biology'
```

Outline

- Overview of the SQL
- SQL Data Definition
- Basic Structure of SQL Queries
- Additional Basic Operations
- ☞ **Set Operations**
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations \cup , \cap , $-$
- Each of the above operations **automatically eliminates duplicates**
- To retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**
 - Suppose a tuple occurs m times in r and n times in s , then, it occurs:
 - $m + n$ times in r **union all** s
 - $\min(m, n)$ times in r **intersect all** s
 - $\max(0, m - n)$ times in r **except all** s

Set Operations

- Find all customers who have a loan, an account, or both:
(select customer_name from depositor)
union [all]
(select customer_name from borrower)
- Find all customers who have both a loan and an account.
(select customer_name from depositor)
intersect [all]
(select customer_name from borrower)
- Find all customers who have an account but no loan.
(select customer_name from depositor)
except [all]
(select customer_name from borrower)

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result
- Multiset versions of some of the relational algebra operators, given multiset relations r_1 and r_2 :
 - $\sigma_\theta(r_1)$: If there are c_1 copies of tuple t_1 in r_1 , and t_1 satisfies selections σ_θ , then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$
 - $\Pi_A(r_1)$: For each copy of tuple t_1 in r_1 , there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1
 - $r_1 \times r_2$: If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2 , there are $c_1 \times c_2$ copies of the tuple $t_1 t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

- E.g., suppose multiset relations $r_1(A, B)$ and $r_2(C)$ are as follows:

$$r_1 = \{(1, a), (2, a)\} \quad r_2 = \{(2), (3), (3)\}$$


- Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, while $\Pi_B(r_1) \times r_2$ would be $\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$
- SQL duplicate semantics:

```
select  $A_1, A_2, \dots, A_n$   
from  $r_1, r_2, \dots, r_m$   
where  $P$ 
```

is equivalent to the multiset version of the expression:

$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

Outline

- Overview of SQL
- SQL Data Definition
- Basic Structure of SQL Queries
- Additional Basic Operations
- Set Operations
-  **Null Values**
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

Null Values

- It is possible for tuples to have a **null** value, signifies an **unknown** value or that a value **does not exist**
- The predicate **is null** can be used to check for null values

```
select loan_number  
from loan  
where amount is null
```

- The result of any arithmetic expression involving null is null
 - E.g. **5 + null** returns **null**
- *Aggregate functions simply ignore null values*


Null Values and Three Valued Logic

- Any comparison with **null** returns **unknown**
 - E.g. $5 < \text{null}$ or $\text{null} <> \text{null}$ or $\text{null} = \text{null}$
- Three-valued logic (三值逻辑) using the truth value **unknown**:
 - **OR**: $(\text{unknown or true}) = \text{true}$, $(\text{unknown or false}) = \text{unknown}$
 $(\text{unknown or unknown}) = \text{unknown}$
 - **AND**: $(\text{true and unknown}) = \text{unknown}$, $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$
 - **NOT**: $(\text{not unknown}) = \text{unknown}$
 - "P is unknown" evaluates to true if predicate P evaluates to unknown
- Result of where clause predicate is treated as **false** if it evaluates to **unknown**

Null Values and Aggregates

- Calculate the sum of all loan amounts
select sum (amount)
from loan
 - Above statement **ignores null** amounts
 - Result is null if there is no non-null amount
- All aggregate operations **except count(*)** ignore tuples with null values on the aggregated attributes

Outline

- Overview of the SQL
- SQL Data Definition
- Basic Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
-  **Aggregate Functions**
- Nested Subqueries
- Modification of the Database

Aggregate Functions

- These functions operate on a set of values of a column of a relation, and return a value
 - **avg**: average value
 - **min**: minimum value
 - **max**: maximum value
 - **sum**: sum of values
 - **count**: number of values

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch
select avg (balance)
from account
where branch_name = 'Perryridge'
- Find the number of tuples in the customer relation
select count ()*
from customer
- Find the number of depositors in the bank
select count (distinct customer_name)
from depositor

Aggregate Functions - Group By

- Find the number of depositors for each branch

```
select branch_name, count (distinct customer_name)  
from depositor, account  
where depositor.account_number = account.account_number  
group by branch_name
```

- **Note:** Attributes in select clause outside of aggregate functions must appear in group by list

```
/*erroneous query*/  
select dept_name, ID, avg(salary)  
from instructor  
group by dept_name
```

Aggregate Functions - Having Clause

- At times, it is useful to state a condition that applies to groups rather than to tuples.
- **E.g.**, find the names of all branches where the average account balance is more than \$1,200.

