COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 8
STRUCTURED QUERY LANGUAGE (SQL)

AGGREGATE FUNCTIONS

- An aggregate function operates on an attribute of a relation and returns a single value (i.e., a table with one row and one column).

count	number of tuples / values	**avg**	average value
stdev	standard deviation of values	**max**	maximum value
sum	sum of values (total)	**min**	minimum value

- For **avg**, **stdev** and **sum** the input must be numbers.
- For other functions, the input can be non-numeric (e.g., strings).
- All aggregate functions, except **count**(*), ignore null values in the input collection and return a value of null for an empty collection.

☞ **The count of an empty collection is defined to be 0.**

AGGREGATE FUNCTIONS: COMPUTATION

Query: Find the average account balance at the Pacific Place branch.

```
select avg(balance) as avgBalance
from Account
where branchName='Pacific Place'
```

Select the tuples where branch name equals Pacific Place.

Project on the balance attribute retaining duplicates.

Calculate the average.

Account

accountNo	balance	branchName
A-102	40000	Star House
A-222	75000	Pacific Place
A-201	90000	Star House
A-215	75000	Pacific Place
A-217	7500	Langham Place
A-224	100000	Pacific Place

accountNo	balance	branchName
A-222	75000	Pacific Place
A-215	75000	Pacific Place
A-224	100000	Pacific Place

avg(balance)

balance
75000
75000
100000

avgBalance
83333.33

AGGREGATE FUNCTIONS: EXAMPLES

Query: Find the number of accounts.

```
select count(*)
from Account;
```

☞ **Remember * stands for *all* attributes.**

Account

accountNo	balance	branchName
A-102	40000	Star House
A-222	75000	Pacific Place
A-201	90000	Star House
A-215	75000	Pacific Place
A-217	7500	Langham Place
A-224	100000	Pacific Place

Same as:

```
select count(branchName)
from Account;
```

Why?

Different from:

```
select count(distinct branchName)
from Account;
```

Why?

Cannot say:

```
select count(distinct *)
from Account;
```

SQL does not allow the use of distinct with count(*).

GROUP BY CLAUSE

A **group by** clause permits aggregate results to be displayed (e.g., max, min, sum, etc.) for groups. For example, **group by** x will get a result for every different value of x.

☞ **Aggregate queries without group by return a single number.**

Query: Find the number of accounts for *each* branch.

```
select branchName, count(*)
from Account
group by branchName;
```


GROUP BY CLAUSE (CONT'D)

Query: Find the number of accounts for *each* branch.

```
select branchName, count(*)
from Account
group by branchName;
```

Group the tuples by branch name.

For each group, count the number of tuples in the group.

Account

accountNo	balance	branchName
A-102	40000	Star House
A-222	75000	Pacific Place
A-201	90000	Star House
A-215	75000	Pacific Place
A-217	7500	Langham Place
A-224	100000	Pacific Place

accountNo	balance	branchName
A-102	40000	Star House
A-201	90000	Star House
A-222	75000	Pacific Place
A-215	75000	Pacific Place
A-224	100000	Pacific Place
A-217	7500	Langham Place

branchName	count(*)
Star House	2
Pacific Place	3
Langham Place	1

GROUP BY CLAUSE: ATTRIBUTES

Query: Find the balance and the number of accounts for *each* branch.

```
select branchName, balance, count(*)
from Account
group by branchName;
```

accountNo	balance	branchName
A-102	40000	Star House
A-201	90000	Star House
A-222	75000	Pacific Place
A-215	75000	Pacific Place
A-224	100000	Pacific Place
A-217	7500	Langham Place

Illegal! Why?

An attribute in the select clause must also appear in the group by clause.

The opposite is not true!

Attributes in the group by clause do not need to appear in the select clause.

GROUP BY CLAUSE: ATTRIBUTES (CONT'D)

Query: Find the balance and the number of accounts for *each* branch.

```
select branchName, balance, count(*)
from Account
group by branchName, balance;
```

branchName	balance	count
Star House	40000	1
Star House	90000	1
Pacific Place	75000	2
Pacific Place	100000	1
Langham Place	7500	1

Either is legal SQL.

```
select branchName, sum(balance), count(*)
from Account
group by branchName;
```

branchName	sum (balance)	count
Star House	130000	2
Pacific Place	25000	3
Langham Place	7500	1

GROUP BY CLAUSE: WITH JOIN

Query: Find the number of depositors for each branch.

```
select branchName, count(distinct clientId)
from Depositor natural join Account
group by branchName;
```

JOIN ⇒ (clientId, accountNo, balance, branchName)

clientId	branchName
1	Star House
2	Pacific Place
4	Langham Place
3	Pacific Place
1	Star House
5	Pacific Place

group by

clientId	branchName
1	Star House
~~1~~	~~Star House~~
4	Langham Place
3	Pacific Place
2	Pacific Place
5	Pacific Place

distinct

clientId	branchName
1	Star House
4	Langham Place
3	Pacific Place
2	Pacific Place
5	Pacific Place

count

branchName	count
Star House	1
Langham Place	1
Pacific Place	3

☞ **Group by and aggregate functions apply to the join result.**

HAVING CLAUSE

The **having** clause allows a condition to be applied to groups rather than to individual tuples.

