Chapter 4

Structured Query Language(SQL)

SQL

- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Derived Relations
- Views
- Modification of the Database
- Joined Relations
- Data Definition Language
- Embedded SQL, ODBC and JDBC

SQL (also pronounced as Sequel)

- SQL is standard language for accessing and manipulating databases.
- The SQL language has several parts:

Data-Definition Language: commands for defining relation schemas, deleting and modifying relation schemas.

Data manipulation lanaguage: commands for insert, delete and modify rows (tuples).

View definition: commands for defining views

Transaction control: commans for specifying begin and end of transaction

Embedded SQL and dynamic SQL: embedding SQL in programming languages

Integrity:commands for specifiying integrity contraints

Authorization:commands for specifying access rights to tables and views

Basic Structure of SQL

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

  select $A_1, A_2, \ldots, A_n$

  from $r_1, r_2, \ldots, r_m$

  where P

  A_i *represents attributes*

  r_i *represents relations*

  P *is a predicate.*

- This query is equivalent to the relational algebra expression.

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

- The result of an SQL query is a relation.

The select Clause

- The select clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- Find the names of all branches in the *loan relation*

- In the "pure" relational algebra syntax,

$$\Pi_{\text{branch-name}}(\mathit{loan})$$

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

```
select *
from loan
```

The select Clause - Notes

NOTE 1: SQL does not permit the '-' character in names, so you would use, for example, *branch_name instead of branch-name in a real* implementation. We use '-' since it looks nicer!

NOTE 2: SQL names are case insensitive, meaning you can use upper case or lower case.

You may wish to use upper case in places where we use bold keywords or reserved words.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To elimination of duplicates, insert the keyword **distinct** after **select.**
- Find the names of all branches in the *loan relations, and remove* duplicates
- The keyword **all** specifies explicitly that duplicates not be removed.

The select Clause (Cont.)

The select clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.

select ***loan-number*, *branch-name*, *amount* ∗ *100***

from *loan*

Returns a relation which is the same as the *loan relations*, except that the attribute *amount is multiplied by 100*.

The where Clause

- The **where clause corresponds to the selection predicate of the** relational algebra. If consists of a predicate involving attributes of the relations that appear in the **from clause.**

- The find all loan number for loans made a the Perryridge branch with loan amounts greater than $1200.

- Comparison results can be combined using the logical connectives **and**, **or**, and **not**.
- Comparisons can be applied to results of arithmetic expressions.
- Comparision operators are <, <=, >, >=, =, <>

The where Clause contd..

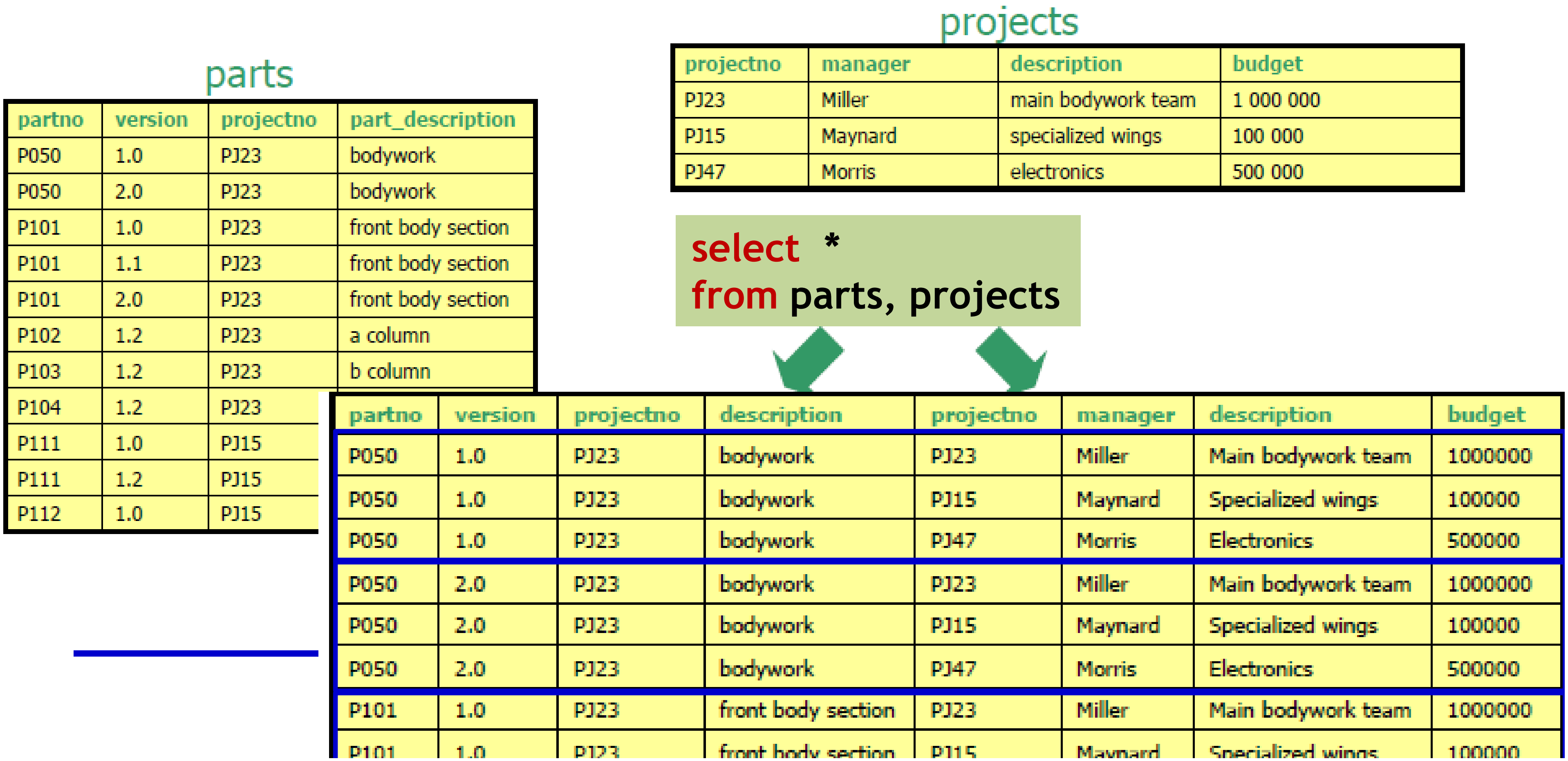
Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns

The where Clause contd..