```
select branch_name, avg (balance)  
from account  
group by branch_name  
having avg (balance) > 1200
```

- **Note**
 - predicates in the **having clause** are applied **after** the information of **groups** whereas
 - predicates in the **where clause** are applied **before** forming **groups**

Aggregate Functions - Having Clause

- E.g., find the average balance for each customer who lives in Harrison and has at least three accounts

```
select depositor.customer_name, avg (balance)  
from depositor, account, customer  
where depositor.account_number=account.account_number  
and depositer.customer_name=customer.customer_name  
and customer_city='Harrison'  
group by depositor.customer_name  
having count(distinct depositor.account_number) >=3
```

Outline

- Overview of SQL
- SQL Data Definition
- Basic Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- **👉 Nested Subqueries**
- Modification of the Database

Nested Subqueries (嵌套子查询)

- SQL provides a mechanism for the nesting of subqueries
- A **subquery** is a **select-from-where** expression that is **nested within** another query in the from clause
- A common use of **subqueries** is to perform
 - tests for **set membership**
 - make **set comparisons**
 - determine **set cardinality (基数)**

Set Membership

- Find all customers who have **both** an account and a loan at the bank

```
select distinct customer_name  
from borrower  
where customer_name in (select customer_name  
from depositor)
```

- Find all customers who have a loan but do not have an account at the bank

```
select distinct customer_name  
from borrower  
where customer_name not in (select customer_name  
from depositor)
```

```
select distinct name  
from instructor  
where name not in ('Mozart', 'Einstein');
```

Set Membership (Cont.)

- Find all customers who have both an account and a loan at the Perryridge branch

```
select distinct customer_name  
from borrower, loan  
where borrower.loan_number=loan.loan_number and  
branch_name="Perryridge" and  
(branch_name, customer_name) in  
(select branch_name, customer_name  
from depositor, account  
where depositor.account_number =account.account_number)
```


Set Comparison

- Find all branches that have greater assets than some branch located in Brooklyn

```
select distinct T.branch_name  
from branch as T, branch as S  
where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

- Same query using **>some** clause

```
select branch_name  
from branch  
where assets > some  
    (select assets  
      from branch  
      where branch_city = 'Brooklyn')
```

Definition of Some Clause

- $F \langle \text{comp} \rangle \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F \langle \text{comp} \rangle t)$, where $\langle \text{comp} \rangle$ can be: $\langle, \leq, >, \geq, =, \neq$

$(5 \langle \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array} \rangle) = \text{true}$ (read: 5 \langle some tuple in the relation)

$(5 \langle \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array} \rangle) = \text{false}$

$(5 = \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$ (since $0 \neq 5$)

(= some) \equiv in
However,
(\neq some) \neq not in

Definition of all Clause

- $F \langle \text{comp} \rangle \text{all } r \Leftrightarrow \forall t \in r (F \langle \text{comp} \rangle t)$

$$(5 \langle \text{all} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array} \rangle) = \text{false}$$

$$(5 \langle \text{all} \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array} \rangle) = \text{true}$$

$$(5 \langle \text{all} \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array} \rangle) = \text{false}$$

$$(5 \langle \neq \text{all} \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array} \rangle) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

$(\neq \text{all}) \equiv \text{not in}$
However, $(= \text{all}) \neq \text{in}$

Example

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name  
from branch  
where assets > all  
    (select assets  
     from branch  
     where branch_city = 'Brooklyn')
```

平均工资最高的系

```
select dept_name  
from instructor  
group by dept_name  
having avg (salary) >= all (select avg (salary)  
                             from instructor  
                             group by dept_name);
```

Test for Empty Relations

- The **exists** construct returns the value **TRUE** if the argument subquery is nonempty
 - **exists** $r \Leftrightarrow r \neq \emptyset$
 - **not exists** $r \Leftrightarrow r = \emptyset$
- E.g., find all customers who have both an account & a loan at the bank

```
select customer_name  
from borrower  
where exists (  
    select *  
    from depositor  
    where depositor.customer_name = borrower.customer_name)
```

Test for Empty Relations

□ Find all customers who have both an account and a loan at the bank

- ✓