Query: Find the names and average balances of all branches where the average account balance is more than $8000.

```
select branchName, avg(balance)
from Account
group by branchName
having avg(balance)>8000;
```

	accountNo	balance	branchName	
✓	A-102	40000	Star House	avg(65000.00)
	A-201	9000	Star House	
	A-222	7500	Pacific Place	
✓	A-215	7500	Pacific Place	avg(83333.33)
	A-224	10000	Pacific Place	
X	A-217	7500	Langham Place	avg(7500.00)

Any condition that appears in the **having** clause refers to the groups and is applied ***after*** the formation of the groups.

Any attribute in the **having** clause that is ***not aggregated*** must appear in the **group by** clause.

HAVING CLAUSE: EVALUATION SEQUENCE

1. Evaluate the from clause to get a relation.
2. If a where clause is present, apply the predicate in the where clause to the from clause result relation before the formation of groups.

   ☞ **Records that do not pass the where clause predicate are eliminated *before* the formation of groups.**

3. Group tuples satisfying the where clause predicate into groups by the group by clause, if present. If the group by clause is absent, the entire set of tuples satisfying the where clause predicate is treated as a group.
4. Apply the having clause, if present, to each group retaining only those groups satisfying the having clause.
5. Apply the aggregate functions in the select clause to get a single result for each group.

☞ **Any attribute present in the having clause *that is not being aggregated* must appear in the group by clause.**

HAVING CLAUSE: EXAMPLE

Query: Find the branch names in Central and Western district where the average account balance is more than $8000.

```
select branchName as branch
from Account natural join Branch
where district='Central and Western'
group by branchName
having avg(balance)>8000;
```

First, find the records that satisfy the **where** clause predicate.

Then, form the groups (include only those tuples that satisfy the **where** clause predicate).

Finally, apply the **having** clause to *each group*.

SQL Evaluation Note

In most relational DBMSs, an alias defined in the select clause cannot be used in most other clauses. This is because the order of execution of a select statement is:

1. from **clause**
2. where **clause**
3. group by **clause**
4. having **clause**
5. select **clause**
6. order by **clause**

Consequently, a column or alias name defined in a clause executed later cannot be used in an earlier clause.

NESTED SUBQUERIES

- Every SQL statement returns a relation as the result.

  ☞ **A relation can be null or contain only a single, atomic value.**

- Consequently, a value or a set of values can be replaced with a SQL statement (i.e., with a subquery).

  ☞ **The query is illegal if the subquery returns the wrong number of tuples or the wrong type for the comparison.**

```
select *
from Loan
where amount>12000;
```

```
select *
from Loan
where amount>(select avg(amount)
              from Loan);
```

This subquery *must* return a single, numeric value else it is illegal.

Subqueries are commonly used to test for set membership, do set comparison or determine set cardinality.

SET MEMBERSHIP: IN

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

```
select distinct clientId
from Borrower
where clientId in (select clientId
                   from Depositor)
```

The **in** operator tests for the presence of set membership (i.e., selects clients in the Borrower set *only if* they are in the Depositor set).
Duplicates are retained.

The *set* of clients who have an account.

SET MEMBERSHIP: NOT IN

Query: Find all clients who have a loan, but do not have an account.

```
select distinct clientId
from Borrower
where clientId not in (select clientId
                       from Depositor);
```

The **not in** operator tests for the absence of set membership (i.e., selects clients in the Borrower set *only if* they are not in the Depositor set).
Duplicates are retained.

The *set* of clients who have an account.

SET COMPARISON: SOME

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

☞ Equivalent to: Find the names of all branches that have greater assets than the minimum assets of any branch located in Central and Western.

```
select branchName
from Branch
where assets >some (select assets
                    from Branch
                    where district='Central and Western');
```

The **where** clause is true if the assets value of a **Branch** tuple is greater than at least one member of the set of all assets values of branches in Central and Western (i.e., greater than the minimum assets value in the set). ***Duplicates are retained.***

The *set* of assets values of all branches in Central and Western.

