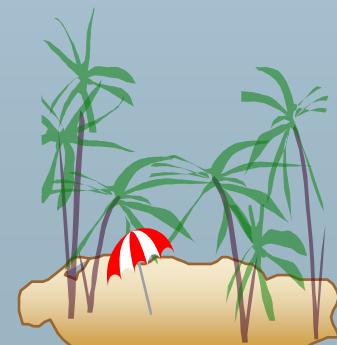




# Chapter 3: Relational Model

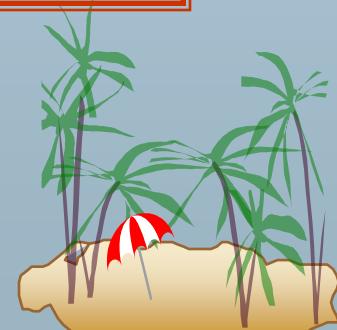
- Structure of Relational Databases
- Relational Algebra
- Tuple Relational Calculus
- Domain Relational Calculus
- Extended Relational-Algebra-Operations
- Modification of the Database
- Views





# Example of a Relation

<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
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





# Basic Structure

- Formally, given sets  $D_1, D_2, \dots, D_n$  a **relation**  $r$  is a subset of  $D_1 \times D_2 \times \dots \times D_n$   
Thus a relation is a set of n-tuples  $(a_1, a_2, \dots, a_n)$  where each  $a_i \in D_i$
- Example: if

*customer-name* = {Jones, Smith, Curry, Lindsay}

*customer-street* = {Main, North, Park}

*customer-city* = {Harrison, Rye, Pittsfield}

Then  $r = \{ (Jones, Main, Harrison),$   
 $\quad (Smith, North, Rye),$   
 $\quad (Curry, North, Rye),$   
 $\quad (Lindsay, Park, Pittsfield) \}$

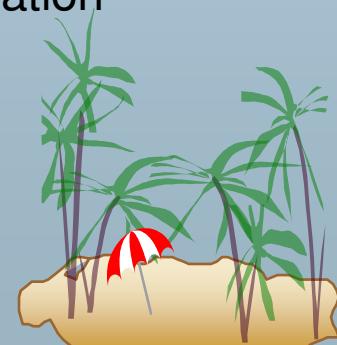
is a relation over *customer-name*  $\times$  *customer-street*  $\times$  *customer-city*





# Attribute Types

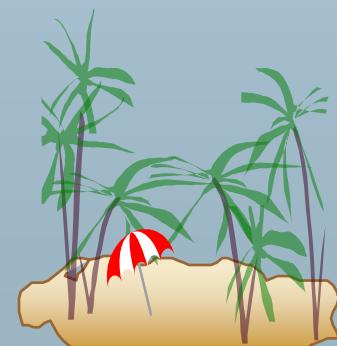
- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**, that is, indivisible
  - ☞ E.g. multivalued attribute values are not atomic
  - ☞ E.g. composite attribute values are not atomic
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations
  - ☞ we shall ignore the effect of null values in our main presentation and consider their effect later





# Relation Schema

- $A_1, A_2, \dots, A_n$  are *attributes*
- $R = (A_1, A_2, \dots, A_n)$  is a *relation schema*
  - E.g. *Customer-schema* =  
 $(\text{customer-name}, \text{customer-street}, \text{customer-city})$
- $r(R)$  is a *relation* on the *relation schema*  $R$ 
  - E.g. *customer* (*Customer-schema*)





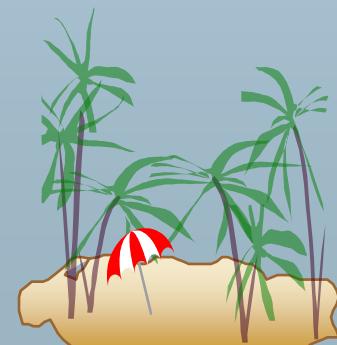
# Relation Instance

- The current values (*relation instance*) of a relation are specified by a table
- An element  $t$  of  $r$  is a *tuple*, represented by a *row* in a table

<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
<i>Jones</i>	Main	Harrison
<i>Smith</i>	North	Rye
<i>Curry</i>	North	Rye
<i>Lindsay</i>	Park	Pittsfield

*customer*

Diagram illustrating a relation instance named *customer*. The table has three attributes: *customer-name*, *customer-street*, and *customer-city*. It contains four tuples (rows). Arrows point from the column headers to the attribute label "attributes (or columns)" and from the row labels to the attribute label "tuples (or rows)".

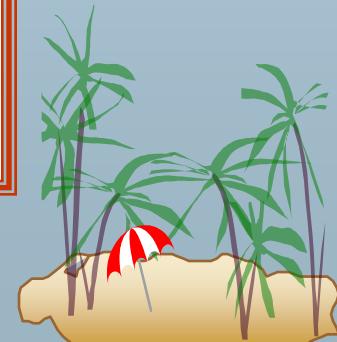




# Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- E.g. *account* relation with unordered tuples

<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750





# Database

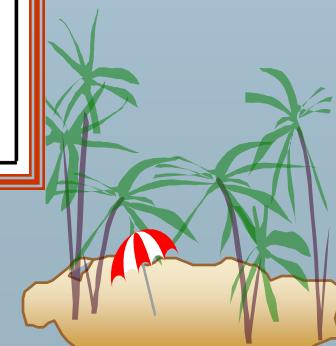
- A database consists of multiple relations
- Information about an enterprise is broken up into parts, with each relation storing one part of the information
  - E.g.: *account*: stores information about accounts
  - depositor*: stores information about which customer owns which account
  - customer*: stores information about customers
- Storing all information as a single relation such as  
*bank(account-number, balance, customer-name, ...)*  
results in
  - ☞ repetition of information (e.g. two customers own an account)
  - ☞ the need for null values (e.g. represent a customer without an account)
- Normalization theory (Chapter 7) deals with how to design relational schemas





# The *customer* Relation

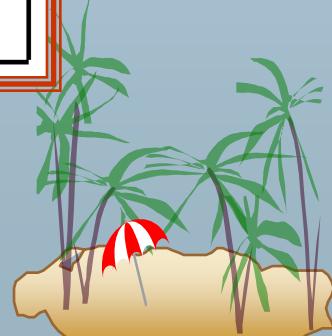
<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton



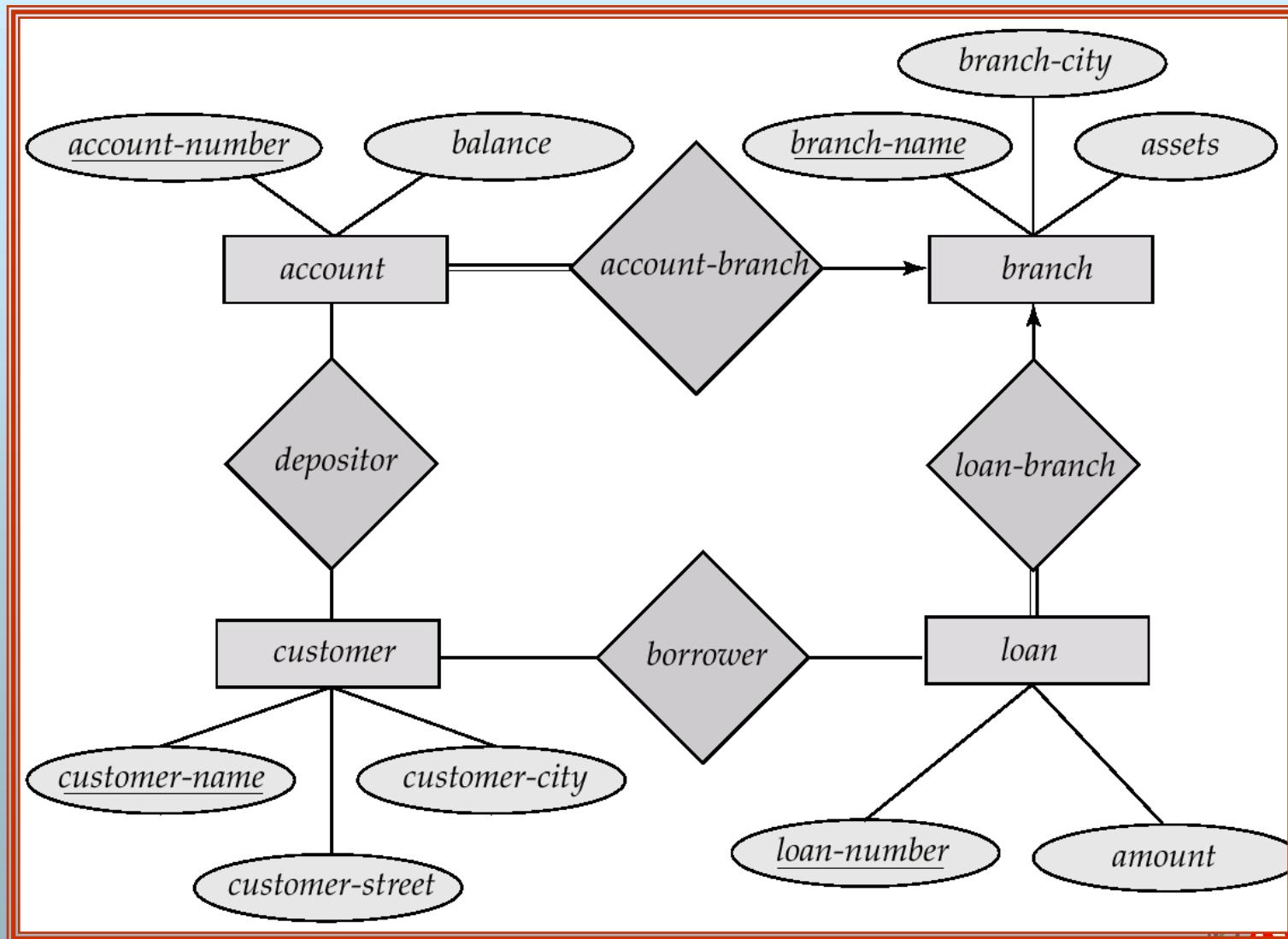


# The *depositor* Relation

<i>customer-name</i>	<i>account-number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305



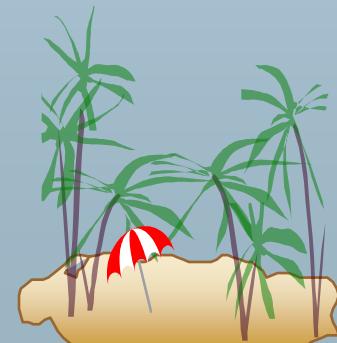
# E-R Diagram for the Banking Enterprise





# Keys

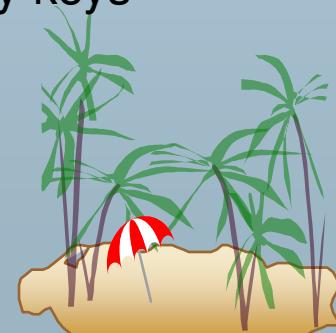
- Let  $K \subseteq R$
- $K$  is a **superkey** of  $R$  if values for  $K$  are sufficient to identify a unique tuple of each possible relation  $r(R)$ 
  - ☞ by “possible  $r$ ” we mean a relation  $r$  that could exist in the enterprise we are modeling.
  - ☞ Example:  $\{customer-name, customer-street\}$  and  $\{customer-name\}$  are both superkeys of *Customer*, if no two customers can possibly have the same name.
- $K$  is a **candidate key** if  $K$  is minimal  
Example:  $\{customer-name\}$  is a candidate key for *Customer*, since it is a superkey (assuming no two customers can possibly have the same name), and no subset of it is a superkey.



