Lecture 5: Basic Query Composition using SQL DML

CSX3006 DATABASE SYSTEMS

ITX3006 DATABASE MANAGEMENT SYSTEMS

Outline

- MORE on SQL DML constructs
  - Ordering the output tuples
  - Rename operations
  - Aggregate Functions and Grouping
  - Effects of null values
  - Set Operations: union, intersect, except (difference)
  - Nested sub queries in where clause
    - in, exists, all, some, any, unique
  - Various join operations in where clause

SQL DML Extensions

- SELECT ... ⇐ Extension to the select clause

  distinct, as, sum, count, min, max, avg, 'arithmetic ops'

- FROM ... ⇐ Extension to from clause

  as, inner join, left outer join, right outer join, full outer join,

  natural, on , using

- WHERE ... ⇐ Extension to where clause

  <, >, =, <>, >=, <=, and, or, not, like, is null, is not null,

  exists, in, any, some, all, unique, 'Sub queries'

GROUP BY ...

HAVING ...

ORDER BY ...

Eliminating Duplicates with Distinct - 1

- Unlike Relational Algebra, SQL DML operators do not eliminate duplicate records (tuples) by default.
  - Need to request explicitly if required with the keyword distinct

customers relation

ID	FirstName	LastName	DOB	city
124	James	Bond	07/07/42	London
286	David	Grant	21/03/57	Bangkok
324	Paul	Jones	02/05/77	Bangkok
442	Miranda	Jeffery	15/08/75	Vienna
522	Susan	Jones	31/08/74	Bangkok

select city
from customers;

city
London
Bangkok
Bangkok
Vienna
Bangkok

select distinct city
from customers;

city
London
Bangkok
Vienna

Eliminating Duplicates with Distinct - 2

customers relation

ID	FirstName	LastName	DOB	City
124	James	Bond	07/07/42	London
286	David	Grant	21/03/57	Bangkok
324	Paul	Jones	02/05/77	Bangkok
442	Miranda	Jeffery	15/08/75	Vienna
522	Susan	Jones	31/08/74	Bangkok

```
select distinct LastName, City
from  customers;
```

LastName	City
Bond	London
Grant	Bangkok
Jones	Bangkok
Jeffery	Vienna

Ordering the display of tuples

- order by
  - Sorts the output of a query based on one or more attributes
  - asc (ascending order -- default) or desc (descending order)

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

- Ordering (Comparison) of values depends on data types
  - Numerical Information: The usual numerical order
  - Textual Information: Lexicographical order (Dictionary Order)
  - Time and Date Information: Chronological Order

Note on Order by clause

- Order by clause affects only the listing order of the output tuples
  - Does not affect the operation of queries; NOT affect what is chosen as output
    - In set and multiset, the order of tuples does NOT matter

Ordering the display of tuples – Example - 1

- Find the names of all customers who have one or more loans at the Perryridge branch along with his/her loan numbers and the loan amount for each loan.
  - Display the result in *alphabetic order* of customer name; when a customer has multiple loans, list the loans in the *descending order* of the loan amounts.

Ordering the display of tuples – Example - 2

- Find the names of all customers who have one or more loans at the Perryridge branch along with his/her loan numbers and the loan amount for each loan.
  - Display the result *in alphabetic order of* customer name; when a customer has multiple loans, list the loans *in the descending order* of the loan amounts.

```
select  customer_name, amount, loan.loan_number
from    borrower, loan
where   borrower.loan_number = loan.loan_number and
        branch_name = 'Perryridge'
order by customer_name asc, amount desc, loan.loan_number asc;
```

Ordering the display of tuples – Example - 3

- What if there are multiple loans with the same amount belonging to the same customer?
  - Will adding keyword distinct to the select list make any difference?

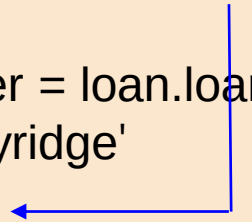
```
select  distinct customer_name, amount, loan.loan_number
from    borrower, loan
where   borrower.loan_number = loan.loan_number and
        branch_name = 'Perryridge'
order by customer_name asc, amount desc, loan.loan_number asc;
```

  - WHY/WHY NOT?

Caution in Using Order By

1. Attributes listed **in the order by clause should be** one or more of those listed **in select clause**

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



2. Use order by clause ONLY WHEN you need tuples displayed in certain order
   - Sorting takes times esp. when dealing with large data sets!!

Rename Operation (as) - 1

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

  old-name* as *new-name

  - **Renaming of *attributes* in *select* clause**
  - **Renaming of *relations* (*table*) in *from* clause**
    - keyword **as** is optional here in SQL standard

Renaming of *attributes* in *select* clause - 1

- When Renaming of Attributes could be useful?
  1. Arithmetic expression is used in the select clause
  2. More meaningful name is used in the output relation

```
select loan_number as loan_id,
       amount * 0.05 as loan_interest
from loan;
```

- The output relation has the following attributes' name:
  - (loan_id, loan_interest)

Renaming of *relations* (*table*) in *from* clause

- When Renaming of Attributes could be useful?
  3. Allows to refer to relations via more meaningful or shorter names in queries (*tuple variables*)

```
select customer_name, borrower.loan_number, amount * 0.05
from borrower, loan
where borrower.loan_number = loan. Loan_number;
```

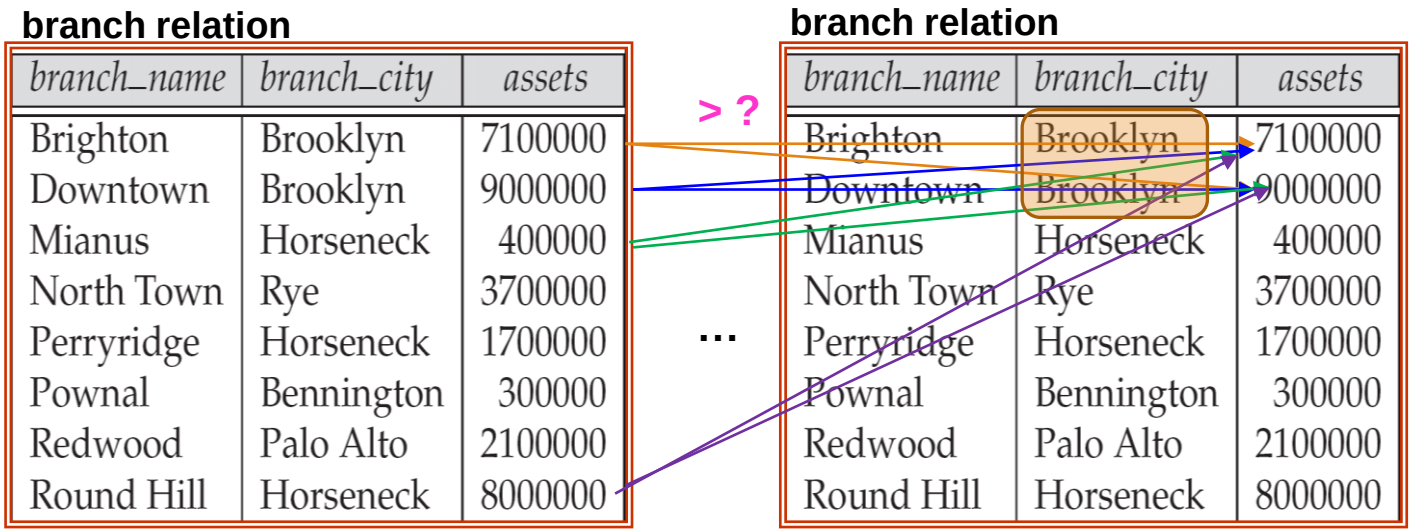
```
select customer_name, b.loan_number, amount * 0.05
from borrower as b, loan as l
where b.loan_number = l.loan_number;
```

- The output relation has the following attributes' name:
  - (customer_name, loan_number, ?column?)

