21121350
Database System

Lecture 3: SQL

Lu Chen (陈璐)
College of Computer Science
Zhejiang University
Spring & Summer 2023
luchen@zju.edu.cn/18868818726

Introduction to SQL

- SQL has been widely used in the world.
  - Structured Query Language (SQL, 结构化查询语言), 1975, IBM. System R. called Structured English QUEry Language (SEQUEL).

- SQL-86: the first standard by ANSI --- American national standard institute, ISO standard.
- SQL-89: minor revision.
- SQL-92: currently supported by most commercial RDBMS.
- SQL-99: a major extension of SQL-92, partly supported now.

SQL Conformance Level

- SQL Conformance levels (标准符合度) can be classified into 4 categories:
  - Entry level SQL (入门级)
  - Transitional SQL (过渡级)
  - Intermediate SQL (中间级)
  - Full SQL (完全级)

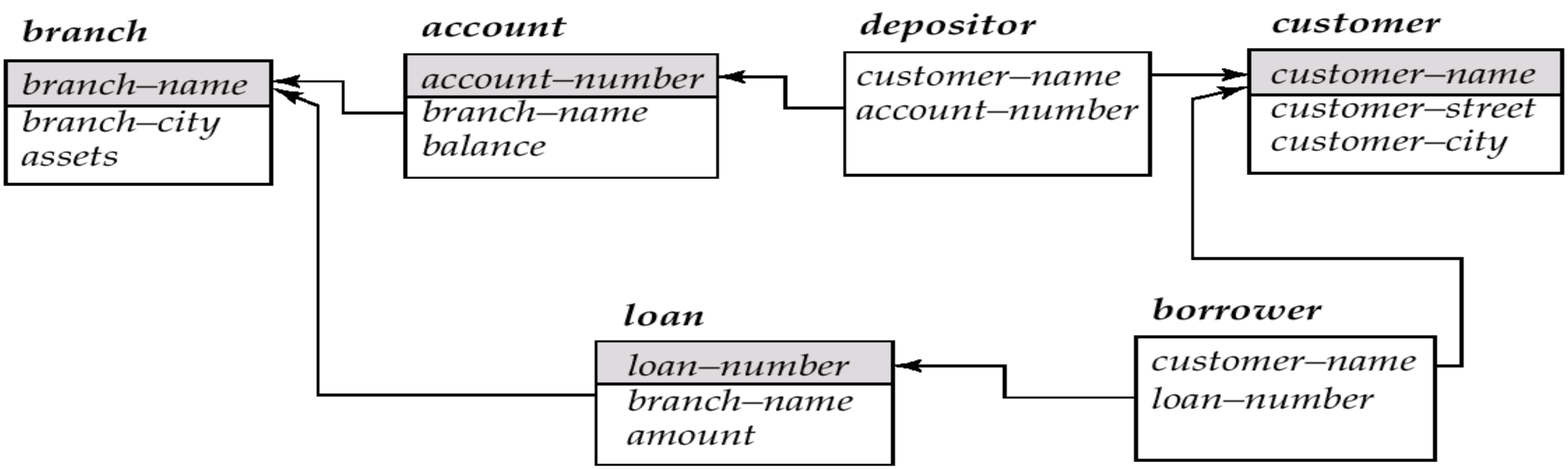
- SQL_92 includes 4 levels
- SQL_99 includes 4 levels
  - Now no DBMS products achieve Full SQL level.

SQL Operations

- SQL includes several parts:
  - Data-Definition Language (DDL)
    - --- Create table, alter table, drop table
    - --- Create index, drop index
    - --- Create view, drop view
    - --- Create trigger…, drop trigger
  - Data-Manipulation Language (DML)
    - --- Select … from
    - --- Insert, delete, update
  - Data-Control Language (DCL)
    - --- Grant, revoke

Banking Example

- *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*)



Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Data Definition Language

- Domain types in SQL
- Create table
- Drop and alter table
- Create index

Data Definition Language (Cont.)

- Example:

  CREATE TABLE *branch*
  (*branch_name*   char(15) not null,
  branch_city   varchar(30),
  assets   numeric(8, 2),
  primary key (*branch_name*))

Cf.:

Super-key

Candidate key

Primary key

Data Definition Language (Cont.)

- The main functions of DDL contain:
  - Define the schema for each relation
  - Define the domain of values associated with each attribute
  - Define the integrity constraints
  - Define the physical storage structure of each relation on disk
  - Define the indices to be maintained for each relations
  - Define the view on relations

Domain Types in SQL

- char(*n*): Fixed length character string, with user-specified length.
- 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, with user-specified precision of *p* digits, with *d* digits to the right of decimal point.
- real, double precision: Floating point and double-precision floating point numbers, with machine-dependent precision.

Domain Types in SQL (Cont.)

- float(n): Floating point number, with user-specified precision of at least *n* digits.
- Null values are allowed in all the domain types. Declaring an attribute to be not null prohibits null values for that attribute.

- date: Dates, containing a (4 digits) year, month and date.
  - E.g., date '2007-2-27'
- Time: Time of day, in hours, minutes and seconds.
  - E.g., time '11:18:16', time '11:18:16.28'
- timestamp: date plus time of day.
  - E.g., timestamp '2011-3-17 11:18:16.28'

Domain Types in SQL (Cont.)

- SQL中有许多函数用于处理各种类型的数据及其类型转换，但各数据库系统中函数的标准化程度不高。

- 不同函数：
  - SqlServer: char(65), substring(s, start, length), getdate(), datalength('abc'), …
  - Oracle: chr(65), substr(s, start, length), sysdate, length('abc'), to_char(*sysdate, 'yyyy/mm/dd'*) 得：2011/03/17, to_date('*11/03/17', 'yy/mm/dd*'), …

- 相同函数:
  - Abs(), exp(), round(), sin(), cos(), …

Create Table

- 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

Integrity Constraints in Create Table

- Not null
- Primary key (A_1, ..., A_n)
- Check (*P*), where *P* is a predicate

- Primary key declaration on an attribute automatically ensures not null in SQL_92 onwards, needs to be explicitly stated in SQL_89

Example: Declare *branch_name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

Method 1:

```
CREATE TABLE branch
    (branch_name char(20) not null,
     branch_city char(30),
     assets integer,
     primary key (branch_name),
     check (assets >= 0));
```

Method 2:

```
CREATE TABLE branch2
    (branch_name char(20)
     primary key,
     branch_city char(30),
     assets integer,
     check (assets >= 0));
```

Drop and Alter Table

- The drop table command deletes all information about the dropped relation from the database.
- DROP TABLE *r*
  - E.g., DROP TABLE *branch2*

Be careful to use the DROP command!!!

Drop and Alter Table (Cont.)

- The alter table command is used to add attributes to an existing relation.

  ALTER TABLE *r* ADD *A D*;

  ALTER TABLE *r* ADD (A_1 D_1, …, A_n D_n);

  where *A* is the name of the attribute to be added to relation *r*, and *D* is the domain of *A.*

- E.g., alter table *loan* add *loan_date* date;
  - All tuples in the relation are assigned *null* as the value for the new attribute.

Drop and Alter Table (Cont.)

- The alter table command can also be used to drop attributes of a relation

  ALTER TABLE *r* DROP *A*

  where *A* is the name of an attribute in relation *r*

- Note that dropping of attributes is not supported by many databases.

