COMP 3311 DATABASE MANAGEMENT SYSTEMS

LECTURE 7 STRUCTURED QUERY LANGUAGE (SQL)

STRUCTURED QUERY LANGUAGE (SQL): OUTLINE

SQL Basic Structure and Operations

Additional Basic Operations

Aggregate Functions

Nested Subqueries and Set Operations

Database Definition

Database Modification

Using SQL in Applications

SQL OVERVIEW

- SQL is the most common relational query language.

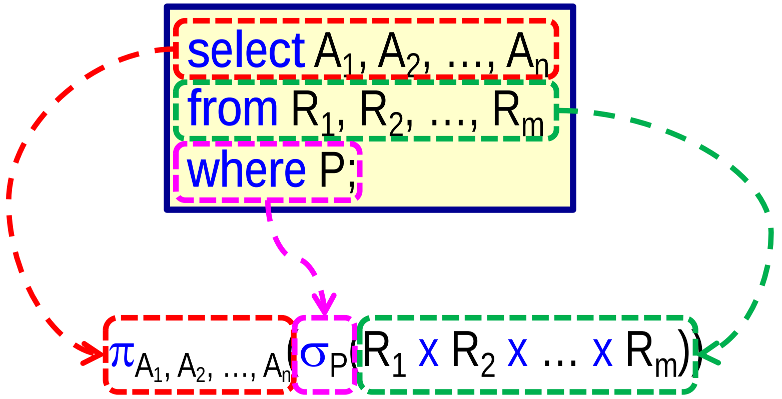
  ☞ **SQL is used in all commercial relational DBMSs.**

- Commercial relational DBMSs have different features of SQL, but the basic structure is the same.
  - Data Manipulation Language
  - Data Definition Language
  - Integrity Constraint Specification
  - View Definition
  - Embedded/Dynamic SQL
  - Transaction Management
  - Security Management

  ⋮

BASIC STRUCTURE OF SQL QUERIES

- SQL is based on set and relational algebra operations with certain modifications and enhancements.
- An SQL query has the basic form:



```
select A1, A2, ..., An
from R1, R2, ..., Rm
where P;
```

$\pi_{A_1, A_2, \ldots, A_n}(\sigma_P(R_1 \times R_2 \times \ldots \times R_m))$

A_i are attributes
R_i are relations
P is a predicate (condition)

The equivalent relational algebra expression.

- The result of an SQL query is a relation (but it may contain duplicates).

☞ **SQL queries can be nested (composed).**

EXAMPLE BANK RELATIONAL SCHEMA

Branch(<u>branchName</u>, branchCity, assets)

Client(<u>clientName</u>, clientStreet, clientCity)

Loan(<u>loanNo</u>, amount, *branchName*)

Account(<u>accountNo</u>, balance, *branchName*)

Borrower(<u>*clientName, loanNo*</u>)

Depositor(<u>*clientName, accountNo*</u>)

Attribute names in italics are foreign key attributes.

PROJECTION: SELECT CLAUSE

- The select clause corresponds to the relational algebra projection (π) operation.

Query: Find the names of all branches in the Loan relation.

```
select branchName
from Loan;
```

= $\pi_{\text{branchName}}(\text{Loan})$

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

```
select *
from Loan;
```

=

```
select loanNo, amount, branchName
from Loan;
```

☞ **Attributes specified in the select clause *must* be defined in the relations in the from clause.**

☞ **SQL *does not* remove duplicates in the result by default.**

PROJECTION: DUPLICATE REMOVAL

- The keyword distinct forces the removal of duplicates.

Query: Find the <u>unique</u> names of all branches in the Loan relation.

```
select distinct branchName
from Loan;
```

$\dashrightarrow$ ***force*** the DBMS to remove duplicates

- The keyword **all** specifies that duplicates <u>*should not*</u> be removed.

```
select all branchName
from Loan;
```

$\dashrightarrow$ ***force*** the DBMS <u>*not*</u> to remove duplicates (same as omitting all)

PROJECTION: ARITHMETIC OPERATIONS

- The select clause can contain arithmetic expressions involving the operators +, –, ÷ and × that can operate on constants or attributes of tuples.

Query: Multiply the amount of each loan by 100.

```
select loanNo, amount*100, branchName
from Loan;
```

This query returns a relation which is the same as the Loan relation, except that the attribute amount is multiplied by 100.

SELECTION: WHERE CLAUSE

- The where clause corresponds to the relational algebra selection predicate (σ) and specifies conditions that tuples in the relations in the from clause must satisfy.