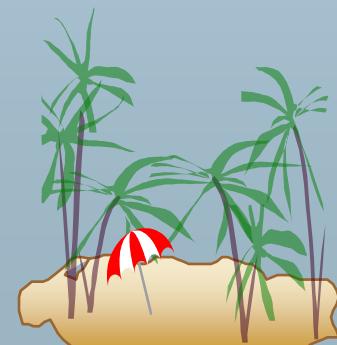
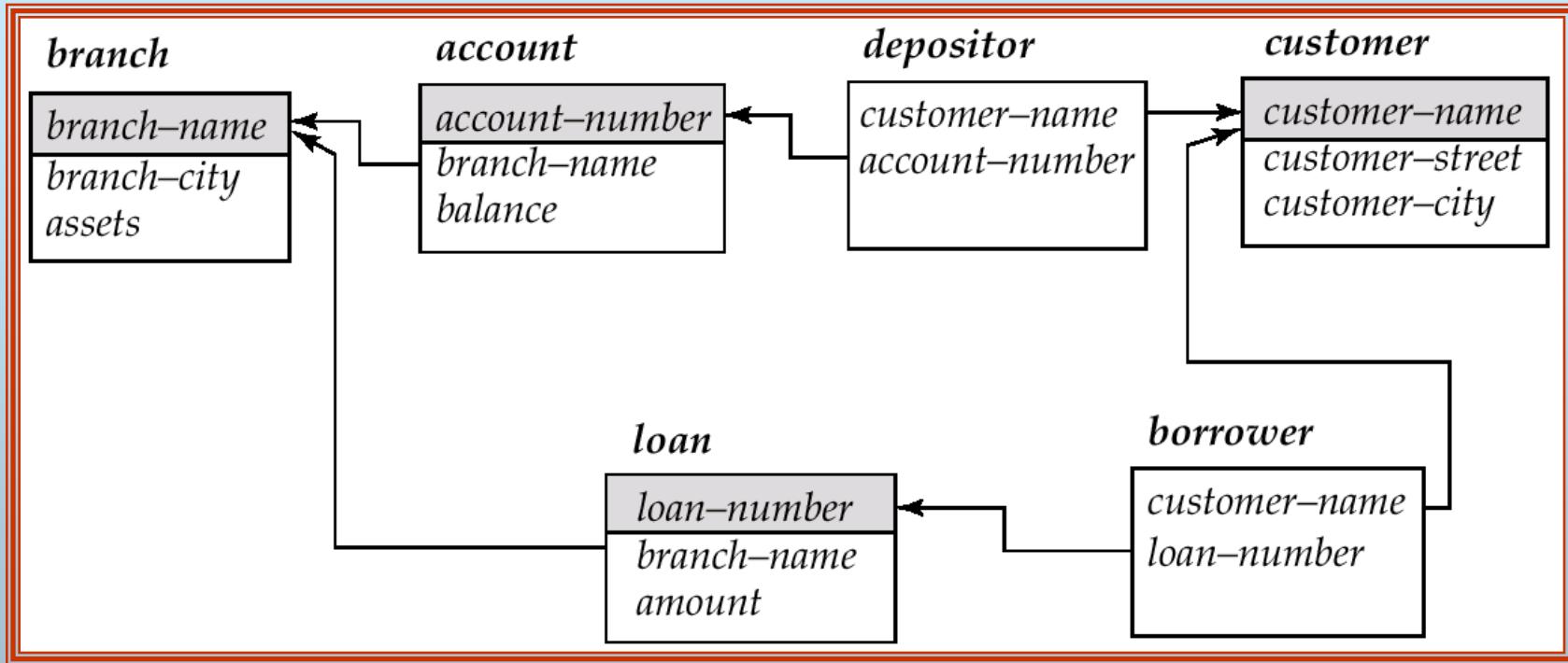


# Determining Keys from E-R Sets

- **Strong entity set.** The primary key of the entity set becomes the primary key of the relation.
- **Weak entity set.** The primary key of the relation consists of the union of the primary key of the strong entity set and the discriminator of the weak entity set.
- **Relationship set.** The union of the primary keys of the related entity sets becomes a super key of the relation.
  - For binary many-to-one relationship sets, the primary key of the “many” entity set becomes the relation’s primary key.
  - For one-to-one relationship sets, the relation’s primary key can be that of either entity set.
  - For many-to-many relationship sets, the union of the primary keys becomes the relation’s primary key



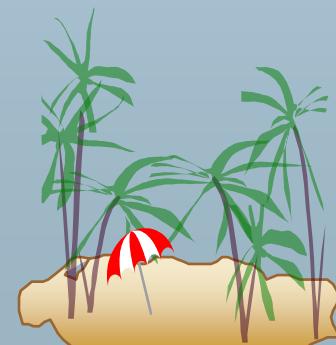
# Schema Diagram for the Banking Enterprise





# Query Languages

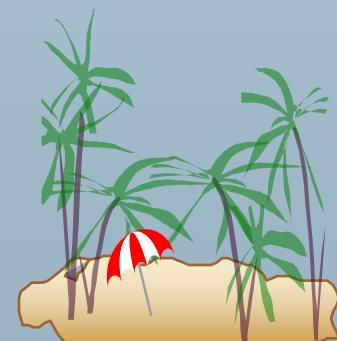
- Language in which user requests information from the database.
- Categories of languages
  - procedural
  - non-procedural
- “Pure” languages:
  - Relational Algebra
  - Tuple Relational Calculus
  - Domain Relational Calculus
- Pure languages form underlying basis of query languages that people use.





# Relational Algebra

- Procedural language
- Six basic operators
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename
- The operators take one or more relations as inputs and give a new relation as a result.





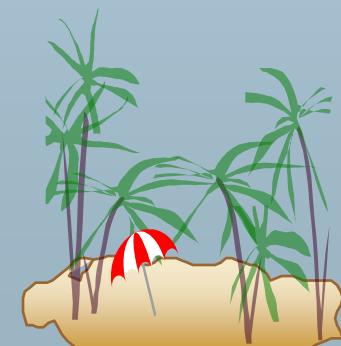
# Select Operation – Example

- Relation  $r$

$A$	$B$	$C$	$D$
$\alpha$	$\alpha$	1	7
$\alpha$	$\beta$	5	7
$\beta$	$\beta$	12	3
$\beta$	$\beta$	23	10

- $\sigma_{A=B \wedge D > 5}(r)$

$A$	$B$	$C$	$D$
$\alpha$	$\alpha$	1	7
$\beta$	$\beta$	23	10





# Select Operation

- Notation:  $\sigma_p(r)$
- $p$  is called the **selection predicate**
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

Where  $p$  is a formula in propositional calculus consisting of **terms** connected by :  $\wedge$  (**and**),  $\vee$  (**or**),  $\neg$  (**not**)

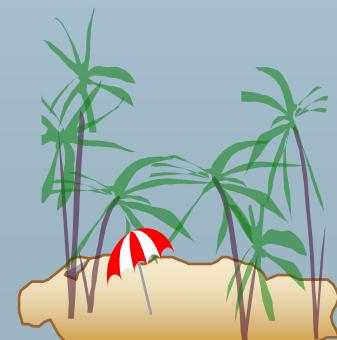
Each **term** is one of:

<attribute>  $op$  <attribute> or <constant>

where  $op$  is one of:  $=, \neq, >, \geq, <, \leq$

- Example of selection:

$\sigma_{\text{branch-name}=\text{"Perryridge}}(account)$





# Project Operation – Example

- Relation  $r$ :

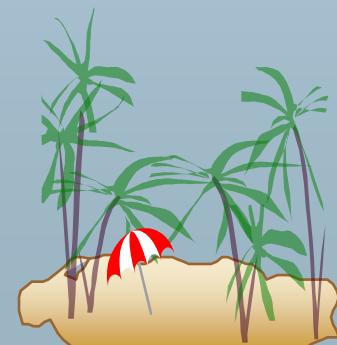
	A	B	C
$\alpha$	10	1	
$\alpha$	20	1	
$\beta$	30	1	
$\beta$	40	2	

- $\Pi_{A,C}(r)$

	A	C
$\alpha$	1	
$\alpha$	1	
$\beta$	1	
$\beta$	2	

=

	A	C
$\alpha$	1	
$\beta$	1	
$\beta$	2	





# Project Operation

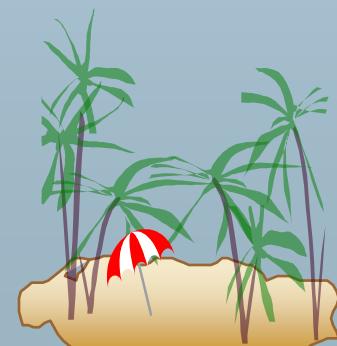
- Notation:

$$\Pi_{A_1, A_2, \dots, A_k}(r)$$

where  $A_1, A_2$  are attribute names and  $r$  is a relation name.

- The result is defined as the relation of  $k$  columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- E.g. To eliminate the *branch-name* attribute of *account*

$$\Pi_{\text{account-number}, \text{balance}}(\text{account})$$





# Union Operation – Example

- Relations  $r, s$ :

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

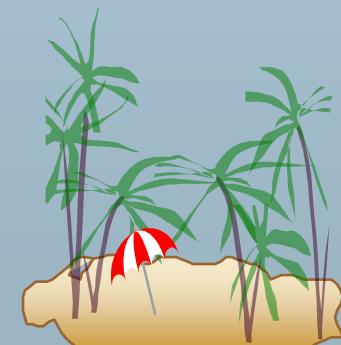
$r$

A	B
$\alpha$	2
$\beta$	3

$s$

$r \cup s$ :

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1
$\beta$	3



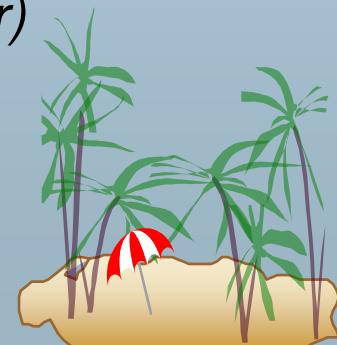


# Union Operation

- Notation:  $r \cup s$
- Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- For  $r \cup s$  to be valid.
  1.  $r, s$  must have the *same arity* (same number of attributes)
  2. The attribute domains must be *compatible* (e.g., 2nd column of  $r$  deals with the same type of values as does the 2nd column of  $s$ )
- E.g. to find all customers with either an account or a loan  
 $\Pi_{customer-name} (depositor) \cup \Pi_{customer-name} (borrower)$





# Set Difference Operation – Example

- Relations  $r, s$ :

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

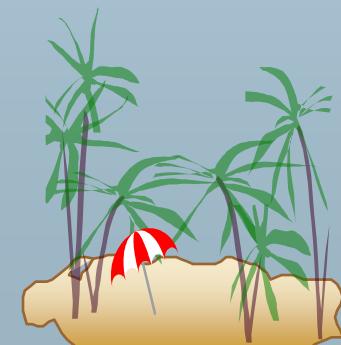
$r$

A	B
$\alpha$	2
$\beta$	3

$s$

$r - s$ :

A	B
$\alpha$	1
$\beta$	1



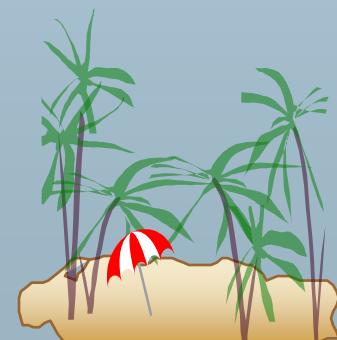


# Set Difference Operation

- Notation  $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \text{ and } t \notin s\}$$

- Set differences must be taken between *compatible* relations.
  - ☞  $r$  and  $s$  must have the *same arity*
  - ☞ attribute domains of  $r$  and  $s$  must be compatible





# Cartesian-Product Operation-Example

Relations  $r, s$ :

A	B
$\alpha$	1
$\beta$	2

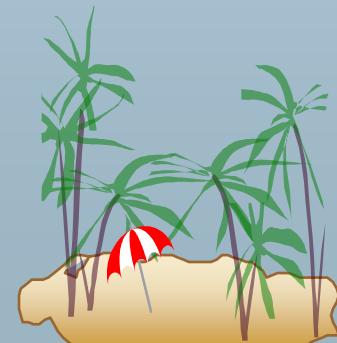
$r$

C	D	E
$\alpha$	10	a
$\beta$	10	a
$\beta$	20	b
$\gamma$	10	b

$s$

$r \times s$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\beta$	10	a
$\alpha$	1	$\beta$	20	b
$\alpha$	1	$\gamma$	10	b
$\beta$	2	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
$\beta$	2	$\beta$	20	b
$\beta$	2	$\gamma$	10	b



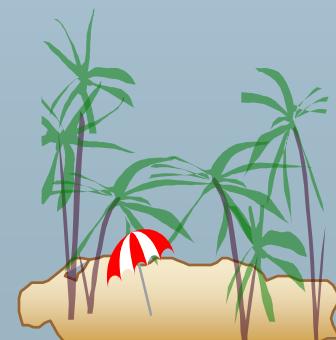


# Cartesian-Product Operation

- Notation  $r \times s$
- Defined as:

$$r \times s = \{t q \mid t \in r \text{ and } q \in s\}$$

- Assume that attributes of  $r(R)$  and  $s(S)$  are disjoint. (That is,  $R \cap S = \emptyset$ ).
- If attributes of  $r(R)$  and  $s(S)$  are not disjoint, then renaming must be used.





