Query Language

- A **query language** is a language in which a user requests information from the database.
- Query languages can be categorized as either **procedural** or **nonprocedural**.
- In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. For example, *relational algebra*.
- In a nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information. For example, *tuple relational calculus* and *domain relational calculus*.
- Most commercial relational database systems offer a query language that includes elements of both procedural and nonprocedural approaches.
- Example: *SQL* (*Structured Query Language*).

Relational Algebra

- It is a procedural query language
- It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The six fundamental operations are:
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename
- The *select*, *project*, and *rename* operations are called unary operators, because they operate on one relation. The remaining three operations are called binary operations, because they operate on pairs of relations.

Consider the following Relational Database

Here, we use the following relational database to illustrate the operations of relational algebra.

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)

Select Operation – Example

- Relation *r*

A	*B*	*C*	*D*
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

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

A	*B*	*C*	*D*
α	α	1	7
β	β	23	10

Select Operation

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

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

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: $=, \neq, >, \geq, <, \leq$

- Examples of selection:
To select those tuples of the *loan* relation where the branch is "Perryridge", we write

$\sigma_{\textit{branch-name = "Perryridge"}}(\textit{loan})$

We can find all tuples of the *loan* relation in which the amount lent is more than $1200 by writing

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

Select Operation

To find those tuples pertaining to *loans* of more than $1200 made by the “Perryridge” branch, we write

$$\sigma_{\textit{branch-name} = \text{“Perryridge”} \wedge \textit{amount} > 1200}(\textit{loan})$$

To find all customers who have the same name as their loan officer, we can write

$$\sigma_{\textit{customer-name} = \textit{banker-name}}(\textit{loan-officer})$$

Project Operation – Example

- Relation *r*:

A	*B*	*C*
α	10	1
α	20	1
β	30	1
β	40	2

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

A	*C*
α	1
α	1
β	1
β	2

=

A	*C*
α	1
β	1
β	2

Project Operation

- Notation:

$$\Pi_{A1,\ A2,\ \ldots,\ Ak}\ (r)$$

  where $A_1,..,A_k$ 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 are removed from result, since relations are sets
- E.g. To eliminate the *branch-name* attribute of *loan*

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

Union Operation – Example

- Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r \cup s$:

A	B
α	1
α	2
β	1
β	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 the name of all customers with either an account or a loan

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

Set Difference Operation – Example

- Relations *r, s:*

A	*B*
α	1
α	2
β	1

r

A	*B*
α	2
β	3

s

r – *s*:

A	*B*
α	1
β	1

Set Difference Operation

- Notation $r - s$
- Defined as:

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

- 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
- For example, we can find the name of all customers of a bank who have an account but not loan by writing

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

Cartesian-Product Operation-Example

Relations *r, s*:

A	*B*
α	1
β	2

r

C	*D*	*E*
α	10	a
β	10	a
β	20	b
γ	10	b

s

r x *s*:

A	*B*	*C*	*D*	*E*
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-Product Operation

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

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

- Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \varnothing$).
- If attributes of *r(R)* and *s(S)* are not disjoint, then renaming must be used.
- For example, suppose that we want to find the names of all customers who have a loan at the "Perryridge" branch. We need the information in both the *loan* and *borrower* relation to do so. For this we can write

$$\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)$$

The *customer-name* column may contain customers who do not have a loan at the "Perryridge" branch. If a customer has a loan in the "Perryridge" branch, then there is some tuple in *borrower* × *loan* that contains this name and *borrower.loan-number = loan.loan-number*. So, if we write,

Cartesian-Product Operation

$\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan))$

We can get only those tuples of *borrower*×*loan* that pertain to customers who have a loan at the “Perryridge” branch.

Finally, since we want only *customer-name*, we do a projection

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

Alternatively, we can also write

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

Composition of Operations

- Relational-algebra operations can be composed together into a relational-algebra expression.
- Example: $\sigma_{A=C}(r \times s)$
- $r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

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

A	B	C	D	E
α	1	α	10	a
β	2	β	20	a
β	2	β	20	b

Composition of Operations

For example, to find the name of those customers who live in "Harrison" city, we write

$\Pi_{\text{customer-name}} (\sigma_{\textit{customer-city} = \textit{"Harrison"}} (\textit{Customer}))$

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

$$\rho_{X(A1, A2, \ldots, An)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to $A1, A2, \ldots, An$.