Query: Find all loan numbers for loans made at the Tsimshatsui branch whose loan amount is greater than $1200.

```
select loanNo
from Loan
where branchName='Tsimshatsui'
        and amount>1200;
```

String values *must* be enclosed in single quotes. Numeric values do not require quotes.

$$\equiv \quad \pi_{loanNo}(\sigma_{branchName='Tsimshatsui' \wedge amount>1200}(Loan))$$

☞ **Attributes specified in a where clause *must* be defined in the relations in the from clause.**

SELECTION: WHERE CLAUSE (CONT'D)

- SQL provides the between operator for convenience.

Query: Find the loan number of loans whose amount is between $100,000 and $200,000 (i.e., ≥$100,000 and ≤$200,000).

```
select loanNo
from Loan
where amount between 100000 and 200000;
```

- Can also use not between (i.e., <$100,000 and >$200,000).

```
select loanNo
from Loan
where amount not between 100000 and 200000;
```

- SQL allows Boolean operators and, or and not to be used in a where clause as well as arithmetic expressions.

NATURAL JOIN: WHERE CLAUSE

- A natural join can be specified by adding the appropriate join condition in the where clause.

Query: Find the name and loan number of all clients.

```
select clientName, borrower.loanNo
from Borrower, Loan
where Borrower.loanNo=Loan.loanNo;
```

Attribute names *must* be qualified if ambiguous.

- SQL provides a shorthand way to specify a natural join.

```
select clientName, loanNo
from Borrower natural join Loan;
```

Attribute names *cannot* be qualified in a natural join. **Why?**

What must be true for these two queries to be equivalent?

CARTESIAN PRODUCT: FROM CLAUSE

- The from clause corresponds to the relational algebra Cartesian-product operation (×).

  Query: Find the Cartesian product of borrower and loan.

```
select *
from Borrower, Loan;
```

- This can also be specified as

```
select *
from Borrower cross join Loan;
```

☞ **A from clause with more than one relation is rarely used without a where clause.**

SET OPERATIONS: UNION, INTERSECT, EXCEPT

- The set operations union, intersect, and except operate on relations and correspond to the relational algebra operations ∪, ∩ and −.

Oracle Note

The keyword minus is used rather than except.

- Each of the set operations automatically removes duplicates.

- The operations union all, intersect all and except all keep all duplicates.

Oracle Note

Only union all is supported.

- 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: EXAMPLES

Query: Find all clients who have a loan, an account, or both.

```
(select clientName from Depositor)
union
(select clientName from Borrower);
```

Query: Find all clients who have both a loan and an account.

```
(select clientName from Depositor)
intersect
(select clientName from Borrower);
```

Query: Find all clients who have an account, but no loan.

```
(select clientName from Depositor)
minus
(select clientName from Borrower);
```

RENAME ATTRIBUTES: AS CLAUSE

- Attributes can be renamed using the as clause:

  old-name as *new-name*

Query: Find the name and loan number of all clients having a loan at the Central branch; replace the attribute name loanNo with the name loanId.

```
select distinct clientName, Borrower.loanNo as loanId
from Borrower, Loan
where Borrower.loanNo=Loan.loanNo
      and branchName='Central';
```

Oracle Note

The keyword as *is optional* in the select clause.

- The SQL standard also allows relations in the from clause to be renamed using the as clause.

Oracle Note

The keyword as *is not allowed* in the from clause.

RENAME RELATIONS

- Renaming relations is convenient for replacing long relation names used multiple times in a query with shorter ones.

Query: Find the client names and their loan numbers for all clients having a loan at *some* branch; replace the column name loanNo with the name loanId.

```
select distinct clientName, B.loanNo loanId
from Borrower B, Loan L
where B.loanNo=L.loanNo;
```

Oracle Note

Relations in the from clause are renamed using an identifier *without* the keyword as.

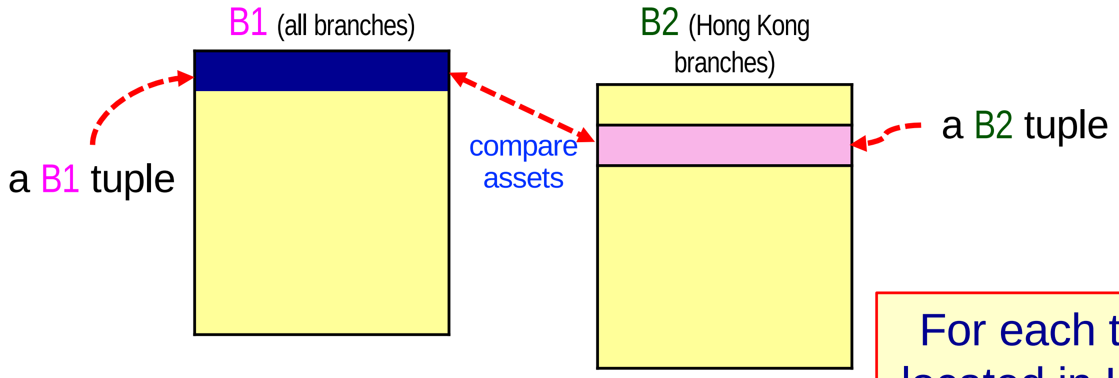
- An identifier for a relation (such as B and L above) is referred to as a *correlation name* in SQL.
  - Also known as *table alias*, *correlation variable* or *tuple variable*.

