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

EXAMPLE RELATIONAL SCHEMA AND DATABASE

(FOR EXERCISES)

Sailor(sailorId, sName, rating, age)

Boat(boatId, bName, color)

Reserves(*sailorId*, *boatId*, rDate)

Attribute names in italics are foreign key attributes.

Sailor

sailorId	sName	rating	age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55
32	Andy	8	25
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	35
85	Art	3	25
95	Bob	3	63
99	Chris	10	30

11 tuples

Reserves

sailorId	*boatId*	rDate
22	101	10/10/17
22	102	10/10/17
22	103	08/10/17
22	104	07/10/17
31	102	10/11/17
31	103	06/11/17
31	104	12/11/17
64	101	05/09/17
64	102	08/09/17
74	103	08/09/17
99	104	08/08/17

11 tuples

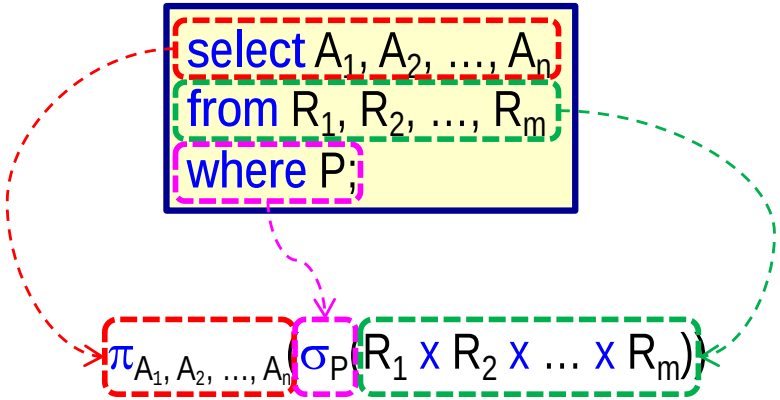
Boat

boatId	bName	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red
105	Serenity	Cyan

5 tuples

SQL BASIC STRUCTURE

- SQL is used in all commercial relational DBMS.
- It is based on set and relational algebra operations with certain modifications and enhancements.
- An SQL query has the basic form:



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

The equivalent relational algebra expression.

- An SQL query result is a relation (but it may contain duplicates).

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

EXAMPLE BANK RELATIONAL SCHEMA

Branch(branchName, district, assets)

Client(clientId, name, address, district)

Loan(loanNo, amount, *branchName*)

Account(accountNo, balance, *branchName*)

Borrower(*clientId*, *loanNo*)

Depositor(*clientId*, *accountNo*)

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;
```

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

- 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 unique names of all branches in the Loan relation.

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

---------> ***force*** the DBMS to remove duplicates

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

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

----------> ***force*** the DBMS *not* 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 Star House branch whose loan amount is greater than $12,000.

```
select loanNo
from Loan
where branchName='Star House'
      and amount>12000;
```

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

$\equiv$ $\pi_{\text{loanNo}}(\sigma_{\text{branchName='Star House'} \wedge \text{amount>12000}}(\text{Loan}))$

☞ **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.

CARTESIAN PRODUCT: FROM CLAUSE

- The **from** clause corresponds to the relational algebra Cartesian-product operation ($\times$).

  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.**

NATURAL JOIN: FROM CLAUSE

- A natural join is specified using the keywords natural join in the from clause.

  Query: Find the client id and loan number of the clients with loans.

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

- A natural join normally will join on all the common attributes of the two relations.

Author(authorId, title, name)
Book(bookId, title, *authorId*)

- The using condition specifies on which attributes to join.

```
select clientId, loanNo
from Borrower join Loan using (loanNo);
```

☞ **The keyword natural is not allowed when including using.**

JOIN: FROM/WHERE CLAUSE

- A join can be specified by adding the appropriate join condition in the **from** clause.

Query: Find the client id, loan number, loan amount and branch name of the clients who have loans.

```
select clientId, Loan.loanNo, amount, branchName
from Loan join Borrower on Loan.loanNo=Borrower.loanNo;
```

Attribute names *must* be qualified if ambiguous.

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

Why?

- The join condition can also be specified in the **where** clause.