```
select customer_name
from borrower
where exists (
  select *
  from depositor
  where depositor.customer_name = borrower.customer_name)
```

- ✓

```
select distinct customer_name
from borrower
where customer_name in (select customer_name
                        from depositor)
```

- ✓

```
select distinct customer_name
from borrower, loan
where borrower.loan_number = loan.loan_number and (branch_name, customer_name) in
  (select branch_name, customer_name
   from depositor, account
   where depositor.account_number = account.account_number)
```

Test for Empty Relations (Cont.)

- Find all customers who have accounts at all branches located in Brooklyn

```
select distinct S.customer_name
from depositor as S
where not exists (
  (select branch_name      /* all branches in Brooklyn  X */
   from branch
   where branch_city = 'Brooklyn')
  except
  (select R.branch_name    /* finds all the branches at which customer
    S.customer_name has an account  Y */
   from depositor as T, account as R
   where T.account_number = R.account_number and
        S.customer_name = T.customer_name))
```

- Note: **not exists** $(X - Y) \Leftrightarrow X - Y = \emptyset \Leftrightarrow X \subseteq Y$

Test for Empty Relations (Cont.)

- Write "relation A contains relation B" as "not exists (B except A)."
- E.g., find all students who have taken all courses offered by the Biology department

```
select distinct S.ID, S.name
from student as S
where not exists ((select course_id
                  from course
                  where dept_name = 'Biology')
except
(select T.course_id
 from takes as T
 where S.ID=T.ID))
```


Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result
- **E.g.**, find all customers who have at most one account at the Perryridge branch

```
select T.customer_name
from depositor as T
where unique (
    select R.customer_name
    from account, depositor as R
    where T.customer_name = R.customer_name and
        R.account_number = account.account_number and
        account.branch_name = 'Perryridge')
```

Examples

找出所有在2009年最多开设一次的课程

等价

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id and
                    R.year = 2009);
```

```
select T.course_id
from course as T
where 1 <= (select count(R.course_id)
            >= from section as R
            where T.course_id= R.course_id and
                  R.year = 2009);
```

Examples

- Find all customers who have at least two accounts at the Perryridge branch.

```
select distinct T.customer_name  
from depositor T  
where not unique(  
    select R.customer_name  
    from account, depositor as R  
    where T.customer_name = R.customer_name and  
        R.account_number = account.account_number and  
        account.branch_name = 'Perryridge')
```

```
select T.course_id  
from course as T  
where not unique (select R.course_id  
    from section as R  
    where T.course_id= R.course_id and  
        R.year = 2009);
```

找出所有在2009年
至少开设两次的课程

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 a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described by
*(select customer_name, borrower.loan_number, branch_name
from borrower, loan
where borrower.loan_number = loan.loan_number)*
- 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
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
- When a view is created, *the query expression is stored in the database*; the expression is substituted into queries using the view

Example

- A view consisting of branches and their customers
create view all_customer as
*(select branch_name, customer_name
from depositor, account
where depositor.account_number = account.account_number)
union
(select branch_name, customer_name
from borrower, loan
where borrower.loan_number = loan.loan_number)*
- Find all customers of the Perryridge branch
*select customer_name
from all_customer
where branch_name = 'Perryridge'*

Derived Relations

- **Derived Relations**

- **E.g.** Find the average account balance of those branches where the average account balance is greater than \$1200.

```
select branch_name, avg (balance)
from account
group by branch_name
having avg (balance) > 1200
```

```
select branch_name, avg_balance
from (select branch_name, avg (balance)
from account
group by branch_name)
as result (branch_name, avg_balance)
where avg_balance > 1200
```

- **Note:** we do not need to use the having clause, since we compute **the temporary (view) relation result in the from clause**, and the attributes of result can be used directly in the where clause

Derived Relations (Cont.)

- E.g. Find the maximum total balance across all branches

```
select max(tot_balance)  
from (select branch_name, sum (balance)  
      from account  
      group by branch_name)  
as branch_total (branch_name, tot_balance)
```


With Clause

- **With** clause allows **views** to be defined **locally** to a query, rather than globally. Analogous to procedures in a programming language
- **E.g.** Find all accounts with the maximum balance.

```
with max_balance(value) as  
    select max(balance)  
    from account
```

```
select account_number  
from account, max_balance  
where account.balance = max_balance.value
```