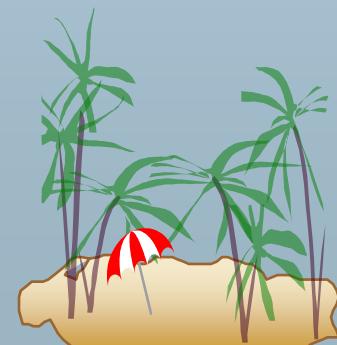
# Composition of Operations

- Can build expressions using multiple operations
- Example:  $\sigma_{A=C}(r \times s)$
- $r \times s$

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\beta$	10	a
$\alpha$	1	$\beta$	20	b
$\alpha$	1	$\gamma$	10	b
$\beta$	2	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
$\beta$	2	$\beta$	20	b
$\beta$	2	$\gamma$	10	b

- $\sigma_{A=C}(r \times s)$

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\beta$	2	$\beta$	20	a
$\beta$	2	$\beta$	20	b





# Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.

Example:

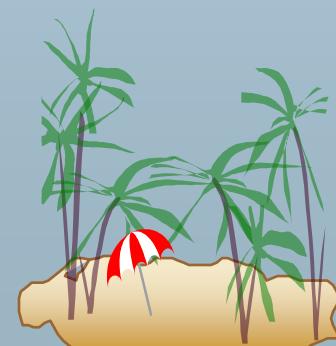
$$\rho_X(E)$$

returns the expression  $E$  under the name  $X$

If a relational-algebra expression  $E$  has arity  $n$ , then

$$\rho_{X(A_1, A_2, \dots, A_n)}(E)$$

returns the result of expression  $E$  under the name  $X$ , and with the attributes renamed to  $A_1, A_2, \dots, A_n$ .





# Banking Example

*branch (branch-name, branch-city, assets)*

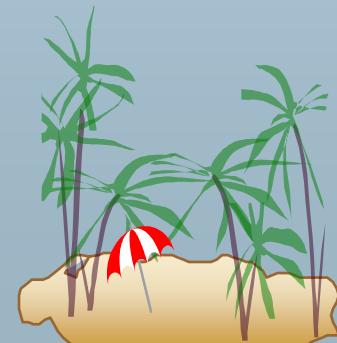
*customer (customer-name, customer-street, customer-only)*

*account (account-number, branch-name, balance)*

*loan (loan-number, branch-name, amount)*

*depositor (customer-name, account-number)*

*borrower (customer-name, loan-number)*



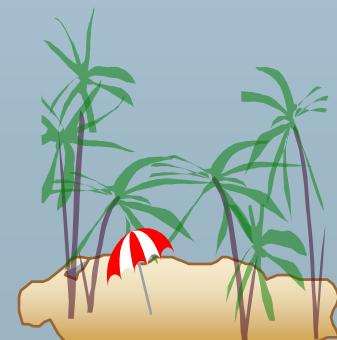


# Example Queries

- Find all loans of over \$1200

$$\sigma_{amount > 1200} (loan)$$

- Find the loan number for each loan of an amount greater than \$1200

$$\Pi_{loan-number} (\sigma_{amount > 1200} (loan))$$


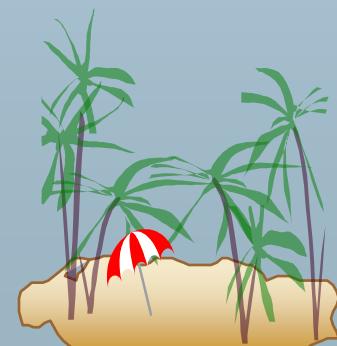


# Example Queries

- Find the names of all customers who have a loan, an account, or both, from the bank

$$\Pi_{customer-name} (borrower) \cup \Pi_{customer-name} (depositor)$$

- Find the names of all customers who have a loan and an account at bank.

$$\Pi_{customer-name} (borrower) \cap \Pi_{customer-name} (depositor)$$


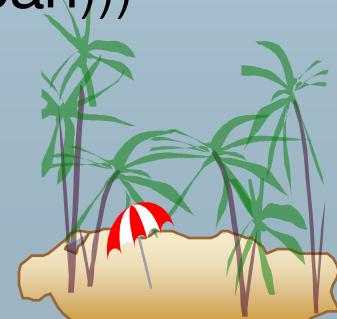


# Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.

$$\Pi_{customer-name} (\sigma_{branch-name = "Perryridge"} \\ (\sigma_{borrower.loan-number = loan.loan-number}(borrower \times loan)))$$

- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$$\Pi_{customer-name} (\sigma_{branch-name = "Perryridge"} \\ (\sigma_{borrower.loan-number = loan.loan-number}(borrower \times loan))) - \\ \Pi_{customer-name}(depositor)$$




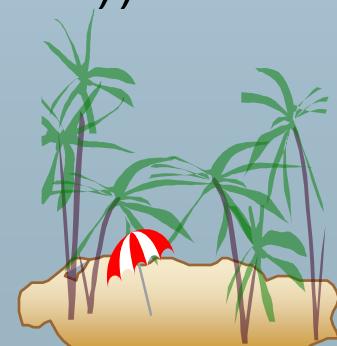
# Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.

- Query 1

$$\prod_{\text{customer-name}} (\sigma_{\text{branch-name} = \text{"Perryridge"} } ( \sigma_{\text{borrower.loan-number} = \text{loan.loan-number}} (\text{borrower} \times \text{loan})))$$

- Query 2

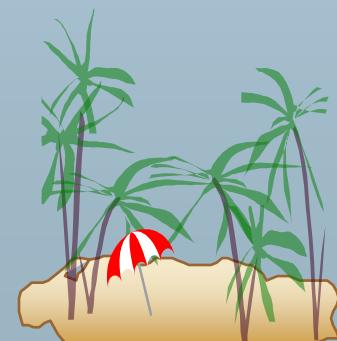
$$\prod_{\text{customer-name}} (\sigma_{\text{loan.loan-number} = \text{borrower.loan-number}} ( \sigma_{\text{branch-name} = \text{"Perryridge"} } (\text{loan}) ) \times \text{borrower}))$$




# Example Queries

Find the largest account balance

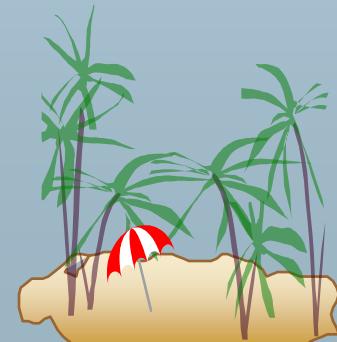
- Rename *account* relation as *d*
- The query is:

$$\Pi_{balance}(account) - \Pi_{account.balance}$$
$$(\sigma_{account.balance < d.balance} (account \times \rho_d (account)))$$




# Formal Definition

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation
- Let  $E_1$  and  $E_2$  be relational-algebra expressions; the following are all relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_p(E_1)$ ,  $P$  is a predicate on attributes in  $E_1$
  - $\Pi_s(E_1)$ ,  $S$  is a list consisting of some of the attributes in  $E_1$
  - $\rho_x(E_1)$ ,  $x$  is the new name for the result of  $E_1$

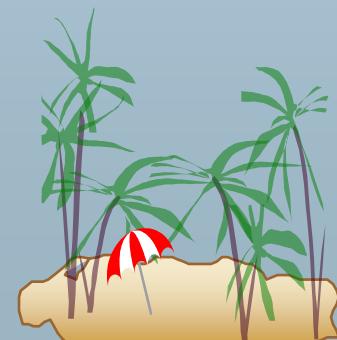




# Additional Operations

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

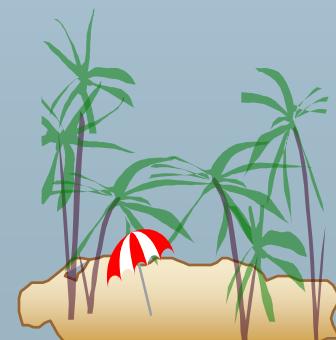
- Set intersection
- Natural join
- Division
- Assignment





# Set-Intersection Operation

- Notation:  $r \cap s$
- Defined as:
- $r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$
- Assume:
  - ☞  $r, s$  have the *same arity*
  - ☞ attributes of  $r$  and  $s$  are compatible
- Note:  $r \cap s = r - (r - s)$





# Set-Intersection Operation - Example

- Relation r, s:

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

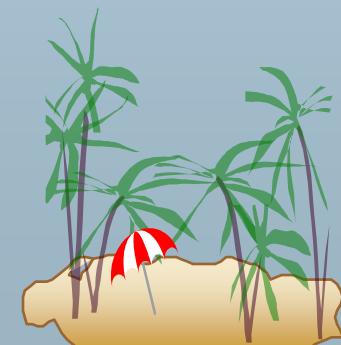
r

A	B
$\alpha$	2
$\beta$	3

s

- $r \cap s$

A	B
$\alpha$	2

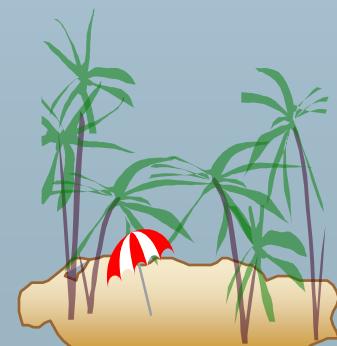




# Natural-Join Operation

- Notation:  $r \bowtie s$
- Let  $r$  and  $s$  be relations on schemas  $R$  and  $S$  respectively.  
Then,  $r \bowtie s$  is a relation on schema  $R \cup S$  obtained as follows:
  - ☞ Consider each pair of tuples  $t_r$  from  $r$  and  $t_s$  from  $s$ .
  - ☞ If  $t_r$  and  $t_s$  have the same value on each of the attributes in  $R \cap S$ , add a tuple  $t$  to the result, where
    - $t$  has the same value as  $t_r$  on  $r$
    - $t$  has the same value as  $t_s$  on  $s$
- Example:
  - $R = (A, B, C, D)$
  - $S = (E, B, D)$
  - ☞ Result schema =  $(A, B, C, D, E)$
  - ☞  $r \bowtie s$  is defined as:

$$\Pi_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$





# Natural Join Operation – Example

- Relations r, s:

A	B	C	D
$\alpha$	1	$\alpha$	a
$\beta$	2	$\gamma$	a
$\gamma$	4	$\beta$	b
$\alpha$	1	$\gamma$	a
$\delta$	2	$\beta$	b

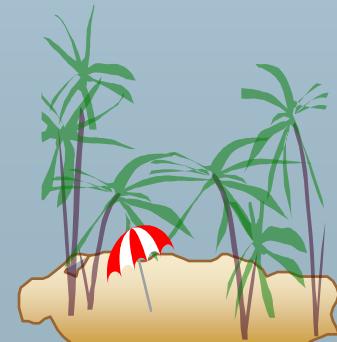
r

B	D	E
1	a	$\alpha$
3	a	$\beta$
1	a	$\gamma$
2	b	$\delta$
3	b	$\in$

s

$r \bowtie s$

A	B	C	D	E
$\alpha$	1	$\alpha$	a	$\alpha$
$\alpha$	1	$\alpha$	a	$\gamma$
$\alpha$	1	$\gamma$	a	$\alpha$
$\alpha$	1	$\gamma$	a	$\gamma$
$\delta$	2	$\beta$	b	$\delta$