Should also rename the attribute!!

Rename Operation (as) - Example - 1

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

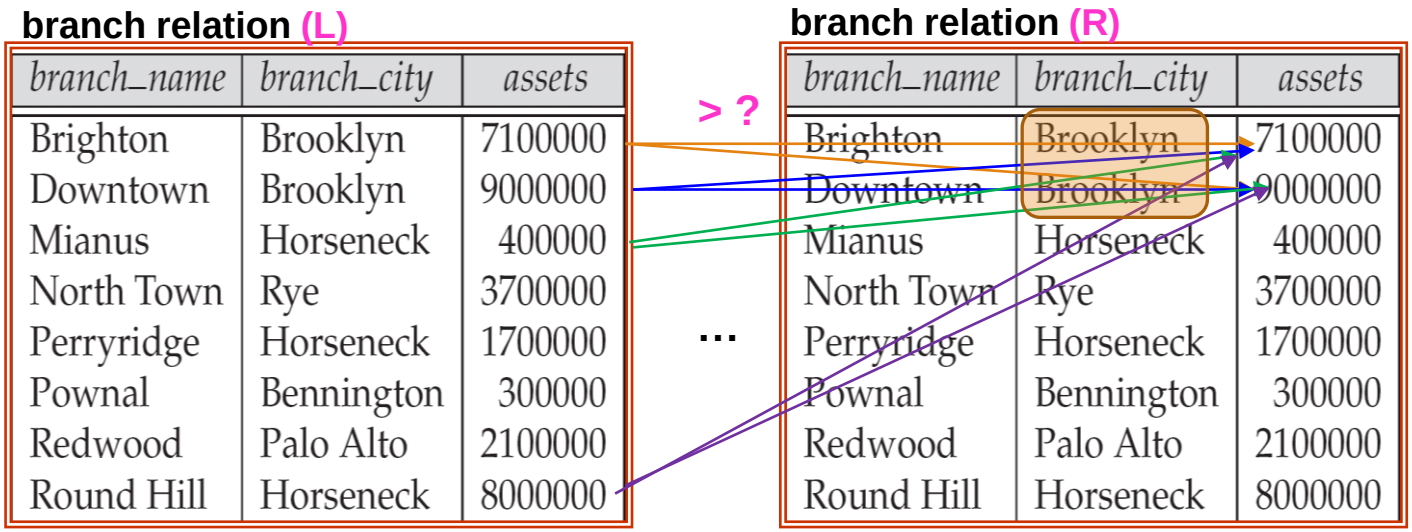



```
select distinct branch_name
from branch, branch
where assets > assets and branch_city = 'Brooklyn';
```

Ambiguous!

Rename Operation (as) - Example - 2

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




```
select distinct L.branch_name
from branch L, branch R
where L.assets > R.assets and R.branch_city = 'Brooklyn'
```

Tuple Variables allow *comparisons of tuples* from the SAME relation!!

Aggregate Functions and Grouping - 1

- SQL supports **aggregate operators** in the select clause
  - **count**, **sum**, **min**, **max**, **avg**
    - Operates on **multisets of values** of a column of a relation
      - For **sum**, **avg**, must be *numerical values*
      - For others, can be *nonnumeric data types*, such as strings
    - **Returns a single value**

Aggregate Functions and Grouping – Example - 1

- Find the average account balance at the Brighton branch.

Aggregate Functions and Grouping – Example - 1

- Find the average account balance at the Brighton branch.

```
select avg(balance) as Average_Balance_in_Brighton
from account
where branch_name = 'Brighton';
```

Result:

	avg numeric
1	825.0000000000000000

```
select round(avg(balance) , 2) as Average_Balance_in_Brighton
from account
where branch_name = 'Brighton';
```

Result:

	round numeric
1	825.00

Aggregate Functions and Grouping – Example - 2

- Find the number of depositors in the bank.

Aggregate Functions and Grouping – Example - 2

- Find the number of depositors in the bank.

```
select count (customer_name)
from depositor;
```

```
select count (distinct customer_name)
from depositor;
```

≠

```
select distinct count (customer_name)
from depositor;
```

Be careful when placing keyword 'distinct'!!

Aggregate Functions and Grouping - 2

- SQL works on multisets

```
select count (a) from r;
select count (distinct a) from r;
/*select distinct sum(c) from r;*/
/*select distinct sum(distinct c) from r;*/

select sum (c) from r;
select sum (distinct c) from r;

select avg (c) from r;
select avg (distinct c) from r;

select max (c) from r;
select max (distinct c) from r;
```

Relation r

a	*b*	*c*
α	α	6
α	β	4
β	β	6
β	β	8

Aggregate Functions and Grouping - 3

select count (*) from r;

- count (*) returns total (counting duplicates separately, and also counting tuples having null values) number of tuples in a relation.
- count (distinct *) is an ILLEGAL expression in SQL

Relation r

a	*b*	*c*
α	α	6
α	β	4
β	β	6
β	β	8

Aggregate Functions and Grouping – Example - 3

- Whether you should eliminate duplicates before an aggregate function is performed or not depends on Business Logic and the Schema Design

- Find the average account balance at the Brighton branch

Aggregate Functions and Grouping – Example - 3

- Find the average account balance at the Brighton branch

```
select avg(balance)
from account
where branch_name = 'Brighton';
```

E.g.,
The average of **3** accounts' balance at Brighton {500, 500, 200} is 400.

```
select avg(distinct balance)
from account
where branch_name = 'Brighton';
```

Using **distinct**, it *eliminate the duplicate value* of 500. As the result, the average of **2** distinct balances {500, 200} is 350.

Aggregate Functions and Grouping – Example - 4

- Find the number of customers who have accounts at the Brighton branch

Aggregate Functions and Grouping – Example - 4

- Find the number of customers who have accounts at the Brighton branch

```
select count(distinct customer_name)
from depositor D, account A
where D.account_number = A.account_number and
        branch_name = 'Brighton';
```

```
select count(customer_name)
from depositor D, account A
where D.account_number = A.account_number and
        branch_name = 'Brighton';
```

Aggregate Functions and Grouping - 4

- **Aggregate functions may also be applied to a group** of tuples rather than every tuple in a relation.
  - A group is formed by keyword **group by** clause
    - **An attribute or attributes** given in the clause are used to form groups
    - Tuples (rows) with the same value on all the attributes listed are placed in a one group

Aggregate Functions and Grouping – Example - 5

- Find the average account balance at each branch

Aggregate Functions and Grouping – Example - 5

- Find the average account balance at each branch

```
select branch_name, avg(balance) as average_balance
from account
group by branch_name;
```

Aggregate Functions and Grouping – Example - 6

- Find the number of depositors in each branch.

Aggregate Functions and Grouping – Example - 6

- Find the number of depositors in each branch.

```
select branch_name, count(distinct customer_name) as customer_count
from depositor D, account A
where D.account_number = A.account_number
group by branch_name;
```

Note that expressions allowed in the **select clause** when a **group by** is used in a query are the grouping attributes and/or aggregate functions.