- The alter table command can also be used to modify the attributes of a relation
  - E.g., ALTER TABLE branch MODIFY (branch_name char(30), assets not null);

Create Index

- CREATE INDEX <*i-name*> ON <*table-name*> (<*attribute-list*>);
  - E.g.,

    create index *b_index* on *branch* (*branch_name*);

    create index *cust_strt_city_index* on *customer* (*customer_city*, *customer_street*);

- CREATE UNIQUE INDEX <*i-name*> ON <*table-name*> (<*attribute-list*>); ---- specify a candidate key
  - E.g.,

    create unique index *uni_acnt_index* on *account* (*account_number*);

- To drop an index
  - E.g.,

    DROP INDEX <*i-name*>

Outline

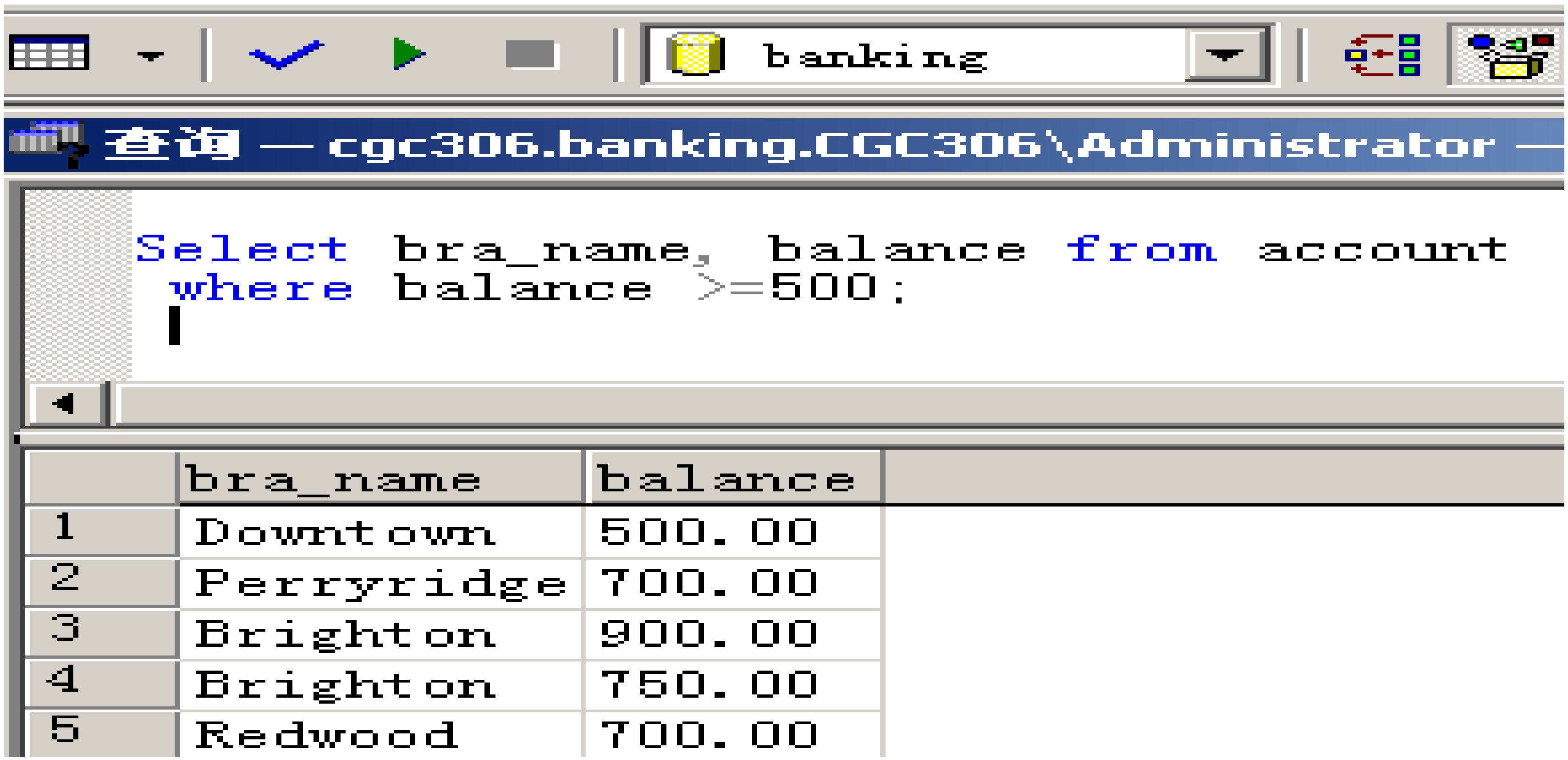
- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Basic Structure

- The select clause
- The where clause
- The from clause
- The rename operation
- String operations
- Ordering the display of tuples
- Duplicates

