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

Basic Structure

- Given sets $A_1, A_2, ..., A_n$ a *relation r* is a subset of
  $A_1 \times A_2 \times ... \times A_n$
  Thus a relation is a *set of n-tuples* $(a_1, a_2, ..., a_n)$ where
  $a_i \in A_i$
- Example: If

  customer-name = {Jones, Smith, Curry, Lindsay}
  customer-street = {Main, North, Park}
  customer-city = {Harrison, Rye, Pittsfield}

  Then *r* = {(Jones, Main, Harrison), (Smith, North, Rye), (Curry, North, Rye), (Lindsay, Park, Pittsfield)} is a relation over
  customer-name × *customer-street* × *customer-city*

Relation Schema

- $A_1, A_2, ..., A_n$ are *attributes*
- $R = (A_1, A_2, ..., A_n)$ is a *relation schema*

  Customer-schema = (*customer-name, customer-street, customer-city*)

- $r(R)$ is a *relation* on the relation schema R

  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.

customer-name	*customer-street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield

customer

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-many relationship sets, above super key is also the primary key.

  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.

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 two or more relations as inputs and give a new relation as a result.

Select Operation

- Notation: $\sigma_P(r)$
- Defined as:

$$\sigma_P(r) = \{t \mid t \in r \textbf{ and } P(t)\}$$

Where P is a formula in propositional calculus, dealing with terms of the form:

$$\begin{array}{ccl} \langle\text{attribute}\rangle & = & \langle\text{attribute}\rangle \text{ or } \langle\text{constant}\rangle \\ & \neq & \\ & > & \\ & \geq & \\ & < & \\ & \leq & \end{array}$$

"connected by": $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)

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

Project Operation

- Notation:

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

  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

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
$\cancel{\alpha}$	$\cancel{1}$
β	1
β	2

=

A	C
α	1
β	1
β	2

Union Operation

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

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

- 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)

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

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

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

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

Cartesian-Product Operation – Example

- Relations *r*, *s*:

A	*B*
α	1
β	2

r

C	*D*	*E*
α	10	+
β	10	+
β	20	–
γ	10	–

s

- $r \times s$

A	*B*	*C*	*D*	*E*
α	1	α	10	+
α	1	β	10	+
α	1	β	20	–
α	1	γ	10	–
β	2	α	10	+
β	2	β	10	+
β	2	β	20	–
β	2	γ	10	–

Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{A=C}(r \times s)$
- $r \times s$
  - Notation: $r \bowtie s$
  - Let r and s be relations on schemas R and S respectively. The result is a relation on schema $R \cup S$ which is obtained by considering 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 $R \cap S$, a tuple t is added to the result, where
    - t has the same value as t_r on r
    - t has the same value as t_s on s

Composition of Operations (Cont.)

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*:

r

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

s

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ϵ

- $r \bowtie s$

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

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 $r \div s$ is a relation on schema $R - S = (A_1, \ldots, A_m)$

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

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 $u \in s$, $tu \notin r$.

Division Operation – Example

- Relations *r*, *s*:

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

r

B
1
2

s

- $r \div s$

A
α
ϵ

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

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 the result of the query.
- Assignment must always be made to a temporary relation variable.
- Example: Write $r \div s$ as

$$
\begin{aligned}
temp1 &\leftarrow \Pi_{R-S}\,(r) \\
temp2 &\leftarrow \Pi_{R-S}\,((temp1 \times s) - \Pi_{R-S,S}(r)) \\
result &= temp1 - temp2
\end{aligned}
$$

  - 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 "Uptown" branches.
  - Query 1

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

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

  - Query 2

$$\Pi_{\textit{customer-name, branch-name}}\ (\textit{depositor} \bowtie \textit{account})$$
$$\div\ \rho_{\textit{temp}(\textit{branch-name})}(\{\ (\text{"}\textit{Downtown}\text{"}),\ (\text{"}\textit{Uptown}\text{"})\ \})$$