Find Resulted Relations

- select branch_name
  from account
  group by branch_name;

- select distinct branch_name
  from account;

- select branch_name
  from account
  where balance > 500
  group by branch_name;

- select distinct branch_name
  from account
  where balance > 500;

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

Aggregate Functions and Grouping - 5

- **group by** clause enables grouping of tuples based on values of attributes
- We may be interested **only** in **groups that satisfies certain conditions**
  - Keyword **having** clause of SQL allows us to **'filter' groups**
    - Specifies condition evaluated against each group

Aggregate Functions and Grouping – Example - 7

- Find the names of all branches whose average account balance is more than $600, along with their average account balances

	branch_name character (15)	average_balance numeric
1	Mianus	700.00
2	Brighton	825.00
3	Redwood	700.00
4	Round Hill	350.00
5	Downtown	500.00
6	Perryridge	400.00

Aggregate Functions and Grouping – Example - 7

- Find the names of all branches whose average account balance is more than $600, along with their average account balances

```
select branch_name, avg(balance) as average_balance
from account
group by branch_name
having avg(balance) > 600;
```

Aggregate Functions and Grouping – Example - 8

- Find the names of all branches in which highest account balance is greater than $600 and average account balance is greater than $500

Aggregate Functions and Grouping – Example - 8

- Find the names of all branches in which highest account balance is greater than $600 and average account balance is greater than $500

```
select branch_name

from account

group by branch_name

having max(balance) > 600 and avg(balance) > 500;
```

Aggregate Functions and Grouping – Example - 9

- Find the average balance for each customer who lives in Palo Alto and has at least two accounts. List the output in the ascending order of the average balance

Aggregate Functions and Grouping – Example - 9

- Find the average balance for each customer who lives in Palo Alto and has at least two accounts. List the output in the ascending order of the average balance

Step 5	**select D**.customer_name, **avg**(balance) **as** *average_balance*
Step 1	**from** depositor **D**, account **A**, customer **C**
Step 2	**where D.**account_number = **A**.account_number **and**
	D.customer_name = **C**.customer_name **and**
	customer_city = 'Palo Alto'
Step 3	**group by D**.customer_name
Step 4	**having count**(**D**.account_number) >= 2
Step 6	**order by** *average_balance;*

Side Note: where vs. having

WHERE CLAUSE

- Filters against individual tuple
  - It CANNOT have aggregate functions in its predicate.

HAVING CLAUSE

- Filter applied on each group of tuples
  - It CAN have predicate conditions that include aggregate functions only.

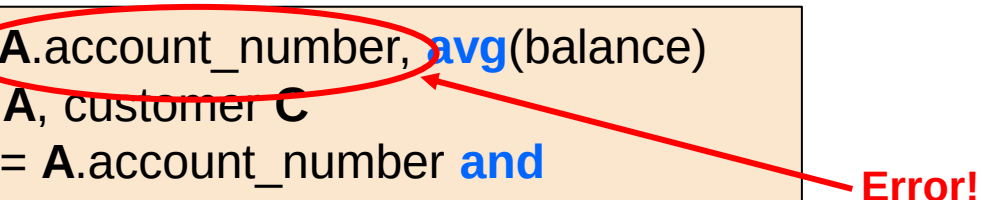
Aggregate Functions and Grouping - 6

- The only attributes that can appear in the select clause in a "grouped" query are
  - Aggregate operators (,which are applied to the group) and/or
  - Grouping attribute(s)

```
select D.customer_name, avg(balance) as average_balance
from depositor D, account A, customer C
where D.account_number = A.account_number and
        D.customer_name = C.customer_name and
        customer_city = 'Palo Alto'
group by D.customer_name
having count(D.account_number ) >= 2;
```

Aggregate Functions and Grouping - 7

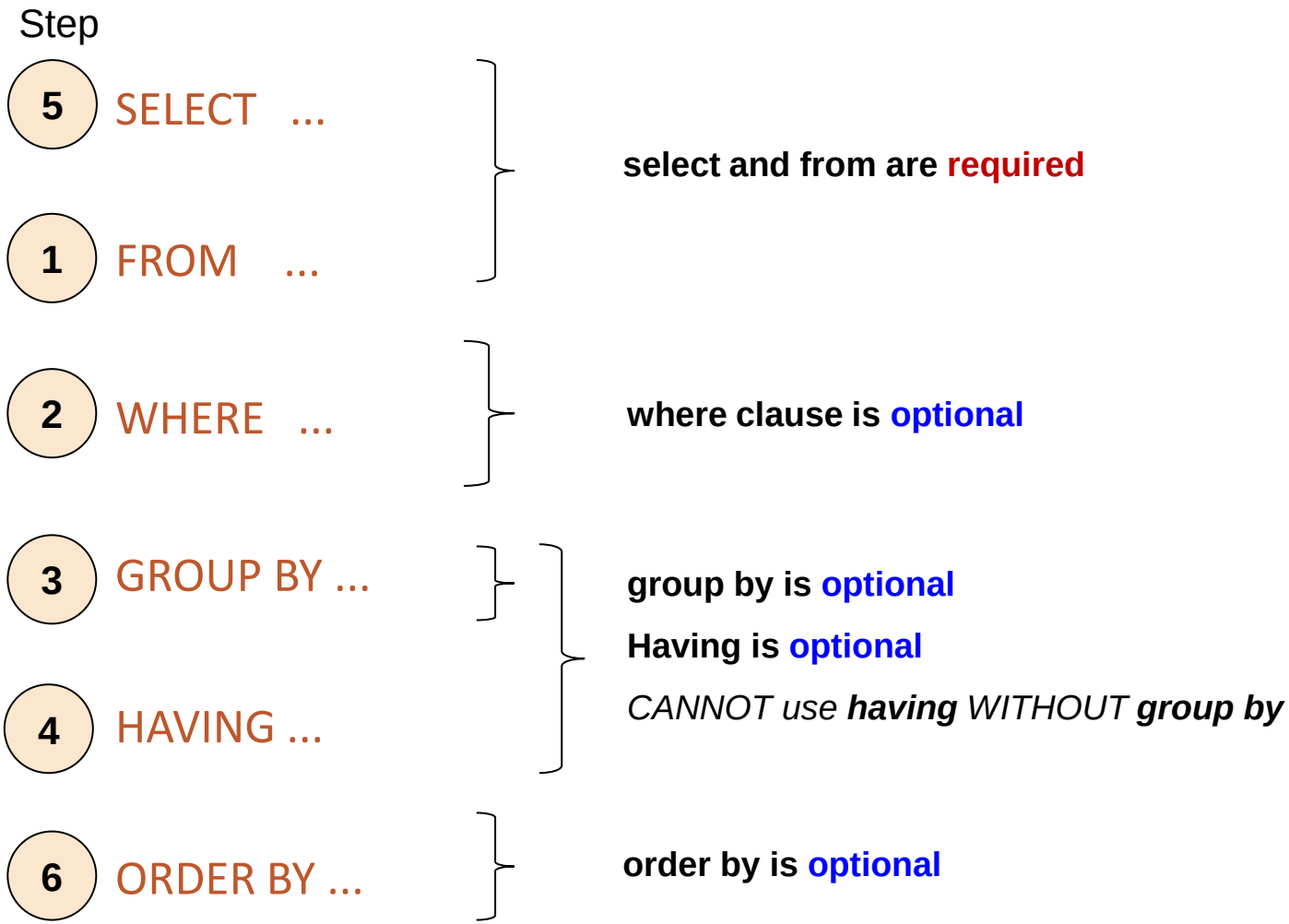
```
select D.customer_name, A.account_number, avg(balance)
from depositor D, account A, customer C
where D.account_number = A.account_number and
        D.customer_name = C.customer_name and
        customer_city = 'Palo Alto'
group by D.customer_name
having count(D.account_number ) >= 2;
```