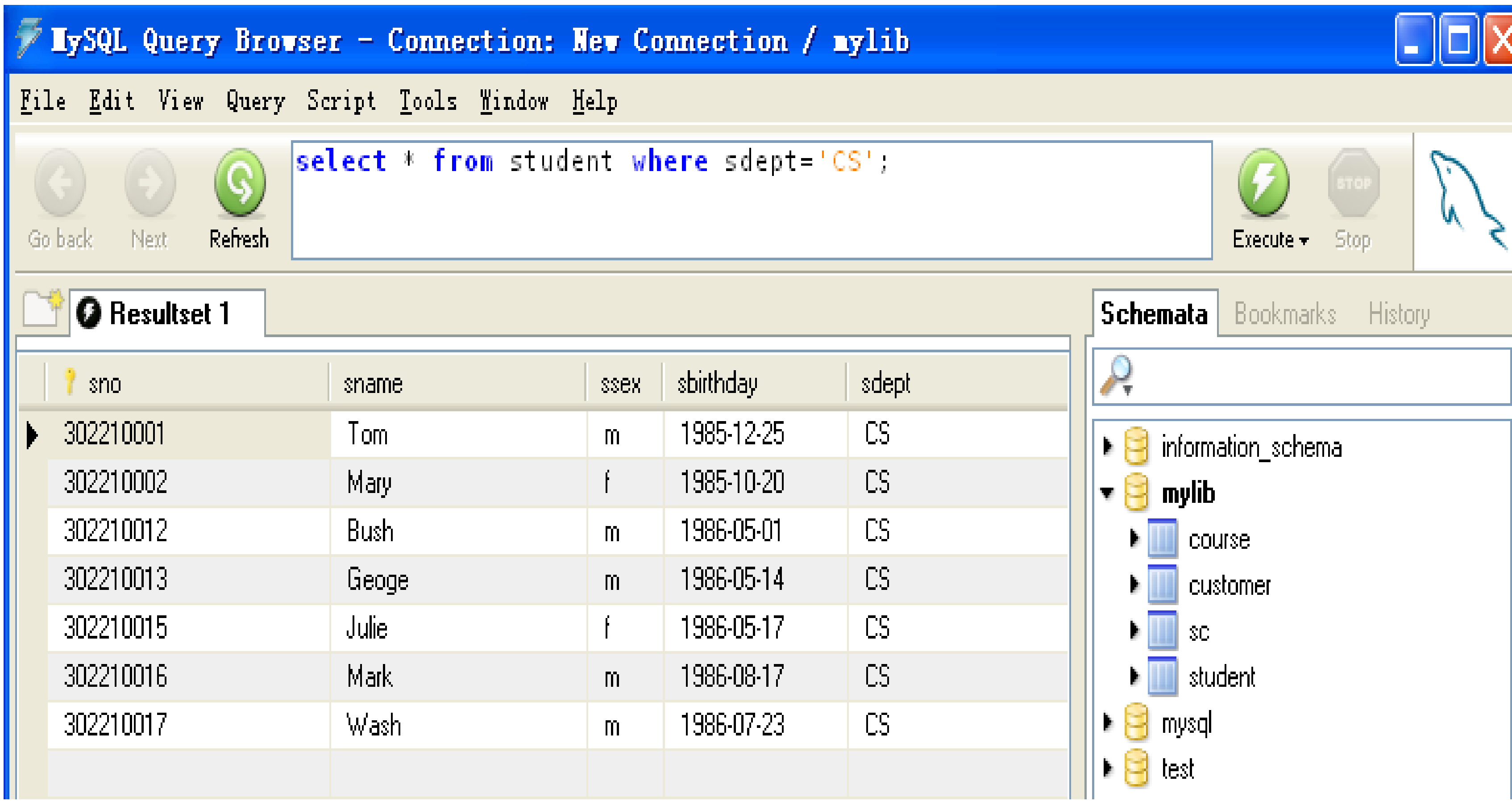
Basic Structure of Select

- Example 1: use Query Analyzer (查询分析器) of SQL Server



Basic Structure of Select (Cont.)

- Example 2: use MySQL Query Browser



Basic Structure of Select (Cont.)

- Example: Select branch_name, balance from account where balance >= 500
- A typical SQL query has the form:

  SELECT $A_1, A_2, ..., A_n$
  FROM $r_1, r_2, ..., r_m$
  WHERE P

  where A_i: attributes, r_i: relations, and P: predicate.

- This query is equivalent to the relational algebra expression below

$$\Pi_{A1,\, A2,\, ...,\, An}(\sigma_P(r_1 \times r_2 \times ... \times r_m))$$

- The result of an SQL query is a relation.

The Select Clause

- E.g., Find the names of all branches in the *loan* relation.

  SELECT *branch_name*
  FROM *loan*

  Not conditional selection

  - The relational algebra: $\Pi_{branch\_name}(loan)$

- Note: SQL does not permit the '-' character in names, and thus use *branch_name* instead of *branch-name* in the real implementation.
  - In text book it uses '-' since it looks nicer!

- Note: SQL names are case insensitive, i.e., you can use capital or small letters.
  - You may wish to use upper case wherever we use bold font.

The Select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword distinct after select.
  - E.g., Find the names of all branches in the *loan* relations, and remove duplicates

    SELECT distinct *branch_name*
    FROM *loan*

- The opposite keyword all allow duplicates, e.g.,

    SELECT all *branch_name*
    FROM *loan*

- By default, duplicates are allowed, i.e., all is the default.

The Select Clause (Cont.)

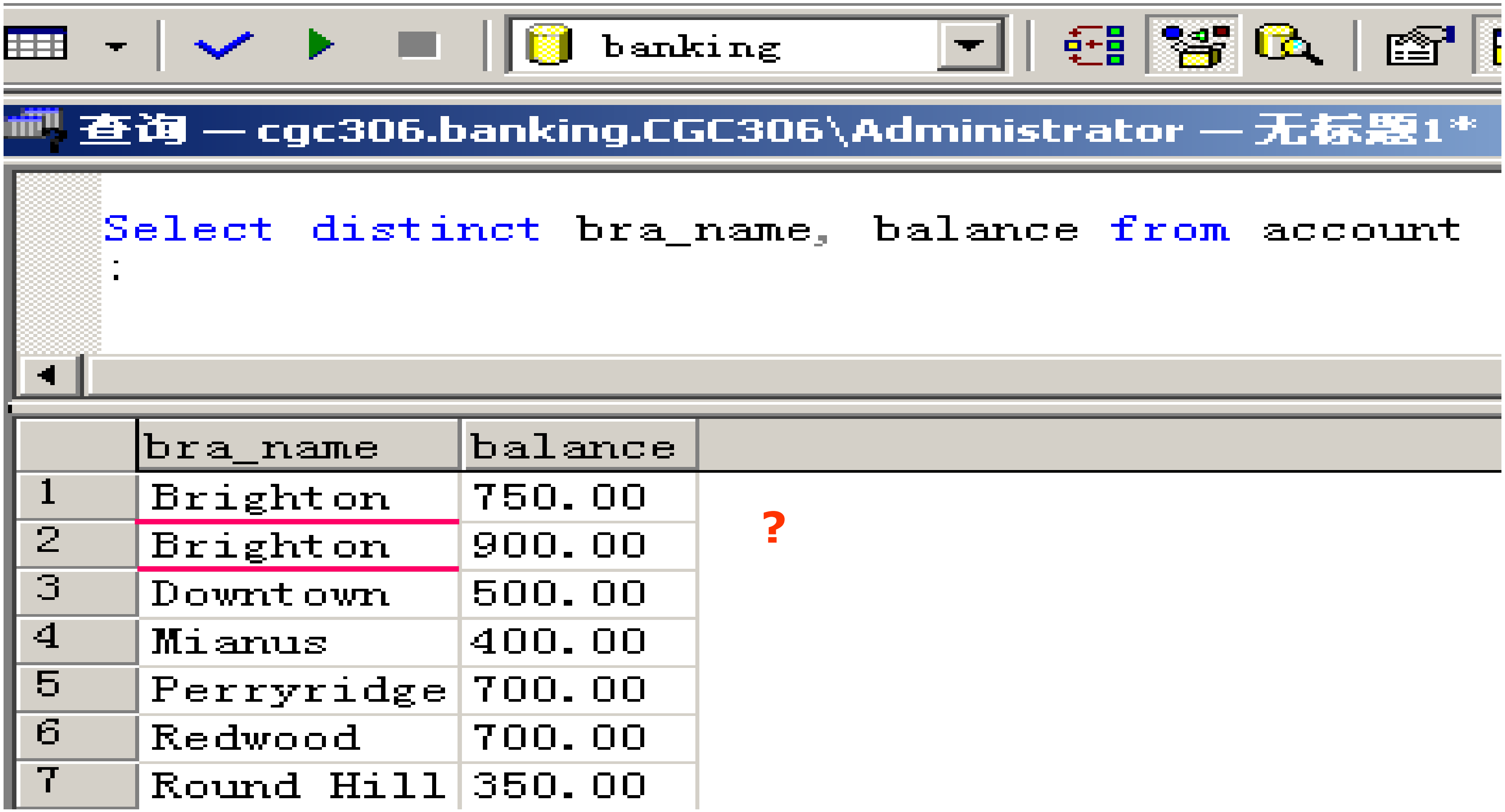
- Example (ORACLE)

```
SQL> select branch_name
from loan;

BRANCH_NAME
Round Hill
Downtown
Perryridge
Perryridge
Downtown
Redwood
Mianus
```

```
SQL> select distinct branch_name
from loan;

BRANCH_NAME
Downtown
Mianus
Perryridge
Redwood
Round Hill
```

The Select Clause (Cont.)

- Example (SQL Server)




```sql
Select distinct bra_name, balance from account
;
```

	bra_name	balance
1	Brighton	750.00
2	Brighton	900.00
3	Downtown	500.00
4	Mianus	400.00
5	Perryridge	700.00
6	Redwood	700.00
7	Round Hill	350.00

?