# Division Operation

$r \div s$

- Suited to queries that include the phrase “for all”.
- Let  $r$  and  $s$  be relations on schemas  $R$  and  $S$  respectively where

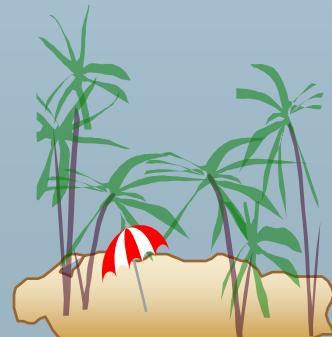
☞  $R = (A_1, \dots, A_m, B_1, \dots, B_n)$

☞  $S = (B_1, \dots, B_n)$

The result of  $r \div s$  is a relation on schema

$R - S = (A_1, \dots, A_m)$

$$r \div s = \{ t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s ( tu \in r ) \}$$





# Division Operation – Example

Relations  $r, s$ :

$A$	$B$
$\alpha$	1
$\alpha$	2
$\alpha$	3
$\beta$	1
$\gamma$	1
$\delta$	1
$\delta$	3
$\delta$	4
$\epsilon$	6
$\epsilon$	1
$\beta$	2

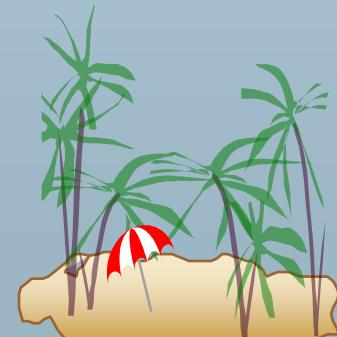
$r \div s$ :

$A$
$\alpha$
$\beta$

$r$

$B$
1
2

$s$





# Another Division Example

Relations  $r, s$ :

A	B	C	D	E
$\alpha$	a	$\alpha$	a	1
$\alpha$	a	$\gamma$	a	1
$\alpha$	a	$\gamma$	b	1
$\beta$	a	$\gamma$	a	1
$\beta$	a	$\gamma$	b	3
$\gamma$	a	$\gamma$	a	1
$\gamma$	a	$\gamma$	b	1
$\gamma$	a	$\beta$	b	1

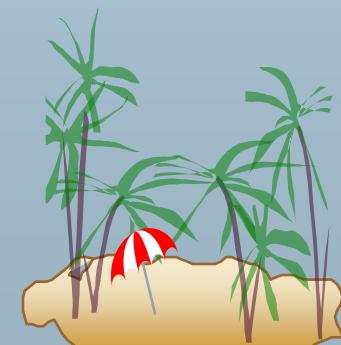
$r$

D	E
a	1
b	1

$s$

$r \div s$ :

A	B	C
$\alpha$	a	$\gamma$
$\gamma$	a	$\gamma$





# Division Operation (Cont.)

## ■ Property

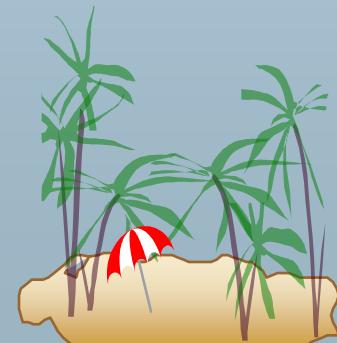
- ❖ Let  $q = r \div s$
- ❖ Then  $q$  is the largest relation satisfying  $q \times s \subseteq r$

## ■ Definition in terms of the basic algebra operation Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

To see why

- ❖  $\Pi_{R-S,S}(r)$  simply reorders attributes of  $r$
- ❖  $\Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$  gives those tuples  $t$  in  $\Pi_{R-S}(r)$  such that for some tuple  $u \in s$ ,  $tu \notin r$ .



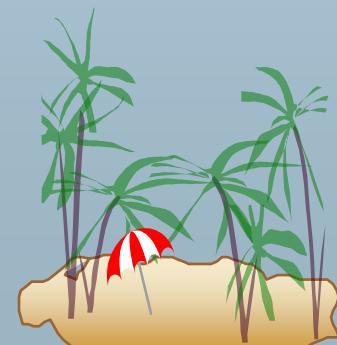


# Assignment Operation

- The assignment operation ( $\leftarrow$ ) provides a convenient way to express complex queries.
  - ❖ Write query as a sequential program consisting of
    - ▀ a series of assignments
    - ▀ followed by an expression whose value is displayed as a result of the query.
  - ❖ Assignment must always be made to a temporary relation variable.
- Example: Write  $r \div s$  as

$$\text{temp1} \leftarrow \Pi_{R-S}(r)$$
$$\text{temp2} \leftarrow \Pi_{R-S} ((\text{temp1} \times s) - \Pi_{R-S,S}(r))$$
$$\text{result} = \text{temp1} - \text{temp2}$$

- ❖ The result to the right of the  $\leftarrow$  is assigned to the relation variable on the left of the  $\leftarrow$ .
- ❖ May use variable in subsequent expressions.





# Example Queries

- Find all customers who have an account from at least the “Downtown” and the Uptown” branches.

Query 1

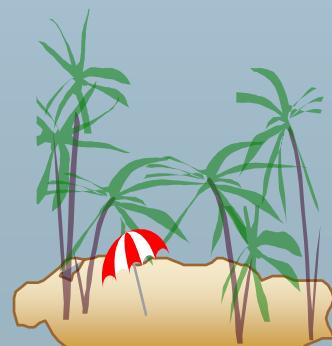
$$\Pi_{CN}(\sigma_{BN=\text{"Downtown"}}(depositor \bowtie account)) \cap$$

$$\Pi_{CN}(\sigma_{BN=\text{"Uptown"}}(depositor \bowtie account))$$

where  $CN$  denotes customer-name and  $BN$  denotes branch-name.

Query 2

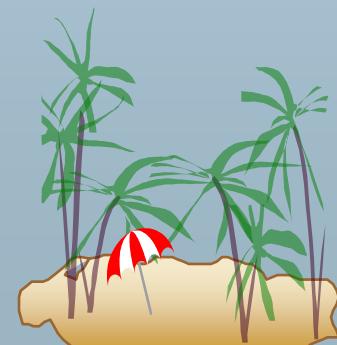
$$\begin{aligned} & \Pi_{customer-name, branch-name}(depositor \bowtie account) \\ & \div \rho_{temp(branch-name)}(\{("Downtown"), ("Uptown")\}) \end{aligned}$$





# Example Queries

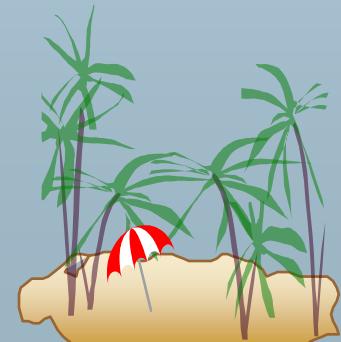
- Find all customers who have an account at all branches located in Brooklyn city.

$$\begin{aligned} & \prod_{customer-name, branch-name} (depositor \bowtie account) \\ & \div \prod_{branch-name} (\sigma_{branch-city = "Brooklyn"} (branch)) \end{aligned}$$




# Extended Relational-Algebra-Operations

- Generalized Projection
- Outer Join
- Aggregate Functions





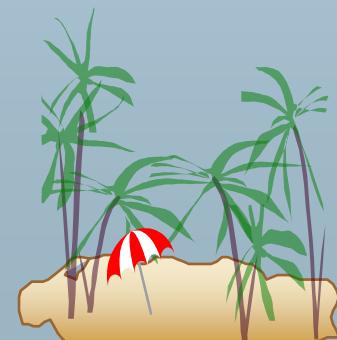
# Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.

$$\Pi_{F_1, F_2, \dots, F_n}(E)$$

- $E$  is any relational-algebra expression
- Each of  $F_1, F_2, \dots, F_n$  are arithmetic expressions involving constants and attributes in the schema of  $E$ .
- Given relation  $\text{credit-info}(\text{customer-name}, \text{limit}, \text{credit-balance})$ , find how much more each person can spend:

$$\Pi_{\text{customer-name}, \text{limit} - \text{credit-balance}} (\text{credit-info})$$





# Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.

**avg**: average value

**min**: minimum value

**max**: maximum value

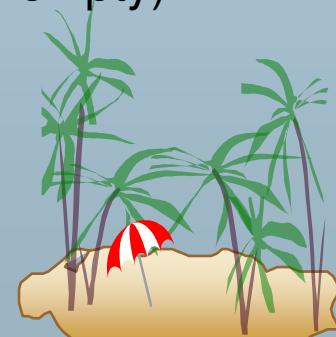
**sum**: sum of values

**count**: number of values

- **Aggregate operation** in relational algebra

$$G_1, G_2, \dots, G_n \text{ } g_{F_1(A_1), F_2(A_2), \dots, F_n(A_n)}(E)$$

- $E$  is any relational-algebra expression
- $G_1, G_2, \dots, G_n$  is a list of attributes on which to group (can be empty)
- Each  $F_i$  is an aggregate function
- Each  $A_i$  is an attribute name





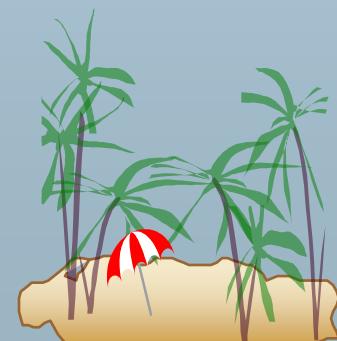
# Aggregate Operation – Example

- Relation  $r$ :

$A$	$B$	$C$
$\alpha$	$\alpha$	7
$\alpha$	$\beta$	7
$\beta$	$\beta$	3
$\beta$	$\beta$	10

$g_{\text{sum}(C)}(r)$

$sum-C$
27





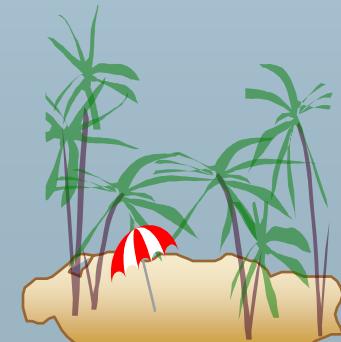
# Aggregate Operation – Example

- Relation *account* grouped by *branch-name*:

<i>branch-name</i>	<i>account-number</i>	<i>balance</i>
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

*branch-name*  $\text{g sum}(\text{balance})$  (*account*)

<i>branch-name</i>	<i>balance</i>
Perryridge	1300
Brighton	1500
Redwood	700

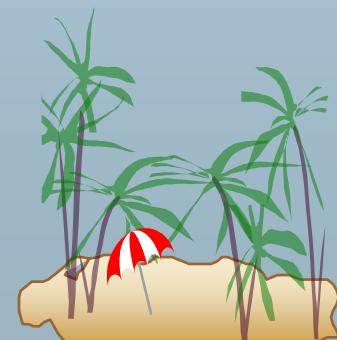




# Aggregate Functions (Cont.)

- Result of aggregation does not have a name
  - ☞ Can use rename operation to give it a name
  - ☞ For convenience, we permit renaming as part of aggregate operation

*branch-name g sum(balance) as sum-balance (account)*





# Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - ☞ *null* signifies that the value is unknown or does not exist
  - ☞ All comparisons involving *null* are (roughly speaking) **false** by definition.
  - ☞ Will study precise meaning of comparisons with nulls later