Practice 5-1
(Use Aggregation Functions)

1. Find average balance of all loans in the bank
2. Display average balance and name of each branch in the bank
3. Count number of account that each customer owns
4. Find number and balance of all loans that have multiple owners.
5. Find the smallest balance of account either from the Downtown or Redwood branch
6. Find the names of all branches whose average loan balance is greater than $1000 and average account balance is less than $600
7. Find the different amount of loan and deposit of Hayes

Structure of SQL Queries



SQL queries and null value

- Be reminded that null is possible value in any domain
  - Refer to the effects of null in relational algebra operations in lecture 3
- SQL follows the exact same rule
- SQL offers special **comparison operators**
  - **is null** returns true if the operand is null
  - **is not null** returns true if the operand is not null

SQL queries and null value – Example - 1

- Find all loan number which appear in the *loan* relation with null values for *amount.*

SQL queries and null value – Example - 1

- Find all loan number which appear in the *loan* relation with null values for *amount.*

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

Whether such a query makes sense or not *depends on the schema of the relation*

- Is the field amount declared not null?

SQL queries and null value – Example - 1

- Find all loan number which appear in the *loan* relation with null values for *amount.*

Let's assume the schema of the relation ALLOWS amount to be **null**

```
select loan_number
from loan
where amount = 0;
```

```
select loan_number
from loan
where amount = 0 or amount is null;
```

SQL queries and null values - 1

- **The result of any arithmetic expression (+, -, *, /) involving *null* is *null.***
  - *null* **+** *3* → *null*
- **The result of any comparisons (=, !=, <, <=, >, >=) involving null is unknown**
  - (null **<** 500) → unknown **(null = null) → unknown**
  - **(null is null) → true** (500 **is null**) → false **in SQL**
- **Three-valued logic using the truth value unknown:**
  - 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*

SQL queries and null values - 2

- If the condition in where/having clause is evaluated to **unknown**, then the result of such clause is **false**.
- Aggregate functions ignores null values except count(*)
- select distinct, and group by treats null value **as any other values**
  - **All null values** are **treated equal and grouped into** its own **group of null**
- order by treats null value **as the largest**
  - Comes last in **asc**
  - Comes first in **desc**

SQL queries and null values – Example - 1

Name	Age	Food
Jenny	33	**null**
Donna	**null**	Pizza
Roy	21	Steak
Sara	34	**null**

member relation

```
select food
from member;
```

Food
null
Pizza
Steak
null

```
select distinct food
from member;
```

Food
null
Pizza
Steak

SQL queries and null values – Example - 2

Name	Age	Food
Jenny	33	**null**
Donna	**null**	Pizza
Roy	21	Steak
Sara	34	**null**

member relation

```
select name, age
from member
where food is null;
```

Name	Age
Jenny	33
Sara	34

SQL queries and null values – Example - 3

Name	Age	Food
Jenny	33	**null**
Donna	**null**	Pizza
Roy	21	Steak
Sara	34	**null**

member relation

```
select *
from member
where age > 25 or food = 'Pizza';
```

Name	Age	Food
Jenny	33	**null**
Donna	**null**	Pizza
Sara	34	**null**

SQL queries and null values – Example - 4

Name	Age	Food
Jenny	33	**null**
Donna	**null**	Pizza
Roy	21	Steak
Sara	34	**null**

member relation

```
select avg(age) as avg_age
from member;
```

avg_age
29.3333

SQL queries and null values – Example - 5

Name	Age	Food
Jenny	33	**null**
Donna	**null**	Pizza
Roy	21	Steak
Sara	34	**null**

member relation

```
select food, sum(age)
from member
group by food;
```

Food	sum(age)
null	67
Pizza	**null**
Steak	21

SQL DML Extensions - 2

(select ... from ... where)

[set operator]

(select ... from ... where)

Operators that expect two or more complete SQL queries as operands:

- union, intersect, except

If needed constant tuples, refer to https://www.postgresql.org/docs/9.5/static/queries-values.html

Set Operations - 1

- SQL supports the following set operations: **union**, **intersect**, **except**
  - Eqivalent to $\cup$, $\cap$, – in relational algebra operators
  - **SQL set operations automatically eliminates duplicates**
  - SQL set operators are binary operators and **require two input relations**
    - The two input relations must be *compatible* (same set of attributes)

Set Operations – Example - 1

- Find all customers who have a loan, an account, or both

Set Operations – Example - 1

- Find all customers who have a loan, an account, **or both**

```
(select customer_name from depositor)
union
(select customer_name from borrower);
```

Set Operations – Example - 2

- Find all customers who have **both** a loan and an account.

Set Operations – Example - 2

- Find all customers who have **both** a loan and an account.

```
(select customer_name from depositor)
intersect
(select customer_name from borrower);
```

Set Operations – Example - 3

- Find all customers who have an account **but no** loan.

Set Operations – Example - 3

- Find all customers who have an account **but no** loan.

```
(select customer_name from depositor)
except
(select customer_name from borrower);
```

Set Operations - 2

- The set operators **automatically eliminates duplicates**
  - **union**, **intersect**, **except**
- **To retain all duplicates**, use the following multiset versions
  - **union all**, **intersect all**, **except all**

Set Operations – Example - 4

M
a
a
b
b
d

r

N
a
a
b
c

s

```
(select * from r)
union
(select * from s);
```

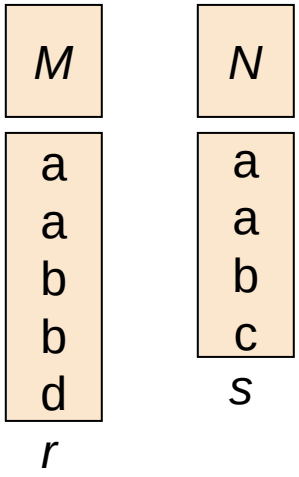
a
b
c
d

```
(select * from r)
union all
(select * from s);
```

m + n

a
a
a
a
b
b
b
c
d

Set Operations – Example - 5



(select * from r)
intersect
(select * from s)

a
b

(select * from r)
intersect all
(select * from s)

a
a
b

min (m,n)

Set Operations – Example - 6

M
a
a
b
b
d

r

N
a
a
b
c

s

(select * from r)
except
(select * from s)

d

(select * from r)
except all
(select * from s)

b
d

max (0,m-n)

SQL DML Extensions - 1

- SELECT ... ⇐ Extension to the select clause

  distinct, as, sum, count, min, max, avg, 'arithmetic ops'

- FROM ... ⇐ Extension to from clause

  as, inner join, left outer join, right outer join, full outer join,

  natural, on, using

- WHERE ... ⇐ Extension to where clause

  <, >, =, <>, >=, <=, and, or, not, like, is not null,

  exists, in, any, some, all, unique, 'Sub queries'

GROUP BY ...

HAVING ...

ORDER BY ...