The Select Clause (Cont.)

- An asterisk * in the select clause denotes all attributes.

  SELECT * FROM *loan*

- However, the select clause can contain arithmetic expressions involving the operations +, –, *, and /, as well as operating on constants or attributes of tuples, e.g.,

  SELECT *loan_number*, *branch_name*, *amount* * 100
  FROM *loan*

The Where Clause

- The WHERE clause specifies conditions that the result must satisfy.
  - E.g., Find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

$$\Pi_{loan\_number}(\sigma_{branch\_name=\text{'Perryridge'} \wedge amount>1200}(loan))$$

SELECT *loan_number*
FROM *loan*
WHERE *branch_name* = ‘Perryridge’ and *amount* > 1200

loan(*loan-number*, *branch-name*, *amount*)

The Where Clause (Cont.)

- In WHERE clause, comparison results can be combined using the logical connectives including AND, OR, and NOT, as well as a BETWEEN comparison operator can be used for specifying a range.
  - E.g., Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (i.e., ≥ \$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 relational algebra, if more than one relation is specified in the FROM clause.

- Find the Cartesian product:

  borrower x *loan* ------

```
SELECT *
FROM borrower, loan
```

The From Clause (Cont.)

- Example: Find the customer name, loan number and loan amount of all customers having a loan at the Perryridge branch.

$$\Pi_{customer\_name,\ loan\_number,\ amount}(\sigma_{branch\_name='Perryridge'}(borrower \bowtie loan))$$

The prefix is necessary.

SELECT *customer_name*, *borrower.loan_number*, *amount*
FROM *borrower*, *loan*
WHERE *borrower.loan_number* = *loan.loan_number* and
branch_name = 'Perryridge'

loan(*loan-number*, *branch-name*, *amount*)
borrower(*customer-name*, *loan-number*)

The Rename Operation --- Column Rename

- The SQL allows renaming relations and attributes using the as clause:

  old_name as new_name

- Example: Find the name, loan number and loan amount of all customers, and 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*

- as is optional
  - SQL Server allows *new-name = column expression*, e.g.,

    SELECT *customer_name*, *loan_id = borrower.loan_number*, *amount*
    FROM *borrower*, *loan*

The Rename Operation --- Column Rename (Cont.)

- Example: Rename the column

```
select bra_name, count( cust_name) as 帐户数 from account A, depositor D
where A.acc_number = D.acc_number
group by bra_name
```

	bra_name	帐户数
1	Brighton	3
2	Downtown	1
3	Mianus	1
4	Perryridge	1
5	Round Hill	1

The Rename Operation --- Tuple Variables

- Tuple variables are defined in the FROM clause via the use of the as clause.
- Example: Find the customer names, their loan numbers and amount for all customers of the bank.

SELECT *customer_name*, *T.loan_number*, *S.amount*
FROM *borrower* as *T*, *loan* as *S*
WHERE *T.loan_number* = *S.loan_number*

for simplification

loan(*loan-number*, *branch-name*, *amount*)
borrower(*customer-name*, *loan-number*)

The Rename Operation --- Tuple Variables (Cont.)

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

SELECT distinct *T.branch_name*
FROM *branch* as *T*, *branch* as *S*
WHERE *T.assets* > *S.assets* and *S.branch_city* = ‘Brooklyn’

for discrimination

branch(*branch-name*, *branch-city*, *assets*)

String Operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using the following two special characters:
  - % --- matches any substring (likes * in the file system).
  - _ --- matches any character (like ? in the file system).

- Note: Fuzzy matching can be achieved (place the WHERE clause, and must be used in conjunction with the LIKE operation).

String Operations (Cont.)

- Example: Find the names of all customers whose name includes the substring “泽” .

  SELECT *customer_name*
  FROM *customer*
  WHERE *customer_name* LIKE ‘%泽%’

  fuzzy match

- Match the name “Main%”

  LIKE ‘Main\%’ escape ‘\’

String Operations (Cont.)

- SQL supports a variety of string operations such as
  - Concatenation (using "||"), e.g.,

    SELECT '客户名=' || customer_name
    FROM *customer*
    WHERE …

    客户名 = Adams
    客户名 = Brooks
    客户名 = Curry

  - Converting from upper to lower case (and vice versa) using Functions lower( ) and upper ( )
  - Finding string length
  - Extracting substrings
  - … …

Ordering the Display of Tuples

- Example: List in alphabetic order the names of all customers having a loan in Perryridge branch.

SELECT distinct *customer_name*
FROM *borrower A*, *loan B*
WHERE *A.loan_number* = *B.loan_number* and
branch_name = 'Perryridge'
order by *customer_name*

Ordering the Display of Tuples (Cont.)

- Example (ORACLE)

SQL> select branch_name from loan;

BRANCH_NAME

Round Hill

Downtown

Perryridge

Perryridge

Downtown

Redwood

Mianus

SQL> select bra_name from loan order by bra_name desc;

BRA_NAME

Round Hill

Redwood

Perryridge

Perryridge

Mianus

Downtown

Downtown

Ordering the Display of Tuples (Cont.)

- We may specify desc for descending order or asc for ascending order, and for each attribute, ascending order is the default.
  - E.g., order by *customer_name* desc

SELECT * FROM *customer*

ORDER BY *customer_city*, *customer_street* desc, *customer_name*

Duplicates

- In traditional relational theory, no duplicates, but we need duplicates in practice.

- Multiset versions of some relational algebra operators, including σ_θ, Π_A, x, which support the multiset.

- 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, 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.)

- Example: Suppose multiset relations $r_1(A, B)$ and $r_2(C)$ are as follows:

  $r_1 = \{(1, a)$
  $\qquad (2, a)\}$

  $r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1) = \{(a)$
  $\qquad (a)\}$

  while $\Pi_B(r_1) \times r_2$ would be:

  $\{\underline{(a, 2), (a, 2)}, \underline{(a, 3), (a, 3), (a, 3), (a, 3)}\}$

Duplicates (Cont.)

- Select statement in SQL also supports the multiset operators including σ_{θ}, Π_A, x.
- SQL duplicate semantics:

  SELECT $A_1, A_2, ..., A_n$
  FROM $r_1, r_2, ..., r_m$
  WHERE P

  is equivalent to the multiset version of the expression below

  $\Pi_{A1,\, A2,\, ...,\, An}(\sigma_P(r_1 \times r_2 \times ... \times r_m))$

✓ To forbid duplicate --- add distinct

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Set Operations

- In SQL, use the set operations including UNION, INTERSECT, and EXCEPT operate on relations as well as correspond to the relational algebra operations $\cup$, $\cap$, and $-$.

- Each of the operations including UNION, INTERSECT, and EXCEPT automatically eliminates duplicates.

- To retain all duplicates, we can use the corresponding multiset versions including 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)

if $m < n$

if $m > n$

Set Operations (Cont.)

- Example 1: Find all customers who have a loan or an account or both.

  (SELECT *customer_name* FROM *depositor*)
  UNION
  (SELECT *customer_name* FROM *borrower*)

- Example 2: Find all customers who have both a loan and an account.

  (SELECT *customer_name* FROM *depositor*)
  INTERSECT
  (SELECT *customer_name* FROM *borrower*)

depositor(*customer-name*, *account-number*)
borrower(*customer-name*, *loan-number*)

Set Operations (Cont.)