```
select clientId, Loan.loanNo, amount, branchName
from Loan, Borrower
where Loan.loanNo=Borrower.loanNo;
```

OUTER JOIN: FROM CLAUSE

- An outer join is specified using the keywords [natural] {full | left | right} outer join in the from clause.

  Query: Find the client id, loan number and district of clients; include also clients who have no loan.

```
select clientId, loanNo, district
from Client natural left outer join Borrower;
```

- The join condition can also be specified using on.

```
select Client.clientId, loanNo, district
from Client left outer join Borrower on Client.clientId=Borrower.clientId;
```

- However, the join condition cannot be specified in the where clause.

What is the difference in the result of these two queries?

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 clientId from Depositor
union
select clientId from Borrower;
```

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

```
select clientId from Depositor
intersect
select clientId from Borrower;
```

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

```
select clientId from Depositor
minus
select clientId 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 clientId, Borrower.loanNo as loanId
from Borrower, Loan
where Borrower.loanNo=Loan.loanNo
        and branchName= 'Pacific Place';
```

Oracle Note

The keyword as ***is optional*** **in the** select **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 clientId, 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.

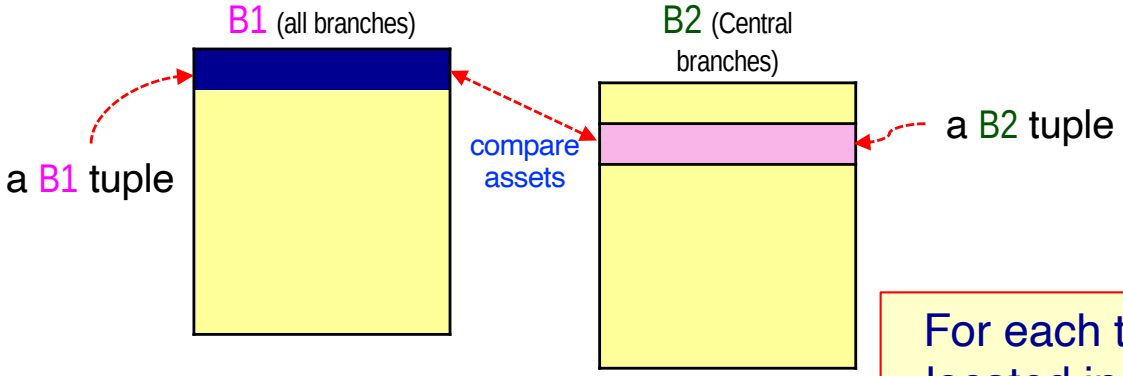
- An identifier for a relation (such as B and L above) is referred to as a *correlation name* in SQL.
  - Also known as *table alias*, *correlation variable* or *tuple variable*.
- While the SQL standard allows relations in the **from** clause to be renamed using the **as** clause, this is not allowed in Oracle.

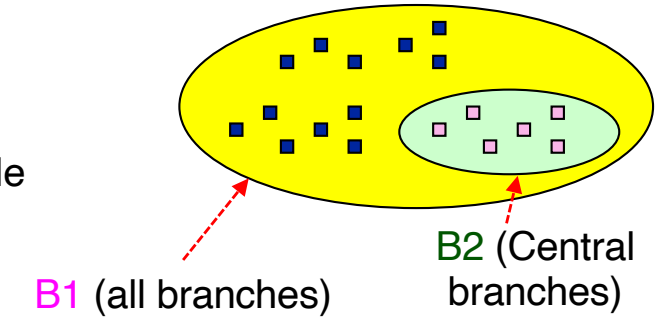
RENAME RELATIONS (CONT'D)

- 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 Central.

```
select distinct B1.branchName
from Branch B1, Branch B2
where B1.assets>B2.assets and B2.district='Central';
```





For each tuple in B1 is there *some* branch located in Central 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 address includes the substring 'Main' (e.g., Mainroad, Mainly Avenue, Mainmount Street, …).

```
select name
from Client
where address 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 name
from Client
where regexp_like (name, '^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 name
from Client
where regexp_like (name, '([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 Pacific Place branch.

```
select distinct name
from Client, Borrower, Loan
where Client.clientId=Borrower.clientId
        and Borrower.loanNo=Loan.loanNo
        and branchName='Pacific Place'
order by name;
```

Ordering options
asc - ascending (default)
desc - descending

- Can sort on multiple attributes.

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

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