Nested Subqueries in Predicate conditions - 1

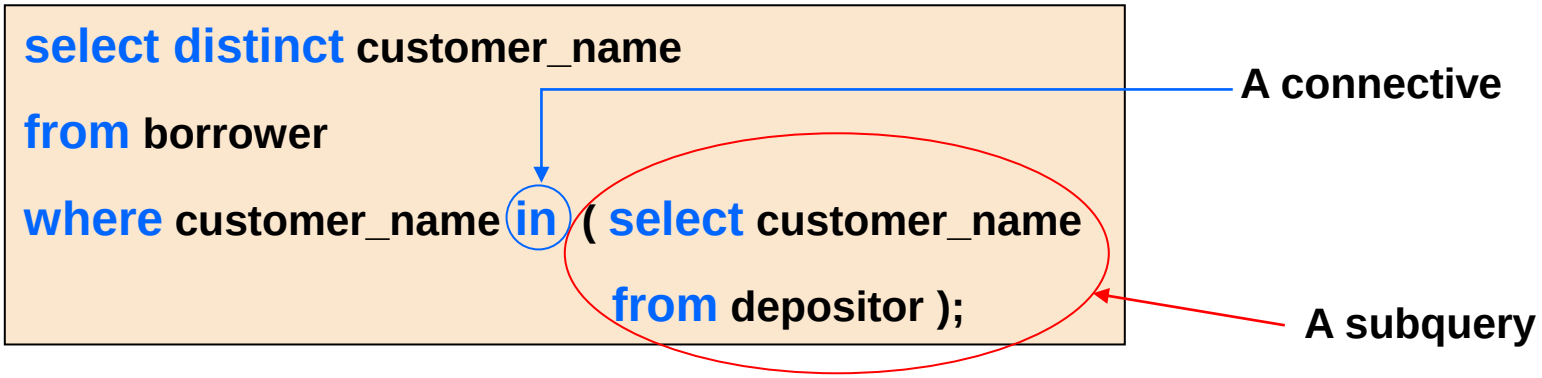
- **A subquery**
  - Is a complete select-from-where (including optional clauses) expression, which is **nested within another query.**
  - can be *a part of predicate condition expression* in **where** clause or **having** clause of an outer query.

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

A subquery

Nested Subqueries in Predicate conditions - 2

- Tuples can be evaluated against the output tuples of the subquery using one of the following **connectives**:
  - **in**, **not in**
  - **some**, **all**
  - **exists**, **not exists**
  - **unique**, **not unique**



Simple Nested Subquery example - 1

- Find the names of customers ***who live in the same city as*** 'Hayes'.

```
select L.customer_name
from customer L, customer R
where L.customer_city = R.customer_city and R.customer_name = 'Hayes' and
        L.customer_name <> 'Hayes';
```

This condition is required since we don't want have Taylor listed in the output

Table L

customer_name	customer_street	customer_city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Table R

customer_name	customer_street	customer_city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Simple Nested Subquery example - 2

- Find the names of customers who live in the same city as ‘Hayes’.

customer_name	customer_street	customer_city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

```
select customer_name
from customer
where customer_name <> ‘Hayes’ and
      customer_city in ( select customer_city
                         from customer
                         where customer_name = ‘Hayes’ ) ;
```

Nested Subquery which finds the city where Hayes lives

Hayes	Main	Harrison

Equality test (=) can be used if the subquery will CERTAINLY return a SINGLE value.

- Otherwise, it would result in error
- “in” connective is preferred as it tests for set membership.

Set Membership Test Using in, not in

- Used to test if a tuple (in the outer query) is a member of a set of tuples (in the subquery)
  - e.g., Find all the customers who **have both a loan and an account** at the bank

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

borrower relation

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

depositor relation

Set Membership Test Using in, not in – Example - 1

- Find all the customers who have both a loan and an account at the bank
- **Firstly** find **members of the set of depositors**

```
select customer_name
from depositor;
```

← Generates the (multi)set of account holders

- **Then,** find those **borrowers** who **also appear in** the above query (depositors)

```
select customer_name
from borrower
where customer_name in
        select customer_name
        from depositor;
```

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

Set Membership Test Using in, not in – Example - 2

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

loan_number	branch_name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

loan relation

customer_name	loan_number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

borrower relation

account_number	branch_name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

account relation

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

depositor relation

Set Membership Test Using in, not in – Example - 2

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

```
( select customer_name
from account A, depositor D
where A.account_number = D.account_number and
            branch_name = 'Perryridge')
intersect
( select customer_name
from loan L, borrower B
where L.loan_number = B.loan_number and   branch_name =
'Perryridge') ;
```

Must be 'Compatible'

```
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 and branch_name = 'Perryridge') ;
```

Set Membership Test Using in, not in – Example - 3

- Find all customers who **have a loan, but do not have an account**

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

borrower relation

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

depositor relation

Set Membership Test Using in, not in – Example - 3

- Find all customers who **have a loan, but do not have an account**

```
(select customer_name from borrower)
except
(select customer_name from depositor);
```

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

Set Membership Test Using in, not in – Example - 4

- Find the names of customers who **have a loan** at the bank **and whose names are neither Smith nor Jones**

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

borrower relation

Set Membership Test Using in, not in – Example - 4

- Find the names of customers **who have a loan** at the bank **and whose names are neither Smith nor Jones**

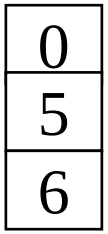
```
select distinct customer_name
from borrower
where customer_name not in ('Smith', 'Jones');
```

Set Comparison Using some, all

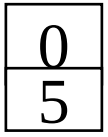
- **> some, >= some, < some, <= some, = some, <> some**
  - **In SQL**, **some** and **any** are ***synonymous***
  - Test if a tuple is >, >=, <, <= , =, <> to **some** (any) tuple of a relation

- **> all, >= all, < all, <= all, = all, <> all**
  - Test if a tuple is >, >=, <, <= , =, <> to **all** tuples of a relation

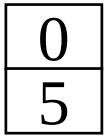
Set Comparison Using some, all - Example - 1

(5 < **some** 

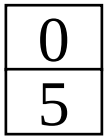
) = true

(5 < **some** 

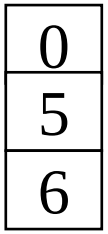
) = ???

(5 = **some** 

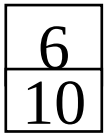
) = ???

(5 ≠ **some** 

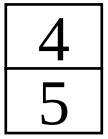
) = ??? (since 0 ≠ 5)

(5 < **all** 

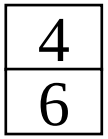
) = false

(5 < **all** 

) = ???

(5 = **all** 

) = ???