- Example 3: Find all customers who have an account but no loan.

  (SELECT *customer_name* FROM *depositor*)
  EXCEPT
  (SELECT *customer_name* FROM *borrower*)

- Notes:
  - Oracle uses UNION, UNION ALL, INTERSECT, and MINUS instead of EXCEPT; but no INTERSECT ALL and MINUS ALL.
  - SQL Server 2000 only supports UNION and UNION ALL.

depositor(*customer-name*, *account-number*) *borrower*(*customer-name*, *loan-number*)

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Aggregate Functions

- These functions (see below) operate on the multiset values of a relation's column, and return a value.

  avg(col): average value

  min(col): minimum value

  max(col): maximum value

  sum(col): sum of values

  count(col): number of values

Aggregate Functions (Cont.)

- Example 1: Find the average account balance at the Perryridge branch.

$$g_{\text{avg(balance)}}(\sigma_{branch\_name=\text{'Perryridge'}}(account))$$

SELECT avg(*balance*) *avg_bal*
FROM *account*
WHERE *branch_name* = 'Perryridge'

Branch_name	*avg_bal*
Perryridge	25000

SELECT *branch_name*, avg(*balance*) *avg_bal*
FROM *account*
WHERE *branch_name* = 'Perryridge'

X

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

account(*account-number*, *branch-name*, *balance*)

Aggregate Functions (Cont.)

- Example 2: Find the average account balance for each branch.

```
SELECT branch_name, avg(balance) avg_bal
FROM account
GROUP BY brach_name
```

Branch_name	*avg_bal*
Downtown	600
Mianus	725
Round Hill	350
Redwood	700

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Aggregate Functions (Cont.)

- Example 3: Find the number of depositors for each branch.

  SELECT *branch_name*, *count*(*customer_name*) *as tot_num*
  FROM *depositor, account*
  WHERE *depositor.account_number=account.account_number*
  GROUP BY *branch_name*

SELECT *branch_name*, *count*(distinct *customer_name*) *as tot_num*
FROM *depositor D*, *account A*
WHERE *D.account_number = A.account_number*
GROUP BY *branch_name*

account(*account-number*, *branch-name*, *balance*)
depositor(*customer-name*, *account-number*)

customer-name	account-number
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Branch-name	tot_num
Downtown	3
Mianus	2
Round Hill	1
Redwood	1

account-number	branch-name	balance
A-101	Downtown	500
A-215	Mianus	700
A-102	Downtown	400
A-305	Round Hill	350
A-201	Downtown	900
A-222	Redwood	700
A-217	Mianus	750

Branch-name	tot_num
Downtown	2
Mianus	2
Round Hill	1
Redwood	1

Aggregate Functions – Having Clause

- Example 4: Find the names of all branches located in city Brooklyn where the average account balance is more than $1,200.

  SELECT *A.branch_name*, avg(*balance*)
  FROM *account A*, *branch B*
  WHERE *A.branch_name* = *B.branch_name* and
  branch_city ='Brooklyn'
  GROUP BY *A.branch_name*
  HAVING avg(*balance*) > 1200

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

branch(*branch-name*, *branch-city*, *assets*)
account(*account-number*, *branch-name*, *balance*)

Summary

- The format of SELECT statement:

  SELECT <[DISTINCT] c_1, c_2, …>
  FROM <r_1, …>
  [WHERE <*condition*>]
  [GROUP BY <c_1, c_2, …> [HAVING <$cond_2$>]]
  [ORDER BY <c_1 [DESC] [, c_2 [DESC|ASC], …]>]

Summary (Cont.)

- The execution order of SELECT:

  From → where → group (aggregate) → having → select → distinct → order by

  - Note that predicates in the having clause are applied after the formation of groups, whereas predicates in the where clause are applied before forming groups.

- Aggregate functions cannot be used in where clause directly.

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Null Values

- Null is a special marker used in SQL, and was first introduced by E.F. Codd.

- The meaning is "missing information" or "inapplicable information", i.e., unknown value or that a value does not exist.

- The result of any arithmetic expression involving '*null'* is *null.*
  - E.g., 5 + null returns null.

- Any comparison with *null* returns "*unknown*"
  - E.g., 5 < *null or null <> null or null = null*

Null Values (Cont.)

- Three-valued logic using the truth value *unknown*: (true, unknown, false)
  - OR: (*unknown* or *true*) = *true*
    (*unknown* or *false*) = *unknown*
    (*unknown* or *unknown*) = *unknown*
  - AND: (*true* and *unknown*) = *unknown*
    (*false* and *unknown*) = *false*
    (*unknown* and *unknown*) = *unknown*
  - NOT: (not *unknown*) = *unknown*

- Result of where clause predicate is treated as *false* if it evaluates to *unknown*.

Null Values (Cont.)

- The predicate *is null*, *is not null* can be used to check for null values.
  - E.g., Find all loan number which appears in the *loan* relation with null values for *amount.*

```
SELECT loan_number
FROM loan
WHERE amount = null
```

X

```
SELECT loan_number
FROM loan
WHERE amount is null
```

- "*P* is *unknown*" evaluates to true if predicate *P* evaluates to *unknown*
- However, aggregate functions simply ignore nulls.

Null Values and Aggregates

- Total all loan amounts

  SELECT sum(*amount*)
  FROM *loan*

  - Above statement ignores null amounts
  - Result is null if there is no non_null amount, i.e., all values of *amount* in *loan* are null.

- All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes.

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Nested Queries

- Set comparison
- Test for empty relations
- Test for absence of duplicate tuples

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A subquery is a select_from_where expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons.

```
Select … from …  where   ...
                  (select … from … where …)
```

Example Query

- Example 1: 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)
```

```
Example 1: SELECT distinct B.customer_name
           FROM borrower B, depositor D
           WHERE B.customer_name = D.customer_name
```

Example Query (Cont.)

- Example 2: Find all customers who have loans at a bank 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*)

Example Query (Cont.)

- Example 3: Find all customers who have both an account and a loan at the Perryridge branch.

Query1: SELECT distinct customer_name
FROM *borrower B*, *loan L*
WHERE *B.loan_number* = *L.loan_number* and
branch_name = 'Perryridge' and
(*branch_name*, *customer_name*) in
(SELECT *branch_name*, *customer_name*
FROM *depositor D*, *account A*
WHERE *D.account_number* = *A.account_number*)

- Note: The above query can be written in a much simpler manner, and its formulation is simply to illustrate SQL features.

Example Query (Cont.)

- **Query 2:**

```
SELECT distinct customer_name
FROM borrower B, loan L
WHERE B.loan_number = L.loan_number and
        branch_name = 'Perryridge' and
        customer_name in
            (SELECT customer_name
             FROM depositor D, account A
             WHERE D.account_number = A.account_number
                   and branch_name = 'Perryridge')
```

Example Query (Cont.)

- **Query 3:**

  SELECT distinct *customer_name*
  FROM *borrower B*, *loan* as *t*
  WHERE *B.loan_number* = *t.loan_number* and
          branch_name = ‘Perryridge’ and
          customer_name in
              (SELECT customer_name
              FROM *depositor D*, *account A*
              WHERE *D.account_number* = *A.account_number*
                  and *branch_name* = *t.branch_name*)

Example Query (Cont.)

- Example 4: Find the account_number with the maximum balance for every branch.

SELECT *account_number*, *balance*
FROM *account*
WHERE *balance* >= max(*balance*)
GROUP BY *branch_name*

(x)

SELECT *account_number*, max(*balance*)
FROM *account*
GROUP BY *branch_name*

(x)

account(*account-number*, *branch-name*, *balance*)

Example Query (Cont.)