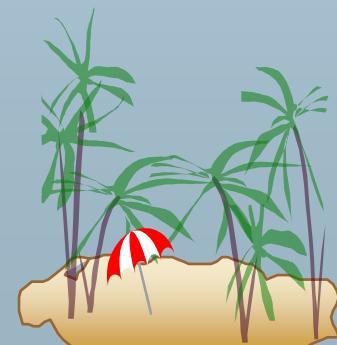
# Outer Join – Example

## ■ Relation *loan*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

## ■ Relation *borrower*

<i>customer-name</i>	<i>loan-number</i>
Jones	L-170
Smith	L-230
Hayes	L-155





# Outer Join – Example

## ■ Inner Join

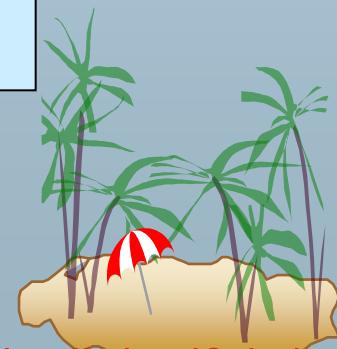
$loan \bowtie Borrower$

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

## ■ Left Outer Join

$loan \leftarrow\bowtie Borrower$

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





# Outer Join – Example

## ■ Right Outer Join

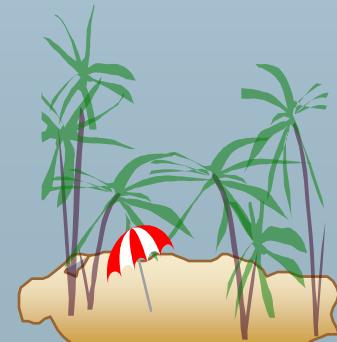
*loan*  $\bowtie$  *borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	<i>null</i>	<i>null</i>	Hayes

## ■ Full Outer Join

*loan*  $\bowtie\bowtie$  *borrower*

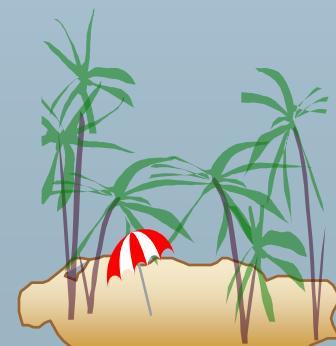
<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>
L-155	<i>null</i>	<i>null</i>	Hayes





# Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*.
- Aggregate functions simply ignore null values
  - ☞ Is an arbitrary decision. Could have returned null as result instead.
  - ☞ We follow the semantics of SQL in its handling of null values
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same
  - ☞ Alternative: assume each null is different from each other
  - ☞ Both are arbitrary decisions, so we simply follow SQL





# Null Values

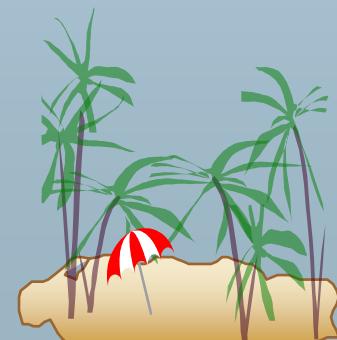
- Comparisons with null values return the special truth value *unknown*
  - ☞ If *false* was used instead of *unknown*, then  $\text{not } (A < 5)$  would not be equivalent to  $A \geq 5$
- Three-valued logic using the truth value *unknown*:
  - ☞ OR:  $(\text{unknown} \text{ or } \text{true}) = \text{true}$ ,  
 $(\text{unknown} \text{ or } \text{false}) = \text{unknown}$   
 $(\text{unknown} \text{ or } \text{unknown}) = \text{unknown}$
  - ☞ AND:  $(\text{true and unknown}) = \text{unknown}$ ,  
 $(\text{false and unknown}) = \text{false}$ ,  
 $(\text{unknown and unknown}) = \text{unknown}$
  - ☞ NOT:  $(\text{not unknown}) = \text{unknown}$
  - ☞ In SQL “*P is unknown*” evaluates to true if predicate *P* evaluates to *unknown*
- Result of select predicate is treated as *false* if it evaluates to *unknown*





# Modification of the Database

- The content of the database may be modified using the following operations:
  - Deletion
  - Insertion
  - Updating
- All these operations are expressed using the assignment operator.





# Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where  $r$  is a relation and  $E$  is a relational algebra query.





# Deletion Examples

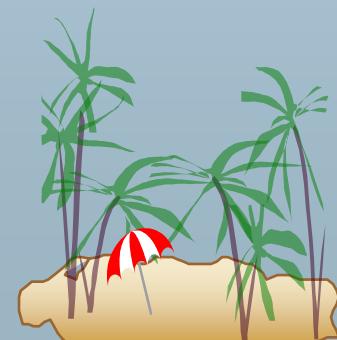
- Delete all account records in the Perryridge branch.

$$\text{account} \leftarrow \text{account} - \sigma_{\text{branch-name} = \text{"Perryridge"} }(\text{account})$$

- Delete all loan records with amount in the range of 0 to 50

$$\text{loan} \leftarrow \text{loan} - \sigma_{\text{amount} \geq 0 \text{ and } \text{amount} \leq 50 }(\text{loan})$$

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{\text{branch-city} = \text{"Needham"} }(\text{account} \bowtie \text{branch})$$
$$r_2 \leftarrow \Pi_{\text{branch-name}, \text{account-number}, \text{balance}}(r_1)$$
$$r_3 \leftarrow \Pi_{\text{customer-name}, \text{account-number}}(r_2 \bowtie \text{depositor})$$
$$\text{account} \leftarrow \text{account} - r_2$$
$$\text{depositor} \leftarrow \text{depositor} - r_3$$




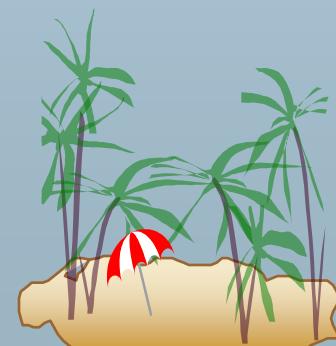
# Insertion

- To insert data into a relation, we either:
  - ☞ specify a tuple to be inserted
  - ☞ write a query whose result is a set of tuples to be inserted
- in relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

where  $r$  is a relation and  $E$  is a relational algebra expression.

- The insertion of a single tuple is expressed by letting  $E$  be a constant relation containing one tuple.



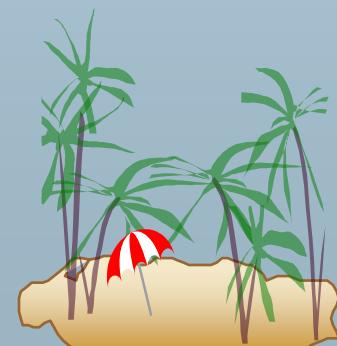


# Insertion Examples

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.

$$\text{account} \leftarrow \text{account} \cup \{(\text{"Perryridge"}, \text{A-973}, 1200)\}$$
$$\text{depositor} \leftarrow \text{depositor} \cup \{(\text{"Smith"}, \text{A-973})\}$$

- Provide as a gift for all loan customers in the Perryridge branch, a \$200 savings account. Let the loan number serve as the account number for the new savings account.

$$r_1 \leftarrow (\sigma_{\text{branch-name} = \text{"Perryridge"}}(\text{borrower} \bowtie \text{loan}))$$
$$\text{account} \leftarrow \text{account} \cup \Pi_{\text{branch-name}, \text{account-number}, 200}(r_1)$$
$$\text{depositor} \leftarrow \text{depositor} \cup \Pi_{\text{customer-name}, \text{loan-number}}(r_1)$$


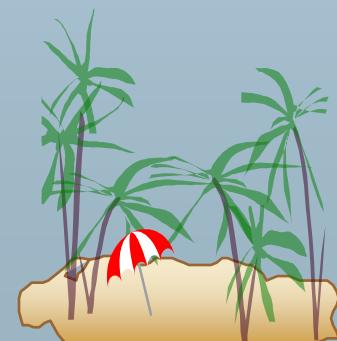


# Updating

- A mechanism to change a value in a tuple without changing *all* values in the tuple
- Use the generalized projection operator to do this task

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_i} (r)$$

- Each  $F_i$  is either
  - ☞ the  $i$ th attribute of  $r$ , if the  $i$ th attribute is not updated, or,
  - ☞ if the attribute is to be updated  $F_i$  is an expression, involving only constants and the attributes of  $r$ , which gives the new value for the attribute





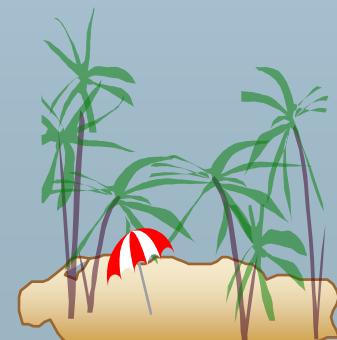
# Update Examples

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \Pi_{AN, BN, BAL} * 1.05 (account)$$

where  $AN$ ,  $BN$  and  $BAL$  stand for *account-number*, *branch-name* and *balance*, respectively.

- Pay all accounts with balances over \$10,000 6 percent interest and pay all others 5 percent

$$\begin{aligned} account \leftarrow & \quad \Pi_{AN, BN, BAL} * 1.06 (\sigma_{BAL > 10000} (account)) \\ & \cup \Pi_{AN, BN, BAL} * 1.05 (\sigma_{BAL \leq 10000} (account)) \end{aligned}$$


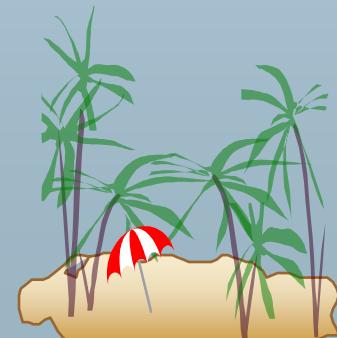


# Views

- In some cases, it is not desirable for all users to see the entire logical model (i.e., all the actual relations stored in the database.)
- Consider a person who needs to know a customer's loan number but has no need to see the loan amount. This person should see a relation described, in the relational algebra, by

$$\Pi_{\text{customer-name}, \text{loan-number}}(\text{borrower} \bowtie \text{loan})$$

- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.





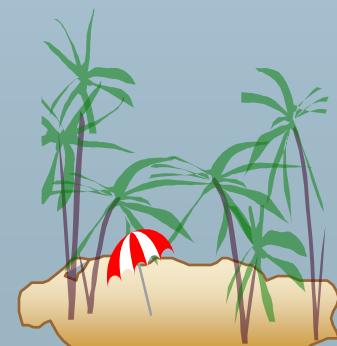
# View Definition

- A view is defined using the **create view** statement which has the form

**create view  $v$  as <query expression>**

where <query expression> is any legal relational algebra query expression. The view name is represented by  $v$ .

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - ☞ Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.





# View Examples

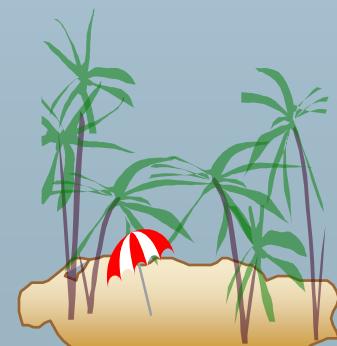
- Consider the view (named *all-customer*) consisting of branches and their customers.