- SQL Includes a **between** comparison operator in order to simplify where clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
- Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, ≥\$90,000 and ≤\$100,000)

The from Clause

- The **from clause corresponds to the Cartesian product operation of the** relational algebra. It lists the relations to be scanned in the evaluation of the expression.



projects

projectno	manager	description	budget
PJ23	Miller	main bodywork team	1 000 000
PJ15	Maynard	specialized wings	100 000
PJ47	Morris	electronics	500 000

parts

partno	version	projectno	part_description
P050	1.0	PJ23	bodywork
P050	2.0	PJ23	bodywork
P101	1.0	PJ23	front body section
P101	1.1	PJ23	front body section
P101	2.0	PJ23	front body section
P102	1.2	PJ23	a column
P103	1.2	PJ23	b column
P104	1.2	PJ23	
P111	1.0	PJ15	
P111	1.2	PJ15	
P112	1.0	PJ15	

select *
from parts, projects

partno	version	projectno	description	projectno	manager	description	budget
P050	1.0	PJ23	bodywork	PJ23	Miller	Main bodywork team	1000000
P050	1.0	PJ23	bodywork	PJ15	Maynard	Specialized wings	100000
P050	1.0	PJ23	bodywork	PJ47	Morris	Electronics	500000
P050	2.0	PJ23	bodywork	PJ23	Miller	Main bodywork team	1000000
P050	2.0	PJ23	bodywork	PJ15	Maynard	Specialized wings	100000
P050	2.0	PJ23	bodywork	PJ47	Morris	Electronics	500000
P101	1.0	PJ23	front body section	PJ23	Miller	Main bodywork team	1000000
P101	1.0	PJ23	front body section	PJ15	Maynard	Specialized wings	100000

The from Clause

- Find the Cartesian product *borrower x loan*

```
    select *
    from borrower, loan
```

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

```
select customer_name, borrower.loan_number, amount
from borrower, loan
where borrower.loan_number = loan.loan_number and
        branch_name = ‘Perryridge’
```

The Rename Operation

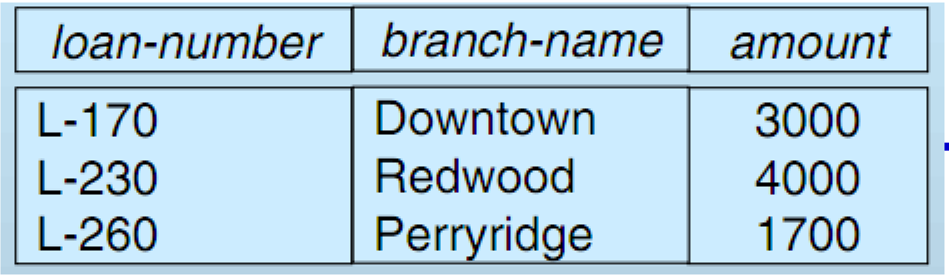
- The SQL allows renaming relations and attributes using the **as** clause:

  old_name **as** ***new_name***

- Find the name, loan number and loan amount of all customers; rename the column name *loan-number as loan_id.*

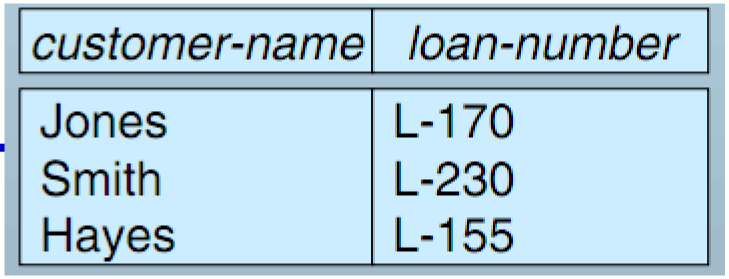
select ***customer_name***, ***borrower.loan_number*** **as** ***loan_id***, ***amount***
from ***borrower***, ***loan***
where ***borrower.loan_number*** = ***loan.loan_number***

Table *loan*



loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Table *customer*



customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Tuple Variables

- Tuple variables are defined in the **from clause** via the use of the **as** clause.

- Find the customer names and their loan numbers for all customers having a loan at some branch.

```
select customer-name, T.loan_number, S.amount
from borrower as T, loan as S
where T.loan-number = S.loan-number
```

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

  Given schema

  branch(branch_name, branch_city, assets)

```
select distinct T.branch-name
from branch as T, branch as S
where T.assets > S.assets and S.branch-city = 'Brooklyn'
```

String Operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:

  percent (%). The % character matches any substring.

  underscore (_). The _ character matches any character.

- Find the names of all customers whose street includes the substring "Main".

```
select customer_name
from customer
where customer_street like '%Main%'
```

- Match the name "Main%"
- **Use escape character '\' before string pattern indicating that string character is to be treated as normal character.**

  where like 'Main\%'

SQL supports a variety of string operations such as concatenation (using "||"), converting from upper to lower case (and vice versa), finding string length, extracting substrings, etc.

String operation practices

table *position*

ID	PositionName
1	Chairman
2	Vice Chairman
3	Director
4	Company Secretary
5	General Manager
6	Assistant General Manager
7	Senior Manager
8	Acting Senior Manager
9	Branch Manager
10	ab\cd
12	ab%cd

```
select  PositionName
from postion
```

WHERE CLAUSES

```
where positionname like ‘_ _ _’
where positionname like ‘_ _ _%’
where positionname like ‘Chair%’
where positionname like ‘%General%’
where positionname like ‘ab\\cd’
where positionname like ‘ab\%cd’
```

Ordering the Display of Tuples

List in alphabetic order the names of all customers having a loan in Perryridge branch

```
select distinct customer-name
from borrower, loan
where borrower loan_number = loan.loan_number and
branch-name = ‘Perryridge’
order by customer_name
```

We may specify **desc** for descending order **or** **asc** for ascending order, for each attribute; ascending order is the default.

E.g. order by customer-name desc

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.

- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.

Set Operation (without duplication)

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

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

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

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

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

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

Set Operation (retaining duplication)

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

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

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

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

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

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

Aggregate Functions

- SQL aggregate functions returns a single value, calculated from values in a column.

  avg: **returns** average value

  min: **returns** minimum value

  max: **returns** maximum value

  sum: **returns** sum of values

  count: **returns** number of rows or values

Aggregate Functions contd..

- Find the average account balance at the Perryridge branch.

```
select avg(balance)
from account
where branch_name = ‘Perryridge’
```

- Find the number of tuples in the *customer relation*.

```
select count (*)
from customer
```

- Find the number of distinct depositors in the bank.

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

Aggregate Functions contd..