(5 ≠ **all** 

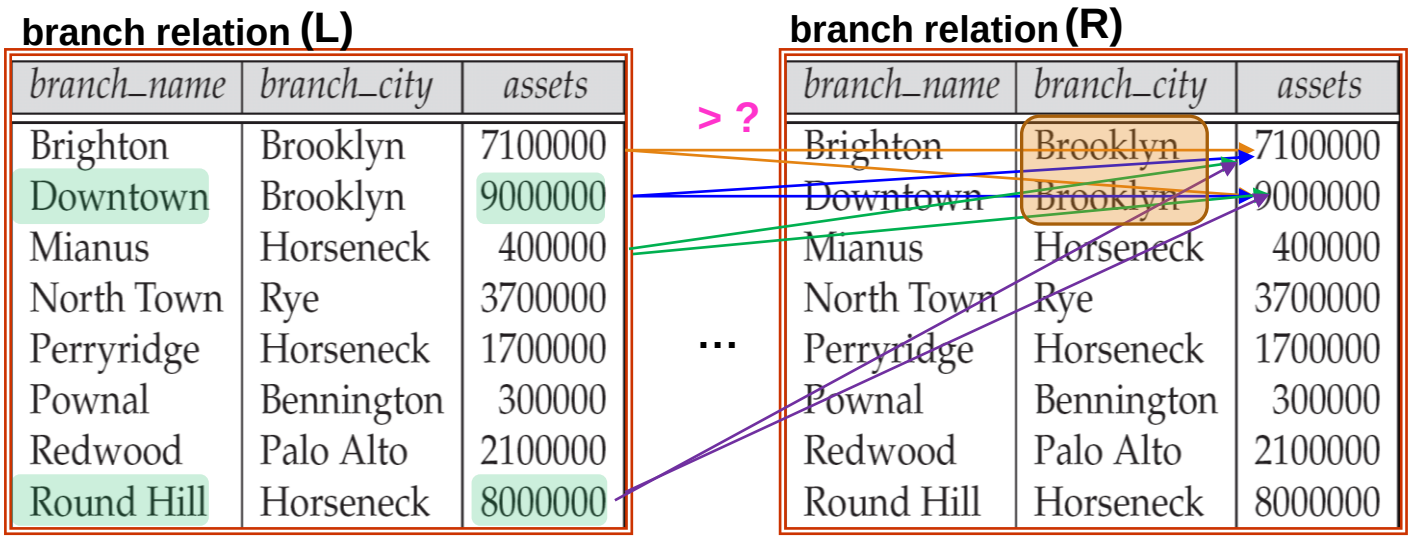
) = ??? (since 5 ≠ 4 and 5 ≠ 6)

(= some) ≡ in
However, **(≠ some) ≠ not in**

(≠ all) ≡ not in
However, **(= all) ≠ in**

Set Comparison Using some, all - Example - 2

- Find all branches that have **greater assets than some of (any) branch located in Brooklyn**.



branch_name	branch_city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

branch_name	branch_city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Set Comparison Using some, all - Example - 2

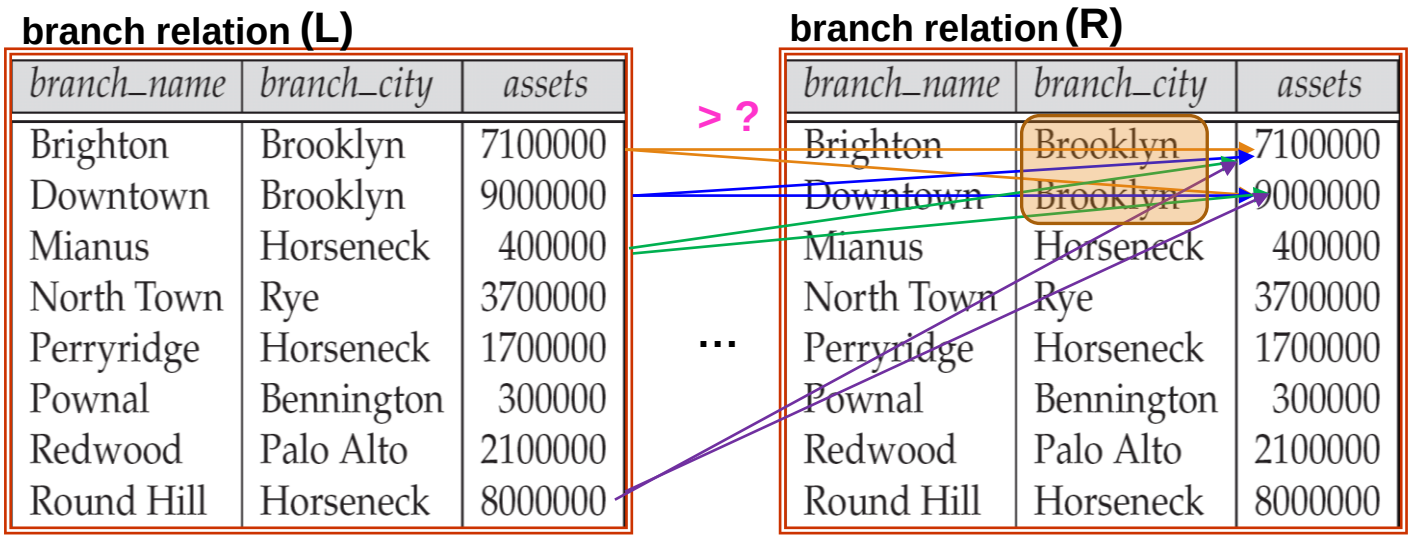
- Find all branches that have **greater assets than some of (any) branch located in Brooklyn**.

```
select distinct  L.branch_name
from branch L, branch R
where L.assets > R.assets and R.branch_city = 'Brooklyn';
```

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

Set Comparison Using some, all - Example - 3

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



branch_name	branch_city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

branch_name	branch_city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Note: the answer is null in this example.

Set Comparison Using some, all - Example - 3

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

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

Note: the answer is null in this example.

Set Comparison Using some, all - Example - 4

- Find the branch that **has the highest average balance**

account_number	branch_name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Avg. bal. = 825

account relation

Set Comparison Using some, all - Example - 4

- Find the branch that **has the highest average balance**

```
select branch_name
from account
group by branch_name
having avg(balance) >= all ( select avg(balance)
                             from account
                             group by branch_name );
```

Note that SQL does **not allow composition** of aggregate functions

e.g) max(avg(balance)) → illegal in SQL

Test for Empty Relation: exists, not exists

- **exists (*relation*)** is true if ***relation*** is not empty
- **not exists (*relation*)** is true if ***relation*** is empty
- What does the query 1 below find?

```
select distinct customer_name
from borrower B
where exists ( select *
                 from depositor D
                   where D.customer_name = B.customer_name );
```

- What does the query 2 below find?

```
select distinct customer_name
from depositor D
where exists ( select *
                 from account A
                   where A.account_number = D.account_number and
                       branch_name = 'Perryridge' );
```

Test for Empty Relation: exists, not exists

- Find the names of customers who **have a loan** with the bank, **but does not have an account.**
  - Express the query using **not exists**

```
select distinct customer_name
from borrower B
where not exists ( select *
                   from depositor D
                   where B.customer_name = D.customer_name );
```

Division Operation in SQL

- Given relation A contains relation B; B is a subset of A; ($B \subseteq A$)
- Standard SQL and most DBMS do NOT support the division operation
  - $A \div B \rightarrow$ In SQL, use not exists (B except A)
- Find all customers who have an account *at all the* branches located in Brooklyn
  - B: all branches located in Brooklyn
  - A: all branches

Division Operation in SQL

- Find all customers who have an account at all the branches located in Brooklyn

```
select distinct S.customer_name
from depositor S
where not exists (    ( select branch_name
                        from branch
                        where branch_city = 'Brooklyn'
                      )                                          } B
                      except
                      ( select R.branch_name
                        from depositor T, account R
                        where T.account_number = R.account_number and
                              S.customer_name = T.customer_name
                      )                                          } A
                  );
```

Division Operation in SQL

- Realize that we cannot express this query using = all, or any of its variants
  - There is no way a branch name will be equal to all the branch names!

SQL DML Extensions - 1

- SELECT ... ⬅ Extension to the select clause

  distinct, as, sum, count, min, max, avg, 'arithmetic ops'

- FROM ... ⬅ Extension to from clause

  as, inner join, left outer join, right outer join, full outer join,

  natural, on , using

- WHERE ... ⬅ Extension to where clause

  <, >, =, <>, >=, <=, and, or, not, like, is null, is not null,

  exists, in, any, some, all, unique, 'Sub queries'

GROUP BY ...

HAVING ...

ORDER BY ...