```
SELECT account_number AN, balance
FROM account  A
WHERE balance >= (SELECT max(balance)
                   FROM account B
                   WHERE A.branch_name = B.branch_name)
ORDER by balance
```

AN	*BALANCE*
A-305	350
A-102	400
A-101	500
A-215	700
A-222	700
A-201	900

```
SELECT account_number, balance
FROM account
GROUP by branch_name
HAVING balance >= max(balance)
ORDER by balance
```

X

Set Comparison

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

```
SELECT distinct branch_name
FROM branch
WHERE assets >  some
              (SELECT assets
               FROM branch
               WHERE branch_city = 'Brooklyn')
```

```
SELECT distinct T.branch_name
FROM branch as T, branch as S
WHERE T.assets > S.assets and S.branch_city = 'Brooklyn'
```

branch(*branch-name*, *branch-city*, *assets*)

Definition of Some Clause

- *C* <*comp*> **some** *r* $\Leftrightarrow \exists\, t \in r$, *C* <*comp*> *t* holds, where <*comp*> could be $<, \leq, >, =$, and $\neq$.

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

(5 < **some** $\begin{array}{|c|}\hline 0\\\hline 5\\\hline\end{array}$) = false

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

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

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

Definition of All Clause

- C $\langle comp \rangle$ **all** $r \Leftrightarrow \forall\, t \in r$, C $\langle comp \rangle$ t holds.

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

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

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

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

$(\neq$ **all**$) \equiv$ **not in**
However, $(=$ **all**$) \not\equiv$ **in**

Example Query

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

Query 1:

```
SELECT branch_name
FROM branch
WHERE assets > all
        (SELECT assets
         FROM branch
         WHERE branch_city = 'Brooklyn')
```

Query 2:

```
SELECT branch_name
FROM branch
WHERE assets > (SELECT max(assets)
                FROM branch
                WHERE branch_city =  'Brooklyn')
```

Test for Empty Relations

- The exists construct returns the value true if the argument subquery is non-empty.
- exists $r \Leftrightarrow r \neq \emptyset$
- not exists $r \Leftrightarrow r = \emptyset$

Example Query

$$\Pi_{\textit{customer-name, branch-name}}(\textit{depositor} \bowtie \textit{account}) \div \Pi_{\textit{branch-name}}\,(\sigma_{\textit{branch-city} = \text{'Brooklyn'}}\,(\textit{branch}))$$

Query 1: SELECT distinct *S.customer_name*
FROM *depositor* as *S*
WHERE not exists (
(SELECT *branch_name*
FROM *branch*
WHERE *branch_city* = 'Brooklyn')
EXCEPT
(SELECT distinct *R.branch_name*
FROM *depositor* as *T*, *account* as *R*
WHERE *T.account_number* = *R.account_number*
and *S.customer_name* = *T.customer_name*))

在Brooklyn的所有支行名字

S.顾客有存款的所有支行名字

- Note that $X - Y = \varnothing \Leftrightarrow X \subseteq Y$

branch(*branch-name*, *branch-city*, *assets*)
account(*account-number*, *branch-name*, *balance*)
depositor(*customer-name*, *account-number*)

Test for Absence of Duplicate Tuples

- The unique construct tests whether a subquery has any duplicate tuples in its result.
- Example: Find all customers who have at most one account at the Perryridge branch.

SELECT *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')

- Oracle 8 and SQL Server 7 do not support unique and not unique

Example Query

- 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')

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Views

- Provide a mechanism to hide certain data from the view of certain users.
- To create a view we use the command:

CREATE VIEW <*v_name*> AS

SELECT c_1, c_2, … From …

CREATE VIEW <*v_name*> (c_1, c_2, …) AS

SELECT e_1, e_2, … FROM …

- Benefits of using views
  - Security
  - Easy to use, support logical independent
- To drop view:

DROP VIEW <V_NAME>

Example Queries

- Create a view consisting of branches and their customer names.

```
CREAT 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))
```

Then we get view: all_customer (*branch_name*, *customer_name*)

Example Queries (Cont.)

- Example: Find all customers at the Perryridge branch.

```
SELECT customer_name
FROM all_customer
WHERE branch_name = 'Perryridge'
```

View and Logical Data Independence

- If relation **S(<u>a</u>, b, c)** is split into two sub relations **S1(<u>a</u>,b)** and **S2(<u>a</u>,c)**

  How to realize the logical data independence?

  1) create table S1 …; create table S2 …
  2) insert into S1 select a, b from S;
     insert into S2 select a, c from S;
  3) drop table S;
  4) create view S(a,b,c) as select a,b,c from S1, S2 where S1.a = S2.a;

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Derived Relations

- Example: Find the average account balance of those branches where the average account balance is greater than $500.

Query 1: SELECT *branch_name*, *avg_bal*
FROM (SELECT *branch_name*, avg(*balance*)
FROM *account*
GROUP BY *branch_name*)
as *result* (*branch_name*, *avg_bal*)
WHERE *avg_bal* > 500

Equivalent to the local view

Query 2: SELECT *branch_name*, *avg*(*balance*)
FROM *account*
GROUP BY *branch_name*
HAVING *avg*(*balance*) > 500

With Clause

- The WITH clause allows views to be defined locally for a query, rather than globally.
- Example: 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*

Define a local view

Use the local view

Complex Query Using With Clause

- Example: 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*, *a_bra_total*) as
SELECT *branch_name*, sum(*balance*)
FROM *account*
GROUP BY *branch_name*

每个Branch
的余额总和

WITH *total_avg*(*value*) as
SELECT avg(*a_bra_total*)
FROM *branch_total*

所有Branch
的余额平均

SELECT *branch_name*, *a_bra_total*
FROM *branch_total A*, *total_avg B*
WHERE *A*.*a_bra_total* >= *B*.*value*

SQL Server 2000 未实现此类WITH的用法，但可以利用建视图来实现同样功能。

Example Queries

- Example 1: Find the students who have enrolled more than 10 courses.

  SELECT *sno*
  FROM *enroll*
  GROUP BY *sno*
  HAVING count(*cno*) > 10

Enrolled(*sno*, *cno*, *grade*)

Example Queries (Cont.)

- Example 2: Find the student names who have enrolled more than 10 courses.

SELECT *sno*, *sname*, Count(*cno*) *course_num* ??
FROM *student*
WHERE *sno* in
(SELECT *sno*
FROM *enroll*
GROUP BY *sno*
HAVING count(*cno*) > 10)

Student(*sno*, *sname*, *ssex*, *sage*, *sdept*)
Enrolled(*sno*, *cno*, *grade*)
Course(*cno*, *cname*, *credits*)

Example Queries (Cont.)

- Example 2: Find the student names who have enrolled more than 10 courses.

  SELECT *TT.sno*, *sname*, *c_num*
  FROM (SELECT *sno*, count(*cno*) as *c_num*
  FROM *enroll*
  GROUP BY *sno*) as *TT*, *student S*
  WHERE *TT.sno* = *S.sno* and *c_num* > 10

Note: The derived table must have its own alias, e.g., *TT* above.
(不管是否被引用，导出表(或称嵌套表)必须给出别名)

Student(*sno*, *sname*, *ssex*, *sage*, *sdept*)
Enrolled(*sno*, *cno*, *grade*)
Course(*cno*, *cname*, *credits*)

Example Queries (Cont.)

- Example 3: Write SQL statement to satisfy each of the following requests.

  Given: *employee*(*id*, *name*, *age*, *gender*, *salary*, *boss*)

  (1) Find the employee who is younger than his/her boss and whose gender is different from his/her boss's.

  (2) Find the employee who has the maximum number of underlings.

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Deletion

- Example: Delete all account records at the Perryridge branch.

  DELETE FROM *account*
  WHERE *branch_name* = ‘Perryridge’

- Formal form:
  DELETE FROM <*table* | *view*>
  [WHERE <*condition*>]

Example Queries

- Example 1: Delete all accounts and relevant information at depositor for 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 B, account A
           WHERE branch_city = 'Needham'
                 and B.branch_name = A.branch_name)
```