- Find the number of depositors for each branch.

```
select branch_name, count (distinct customer_name)
from depositor, account
where depositor.account_number = account.account_number
group by branch-name
```

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

GROUP BY Clause (1 of 2)

- The GROUP BY clause allows to apply aggregate functions to groups of rows.
- **Note**: Columns in the SELECT clause outside of aggregate functions must appear in the GROUP BY list.
- **Example**:
  How many different versions do exist for each part?

```
SELECT      partno, COUNT(*) AS number_of_versions
FROM        parts
GROUP BY    partno
```

GROUP BY clause (2 of 2)

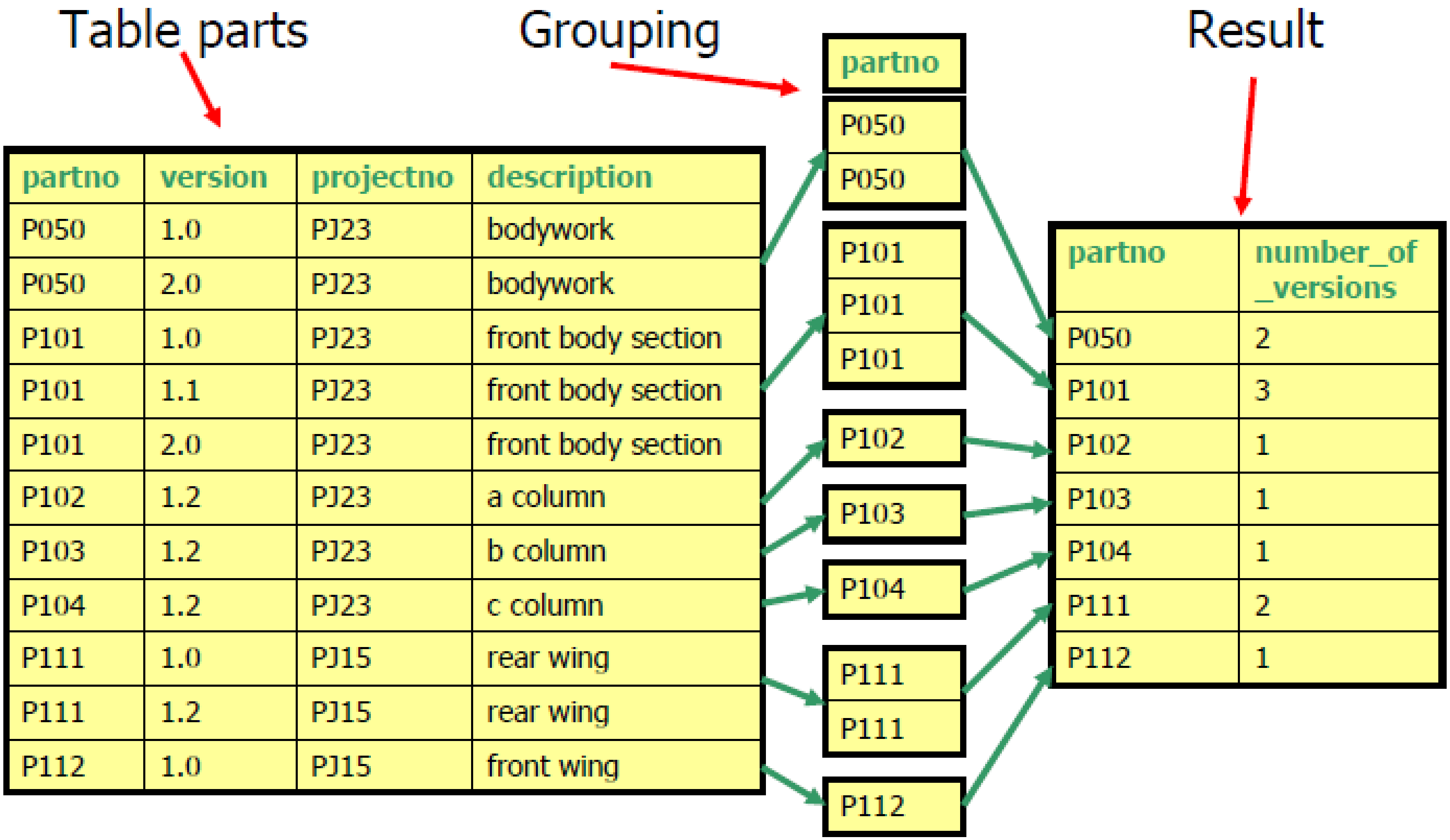


Table parts

partno	version	projectno	description
P050	1.0	PJ23	bodywork
P050	2.0	PJ23	bodywork
P101	1.0	PJ23	front body section
P101	1.1	PJ23	front body section
P101	2.0	PJ23	front body section
P102	1.2	PJ23	a column
P103	1.2	PJ23	b column
P104	1.2	PJ23	c column
P111	1.0	PJ15	rear wing
P111	1.2	PJ15	rear wing
P112	1.0	PJ15	front wing

Grouping

partno
P050 P050
P101 P101 P101
P102
P103
P104
P111 P111
P112

Result

partno	number_of _versions
P050	2
P101	3
P102	1
P103	1
P104	1
P111	2
P112	1

Aggregate Functions – having clause

Find the names of all branches where the average account balance is more than $1,200.

```
select branch_name, avg (balance)
from account
group by branch_name
having avg (balance) > 1200
```

Note: predicates in the **having clause are applied after the** formation of groups whereas predicates in the **where clause are** applied before forming groups

Null values

- SQL allows the use of null values to indicate absence of information about the value of an attribute.

List all loan numbers having loan amount

```
select loan-number
from loan
where amount is not null
```

List all loan numbers having no loan amount

```
select loan-number
from loan
where amount is null
```

Predicate is null tests for the presence of null values

Predicate is not null tests for the abscence of null values

Null values in operation

- *null signifies an 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 , null <> null , null = null