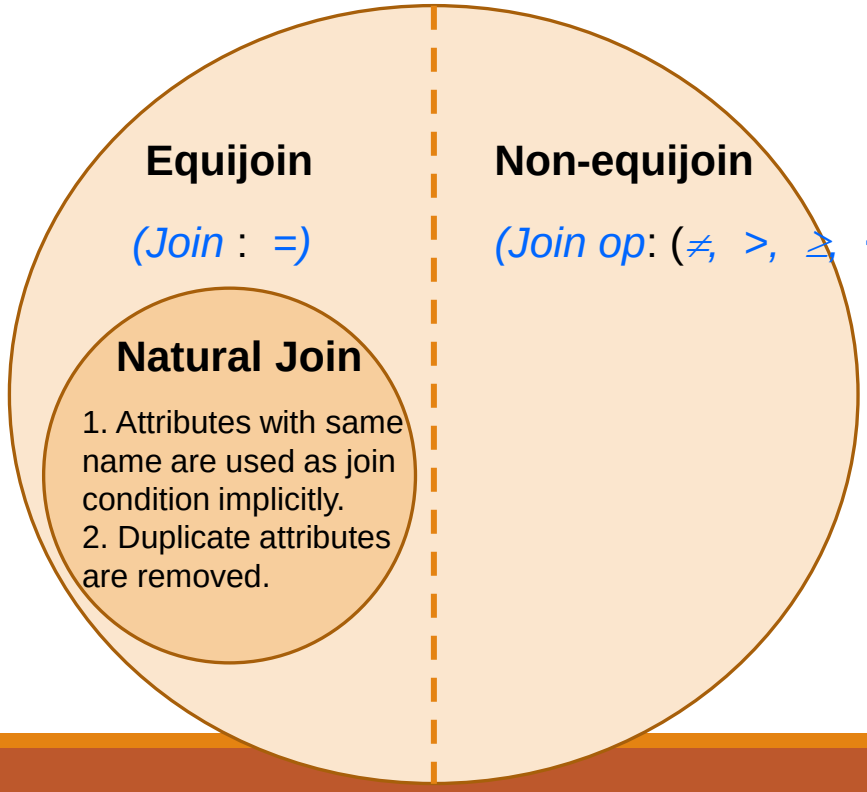
Types of Join

inner join

(Includes the only tuples that matches the join condition)

Theta Join

(Join op is one of: $=, \neq, >, \geq, <, \leq$*)*



outer join

(Includes the tuples that matches the join condition **AND …**)

Left outer Join
(include tuples from the left relation)

Right outer Join
(include tuples from the right relation)

Full outer Join
(include tuples from both left and right relation)

Note: Missing info. is filled with null.

Join Operators in SQL

- SQL provides various **join operations that can be expressed in where clause**
  - **cross join** (Cartesian product)
  - **inner Join** (Theta Join)
  - **natural join**
  - **left outer join**
  - **right outer join**
  - **full outer join**

These can be expressed using basic select... from.... where

These are new operators, but can be expressed in a complex query involving the union operator

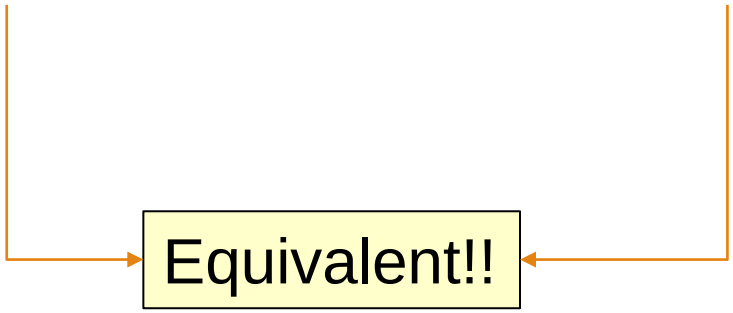
Cross Join

CROSS JOIN

```
select customer_name, balance
from depositor cross join account;
```

CARTESIAN PRODUCT

```
select customer_name, balance
from depositor, account;
```



Inner Join … On (theta join)

- SQL keyword **inner join** (or simply **join**) specifies inner join between two relations
- Join condition is specified by **on**


```
select customer_name, balance
from depositor inner join account on
        account.account_number = depositor.account_number;
```

```
select customer_name, balance
from depositor join account on
        account.account_number = depositor.account_number;
```

Equivalent to

```
select customer_name, balance
from depositor, account
where account.account_number = depositor.account_number;
```

Inner Join … On (theta join)

- The output relation from the inner join … on includes **all attributes** from both relations **including duplicate attributes** which are listed twice
- What's the schema of the output relation of

  account A join depositor D on D.account_number = A.account_number ?

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

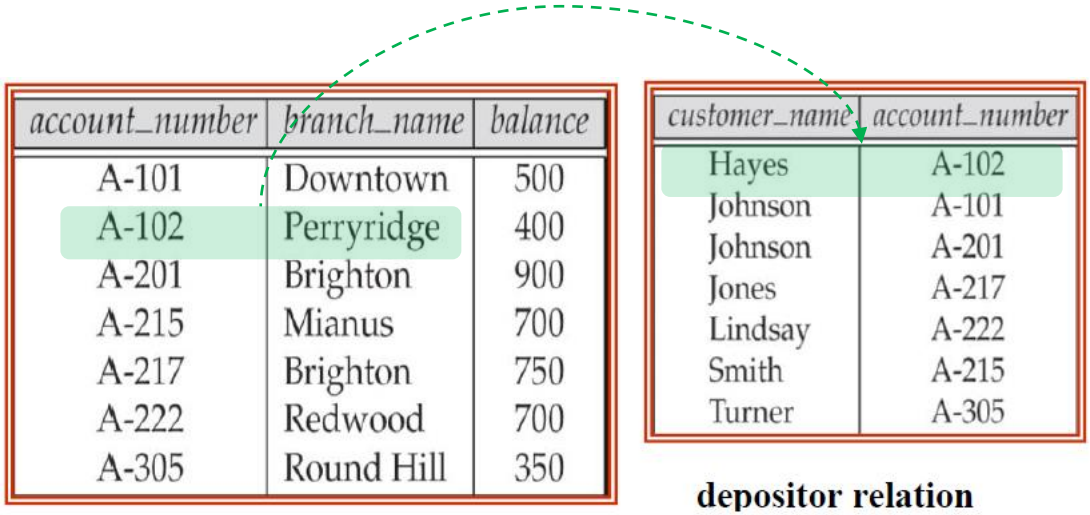
account relation

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

depositor relation

Inner Join … On – Example 1

- Find the names of all customers who have an account at the Perryridge branch



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

account relation

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

depositor relation

Inner Join … On – Example 1

- Find the names of all customers who have an account at the Perryridge branch

```
select distinct customer_name
from depositor D join account A on
        A.account_number = D.account_number
where branch_name = 'Perryridge';
```

```
select distinct customer_name
from depositor join account on
        account.account_number = depositor.account_number
        and branch_name = 'Perryridge';
```

```
select distinct customer_name
from depositor, account
where account.account_number = depositor.account_number
and branch_name = 'Perryridge';
```