branch(<u>*branch-name*</u>, *branch-city*, *assets*)
account(<u>*account-number*</u>, *branch-name*, *balance*)
depositor(<u>*customer-name*</u>, <u>*account-number*</u>)

Example Queries (Cont.)

- Example 1: Delete all accounts and relevant information at depositor for every branch located in Needham city.

DELETE FROM *account*, *depositor*, *branch*
WHERE *account.accont_number* = *depositor.account_number*
and *branch.branch_name* = *account.branch_name*
and *branch_city* = 'Needham'

(X)

branch(*branch-name*, *branch-city*, *assets*)
account(*account-number*, *branch-name*, *balance*)
depositor(*customer-name*, *account-number*)

Example Queries (Cont.)

- Example 2: Delete the record of all accounts with balances below the average at the bank.

  DELETE FROM *account*
  WHERE *balance* < (SELECT avg(*balance*)
  FROM *account*)

  Problem: as we delete tuples from *account*, the average balance changes.
  Solution actually used in SQL:
  1. First, compute avg balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing avg or retesting the tuples)
  3. 在同一SQL语句内，除非外层查询的元组变量引入内层查询，否则层查询只进行一次。

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')

Format:

INSERT INTO <*table*|*view*> [(c_1, c_2,…)]
VALUES (e_1, e_2, …)

INSERT INTO <*table*|*view*> [(c_1, c_2,…)]
SELECT e_1, e_2, …
FROM …

Example Queries

- Example 1: Add a new tuple to *account* with *balance* set to null.

  INSERT INTO *account*
  VALUES (‘A_777’, ‘Perryridge’, *null*)
  or equivalently
  INSERT INTO *account* (*account_number*, *branch_name*)
  VALUES (‘A_777’, ‘Perryridge’)

Example Queries (Cont.)

- Example 2: Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.
  - Add one record to account and depositor.

- Step 1: insert into *account*

  SELECT *loan_number*, *branch_name*, 200
  FROM *loan*
  WHERE *branch_name* = 'Perryridge'

- Step 2: insert into *depositor*

  SELECT *customer_name*, *A.loan_number*
  FROM *loan A*, *borrower B*
  WHERE *A.branch_name* = '*Perryridge*' and
  A.loan_number = *B.loan_number*

Insertion (Cont.)

- The "select from where" statement is fully evaluated before any of its results are inserted into the relation.

- Thus, the statement below is right:

  INSERT INTO *table*1
  SELECT *
  FROM *table*1

Updates

- Format of update statement:

  UPDATE <*table* | *view*>

  SET <c_1 = e_1 [, c_2 = e_2, …]> [WHERE <*condition*>]

Updates (Cont.)

- Example: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
- Need two update statements:

UPDATE *account*
SET *balance* = *balance* $*$ 1.06
WHERE *balance* > 10000
UPDATE *account*
SET *balance* = *balance* $*$ 1.05
WHERE *balance* $\leq$ 10000

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

Case Statement for Conditional Updates

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

UPDATE *account*

SET *balance* = case

when *balance* <= 10000

then *balance* * 1.05

else *balance* * 1.06

end

Update of a View

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

  CREATE VIEW *branch_loan* as
  SELECT *branch_name*, *loan_number*
  FROM *loan*

- Add a new tuple to *branch_loan.*

  INSERT INTO *branch_loan*
  VALUES (‘Perryridge’, ‘L-307’)

  This insertion will be translated into:

  INSERT INTO *loan*
  VALUES (‘L-307’, ‘Perryridge’, *null*)

建立在单个基本表上的视图，且视图的列对应表的列，称为“行列视图”。

Update of a View (Cont.)

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed, e.g.,

  CREATE VIEW *all_customer* as
  SELECT *branch_name*, *customer_name*
  FROM *account*, *depositor*
  WHERE …
  UNION
  SELECT *branch_name*, *customer_name*
  FROM *loan*, *borrower*
  WHERE …

  - E.g., INSERT INTO *all_customer*
    VALUES ('Second Bank', 'W. Bush')
  - DELETE FROM *all_customer*
    WHERE ...

X

Summary

- View 是虚表，对其进行的所有操作都将转化为对基表的操作。

- 查询操作时，VIEW与基表没有区别，但对VIEW的更新操作有严格限制，如只有行列视图，可更新数据。

- Most SQL implementations allow updates only on simple views defined on a single relation and without aggregates

Indexes

- **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*

Transactions

- A transaction is a sequence of queries and data update statements executed as a single logical unit.
  - Transactions are started implicitly and terminated by one of
    - COMMIT WORK: makes all updates of the transaction permanent in the database.
    - ROLLBACK WORK: undoes all updates performed by the transaction.

Transactions (Cont.)

- Motivating example:

  UPDATE *account* SET *balance* = *balance* – 100 WHERE *account_number* = ‘A-101’
  UPDATE *account* SET *balance* = *balance* + 100 WHERE *account_number* = ‘A-201’

  - If one steps succeeds and the other fails, database is in an inconsistent state. Therefore, either both steps should succeed or neither. ___ Atomicity

- 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.

  UPDATE *account* SET *balance* = *balance* – 100 WHERE *account_number* = ‘A-101’
  UPDATE *account* SET *balance* = *balance* + 100 WHERE *account_number* = ‘A-201’
  COMMIT WORK

Transactions (Cont.)

- The four properties of transaction are required: atomicity, isolation, consistency, durability (see chpt15).
- In some database systems, each SQL statement that executes successfully is automatically committed.
  - Each transaction would then consist of only a single statement.
  - Automatic commit can usually be turned off, allowing multi_statement transactions, but how to do so depends on the database system.
  - In MySQL:

    >SET AUTOCOMMIT=0;
- Another option in SQL 1999: enclose statements within

  begin atomic

  ……

  end

Transactions (Cont.)