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

Null and Three-valued logic

- Three-valued logic using the truth value *unknown:*
- OR operation

  unknown or true= true

  unknown or false = unknown

  unknown or unknown = unknown
- AND operation

  true and unknown = unknown

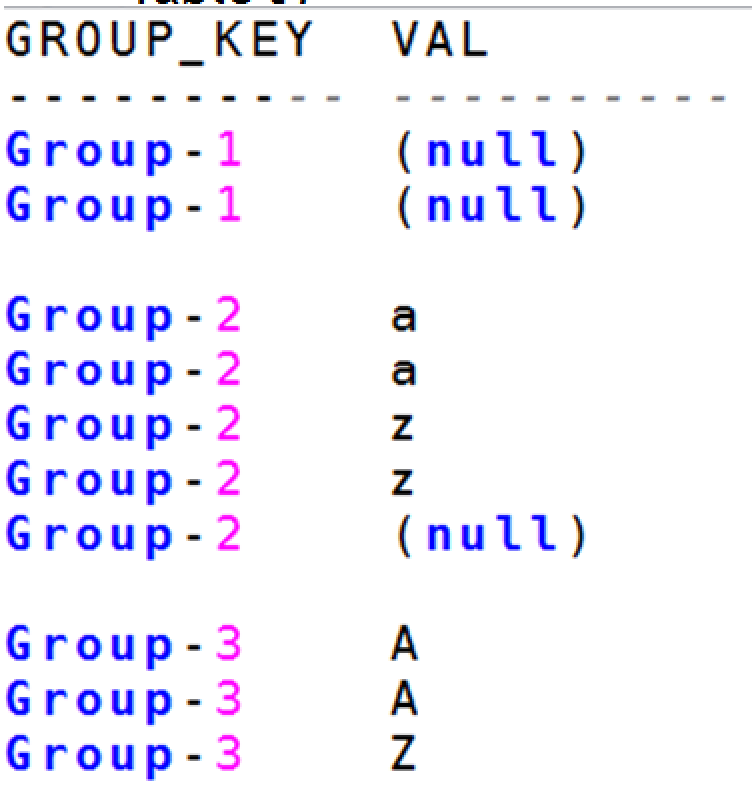
  false and unknown=false

  unknown and unknown = unknown
- NOT:

  not (unknown) = unknown
- Except for count(*), ignore null values in their input collection.

A test

Table *t1*

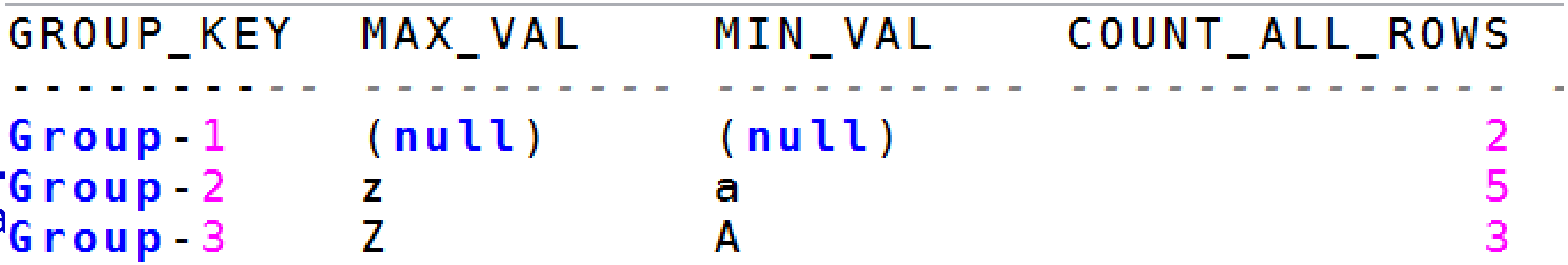


```
GROUP_KEY    VAL
------------ ----------
Group-1      (null)
Group-1      (null)

Group-2      a
Group-2      a
Group-2      z
Group-2      z
Group-2      (null)

Group-3      A
Group-3      A
Group-3      Z
```

Query:

```
select group_key ,
MAX( VAL ) as max_val ,
MIN( VAL ) as min_val,
COUNT( * ) as count_all_rows
from t1
group by group_key
 order by group_key ;
```

Result of query



```
GROUP_KEY    MAX_VAL      MIN_VAL      COUNT_ALL_ROWS
------------ ------------ ------------ ----------------
Group-1      (null)       (null)                      2
Group-2      z            a                           5
Group-3      Z            A                           3
```

Nested Queries (1)

- 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, and set cardinality.

Nested Queries (2)

Table *Persons*

PNo	LastName	FirstName
1	Shrestha	Hari
2	Maharjan	Ram
3	Shakya	Krishna

Table *PersonTelephone*

PNo	Telephone
1	01223431
1	01223433
3	01554200

Comparison operator (=, >=, <=, >, <, like, between) operator does not allows you to specify multiple values in a WHERE clause.

```
SELECT  *
FROM Persons
WHERE LastName = (SELECT LastName FROM Persons WHERE
Person_No=1)
```

IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT  *
FROM Persons
WHERE LastName IN ('Hansen','Pettersen')
```

Nested Queries (3)

- 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)
```

- Find all customers who have a loan at the 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)
```

Nested Queries....Set Comparison 1

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

```
select distinct T.branch-name
from branch as T, branch as S
where T.assets > S.assets and S.branch-city = ‘Brooklyn’
```

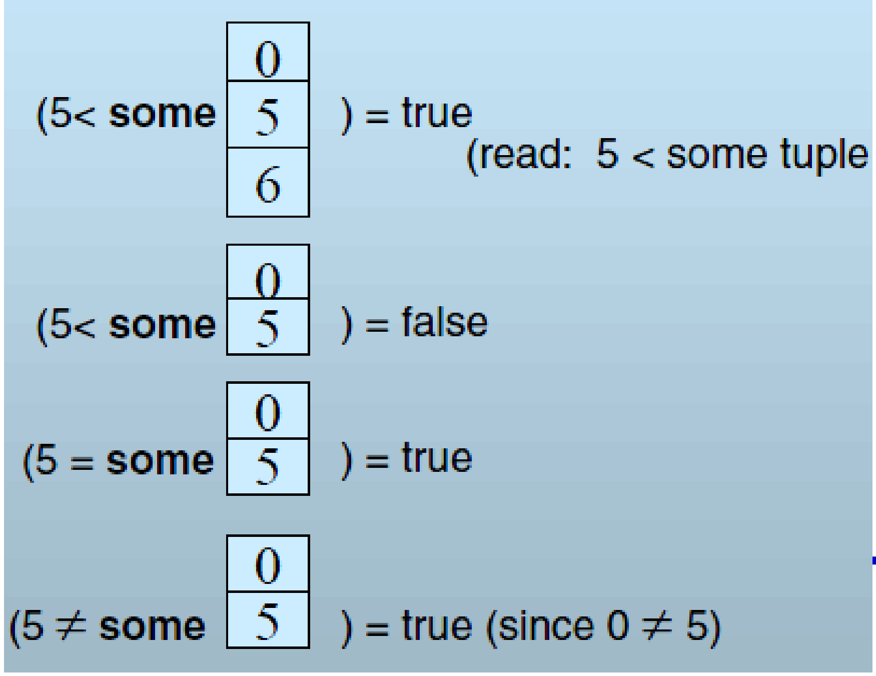
- Same query using > some clause

```
select branch_name
from branch
where assets > some
                        (select assets
                        from branch
                        where branch-city = ‘Brooklyn’)
```

Nested Queries....Set Comparison 2

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