SET COMPARISON: SEMANTICS OF SOME

(5 < **some** | 0 | 5 | 6 |) returns true (since 5 is less than the maximum value 6 in the set)

(5 < **some** | 0 | 5 |) returns false (since 5 is not less than the maximum value 5 in the set)

(5 = **some** | 0 | 5 |) returns true (since 5 = 5)

(5 ≠ **some** | 0 | 5 |) returns true (since 5 ≠ 0)

Note

(= **some**) is equivalent to **in**

(≠ **some**) is not equivalent to **not in**

SET COMPARISON: ALL

Query: Find the names of those branches that have greater assets than *all* branches located in Central and Western.

☞ Equivalent to: Find the names of all branches that have greater assets than the maximum assets of any branch located in Central and Western.

```
select branchName
from Branch
where assets >all (select assets
                   from Branch
                   where district='Central and Western');
```

The **where** clause is true if the assets value of a **Branch** tuple is greater than each of the members of the set of all assets values of branches in Central and Western (i.e., greater than the maximum assets value in the set). ***Duplicates are retained.***

The *set* of assets values of all branches in Central and Western.

SET COMPARISON: SEMANTICS OF ALL

(5 < all | 0 / 5 / 6 |) returns false (since 5 is not less than all of the members in the set)

(5 < all | 6 / 9 |) returns true (since 5 is less than all of the members in the set)

(5 = all | 4 / 5 |) returns false (since 5 ≠ 4)

(5 ≠ all | 6 / 9 |) returns true (since 5 ≠ 6 or 9)

Note

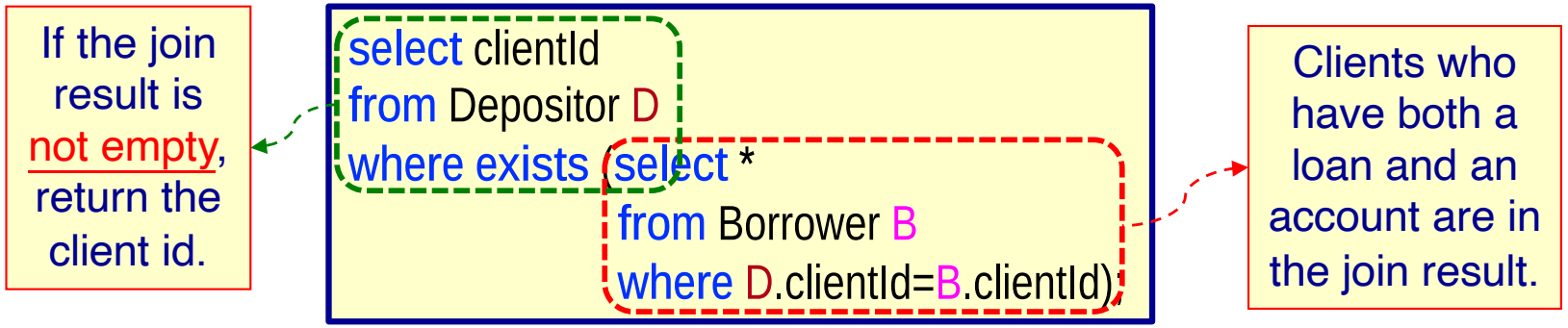
(≠ all) is equivalent to not in

(= all) is not equivalent to in

EMPTY RELATION TEST

- The **exists** operator returns true if the subquery is *not empty* (i.e., the subquery returns at least one tuple).

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



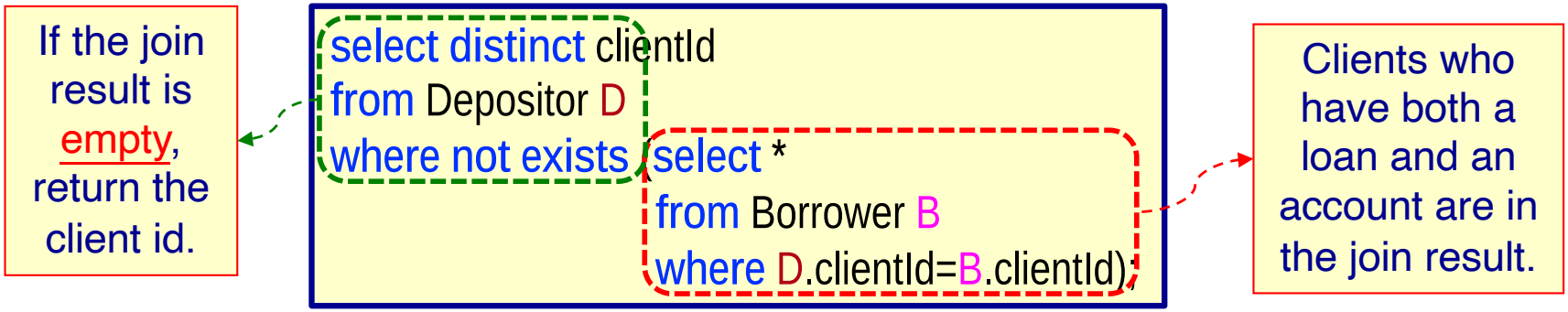
Scoping rules for correlation names (aliases) in subqueries.