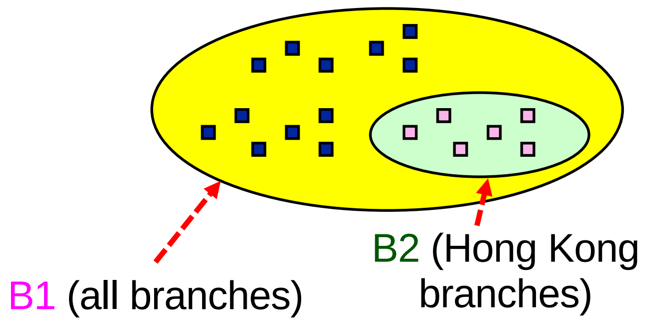
RENAME RELATIONS (cont'd)

- Renaming a relation is required when we want to compare tuples in the same relation (self-join).

Query: Find the names of all branches that have greater assets than some (i.e., at least one) branch located in Hong Kong.

```
select distinct B1.branchName
from Branch B1, Branch B2
where B1.assets>B2.assets and B2.branchCity='Hong Kong';
```





For each tuple in B1 is there *some* branch located in Hong Kong that has less assets? If yes, add the B1 tuple to the result.

STRING PATTERN MATCHING: LIKE OPERATOR

- The like operator is used for matching characters in strings.
- Character attributes can be compared to a pattern using:

  % (percent) matches any substring.

  _ (underscore) matches any single character.

Query: Find the name of all clients whose streets include the substring 'Main' (e.g., Mainroad, Mainly Avenue, Mainmount Street, ...).

```
select clientName
from Client
where clientStreet like '%Main%';
```

☞ **Pattern matching is *usually* case-sensitive.**

STRING PATTERN MATCHING: LIKE OPERATOR (CONT'D)

- To include the special pattern matching characters in a string, SQL allows the specification of an escape character.
  - Suppose we use backslash (\) as the escape character.
    - like '20\%%' escape '\' matches all strings beginning with "20%"
    - like 'pair_%' escape '\' matches all strings beginning with "pair_"

- To include a single quote in a string, use two single quotes.
  - like 'Toms''s%' matches all strings beginning with "Tom's"

STRING PATTERN MATCHING: REGEXP_LIKE OPERATOR

- The regexp_like operator is used for specifying patterns similar to that used in Unix regular expressions.

Query: Find the names of those clients whose names begin with Steven or Stephen (i.e., the name begins with ‘Ste’ followed by either ‘v’ or ‘ph’ followed by ‘en’ followed by any other characters).

```
select clientName
from Client
where regexp_like (clientName, '^Ste(v|ph)en');
```

Query: Find the names of those clients with a double vowel (i.e., double a, e, i, o or u) in their name, regardless of case.

```
select clientName
from Client
where regexp_like (clientName, '([aeiou])\1', 'i');
```

STRING PATTERN MATCHING: REGEXP_LIKE OPERATOR

Usage: regexp_like(*source_string*, *pattern*, [*match_parameter*])

where:

- *source_string* is a search value (usually an attribute name);
- *pattern* is a regular expression;
- *match_parameter* specifies a matching behaviour as follows
  - 'i' specifies case-insensitive matching.
  - 'c' specifies case-sensitive matching.
  - 'n' allows the period (.), which is normally the match-any-character wildcard character, to match the newline character.
  - 'm' treats the source string as multiple lines.

  If *match_parameter* is omitted then:
  - The default case sensitivity is used (usually case-sensitive).
  - A period (.) does not match the newline character.
  - The source string is treated as a single line.

ORDERING RESULT TUPLES: ORDER BY CLAUSE

Query: Find, in alphabetic order, the names of all clients having a loan at the Central branch.

```
select distinct clientName
from Borrower, Loan
where Borrower.loanNo=Loan.loanNo
          and branchName='Central'
order by clientName;
```

Ordering options

asc	- ascending (default)
desc	- descending

- Can sort on multiple attributes.

  e.g., order by clientName desc, amount asc

☞ **Since sorting a large number of tuples may be costly, it is desirable to use the order by clause only when necessary.**