Rename Operation

For example, to find the largest account balance, we write

$$\Pi_{balance}(account) - \Pi_{account.balance}$$

$$(\sigma_{account.balance < d.balance} (account \times \rho_{d} (account)))$$

To find the names of all customers who live on the same street and in the same city as Smith, we write

$$\Pi_{customer.customer\text{-}name}(\sigma_{customer.customer\text{-}street = smith\text{-}addr.street \wedge customer.customer\text{-}city = smith\text{-}addr.city} (customer \times \rho_{smith\text{-}addr(street,city)} (\Pi_{customer\text{-}street,\ customer\text{-}city} (\sigma_{customer\text{-}name = \text{“Smith”}} (customer)))))$$

Formal Definition of Relational Algebra

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation; that is by listing its tuples within { }, for example { (A-101, Downtown,500), (A-215, Mianus, 700) }
- 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 x E_2
  - $\sigma_p (E_1)$, P is a predicate on attributes in E_1
  - $\prod_s(E_1)$, 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 - Example

- Relation r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- r ∩ s

A	B
α	2

Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:
- $r \cap s = \{ t \mid t \in r \textbf{ and } t \in s \}$
- Assume:
  - r, s have the *same arity*
  - attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$
- For example, to find the name of all customers who have both a loan and an account, we can write

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

Natural-Join Operation

- The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It performs Cartesian product and a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.
- For example, to find the names of all customers who have a loan at the bank, along with the loan number and the loan amount, we can use Cartesian product as follows:

$$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower\ x\ loan))$$

but we can express the above query by using natural join as follows:

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

Note: $r \bowtie s = r\ x\ s$ if $R \cap S = \phi$

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:

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

Natural Join Operation – Example

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	$\in$

s

$r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Other Examples

- To find the names of all branches with customers who have an account in the bank and who live in Harrison, we write

  $\Pi_{branch\text{-}name}(\sigma_{customer\text{-}city = \text{“Harrison”}}(customer \bowtie account \bowtie depositor)$

- To find all customers who have both a loan and an account at the bank, we write

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

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
  - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - $S = (B_1, \ldots, B_n)$

  The result of r ÷ s is a relation on schema

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

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

Division Operation – Example

Relations *r, s*:

A	*B*
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
ϵ	6
ϵ	1
β	2

r

B
1
2

s

r ÷ *s*:

A
α
β

Another Division Example

Relations *r, s*:

A	*B*	*C*	*D*	*E*
α	a	α	a	*1*
α	a	γ	a	*1*
α	a	γ	b	*1*
β	a	γ	a	*1*
β	a	γ	b	*3*
γ	a	γ	a	*1*
γ	a	γ	b	*1*
γ	a	β	b	*1*

r

D	*E*
a	*1*
b	*1*

s

$r \div s$:

A	*B*	*C*
α	a	γ
γ	a	γ

Division Operation (Cont.)

- For example, to find the name of all customers who have an account at *all* branches located in Brooklyn, we write

$$\prod_{customer\text{-}name,\ branch\text{-}name} (depositor \bowtie account) \div \prod_{branch\text{-}name} (\sigma_{branch\text{-}city = \text{“Brooklyn”}} (branch))$$

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

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

  - 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.

Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions
- Outer Join

Generalized Projection

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

$$\Pi_{F1, F2, \ldots, Fn}(E)$$

- E is any relational-algebra expression
- Each of F_1, F_2, …, F_n are arithmetic expressions involving constants and attributes in the schema of E.
- For example, given relation *credit-info(customer-name, limit, credit-balance),* find how much more each person can spend:

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

The attribute resulting from the expression *limit - credit-balance* does not have a name and we can give it name as follows:

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

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

$$_{G1, G2, \ldots, Gn}\mathcal{g}_{F1(A1), F2(A2), \ldots, Fn(An)}(E)$$

  - E is any relational-algebra expression
  - $G_1, G_2 \ldots, G_n$ 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*
α	α	7
α	β	7
β	β	3
β	β	10

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

sum-C
27

Aggregate Operation – Example

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

branch-name	*account-number*	*balance*
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$$branch\text{-}name\ \mathcal{G}\ sum(balance)\ (account)$$

branch-name	*balance*
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

    $branch\text{-}name\ \mathcal{G}\ \mathbf{sum}(balance)\ \mathbf{as}\ sum\text{-}balance,\ \mathbf{max}(balance)\ \mathbf{as}\ max\text{-}balance\ (account)$
- There are cases where we must eliminate multiple occurrences of a value before computing an aggregate function. For this, we use the function name with the addition of the hyphenated string **"distinct"** appended to the end of the function name (for example, **count-distinct**). For example, to find the number of branches appearing in the *pt-works* relation, we write

  $\mathcal{G}\ \mathbf{count\text{-}distinct}(branch\text{-}name)(pt\text{-}works)$

Outer Join

- An extension of the natural join operation that avoids loss of information.
- Computes the natural join and then adds tuples form one relation that does not match tuples in the other relation to the result of the natural join.
- Uses *null* values; *null* signifies that the value is unknown or does not exist
- There are three forms of outer join: ***left outer join***, ***right outer join*** and ***full outer join***.
- The *left outer join* takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation and adds them to the result of the natural join.
- Similarly, *right outer join* takes all tuples in the right relation that did not match with any tuple in the left relation, pads the tuples with null values for all other attributes from the left relation and adds them to the result of the natural join.
- The *full outer join* does both of the *left outer join* and *right outer join* operations.

Outer Join – Example

- 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

Outer Join – Example

- **Inner Join**

  loan ⋈ *Borrower*

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

- **Left Outer Join**

  loan ⟕ *Borrower*

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

Outer Join – Example

- **Right Outer Join**

  loan ⟖ *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

- **Full Outer Join**

  loan ⟗ *borrower*

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

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.

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

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

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

- Delete all accounts at branches located in Needham.

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

Insertion

- To insert data into a relation, we either:
  - specify a tuple to be inserted or
  - 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.

  $account \leftarrow account \cup \{(\text{“Perryridge”}, \text{A-973}, 1200)\}$

  $depositor \leftarrow depositor \cup \{(\text{“Smith”}, \text{A-973})\}$

- 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.

  $r_1 \leftarrow (\sigma_{branch\text{-}name = \text{“Perryridge”}} (borrower \bowtie loan))$

  $account \leftarrow account \cup \prod_{loan\text{-}number,\ branch\text{-}name,200} (r_1)$

  $depositor \leftarrow depositor \cup \prod_{customer\text{-}name,\ loan\text{-}number} (r_1)$

Updating

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

$$r \leftarrow \prod_{F1,\, F2,\, \ldots,\, Fl,} (r)$$

- Each F_i is either
  - the ith attribute of r, if the ith 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

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

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). Security considerations may require that certain data be hidden from users.
- 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

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

- 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 the definition of the view itself, rather than the result of evaluation the relational algebra expression that defines the view; whenever a view relation appears in a query, it is replaced by the stored query expression.

View Examples

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

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

$$\Pi_{\textit{branch-name, customer-name}} (\textit{depositor} \bowtie \textit{account})$$
$$\cup \Pi_{\textit{branch-name, customer-name}} (\textit{borrower} \bowtie \textit{loan})$$

- Using the view *all-customer*, we can find all customers of the Perryridge branch by writing:

$$\Pi_{\textit{customer-name}} (\sigma_{\textit{branch-name} = \text{“Perryridge”}} (\textit{all-customer}))$$

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**

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

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

$$\textit{branch-loan} \leftarrow \textit{branch-loan} \cup \{(\text{“Perryridge”, L-37})\}$$

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

    $v \leftarrow v \cup$ (L-99, 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