```
select branch_name
from branch
where assets > all (select assets
                    from branch
                    where branch-city = ‘Brooklyn’)
```



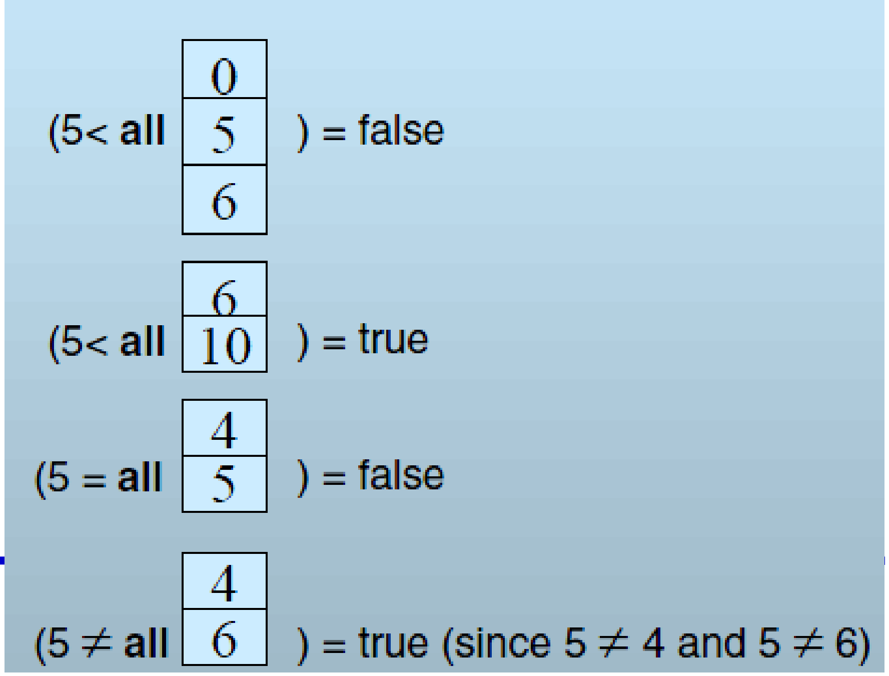
$(5 < \textbf{some} \begin{array}{|c|}\hline 0\\ \hline 5\\ \hline 6\\ \hline\end{array}) = \text{true}$

(read: 5 < some tuple

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

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

$(5 \neq \textbf{some} \begin{array}{|c|}\hline 0\\ \hline 5\\ \hline\end{array}) = \text{true (since } 0 \neq 5\text{)}$



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

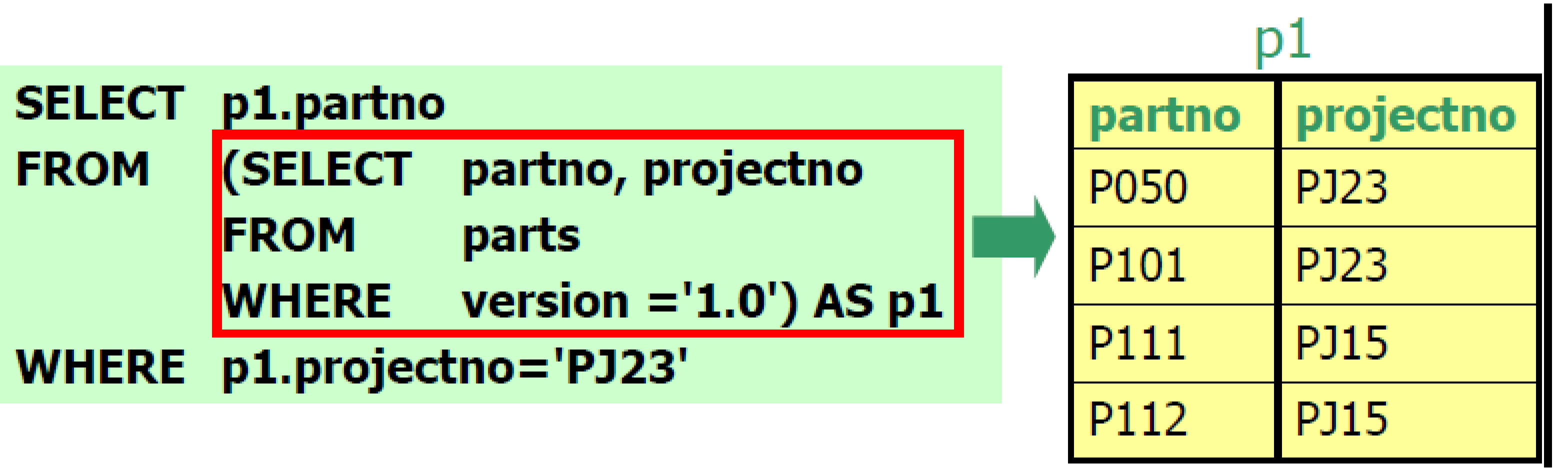
$(5 < \textbf{all} \begin{array}{|c|}\hline 6\\ \hline 10\\ \hline\end{array}) = \text{true}$

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

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

Derived Tables

- Several types of tables may be provided in a FROM clause:
  - base tables
  - views
  - tables that are calculated by nested select statements
- Which parts of version 1.0 are designed by project PJ23?



```
SELECT  p1.partno
FROM    (SELECT   partno, projectno
         FROM     parts
         WHERE    version ='1.0') AS p1
WHERE   p1.projectno='PJ23'
```

p1

partno	projectno
P050	PJ23
P101	PJ23
P111	PJ15
P112	PJ15

Derived Tables

Find the average account balance of those branches where the average account balance is greater than $1200.

```
select branch-name, avg-balance
from (select branch-name, avg (balance)
          from account
          group by branch-name)
          as result (branch-name, avg-balance)
where avg-balance > 1200
```

Note that we do not need to use the having clause, since we compute the temporary relation result in the from clause.

Modification of the Database: *Deletion (1)*

- SQL expression for deletion

  delete from r

  where p

- The DELETE statement allows to remove rows from a table.
- The WHERE clause may be used to specify the rows to be deleted.

- Delete all account

  delete from account

Modification of the Database – *Deletion (2)*

- Delete all account records at the Perryridge branch

- Delete all accounts at every branch located in Needham city.

Modification of the Database – *Deletion (2)*

- Delete all account records at the Perryridge branch

```
delete from account
where branch_name = ‘Perryridge’
```

- Delete all accounts at every branch located in Needham city.