Example Queries

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

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

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 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 (branch-name, account-number, balance)

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

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Example Queries

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

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

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

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

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

Example Queries

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

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

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

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

Example Queries

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

$$\{t \mid \exists s \in \textit{borrower}(t[\textit{customer-name}] = s[\textit{customer-name}]$$
$$\land \exists u \in \textit{loan}(u[\textit{branch-name}] = \text{“Perryridge”}$$
$$\land u[\textit{loan-number}] = s[\textit{loan-number}]))\}$$

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

$$\{t \mid \exists s \in \textit{borrower}(t[\textit{customer-name}] = s[\textit{customer-name}]$$
$$\land \exists u \in \textit{loan}(u[\textit{branch-name}] = \text{“Perryridge”}$$
$$\land u[\textit{loan-number}] = s[\textit{loan-number}])$$
$$\land \not\exists v \in \textit{depositor}\ (v[\textit{customer-name}] = t[\textit{customer-name}]\}$$

Example Queries

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

$$\{t \mid \exists\, s \in \textit{loan}\, (s[\textit{branch-name}] = \text{“Perryridge”}$$
$$\quad \wedge\, \exists\, u \in \textit{borrower}\, (u[\textit{loan-number}] = s[\textit{loan-number}]$$
$$\qquad \wedge\, t[\textit{customer-name}] = u[\textit{customer-name}]$$
$$\quad \wedge\, \exists\, v \in \textit{customer}\, (u[\textit{customer-name}] = v[\textit{customer-name}]$$
$$\qquad \wedge\, t[\textit{customer-city}] = v[\textit{customer-city}])))\}$$

Example Queries

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

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

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

Domain Relational Calculus

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

$$\{< x_1,\ x_2,\ ...,\ x_n > \ | \ P(x_1,\ x_2,\ ...,\ x_n)\}$$

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

Example Queries

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

$$\{< b, l, a > \mid < b, l, a > \in \textit{loan} \land a > 1200\}$$

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

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

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

$$\{< c, a > \mid \exists\, l\, (< c, l > \in \textit{borrower} \land \exists\, b\, (< b, l, a > \in \textit{loan} \land b = \text{“Perryridge”}))\}$$

Example Queries

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

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

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

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

Safety of Expressions

$$\{< x_1,\ x_2, ..., x_n > \ | \ P\,(x_1,\ x_2, ..., x_n)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *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 x in *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 *dom*(P_1).

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.

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

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

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

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 **false** by definition.

Outer Join – Example

- Relation *loan*

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

- Relation *borrower*

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

Outer Join – Example

- *loan* $\bowtie$ *Borrower*

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

- *loan* ⟕ *borrower*

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

Outer Join – Example

- *loan* ⟕ *Borrower*

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

- *loan* ⟗ *borrower*

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

Aggregate Functions

- Aggregation operator $\mathcal{G}$ 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

$$_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1\ A_1,\ F_2\ A_2, \ldots,\ F_m\ A_m}(E)$$

  - E is any relational-algebra expression
  - $G_1, G_2, \ldots, G_n$ is a list of attributes on which to group
  - F_i is an aggregate function
  - A_i is an attribute name

Aggregate Function – Example

- Relation r:

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

- $\mathbf{sum}_C(r)$

sum-C
27

Aggregate Function – 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

- $_{\textit{branch-name}}\mathcal{G}_{\textbf{sum}\ \textit{balance}}(\textit{account})$

branch-name	*sum-balance*
Perryridge	1300
Brighton	750
Redwood	700

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 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
  - 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.

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

- 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.

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

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

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(r)$$

  - 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.

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

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

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

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

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.)
- 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 part 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 epression. Rather, a view definition causes the saving of an expression to be 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**

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

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

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

Updates Through Views

- 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 (“Perryridge”, L-37, *null*) into the *loan* relation

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 and only if there is a path in the dependency graph from v_2 to v_1.
- 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.