Complex Query using with Clause

- E.g. Find all branches where the total account deposit is greater than the average of the total account deposits at all branches

with branch_total (branch_name, value) as

select branch_name, sum (balance)

from account

group by branch_name

with branch_total_avg (value) as

select avg (value)

from branch_total

select branch_name

from branch_total, branch_total_avg

where branch_total.value >= branch_total_avg.value

Scalar Subquery

- Scalar subquery(标量子查询) is used where **a single value** is expected
- E.g. List all departments along with the number of instructors in each department

```
select dept_name,  
       (select count(*)  
        from instructor  
        where department.dept_name = instructor.dept_name)  
       as num_instructors  
from department;
```

- **Note:** Runtime error if subquery returns more than one tuple

Outline

- Overview of SQL
- SQL Data Definition
- Basic Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- **Modification of the Database**

Modification of the Database - Deletion

- E.g. Delete all accounts at every branch located in Needham city

```
delete from account  
where branch_name in (select branch_name  
from branch  
where branch_city = 'Needham')
```



```
delete from depositor  
where account_number in  
(select account_number  
from branch, account  
where branch_city = 'Needham'  
and branch.branch_name = account.branch_name)
```

Example

- E.g. Delete the records of all accounts with balances below the average at the bank

delete from account

*where balance < (select avg(balance)
from account)*



- **Note:** as we delete tuples from account, **the average balance changes**
- **Solution used in SQL:**
 - First, compute avg balance and find all tuples to delete
 - Next, delete all tuples found above (without recomputing avg or retesting the tuples)

Modification of the Database - Insertion

- Add a new tuple to account

```
insert into account  
  values ('A-9732', 'Perryridge', 1200)
```

or equivalently

```
insert into account (branch_name, balance, account_number)  
  values ('Perryridge', 1200, 'A-9732')
```

- Add a new tuple to account with balance setting to null

```
insert into account  
  values ('A-777', 'Perryridge', null)
```

Modification of the Database - Insertion

- Provide as a gift for all loan customers of the Perryridge branch, i.e., a \$200 saving account. Let the loan number serve as the account number for the new saving account

```
insert into account
```

```
    select loan_number, branch_name, 200  
    from loan  
    where branch_name = 'Perryridge'
```

```
insert into depositor
```

```
    select customer_name, loan_number  
    from loan, borrower  
    where loan.loan_number = borrower.loan_number  
        and branch_name = 'Perryridge'
```

- **Note:** The *select from where* statement is *fully evaluated before* any of its results are *inserted* into the relation. Otherwise, queries like *insert into table1 select * from table2* would cause problems

Modification of the Database - Updates

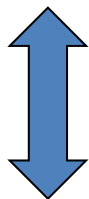
- Increase all accounts with balances over \$10,000 by 6%, and all other accounts receive an increase of 5%.

- Write two update statements:

- update account*

- set balance = balance * 1.06*

- where balance > 10000*



- update account*

- set balance = balance * 1.05*

- where balance ≤ 10000*

- The **order** is important
- Can be done better using the case statement (next slide)

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.

```
update account  
  set balance = case  
    when balance <= 10000 then balance *1.05  
    else balance * 1.06  
  end
```

```
case  
  when  $pred_1$  then  $result_1$   
  when  $pred_2$  then  $result_2$   
  ...  
  when  $pred_n$  then  $result_n$   
  else  $result_0$   
end
```

Review Terms

- **DDL**:Data definition language
- **DML**:Data manipulation language
- Database schema
- Database instance
- Relation schema
- Relation instance
- Primary key
- Foreign key
 - Referencing relation
 - Referenced relation
- Query language
- SQL query structure
 - **select clause**
 - **from clause**
 - **where clause**
- **Natural join** operation
- **as clause**
- **order by clause**
- Tuple variable
- Set operations
 - **Union**
 - **Intersect**
 - **except**
- **Null** values
 - Truth value "unknown"
- **Aggregate functions**
 - **avg, min, max, sum, count**
 - **group by**
 - **having**
- **Nested subqueries**
- Set comparisons
 - **{<, ≤, >, ≥}{some, all}**
 - **exists**
 - **unique**
- **with clause**
- Scalar subquery
- Database modification
 - **Deletion**
 - **Insertion**
 - **Updating**

Homework

- **Further Reading**
 - Chapter 3
- **Exercises**
 - 3.8, 3.9
 - 3.15, 3.16, 3.17, 3.21
- **Submission**
 - Deadline: 12:00pm, April 20, 2024

End of Lecture 3