```
delete from account
where branch_name in (select branch_name
                      from branch
                      where branch_city = ‘Needham’)
```

Modification of the Database – *Deletion (3)*

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

```
delete from account
where balance < (select avg (balance)
                 from account)
```

Modification of the Database – *Insertion (1)*

The INSERT statement allows two different ways to provide the rows that should be inserted into a table:

- list of values
- table value constructor, i.e., the result of a query

Tuples to be inserted must be

i. Attribute values must be member of attribute's domain

ii. Must be of the correct arity.

- SQL expression for Insertion

```
Insert into r (column1, column2, ...)
   values (value1, value2, ...)
```

Modification of the Database – *Insertion (2)*

- 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')
```

Modification of the Database – *Insertion (3)*

- Add a new tuple to *account with balance set to null*

```
insert into account
values ('A-777','Perryridge', null)
```

- 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

```
insert into account
        select loan-number, branch-name, 200
        from loan
        where branch-name = 'Perryridge'
```

Modification of the Database – *Updation(1)*

The UPDATE statement allows to change some values of a row without changing all the values in a row

- Use the SET clause to assign the new values
- Use the WHERE clause to specify the rows that should be updated

SQL expression for Insertion

```
update r
set  column_1 = value1, column_2 = value2
where p
```

Modification of the Database – *Updation(2)*

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

- The order is important
- Can be done better using the **case statement**

Modification of the Database – *Updation(2)*

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

```
update account
set balance = balance * 1.06
where balance > 10000

update account
set balance = balance * 1.05
where balance ≤ 10000
```

- The order is important
- Can be done better using the **case statement**

Modification of the Database – *Updation(2)*

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

```
update account
set balance = case
                  when balance <= 10000 then balance *1.05
                  else balance * 1.06
              end
```

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 as <query expression>*

Where

<query expression> is any legal expression.

The view name is represented by *v.*

- A view is a named virtual table (query) that is computed from one ore more underlying tables (base tables or even views).

View Examples – *Creating view*

A view consisting of branches and their customers

```
create 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)
```

View Examples – *Using view in select*

- Now find all customers of the Perryridge branch by using views.

```
select customer_name
from all_customer
where branch_name = ‘Perryridge’
```

Views advantages and properties

- Advantages
  - More user friendly
  - Higher degree of data independence
- Properties of views
  - A view can be handled like a table
  - Views on views are possible
  - Limited updates: updatable and non-updatable views

With Clause

- with clause allows temporary views
- 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
```

Updatable Views

Informal rule: For a view to be updatable, the DBMS must be able to trace any row or column back to its row or column in the source table

Definition by ISO standard: A view is updatable if and only if:

- ✓ DISTINCT is not specified
- ✓ Every element in the SELECT list of the defining query is a column name (rather than a constant, expression, or aggregate function) and no column name appears more than once.
- ✓ The FROM clause specifies only one table: i.e., single source for the view, no JOIN, UNION, INTERSECT, or EXCEPT
- ✓ The WHERE clause does not include any nested SELECTs that references the table in the FROM clause
- ✓ There is no GROUP BY or HAVING clause in the defining query

Test for Empty relations

- SQL includes a feature for testing whether a subquery has any tuples in the result.
- The exists construct returns the value true if the argument subquery is nonempty.
- E.g. Find all customers who have both an account and loan at the bank.

```
select customer_name
from borrower
where exists (select * from depositor
        where depositor.customer_name=borrower.customer_name)
```

Test for Empty relations

- E.g. Find all customers who have only loan at the bank.