- Transaction example :
- **SET AUTOCOMMIT=0**

UPDATE account **SET** balance=balance-100 **WHERE** ano='1001';

UPDATE account **SET**balance=balance+100 **WHERE** ano='1002';

COMMIT;

UPDATE account **SET** balance=balance -200 **WHERE** ano='1003';

UPDATE account **SET** balance=balance+200 **WHERE** ano='1004';

COMMIT;

UPDATE account **SET** balance=balance+balance*2.5%;

COMMIT;

Outline

- Data Definition Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Views
- Derived Relations
- Modification of the Database
- Joined Relations

Joined Relations

- Join operations take as input two relations and return as a result another relation.
- 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_2, ..., A_n)

loan_number	*branch_name*	*amount*
L_170	Downtown	3000
L_230	Redwood	4000
L_260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	**L_170**
Smith	**L_230**
Hayes	**L_155**

borrower

假设由于某种原
因造成帐目不符

Inner Join

loan $\bowtie$ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L_170	Downtown	3000	Jones
L_230	Redwood	4000	Smith

Left Out Join

loan ⟕ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L_170	Downtown	3000	Jones
L_230	Redwood	4000	Smith
L_260	Perryridge	1700	***null***

$R ⟕ S = (R \bowtie S) \cup ((R - \Pi_R(R \bowtie S)) \times \{(null, \ldots)\})$

Right Out Join

loan ⟖ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L_170	Downtown	3000	Jones
L_230	Redwood	4000	Smith
L_155	***null***	***null***	Hayes

Full Outer Join

loan ⟗ *borrower*

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

Combination of Join Type and Join Condition

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)

自然连接：*R* natural {inner join, left join, right join, full join} *S*

非自然连接：*R* {inner join, left join, right join, full join} *S* { on <连接条件判别式>; using (<同名的等值连接属性名>) }

★ Key word *Inner*, *outer* is optional

Natural join: 以同名属性相等作为连接条件
Inner join: 只输出匹配成功的元组
Outer join: 还要考虑不能匹配的元组

Joined Relations in SQL

- Select * from *loan* natural inner join *borrower*.

loan_number	*branch_name*	*amount*	*customer_name*
L_170	Downtown	3000	Jones
L_230	Redwood	4000	Smith

- Select * from *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 in SQL (Cont.)

- Select * from *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

★ 非自然连接，容许不同名属性的比较，且结果关系中不消去重名属性。

- Select * from *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 in SQL (Cont.)

- Select * from *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

- 使用using的连接类似于natural连接，但仅以using列出的公共属性为连接条件.

Joined Relations (Cont.)

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

connection condition

SELECT *customer_name*
FROM (*depositor* natural full outer join *borrower*)
WHERE *account_number* is *null* or *loan_number* is *null*

depositor(*customer-name*, *account-number*)
borrower(*customer_name*, *loan_number*)

主流商用数据库中外连接的表示

- SQL Server:
  - SELECT *loan.loan_number*, *branch_name*, *amount*, *customer_name*
    FROM *loan* left outer join *borrower* on
    loan.loan_number = *borrower.loan_number*
  - SELECT *loan.loan_number*, *branch_name*, *amount*, *customer_name*
    FROM *loan*, *borrower*
    WHERE *loan.loan_number* *= *borrower.loan_number*

*= denotes left join

- Note: *= denotes left join, and =* denotes right join.

主流商用数据库中外连接的表示 (Cont.)

- ORACLE:
  - SELECT *loan.loan_number*, *branch_name*, *amount*, *customer_name*
    FROM *loan*, *borrower*
    WHERE *loan.loan_number* = *borrower.loan_number* (+)

denotes left join

- Note:
  - Right join: *loan.loan_number* (+) = *borrower.loan_number* [相当于在（+）侧附加一特殊空行与另一表中的连接属性匹配].
  - Left join: *loan.loan_number* = *borrower.loan_number* (+).

Synthetic examples

- Example 1: Consider the following relational schema: *part*(*id*, *name*, *color*, *weight*, *sub_part*), transfer the following SQL query into the relational algebra expression:

  SELECT *part2.id*

  FROM *part as part1*, *part as part2*

  WHERE *part1.id = part2.sub_part* AND *part1.color* = 'red' AND *part2.color* = 'blue' AND *part1.weight – part2.weight* >100

$$\Pi_{p2.id}\ (\sigma_{p1.weight-p2.weight\ >100}\ (\sigma_{p1.id=p2.sub\_part}\ ((\sigma_{p1.color=\text{'red'}}(\rho_{p1}(part)))\ x\ (\sigma_{p2.color=\text{'blue'}}(\rho_{p2}(part))))))$$

Synthetic examples (Cont.)

- Example 2: Consider the student database below:

  student(<u>*student-no*</u>, *student-name*, *sex*, *age*, *dept-name*)

  course(<u>*course-no*</u>, *course-name*, *credit*)

  study(<u>*student-no*</u>, <u>*course-no*</u>, *score*)

  Please give the SQL statements for each of the following requirements:

(1) Find the names of students who have studied course ‘Database System’ and sort results by ascending score.

(2) Find the names of students who get the best score in course ‘Database System’.

(3) Find the names of courses that have maximum average score.

Synthetic examples (Cont.)

- (1) Find the names of students who have studied course ‘Database System’ and sort results by ascending score.

SELECT *student_name*
FROM *student S*, *study T*, *course C*
WHERE *S.student_no* = *T.student_no* and
T.course_no = *C.course_no* and
course_name = ‘Database System’
ORDER BY *score*

Synthetic examples (Cont.)

- (2) Find the names of students who get the best score in course 'Database System'.

```
SELECT student_name
FROM student S, study T, course C
WHERE S.student_no = T.student_no and
        T.course_no = C.course_no and
        course_name = 'Database System' and
        T.score >= (SELECT max(score)
                      FROM study T, course C
                      WHERE T.course_no = C.course_no and
                            course_name = 'Database System')
```

Synthetic examples (Cont.)

- (3) Find the names of courses that have maximum average score.

Method 1:

```
SELECT course_name
FROM course
WHERE course_no in
        (SELECT course_no
         FROM study
         GROUP BY course_no
         HAVING avg(score) >= all
                (SELECT avg(score)
                 FROM study
                 GROUP BY course_no))
```

Synthetic examples (Cont.)

- (3) Find the names of courses that have maximum average score.

Method 2:

```
WITH course_avg(course_no, score_avg) as
    SELECT course_no, avg(score)
    FROM study
    GROUP BY course_no
    SELECT course_name
    FROM course
    WHERE course_no in
        (SELECT course_no
         FROM course_avg
         WHERE score_avg =
            (SELECT max(score_avg)
             FROM course_avg))
```

Summary

- The properties of SQL are as follows:
  - Non-procedural language
    - Declarative programming
    - Imperative programming (Fortran, C or Pascal)
  - Set oriented operations, not recorder oriented
  - Integrate the DDL, DML, DCL into one
  - Support the three level schemas and data independent

- SQL is growing fast
  - Procedural, O-O, XML, …

Summary (Cont.)

- Create table, alter table, drop table
- Create view, drop view
- [Create index, drop index, assertion, trigger, …]
- SELECT <[DISTINCT] c_1, c_2, …> FROM <r_1, …>
  [WHERE <*condition1*>] [GROUP BY <c_1, c_2, …>
  [HAVING <*condition2* >]] [ORDER BY <c_1 [DESC] [, *c2* [DESC | ASC], …]
  - [<*condition*>]: =, >, …; > all, > some, …; in, not in, between; like; exists, not exists
  - Join, With
  - Union, union all, intersect, except
- 问题：查询速度、汉字…

Q & A

Your questions and suggestions are expected for me.

Thanks a lot!

Questions?

Exercises: 3.8, 3.9, 3.10, and 3.15 (see Pages 117-120)