**create view *all-customer* as**

$$\begin{aligned} & \Pi_{\text{branch-name}, \text{customer-name}} (\text{depositor} \bowtie \text{account}) \\ & \cup \Pi_{\text{branch-name}, \text{customer-name}} (\text{borrower} \bowtie \text{loan}) \end{aligned}$$

- We can find all customers of the Perryridge branch by writing:

$$\begin{aligned} & \Pi_{\text{customer-name}} \\ & (\sigma_{\text{branch-name} = \text{"Perryridge"}} (\text{all-customer})) \end{aligned}$$





# Updates Through View

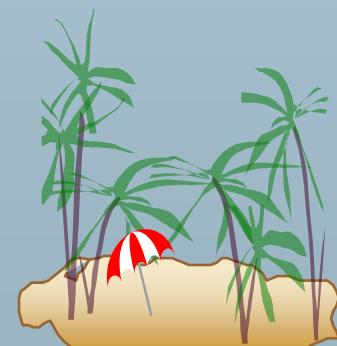
- Database modifications expressed as views must be translated to modifications of the actual relations in the database.
- Consider the person who needs to see all loan data in the *loan* relation except *amount*. The view given to the person, *branch-loan*, is defined as:

**create view branch-loan as**

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

- Since we allow a view name to appear wherever a relation name is allowed, the person may write:

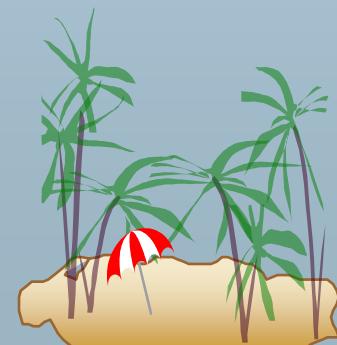
*branch-loan*  $\leftarrow$  *branch-loan*  $\cup$  {("Perryridge", L-37)}





# Updates Through Views (Cont.)

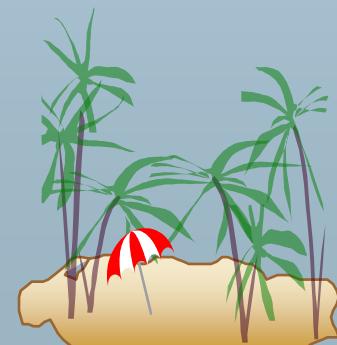
- The previous insertion must be represented by an insertion into the actual relation *loan* from which the view *branch-loan* is constructed.
- An insertion into *loan* requires a value for *amount*. The insertion can be dealt with by either.
  - ☞ rejecting the insertion and returning an error message to the user.
  - ☞ inserting a tuple ("L-37", "Perryridge", *null*) into the *loan* relation
- Some updates through views are impossible to translate into database relation updates
  - ☞ create view *v* as  $\sigma_{branch-name = "Perryridge"}(account)$   
 $v \leftarrow v \cup (L-99, \text{Downtown}, 23)$
- Others cannot be translated uniquely
  - ☞ *all-customer*  $\leftarrow all-customer \cup \{("Perryridge", "John")\}$ 
    - ☞ Have to choose loan or account, and create a new loan/account number!





# Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation  $v_1$  is said to *depend directly* on a view relation  $v_2$  if  $v_2$  is used in the expression defining  $v_1$
- A view relation  $v_1$  is said to *depend on* view relation  $v_2$  if either  $v_1$  depends directly to  $v_2$  or there is a path of dependencies from  $v_1$  to  $v_2$
- A view relation  $v$  is said to be *recursive* if it depends on itself.





# View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view  $v_1$  be defined by an expression  $e_1$  that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

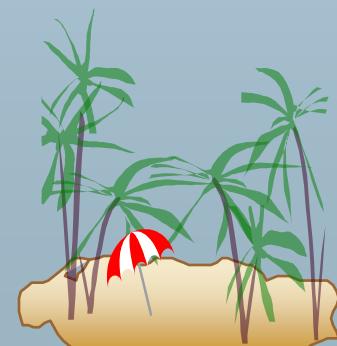
**repeat**

    Find any view relation  $v_i$  in  $e_1$

    Replace the view relation  $v_i$  by the expression defining  $v_i$

**until** no more view relations are present in  $e_1$

- As long as the view definitions are not recursive, this loop will terminate



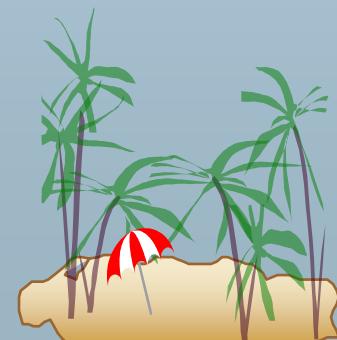


# Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form

$$\{t \mid P(t)\}$$

- It is the set of all tuples  $t$  such that predicate  $P$  is true for  $t$
- $t$  is a *tuple variable*,  $t[A]$  denotes the value of tuple  $t$  on attribute  $A$
- $t \in r$  denotes that tuple  $t$  is in relation  $r$
- $P$  is a *formula* similar to that of the predicate calculus





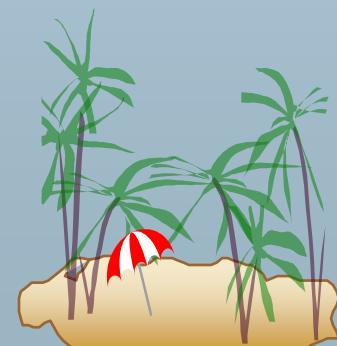
# Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g.,  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $>$ ,  $\geq$ )
3. Set of connectives: and ( $\wedge$ ), or ( $\vee$ ), not ( $\neg$ )
4. Implication ( $\Rightarrow$ ):  $x \Rightarrow y$ , if  $x$  if true, then  $y$  is true

$$x \Rightarrow y \equiv \neg x \vee y$$

5. Set of quantifiers:

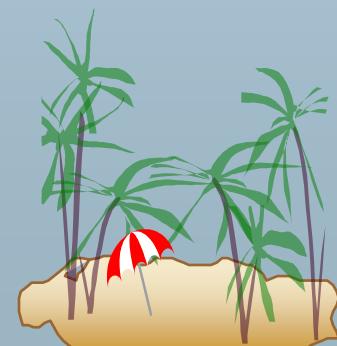
- $\exists t \in r (Q(t)) \equiv$  "there exists" a tuple in  $t$  in relation  $r$  such that predicate  $Q(t)$  is true
- $\forall t \in r (Q(t)) \equiv Q$  is true "for all" tuples  $t$  in relation  $r$





# Banking Example

- *branch (branch-name, branch-city, assets)*
- *customer (customer-name, customer-street, customer-city)*
- *account (account-number, branch-name, balance)*
- *loan (loan-number, branch-name, amount)*
- *depositor (customer-name, account-number)*
- *borrower (customer-name, loan-number)*





# Example Queries

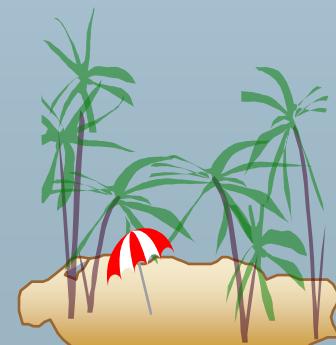
- Find the *loan-number*, *branch-name*, and *amount* for loans of over \$1200

$$\{t \mid t \in \text{loan} \wedge t[\text{amount}] > 1200\}$$

- Find the loan number for each loan of an amount greater than \$1200

$$\{t \mid \exists s \in \text{loan} (t[\text{loan-number}] = s[\text{loan-number}] \wedge s[\text{amount}] > 1200)\}$$

Notice that a relation on schema [*loan-number*] is implicitly defined by the query



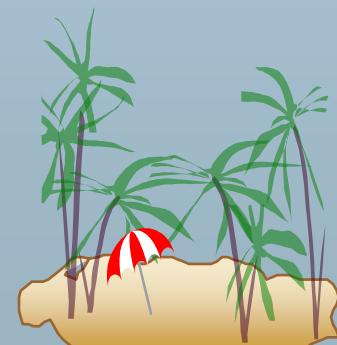


# Example Queries

- Find the names of all customers having a loan, an account, or both at the bank

$$\{t / \exists s \in \text{borrower}( t[\text{customer-name}] = s[\text{customer-name}] ) \\ \vee \exists u \in \text{depositor}( t[\text{customer-name}] = u[\text{customer-name}] )\}$$

- Find the names of all customers who have a loan and an account at the bank

$$\{t / \exists s \in \text{borrower}( t[\text{customer-name}] = s[\text{customer-name}] ) \\ \wedge \exists u \in \text{depositor}( t[\text{customer-name}] = u[\text{customer-name}] )\}$$


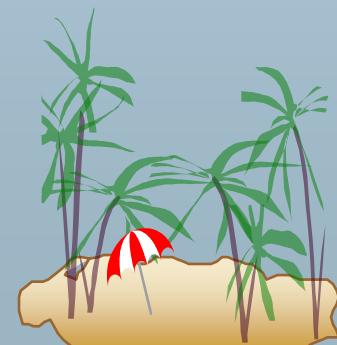


# Example Queries

- Find the names of all customers having a loan at the Perryridge branch

$$\{t / \exists s \in \text{borrower}(t[\text{customer-name}] = s[\text{customer-name}] \wedge \exists u \in \text{loan}(u[\text{branch-name}] = \text{"Perryridge"} \wedge u[\text{loan-number}] = s[\text{loan-number}]))\}$$

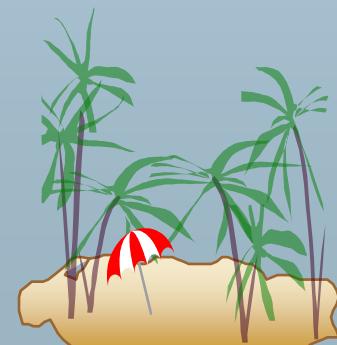
- Find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank

$$\{t / \exists s \in \text{borrower}( t[\text{customer-name}] = s[\text{customer-name}] \wedge \exists u \in \text{loan}(u[\text{branch-name}] = \text{"Perryridge"} \wedge u[\text{loan-number}] = s[\text{loan-number}]) \wedge \text{not } \exists v \in \text{depositor}(v[\text{customer-name}] = t[\text{customer-name}]) \}$$




# Example Queries

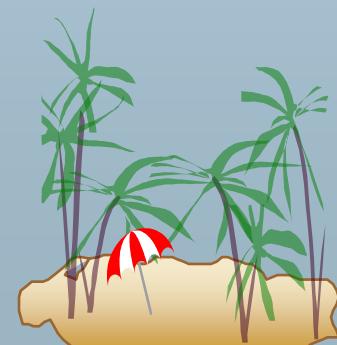
- Find the names of all customers having a loan from the Perryridge branch, and the cities they live in

$$\{t \mid \exists s \in \text{loan}(s[\text{branch-name}] = \text{"Perryridge"}) \wedge \exists u \in \text{borrower}(u[\text{loan-number}] = s[\text{loan-number}] \wedge t[\text{customer-name}] = u[\text{customer-name}]) \wedge \exists v \in \text{customer}(u[\text{customer-name}] = v[\text{customer-name}] \wedge t[\text{customer-city}] = v[\text{customer-city}]))\}$$




# Example Queries

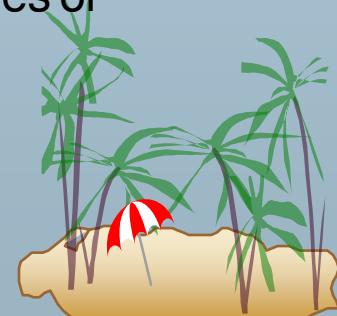
- Find the names of all customers who have an account at all branches located in Brooklyn:

$$\{t \mid \exists c \in \text{customer} (t[\text{customer.name}] = c[\text{customer-name}]) \wedge \\ \forall s \in \text{branch} (s[\text{branch-city}] = \text{"Brooklyn"} \Rightarrow \\ \exists u \in \text{account} (s[\text{branch-name}] = u[\text{branch-name}]) \\ \wedge \exists s \in \text{depositor} (t[\text{customer-name}] = s[\text{customer-name}] \\ \wedge s[\text{account-number}] = u[\text{account-number}]))\}$$




# Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example,  $\{t \mid \neg t \in r\}$  results in an infinite relation if the domain of any attribute of relation  $r$  is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression  $\{t \mid P(t)\}$  in the tuple relational calculus is *safe* if every component of  $t$  appears in one of the relations, tuples, or constants that appear in  $P$ 
  - ☞ NOTE: this is more than just a syntax condition.
    - ☞ E.g.  $\{ t \mid t[A]=5 \vee \text{true} \}$  is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in  $P$ .