- A correlation name defined in a subquery can be used only in the subquery itself or in any subquery contained in the subquery (e.g., D can be used in the nested select; B cannot be used in the outer select).
- Locally defined correlation names override globally defined names.

EMPTY RELATION TEST (CONTD)

- The **not exists** operator returns true if the subquery is *empty* (i.e., the subquery returns no rows).

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



☞ The **not exists** operator can be used to simulate set containment.

relation A contains relation B ⇔ not exists (B minus A)

Not implemented in Oracle.

DUPLICATE TUPLES TEST: UNIQUE

- The **unique** operator tests for the *non existence* (i.e., absence) of duplicate tuples in a subquery.

  ☞ **Returns true if the subquery contains no duplicate tuples.**

Query: Find all clients who have *only one account* at the Star House branch.

See later slide for an alternate way to answer this query.

```
select D1.clientId
from Depositor D1
where unique (select D2.clientId
              from Account, Depositor D2
              where D1.clientId=D2.clientId
                    and D2.accountNo=Account.accountNo
                    and Account.branchName='Star House');
```

For each depositor D1, check …

Clients at Star House with the same name as D1.

Find depositors with the same name as D1.

Not implemented in Oracle.

DUPLICATE TUPLES TEST: NOT UNIQUE

- The **not unique** operator tests for the *existence* (i.e., presence) of duplicate tuples in a subquery.

☞ **Returns true if the subquery contains two or more duplicate tuples.**

Query: Find all clients who have *at least two accounts* at the Star House branch.

See later slides for an alternate way to answer this query.

```
select D1.clientId
from Depositor D1
where not unique (select D2.clientId
                  from Account, Depositor D2
                  where D1.clientId=D2.clientId
                        and D2.accountNo=Account.accountNo
                        and Account.branchName='Star House');
```

☞ **Fails if tuples contain null values.**

DUPLICATE TUPLES TEST: REVISITED

- The **group by** and **having** clauses can test for the *non existence* (absence) and *existence* (presence) of duplicate tuples.

Query: Find all clients who have *only one account* at the Star House branch.

```
select clientId
from Depositor D, Account A
where D.accountNo=A.accountNo
        and branchName='Star House'
group by clientId
having count(*)=1;
```

Query: Find all clients who have *at least two accounts* at the Star House branch.

How would you answer this query?

SUBQUERIES IN THE FROM CLAUSE

- The **from** clause can contain a subquery expression.

Why?

Query: Find the name(s) of branches whose average balance is greater than the average account balance.

```
select branchName
from (select branchName, avg(balance) as avgBalance
      from Account
      group by branchName) result
where avgBalance>(select avg(balance)
                  from Account);
```

The average balance of all accounts.

result

branchName	avg Balance
Star House	65000.00
Langham Place	7500.00
Pacific Place	83333.33

The result relation contains the branch name and average balance of each branch.

☞ **The relation result is called a *derived (temporary) relation*.**

SUBQUERIES IN THE FROM CLAUSE (CONTD)

Query: Find the name and average balance of branches with the *maximum average* account balance.

```
select branchName, avgBalance
from (select branchName, avg(balance) as avgBalance
      from Account
      group by branchName) result
where avgBalance=(select max(avgBalance)
                  from result);
```

result

branchName	avg Balance
Star House	65000.00
Langham Place	7500.00
Pacific Place	83333.33

The result relation contains the branch name and average balance of each branch.

The maximum average balance in the result relation.

Oracle Note

This query is *not allowed in Oracle* due to Oracle's scoping rules. (The scope of the result relation is restricted to the outer select clause.)

See the next slide.

WITH CLAUSE

- Allows a derived (temporary) relation to be defined that is available only to the query in which the **with** clause occurs.

Query: Find the name and average balance of branches with the *maximum average* account balance.

```
with result (branchName, avgBalance) as
    (select branchName, avg(balance)
     from Account
     group by branchName)
select branchName, avgBalance
from result
where avgBalance=(select max(avgBalance)
                  from result);
```

result

branchName	avg Balance
Star House	65000.00
Langham Place	7500.00
Pacific Place	83333.33

The result relation contains the branch name and average balance of each branch.

Oracle Note
This query *is allowed* in Oracle.

The maximum average balance in the result relation.