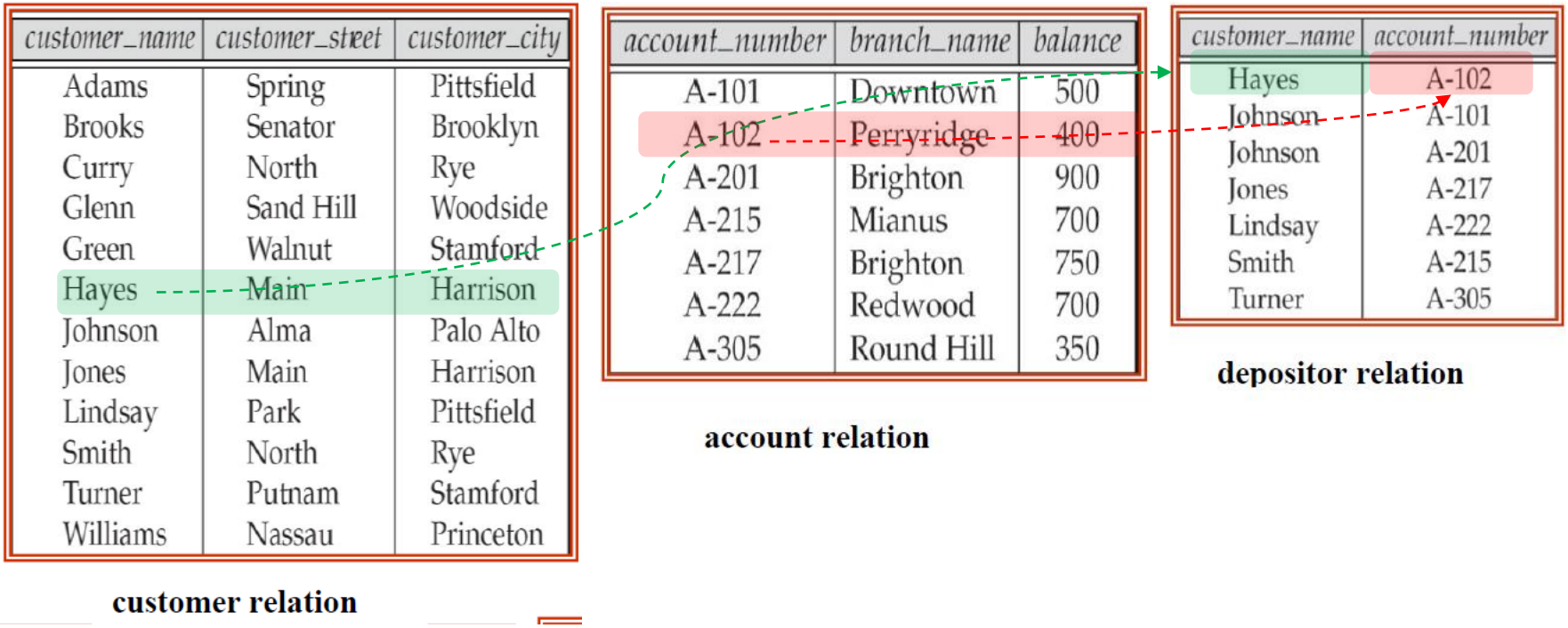
```
select distinct customer_name
from depositor D
where exists ( select *
                from account A
where A.account_number = D.account_number and branch_name = 'Perryridge');
```

Inner Join … On

- Joining more than two tables
  - Inputs of a join operation are tables
  - Output of a join operation is a table
  - Therefore, output of a join can be used as an input to another join

Inner Join … On – Example 2

- Find the names of customers who have an account in ‘Perryridge’ branch and who live in the city of Harrison.



customer_name	customer_street	customer_city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

customer relation

account_number	branch_name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

account relation

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

depositor relation

Inner Join … On – Example 2

- Find the names of customers who have an account in ‘Perryridge’ branch and who live in the city of Harrison.

```
select distinct C.customer_name
from depositor D join account A on A.account_number = D.account_number
                 join customer C on C.customer_name = D.customer_name
where branch_name = ‘Perryridge’ and customer_city = ‘Harrison’;
```

Inner Join … Using - 1

- **Equijoins based on the attributes with the same name** can be expressed using the **inner join … using** in SQL
  - The keyword **inner** **can be omitted**

```
select customer_name
from depositor join account  on
            account.account_number = depositor.account_number;
```

```
select customer_name
from depositor join account  using (account_number);
```

Attributes which are common to both relations

The schema of the output relation of **inner join ... using** eliminates the duplicate attributes.

Inner Join … Using - 2

- Inner join … using are most useful when
  - input joins **have multiple common attributes** and
  - we **want to create an equijoin based only on the values of the *only some of the common attributes***.

A	B	C	D
1	I	M	5
2	J	N	6
3	K	O	7
4	L	P	8

r1

B	C	D	E
I	M	2	I
U	N	6	J
K	O	7	N
L	T	9	T

r2

```
select *
from r1 join r2 using (B,C)
```

A	B	C	r1.D	r2.D	E
1	I	M	5	2	I
3	K	O	7	7	N

Natural Join

- Join two tables based on the **equality** of values of **all** the **common attributes** (simplified expression)
  - **natural inner join**
  - **natural join**

Equivalent;
Note that keyword inner can be omitted

```
select customer_name
from depositor natural join account;
```

The schema of the output relation of natural join **eliminates the duplicate attributes**.

```
select customer_name
from depositor join account  using (account_number);
```

```
select customer_name
from depositor join account  on
        account.account_number = depositor.account_number;
```

Outer Join - 1

- SQL supports the same set of outer join operations as in the relational algebra
  - **left outer join**
  - **right outer join**
  - **full outer join**

left join
right join
full join

Keyword outer can be omitted

Outer Join - 2

- As in inner joins, the outer join operators need to have one of the join condition specifier
  - natural
    - e.g., r1 natural left join r2
  - using
    - e.g., r1 right join r2 using(a1, a2)
  - on
    - e.g., r1 right join r2 on r1.a1 = r2.b2

Left Outer Join Example

Name	Age	Food
Jenny	33	Burger
Donna	22	Pizza
Roy	21	Steak
Sara	34	Pasta

member relation

Food	Day
Pizza	Monday
Burger	Tuesday
Salad	Wednesday
Pasta	Thursday
Tacos	Friday

menu relation

```
select *
from member natural left join menu;
```

```
select *
from member left join menu using (food);
```

Name	Age	Food	Day
Jenny	33	Burger	Tuesday
Donna	22	Pizza	Monday
Roy	21	Steak	null
Sara	34	Pasta	Thursday

Not Same

```
select *
from member left join menu on member.food = menu.food;
```

Right Outer Join Example

Name	Age	Food
Jenny	33	Burger
Donna	22	Pizza
Roy	21	Steak
Sara	34	Pasta

member relation

Food	Day
Pizza	Monday
Burger	Tuesday
Salad	Wednesday
Pasta	Thursday
Tacos	Friday

menu relation

```
select *
from member right join menu on member.food = menu.food;
```

Name	Age	member.Food	menu.Food	Day
Donna	22	Pizza	Pizza	Monday
Jenny	33	Burger	Burger	Tuesday
Sara	34	Pasta	Pasta	Thursday
null	null	null	Salad	Wednesday
null	null	null	Tacos	Friday

Full Outer Join Example

Name	Age	Food
Jenny	33	Burger
Donna	22	Pizza
Roy	21	Steak
Sara	34	Pasta

member relation

Food	Day
Pizza	Monday
Burger	Tuesday
Salad	Wednesday
Pasta	Thursday
Tacos	Friday

menu relation

```
select *
from member natural full join menu;
```

Name	Age	Food	Day
Donna	22	Burger	Tuesday
Jenny	33	Pizza	Monday
Roy	21	Steak	null
Sara	34	Pasta	Thursday
null	null	Salad	Wednesday
null	null	Tacos	Friday