```
select customer_name
from borrower
where not exists (select * from depositor
        where depositor.customer_name=borrower.customer_name)
```

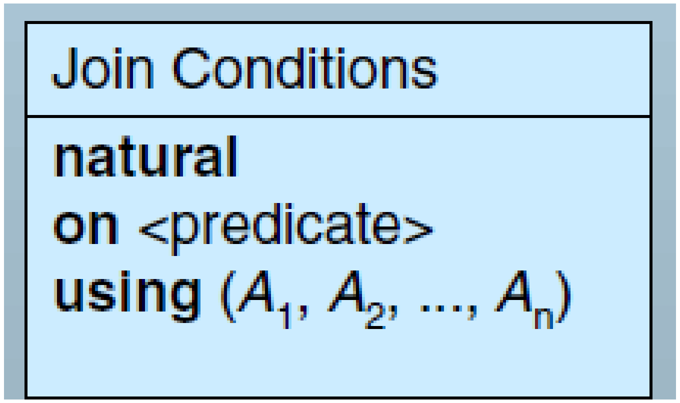
Scoping rule

Scoping rules are analogous to the scoping rule of varibles in programming language.

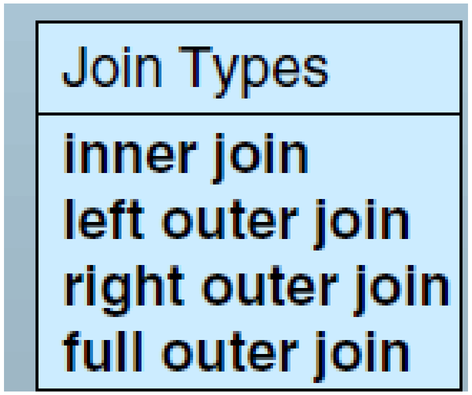
```
select customer_name
from borrower as B
where exists (select * from depositor as D
        where D.customer_name=B.customer_name)
```

Joins

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from clause**
- 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.



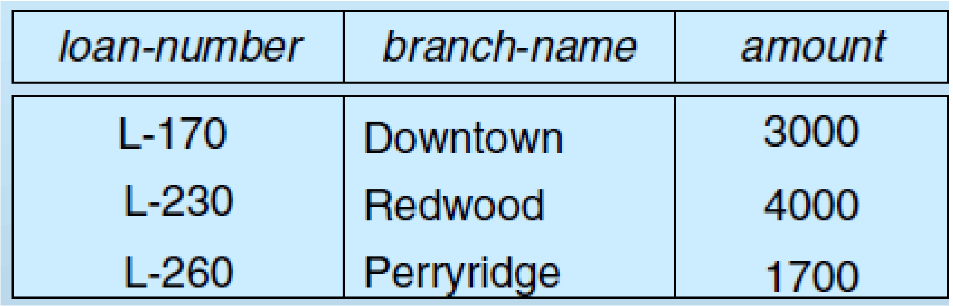
Join Conditions
natural **on** <predicate> **using** (A_1, A_2, ..., A_n)



Join Types
inner join **left outer join** **right outer join** **full outer join**

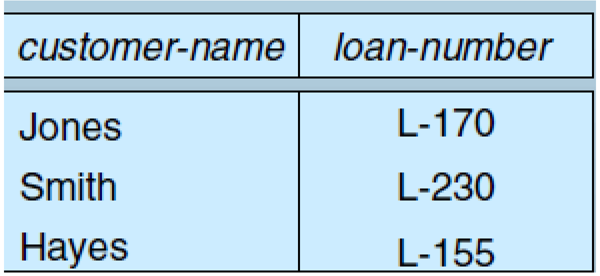
Joined Relations – tables for Examples

- Relation *loan*



loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

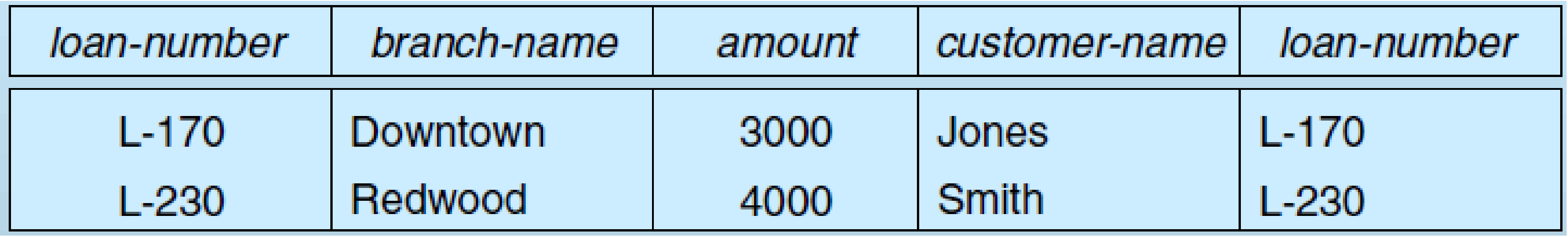


customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

- Note: borrower information missing for L-260 and loan information missing for L-155

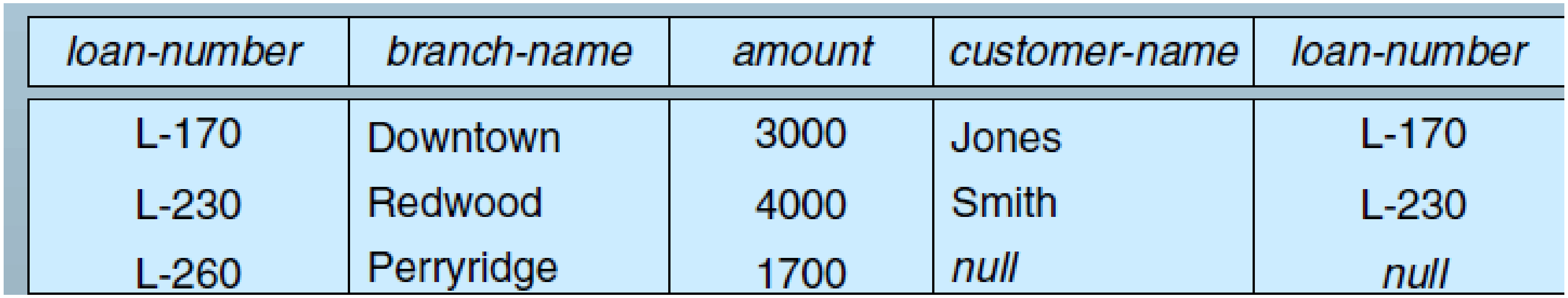
Inner Joins

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

loan left outer join borrower on
loan.loan-number = borrower.loan-number



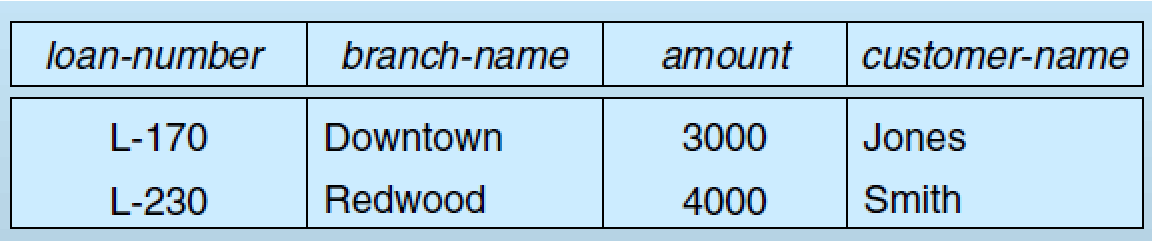
loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Join condition and keywords

- Use of a join condition
  - mandatory for outer joins.
  - Optional for inner joins (if omitted, a result is cartesian product)
- Inner and outer keywords are also optional because it can be still distinguished with rest of the join types.

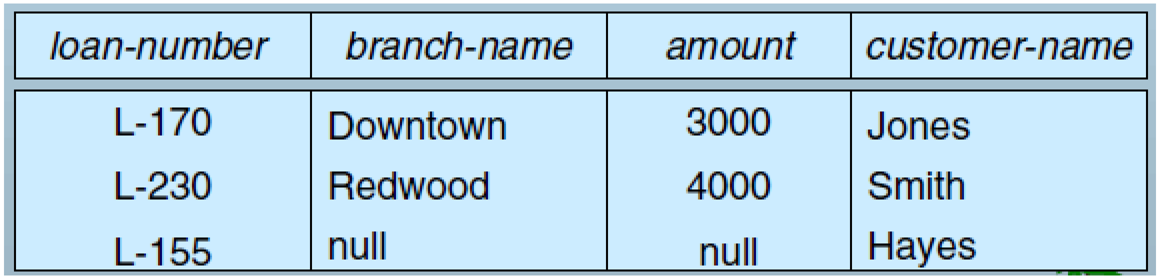
Natural join examples

- *loan* ***natural inner join*** ***borrower***



loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- 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

Result of Natural Join

- The ordering of the attributes in the result of a natural join is
  - The join attributes that is the attributes common to both tables appear first, in the order in which they appear in the left-hand-side table.
  - Next come all the non-join attributes of the left hand side table
  - Finally, all non-join attributes the right hand side table.

Full outer join

- It is a combination of the left and right outer join types.
  - The operation computes the result of the inner join
  - On the tuples of left hand side relation, it adds null values that did not match any from the right-hand-side and adds them to result.
  - On the tuples of right hand side relation, it adds null values that did not match any from the left hand side and adds them to the result.

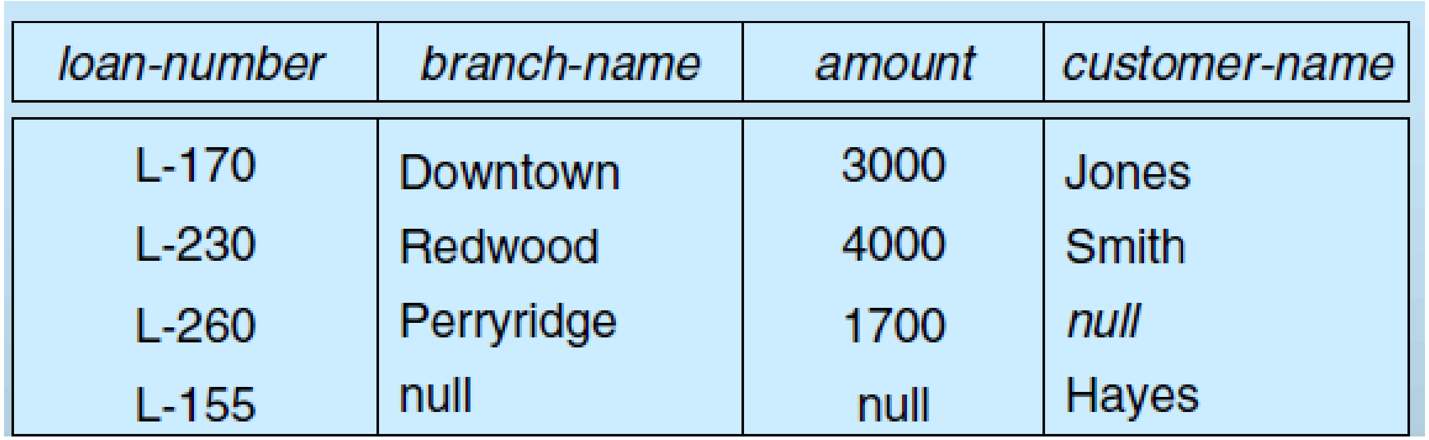
The *using* condition

- The join condition using (A_1, A_2, ...A_n) is similar to the natural join condition except that the join attributes are the attributes A_1, A_2, ...A_n rather than all attributes that are common to both relations.

- The attributes A_1, A_2, ...A_n must consits of only attributes that are common to both relations.

Full outer join examples

- *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

Examples

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