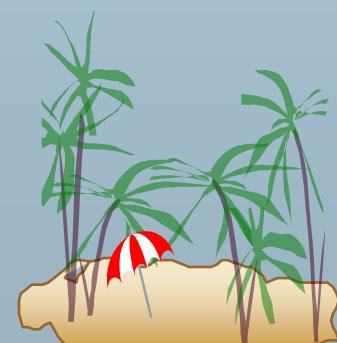


# Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:

$$\{ < x_1, x_2, \dots, x_n > \mid P(x_1, x_2, \dots, x_n) \}$$

- ☞  $x_1, x_2, \dots, x_n$  represent domain variables
- ☞  $P$  represents a formula similar to that of the predicate calculus





# Example Queries

- Find the *loan-number*, *branch-name*, and *amount* for loans of over \$1200

$$\{< l, b, a > \mid < l, b, a > \in \text{loan} \wedge a > 1200\}$$

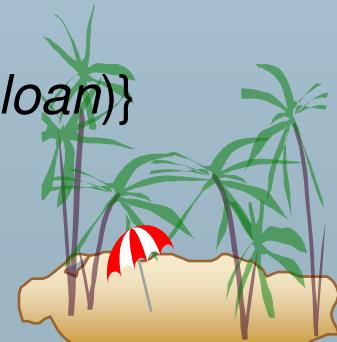
- Find the names of all customers who have a loan of over \$1200

$$\{< c > \mid \exists l, b, a (< c, l > \in \text{borrower} \wedge < l, b, a > \in \text{loan} \wedge a > 1200)\}$$

- Find the names of all customers who have a loan from the Perryridge branch and the loan amount:

$$\{< c, a > \mid \exists l (< c, l > \in \text{borrower} \wedge \exists b (< l, b, a > \in \text{loan} \wedge b = \text{"Perryridge"}))\}$$

or  $\{< c, a > \mid \exists l (< c, l > \in \text{borrower} \wedge < l, \text{"Perryridge"}, a > \in \text{loan})\}$



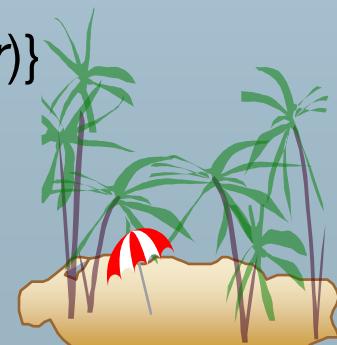


# Example Queries

- Find the names of all customers having a loan, an account, or both at the Perryridge branch:

$$\{< c > \mid \exists l (< c, l > \in \text{borrower} \\ \wedge \exists b, a (< l, b, a > \in \text{loan} \wedge b = \text{"Perryridge"})) \\ \vee \exists a (< c, a > \in \text{depositor} \\ \wedge \exists b, n (< a, b, n > \in \text{account} \wedge b = \text{"Perryridge"}))\}$$

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

$$\{< c > \mid \exists s, n (< c, s, n > \in \text{customer}) \wedge \\ \forall x, y, z (< x, y, z > \in \text{branch} \wedge y = \text{"Brooklyn"}) \Rightarrow \\ \exists a, b (< x, y, z > \in \text{account} \wedge < c, a > \in \text{depositor})\}$$


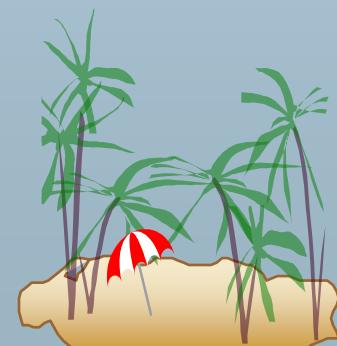


# Safety of Expressions

$$\{ < x_1, x_2, \dots, x_n > \mid P(x_1, x_2, \dots, x_n) \}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from  $\text{dom}(P)$  (that is, the values appear either in  $P$  or in a tuple of a relation mentioned in  $P$ ).
2. For every “there exists” subformula of the form  $\exists x (P_1(x))$ , the subformula is true if and only if there is a value of  $x$  in  $\text{dom}(P_1)$  such that  $P_1(x)$  is true.
3. For every “for all” subformula of the form  $\forall x (P_1(x))$ , the subformula is true if and only if  $P_1(x)$  is true for all values  $x$  from  $\text{dom}(P_1)$ .





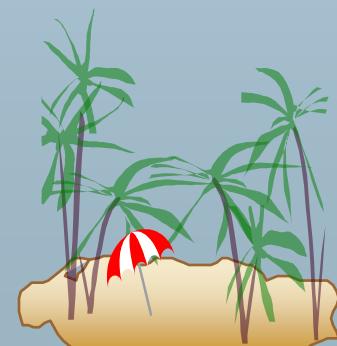
Das Bild kann zurzeit nicht angezeigt werden.

# End of Chapter 3



# Result of $\sigma_{branch-name = "Perryridge"}(loan)$

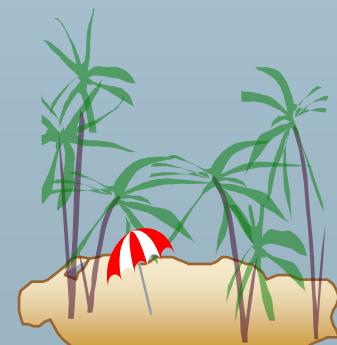
<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-15	Perryridge	1500
L-16	Perryridge	1300





# Loan Number and the Amount of the Loan

<i>loan-number</i>	<i>amount</i>
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

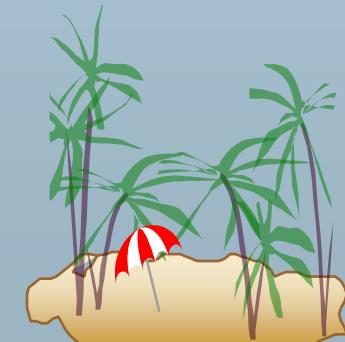




# Names of All Customers Who Have Either a Loan or an Account

*customer-name*

Adams  
Curry  
Hayes  
Jackson  
Jones  
Smith  
Williams  
Lindsay  
Johnson  
Turner

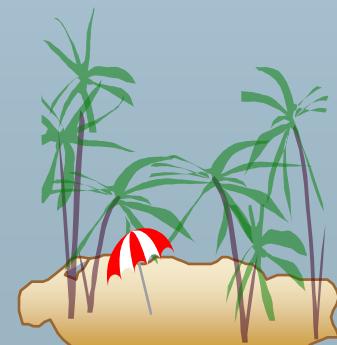




# Customers With An Account But No Loan

*customer-name*

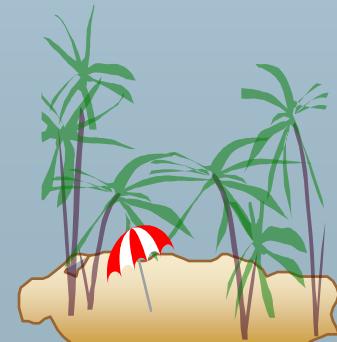
Johnson  
Lindsay  
Turner





# Result of *borrower* × *loan*

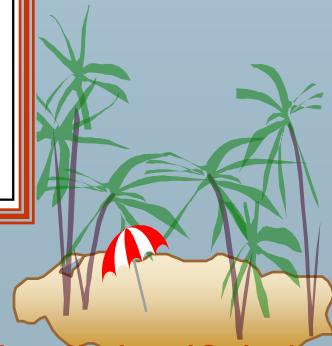
customer-name	borrower.	loan.	branch-name	amount
	loan-number	loan-number		
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500





## Result of $\sigma_{branch-name = "Perryridge"} (borrower \times loan)$

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

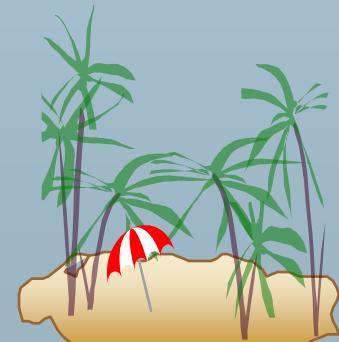




# Result of $\Pi_{customer-name}$

*customer-name*

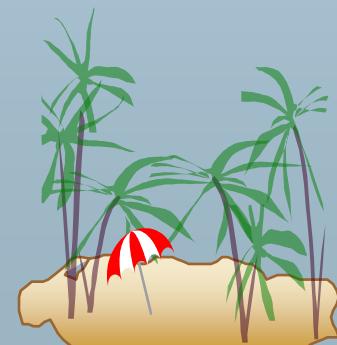
Adams  
Hayes





# Result of the Subexpression

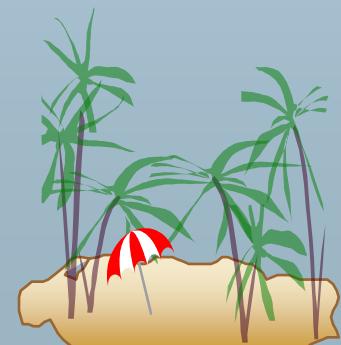
<i>balance</i>
500
400
700
750
350





# Largest Account Balance in the Bank

<i>balance</i>
900

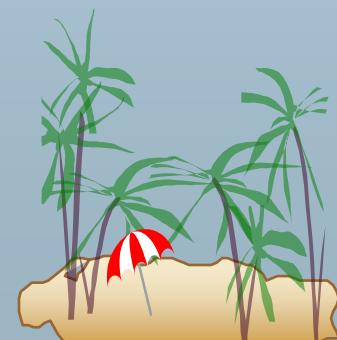




# Customers Who Live on the Same Street and In the Same City as Smith

*customer-name*

Curry  
Smith

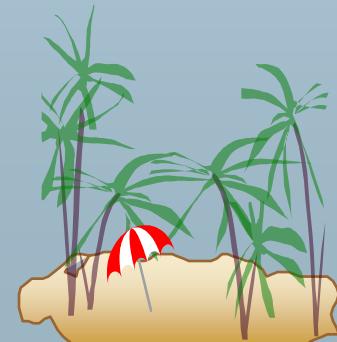




# Customers With Both an Account and a Loan at the Bank

*customer-name*

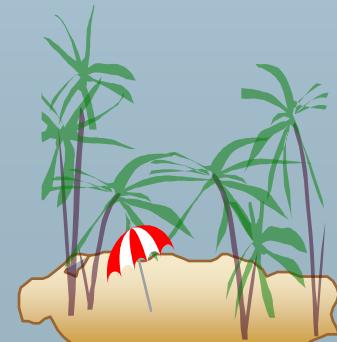
Hayes  
Jones  
Smith





# Result of $\Pi_{customer-name, loan-number, amount}$ **(borrower $\bowtie$ loan)**

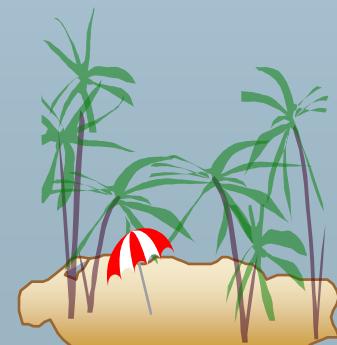
<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000





# Result of $\Pi_{branch-name}(\sigma_{customer-city = "Harrison"}(customer \bowtie account \bowtie depositor))$

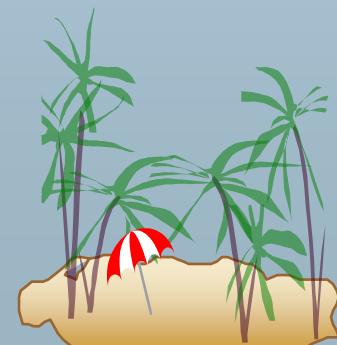
<i>branch-name</i>
Brighton
Perryridge





# Result of $\Pi_{branch-name}(\sigma_{branch-city = "Brooklyn"}(branch))$

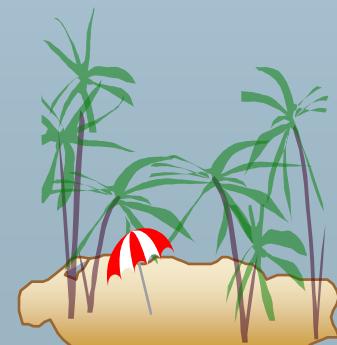
<i>branch-name</i>
Brighton
Downtown





## Result of $\Pi_{customer-name, branch-name}(depositor \bowtie account)$

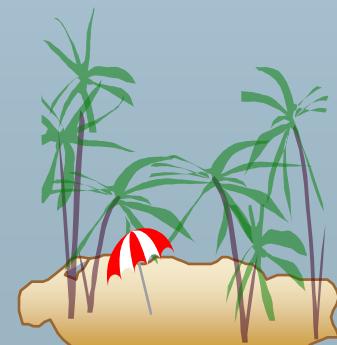
<i>customer-name</i>	<i>branch-name</i>
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill





# The *credit-info* Relation

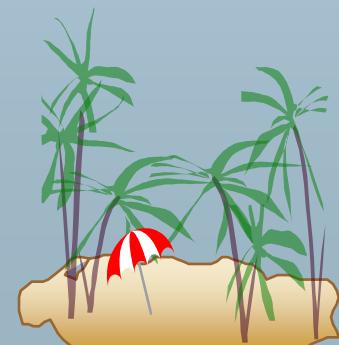
<i>customer-name</i>	<i>branch-name</i>
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill





# Result of $\Pi_{customer-name, (limit - credit-balance)} as credit-available(credit-info)$ .

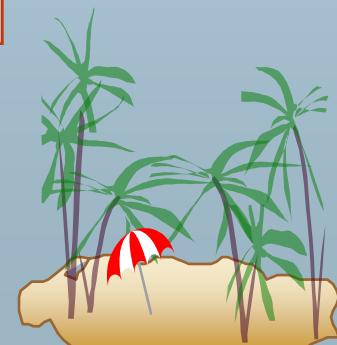
<i>customer-name</i>	<i>credit-available</i>
Curry	250
Jones	5300
Smith	1600
Hayes	0





# The *pt-works* Relation

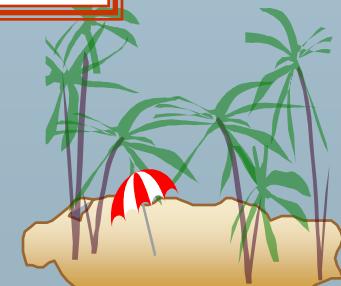
<i>employee-name</i>	<i>branch-name</i>	<i>salary</i>
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600





# The *pt-works* Relation After Grouping

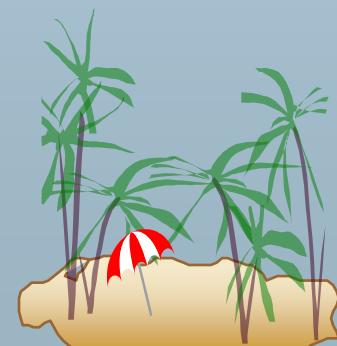
<i>employee-name</i>	<i>branch-name</i>	<i>salary</i>
Rao	Austin	1500
Sato	Austin	1600
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300





# Result of $\text{branch-name} \sum \text{sum}(\text{salary})$ (pt-works)

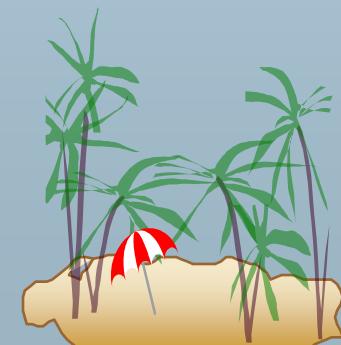
<i>branch-name</i>	<i>sum of salary</i>
Austin	3100
Downtown	5300
Perryridge	8100





# Result of $\text{branch-name} \sum \text{sum salary}, \max(\text{salary})$ as **max-salary (pt-works)**

<i>branch-name</i>	<i>sum-salary</i>	<i>max-salary</i>
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300



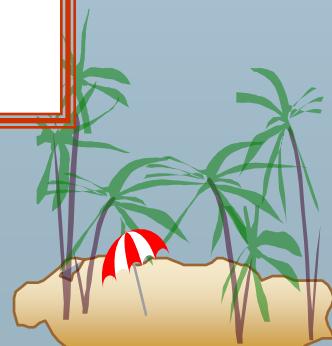


# The *employee* and *ft-works* Relations

<i>employee-name</i>	<i>street</i>	<i>city</i>
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

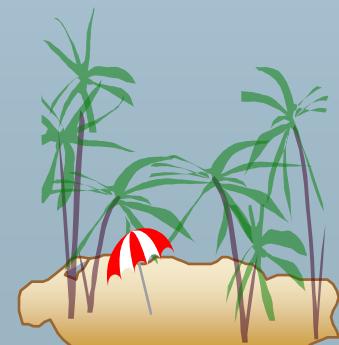
<i>employee-name</i>	<i>branch-name</i>	<i>salary</i>
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500





# The Result of *employee* $\bowtie$ *ft-works*

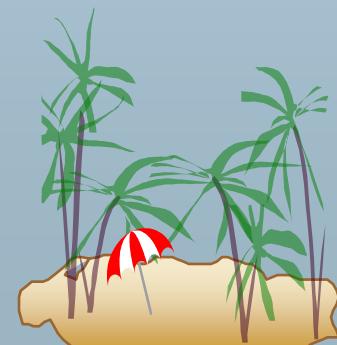
<i>employee-name</i>	<i>street</i>	<i>city</i>	<i>branch-name</i>	<i>salary</i>
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500





# The Result of *employee* $\bowtie$ *ft-works*

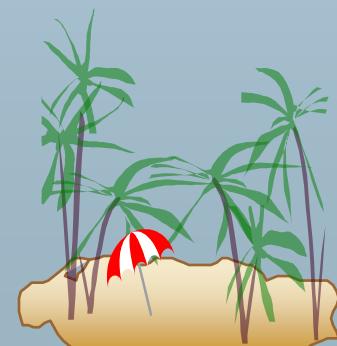
<i>employee-name</i>	<i>street</i>	<i>city</i>	<i>branch-name</i>	<i>salary</i>
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	<i>null</i>	<i>null</i>





# Result of employee $\bowtie$ ft-works

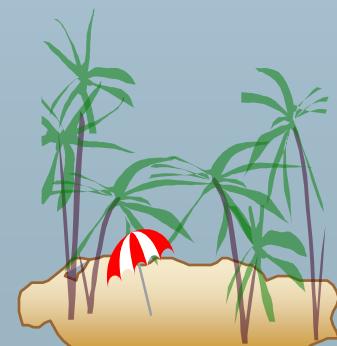
<i>employee-name</i>	<i>street</i>	<i>city</i>	<i>branch-name</i>	<i>salary</i>
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	<i>null</i>	<i>null</i>	Redmond	5300





# Result of employee $\bowtie$ ft-works

<i>employee-name</i>	<i>street</i>	<i>city</i>	<i>branch-name</i>	<i>salary</i>
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	<i>null</i>	<i>null</i>
Gates	<i>null</i>	<i>null</i>	Redmond	5300



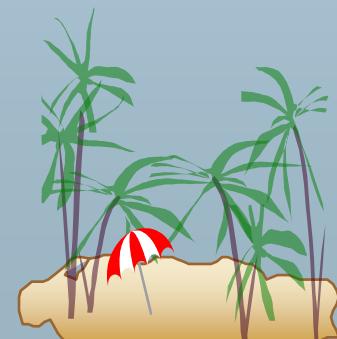


# Tuples Inserted Into *loan* and *borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
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
<i>null</i>	<i>null</i>	1900

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	<i>null</i>

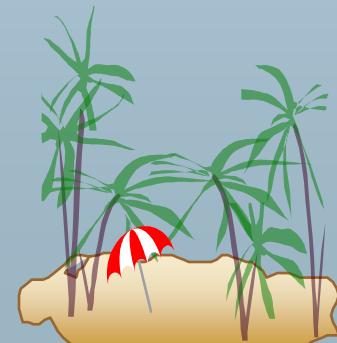




# Names of All Customers Who Have a Loan at the Perryridge Branch

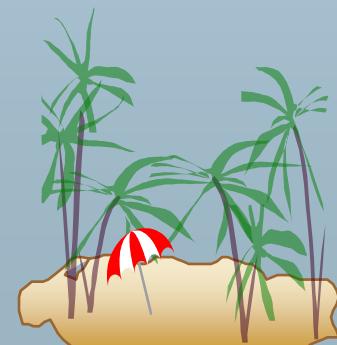
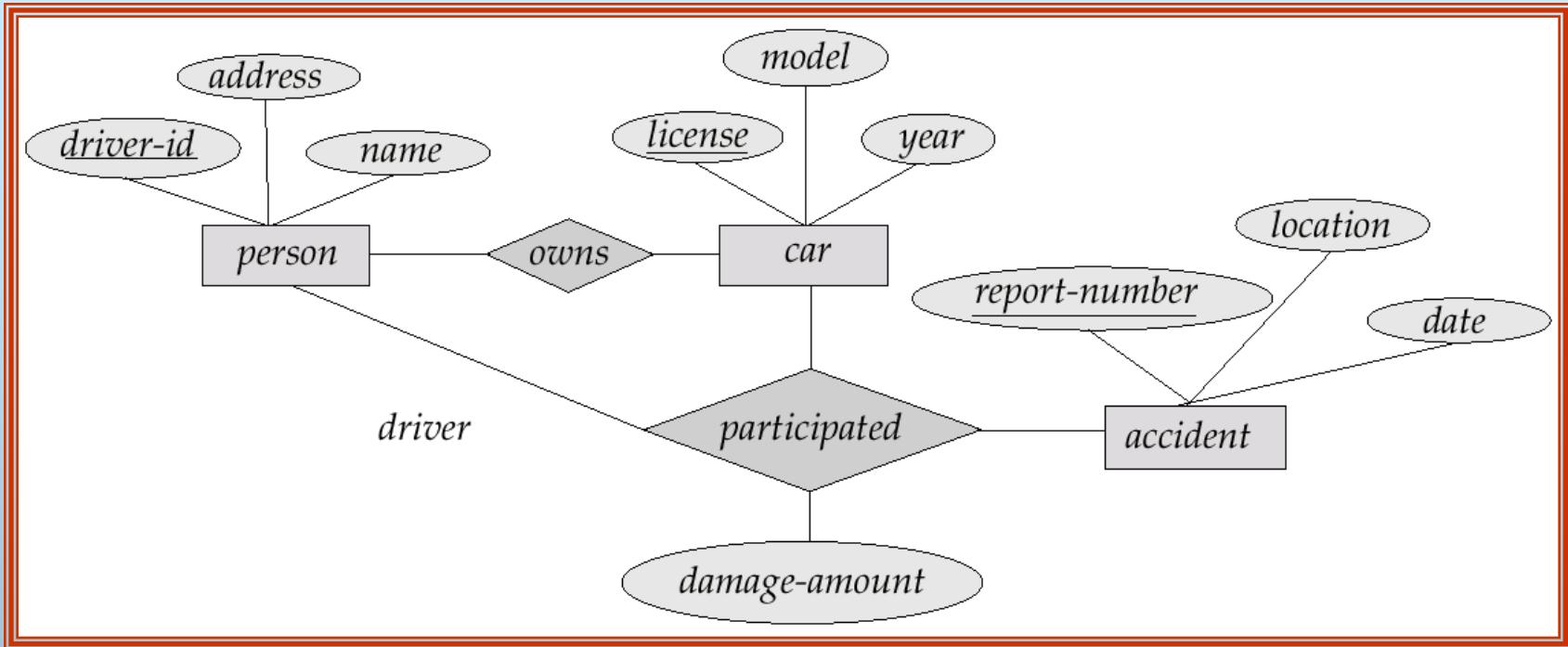
*customer-name*

Adams  
Hayes





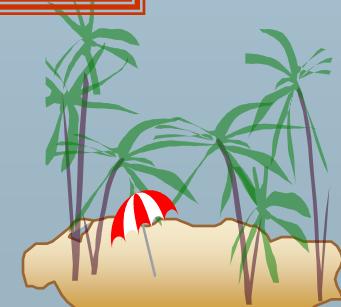
# E-R Diagram





# The *branch* Relation

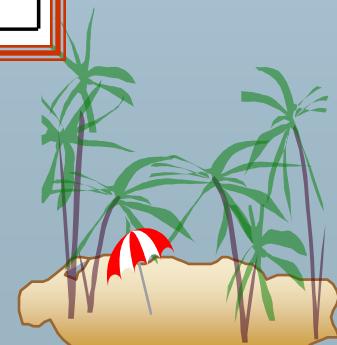
<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000





# The *loan* Relation

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
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





# The *borrower* Relation

<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