```
select customer_name
from (depositor natural full outer join borrower)
where account_number is null or loan_number is null
```

- find all customers who have an account but no loan

```
select d_CN
from (depositor left outer join borrower
  on
  depositor.customer_name=borrower.customer_name)
   as db1(d_CN, account_number, b_CN, loan_number)
where loan_number is null
```

Data Definition Language (DDL)

- The data definitoin lanauge is a means by which the set of relations in a database are specified.
- This also allows to specifiy information about each relations
  - The schema for each relation
  - The domain of values associated with each attribute
  - The integrity constraints
  - The set of indices to be maintained for each relation
  - The security and authorization information for each relation
  - The physical storage structure of each relation on disk

Domain types in SQL

char(n). Fixed length character string, with user-specified length *n*.

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 *n digits to the right of decimal point*.

example: numeric(3,1) allows 44.5, 2.3 to be stored but not ~~444.5~~ or ~~0.32~~

real, double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.

float(n). Floating point number, with user-specified precision of at least *n* digits.

Date/Time Types in SQL (Cont.)

Date: A calendar date, containing a (4 digit) year, month and date E.g. date ‘2001-7-27’

time: The time of day, in hours, minutes and seconds.

E.g. time ’09:00:30’ time ’09:00:30.75’

timestamp: date plus time of day

E.g. timestamp ‘2001-7-27 09:00:30.75’

Interval: period of time

E.g. Interval ‘1’ day

- Subtracting a date/time/timestamp value from another gives an interval value
- Interval values can be added to date/time/timestamp values

Extract values

- Can extract values of individual fields from date/time/timestamp
- E.g. extract (year from r.starttime)
- Can cast string types to date/time/timestamp
- E.g. cast <string-valued-expression> as date

Create Table Construct

- 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>,
...,
<integrity-constraintk>)

- *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*

Example:

create table *branch*
(*branch_name* char(15) not null,
Branch_city char(30),
assets integer)

Integrity Constraints in Create Table

- primary key (A_1, A_2 ..., A_n):

  The primary key specification says that attributes A_1, A_2, .., A_n form the primary key.

  The Primary key attributes are required to be non null and unique, that is no tuple can have a null value for a primary attribute and no two tuples in the relation can be equal on the primary-key attributes.

  Primary key specification is optional, it is generally a good idea to specifiy a primary key for each relation.

  If a newly inserted or modified tuple in relation has null values for any primary key or if the tuple has the same value on the primary-key attributes as does another tuple in the relation, SQL flags an error and prevent from any change.

- check (P): The check clause specifies a predicate P that must be satisfied by every tuple in the relation.

Integrity Constraints in Create Table

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

```
create table branch
(branch_name char(15),
branch_city char(30),
assets integer,
primary key (branch_name),
check (assets >= 0))
```

- primary key declaration on an attribute automatically ensures not null in SQL-92 onwards.

Null and unique

- By default null is a legal value for every attribute in SQL except for those attribute which are specifically stated to be not null.

  account_number char(10) not null

- SQL also supports an integrity constraint

  unique (A_1, A_2, … A_n)

  Which specifies that the attributes A_1, A_2, … A_n form a

  - candidate key that is no two tuples in the relation can be equal.
  - But, candidate key attributes are permitted to be null unless they have explicitly been declared with not null.

Example

```
create table student
 (name char(15) not null,
 student_id char(10),
 degree_level  char(15),
 primary key(student_id),
 check(degree_level in (‘Bachelors’, ‘Masters’, ‘Degree’)))
```

Drop Table Constructs

- The **drop table** command deletes all information about the dropped/removed relation from the database.

drop table *r* — *the statement removes relation from the database*

delete table *r* — *the statement removes all the tuples in r, it retains the table r.*

Alter Table Constructs

- The **alter table** command is used to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the **alter table** command is

  alter table *r* add *A D*

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

- 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 of relation r to be deleted

  Dropping of attributes not supported by many databases

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, and Cobol.
- A language to which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise embedded SQL.
- EXEC SQL statement is used to identify embedded SQL request to the preprocessor
- EXEC SQL <embedded SQL statement > END-EXEC
- Note: this varies by language. E.g. the Java embedding uses

  # SQL { …. }

Example Query

From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.

Specify the query in SQL and declare a *cursor for it*

```
EXEC SQL
     declare c cursor for
    select customer_name, customer_city
    from depositor, customer, account
    where depositor.customer_name = customer.customer_name
       and depositor account_number = account.account_number
       and account.balance >:amount
END-EXEC
```

Embedded SQL (Cont.)

- The open statement causes the query to be evaluated

```
EXEC SQL open c END-EXEC
```

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

```
EXEC SQL fetch c into :cn, :cc END-EXEC
```

- Repeated calls to fetch get successive tuples in the query result
- The close statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL close c END-EXEC
```

Dynamic Queries

- Dynamic Queries allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```
        char * sqlprog = “update account
                     set balance = balance * 1.05
                     where account_number = ?”

EXEC SQL prepare dynprog from :sqlprog;
        char account [10] = “A-101”;
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

Open DataBase Connectivity(ODBC) standard

- Open DataBase Connectivity(ODBC) standard

  standard for application program to communicate with a database server.
- application program interface (API) to
  - ✔ open a connection with a database,
  - ✔ send queries and updates,
  - ✔ get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC