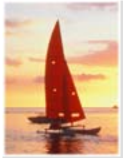

Relational Model

Database System Concepts, 5th Ed.

Chapter 2: Relational Model

- Structure of Relational Databases
- Fundamental Relational-Algebra-Operations
- Additional Relational-Algebra-Operations
- Extended Relational-Algebra-Operations
- Modification of the Database

Example of a Relation

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

Basic Structure

- Formally, given sets D_1, D_2, …. D_n a **relation** *r* is a subset of

  $D_1 \times D_2 \times \ldots \times D_n$

  Thus, a relation is a set of *n*-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$

- Example: If
  - *customer_name* = {Jones, Smith, Curry, Lindsay, …}
    /* Set of all customer names */
  - *customer_street* = {Main, North, Park, …} /* set of all street names*/
  - *customer_city* = {Harrison, Rye, Pittsfield, …} /* set of all city names */

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

  is a relation over

  customer_name x customer_street x 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. the value of an attribute can be an account number, but cannot be a set of account numbers
- Domain is said to be atomic if all its members are 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

- $A_1, A_2, \ldots, A_n$ are *attributes*
- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*
  - Ordering of attributes is important!

  Example:

  Customer_schema = (*customer_name, customer_street, customer_city*)
- *r*(*R*) denotes a *relation r* on the *relation schema R*

  Example:

  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

attributes (or columns)

customer_name	*customer_street*	customer_city
Jones	Main	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield

tuples (or rows)

customer

Relations are Unordered

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

account_number	*branch_name*	*balance*
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

  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., if two customers own an account (What gets repeated?)
  - the need for null values
    - e.g., to represent a customer without an account
- Normalization theory (Chapter 7: Relational Database Design) deals with how to design relational schemas

The *customer* Relation

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

The *depositor* Relation

customer_name	*account_number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Keys

- Let K ⊆ 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
    - In real life, an attribute such as *customer_id* would be used instead of *customer_name* to uniquely identify customers, but we omit it to keep our examples small, and instead assume customer names are unique.

Keys (Cont.)

- *K* is a **candidate key** if *K* is minimal
  Example: {*customer_name*} is a candidate key for *Customer*, since it is a superkey and no subset of it is a superkey.
- **Primary key:** a candidate key chosen as the principal means of identifying tuples within a relation
  - Should choose an attribute whose value never, or very rarely, changes.
  - E.g. email address is unique, but may change

Query Languages

- Language in which user requests information from the database.
- Categories of languages
  - Procedural
  - Non-procedural, or declarative
- “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: σ
  - project: $\prod$
  - union: $\cup$
  - set difference: –
  - Cartesian product: ×
  - rename: ρ
- The operators take one or two relations as inputs and produce a new relation as a result.

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:

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

where p is a formula in *propositional calculus*:

formula := term
term <conj> term
(term)

term := expr
expr <op> expr
(expr)

expr := attribute
constant

<conj> is one of: $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)

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

- Example of selection:

$$\sigma_{branch\_name=\text{'Perryridge'}}(account)$$

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:

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

  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
- Example: To eliminate the *branch_name* attribute of *account*

$$\Pi_{account\_number,\ balance}\ (account)$$

Union Operation – Example

- Relations *r, s:*

r

A	*B*
α	1
α	2
β	1

s

A	*B*
α	2
β	3

- r ∪ 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** (example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)
- Example: 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
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

Cartesian-Product Operation – Example

- Relations *r, s*:

r

A	B
α	1
β	2

s

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

- $r \times s$

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

Cartesian-Product Operation

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

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

  where *tq* means the concatenation of tuples *t* and *q* to produce a single tuple.
- 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

A	B
α	1
β	2

s

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

- $r \times s$

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	10	a
β	2	β	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 of naming a relation:

$$\rho_X(E)$$

returns the expression *E* under the name *X*

- Example of naming a relation and its attributes:

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

$$\rho_{x(A_1, A_2, \ldots, A_n)}(E)$$

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

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

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower (customer_name, loan_number)

- Find all loans of over $1200

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

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

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

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

$$\Pi_{customer\_name}\ (borrower) \cup \Pi_{customer\_name}\ (depositor)$$

Example Queries

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower (customer_name, loan_number)

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

$$\Pi_{customer\_name}\ (\sigma_{branch\_name=\text{"Perryridge"}}\ (\sigma_{borrower.loan\_number\ =\ loan.loan\_number}\ (borrower \times loan)))$$

- 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.

$$\Pi_{customer\_name}\ (\sigma_{branch\_name\ =\ \text{"Perryridge"}}\ (\sigma_{borrower.loan\_number\ =\ loan.loan\_number}(\text{borrower} \times \text{loan})))\ -\ \Pi_{customer\_name}(\text{depositor})$$

Example Queries

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

- Query 1

$$\Pi_{customer\_name}\,(\sigma_{branch\_name = \text{“Perryridge”}}\,(\sigma_{borrower.loan\_number = loan.loan\_number}\,(borrower \times loan)))$$

- Query 2

$$\Pi_{customer\_name}(\sigma_{loan.loan\_number = borrower.loan\_number}\,((\sigma_{branch\_name = \text{“Perryridge”}}\,(loan)) \times borrower))$$

Example Queries

account (account_number, branch_name, balance)

- Find the largest account balance
  - Strategy:
    - Find those balances that are *not* the largest
      - Rename *account* relation as *d* so that we can compare each account balance with all others
    - Use set difference to find those account balances that were *not* found in the earlier step.
  - The query is:

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

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
  - $\prod_S(E_1)$, 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

- 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
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cap s$

A	B
α	2

Natural-Join Operation

- Notation: r ⋈ s
- Let *r* and *s* be relations on schemas *R* and *S* respectively.
  Then, r ⋈ 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* ⋈ *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*
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

Division Operation

- Notation: $r \div s$
- 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

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

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

  where *tu* means the concatenation of tuples *t* and *u* to produce a single tuple.

Division Operation – Example

- Relations *r, s*:

r

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
$\in$	6
$\in$	1
β	2

s

B
1
2

- $r \div s$

A
α
β

Another Division Example

- Relations *r, s*:

r

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*

s

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

- $r \div s$

A	*B*	*C*
α	a	γ
γ	a	γ

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

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

  - 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.

Bank Example Queries

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

$$\Pi_{customer\_name}\ (borrower) \cap \Pi_{customer\_name}\ (depositor)$$

- Find the name of all customers who have a loan at the bank and the loan amount

$$\Pi_{customer\_name,\ loan\_number,\ amount}\ (borrower \bowtie loan)$$

Bank Example Queries

- Find all customers who have an account from at least the “Downtown” and the Uptown” branches.
  - Query 1

$$\Pi_{customer\_name}\,(\sigma_{branch\_name\,=\,\text{“Downtown”}}\,(depositor \bowtie account\,)) \cap$$

$$\Pi_{customer\_name}\,(\sigma_{branch\_name\,=\,\text{“Uptown”}}\,(depositor \bowtie account))$$

  - Query 2

$$\Pi_{customer\_name,\ branch\_name}\,(depositor \bowtie account)$$
$$\div\ \rho_{\ temp(branch\_name)}\,(\{(\text{“Downtown”}),\ (\text{“Uptown”})\})$$

Note that Query 2 uses a constant relation.

Bank Example Queries

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

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

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.

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

- *E* is any relational-algebra expression
- Each of F_1, F_2, …, F_n are 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:

$$\prod_{customer\_name,\ limit\ -\ credit\_balance} (credit\_info)$$

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 $\mathcal{G}$ algebra

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

  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

- $\mathcal{G}_{\textbf{sum}(C)}(r)$

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

$$_{branch\_name}\,\mathcal{G}_{\,\mathbf{sum}(balance)}\,(account)$$

branch_name	**sum**(*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

    $$\rho_{x(branch\_name, sum\_balance)} \left({}_{branch\_name} \mathcal{G}_{\textbf{sum}(balance)} (account) \right)$$

  - For convenience, we permit renaming as part of aggregate operation

    $${}_{branch\_name} \mathcal{G}_{\textbf{sum}(balance) \textbf{ as } sum\_balance} (account)$$

Outer Join

- An extension of the join operation that avoids loss of information.
- Example of natural join:

loan

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

loan ⋈ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Outer Join (cont.)

- Computes the join and then adds tuples form one relation that does 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.
    - We shall study precise meaning of comparisons with nulls later

Left Outer Join – Example

- Left Outer Join

loan

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

loan ⟕ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*

Right Outer Join – Example

- Right Outer Join

loan

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

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 – Example

- Full Outer Join

loan

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

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.

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

- 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.

branch (*branch_name, branch_city, assets*)
account (*account_number, branch_name, balance*)
depositor (*customer_name, account_number*)

$$r_1 \leftarrow \sigma_{branch\_city = \text{“Needham”}} (account \bowtie branch)$$

$$r_2 \leftarrow \Pi_{account\_number,\ branch\_name,\ balance} (r_1)$$

$$r_3 \leftarrow \Pi_{customer\_name,\ account\_number} (r_2 \bowtie \text{depositor})$$

$$account \leftarrow \text{account} - r_2$$

$$depositor \leftarrow \text{depositor} - r_3$$

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.

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

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

- 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.

account (*account_number, branch_name, balance*)
loan (*loan_number, branch_name, amount*)
depositor (*customer_name, account_number*)
borrower (*customer_name, loan_number*)

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

$$\textit{account} \leftarrow \textit{account} \cup \Pi_{loan\_number,\ branch\_name,\ 200}\,(r_1)$$

$$\text{depositor} \leftarrow \textit{depositor} \cup \Pi_{customer\_name,\ loan\_number}\,(r_1)$$

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

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

- 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

account (*account_number, branch_name, balance*)

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

$$account \leftarrow \Pi_{account\_number,\ branch\_name,\ balance*1.05}\ (account)$$

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

$$account \leftarrow \Pi_{account\_number,\ branch\_name,\ balance*1.06}\ (\sigma_{balance > 10000}\ (account\ ))$$
$$\cup\ \Pi_{account\_number,\ branch\_name,\ balance*1.05}\ (\sigma_{balance \leq 10000}\ (account))$$

Views

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

$$\Pi_{customer\_name,\ loan\_number}\ (borrower \bowtie loan)$$

- A **view** provides a mechanism to hide certain data from the view of certain users.
- 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 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.
- When a view is created, the *query expression* is stored in the database; the expression is substituted into queries using the view.
  - So view is not the same as creating a new relation by evaluation the query expression.

Example Queries

- A view consisting of branches and their customers

create view *all_customer* **as**

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

- Find all customers of the Perryridge branch

$$\Pi_{customer\_name}(\sigma_{branch\_name = \text{'Perryridge'}}(all\_customer))$$

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

Update of a 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 *loan_branch* **as**

  $$\Pi_{loan\_number,\ branch\_name}\,(loan)$$

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

  $$loan\_branch \leftarrow loan\_brach \cup \{(\text{'L-37'}, \text{'Perryridge'})\}$$

Update of a View (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 the tuple

    ('L-37', 'Perryridge', *null*)

    into the *loan* relation.

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 (v), 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*

Example Queries

- *loan* (*loan_number, branch_name, amount*)

- Find the *loan_number, branch_name,* and *amount* for loans of over $1200

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

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

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

  - Notice that a relation on schema (*loan_number*) is implicitly defined by the query.

- *Relation schema* of an expression is determined by either of:
  - If $\boldsymbol{t} \in \boldsymbol{r}$ is present in the expression, the resulting schema is of ***r***
  - Otherwise the resulting schema is determined by all attributes of ***t*** used in the expression.
    - Note: If t[A] is used more than once, the attribute A is in the relation schema just once!!!

Example Queries

- *depositor* (*customer_name, account_number*)
- *borrower* (*customer_name, loan_number*)

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

$$\{t \mid \exists s \in borrower\ (\ t\ [customer\_name\ ] = s\ [customer\_name\ ]) \\ \quad \vee \exists u \in depositor\ (\ t\ [customer\_name\ ] = u\ [customer\_name\ ])\ \}$$

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

$$\{t \mid \exists s \in borrower\ (\ t\ [customer\_name\ ] = s\ [customer\_name\ ]) \\ \quad \wedge \exists u \in depositor\ (\ t\ [customer\_name\ ] = u\ [customer\_name]\ )\ \}$$

Example Queries

- *loan* (*loan_number, branch_name, amount*)
- *depositor* (*customer_name, account_number*)
- *borrower* (*customer_name, loan_number*)

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

$$\{t \mid \exists s \in borrower\ (t\,[customer\_name\,] = s\,[customer\_name\,]$$
$$\wedge\ \exists u \in loan\ (u\,[branch\_name\,] = \text{“Perryridge”}$$
$$\wedge\ u\,[loan\_number\,] = s\,[loan\_number\,]\,)\,)\,\}$$

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

$$\{t \mid \exists s \in borrower\ (t\,[customer\_name\,] = s\,[\text{customer\_name}\,]$$
$$\wedge\ \exists u \in loan\ (u\,[branch\_name\,] = \text{“Perryridge”}$$
$$\wedge\ u\,[loan\_number\,] = s\,[\text{loan\_}number\,]\,)\,)$$
$$\wedge\ \neg\ \exists v \in depositor\ (v\,[customer\_name\,] =$$
$$t\,[customer\_name\,]\,)\,\}$$

Example Queries

- *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*)

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

$$
\begin{aligned}
\{t \mid \exists s \in loan\ (&s[branch\_name] = \text{“Perryridge”} \\
&\wedge \exists u \in borrower\ (u[loan\_number] = s[loan\_number] \\
&\qquad \wedge\ t[customer\_name] = u[customer\_name]) \\
&\wedge \exists\ v \in customer\ (u[customer\_name] = v[customer\_name] \\
&\qquad \wedge\ t[customer\_city] = v[customer\_city]))\}
\end{aligned}
$$

Example Queries

- *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*)

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

$$
\begin{aligned}
&\{t \mid \exists\, r \in customer\ (t\,[customer\_name\,] = r\,[customer\_name\,]) \land \\
&\quad (\forall\, u \in branch\ (u\,[branch\_city\,] = \text{“Brooklyn”} \Rightarrow \\
&\qquad \exists\, s \in depositor\ (r\,[customer\_name\,] = s\,[customer\_name\,] \\
&\qquad \land \exists\, w \in account\ (w[account\_number\,] = s\,[account\_number\,] \\
&\qquad \land (w\,[branch\_name\,] = u\,[branch\_name\,]\,)\,)\,) \\
&\quad)\,)\,\}
\end{aligned}
$$

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. $\{ t \mid t[A] = 5 \lor \textbf{true} \}$ 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:

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

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

Example Queries

- *loan* (*loan_number, branch_name, amount*)
  depositor (*customer_name, account_number*)
  borrower (*customer_name, loan_number*)

- Find the *loan_number, branch_name,* and *amount* for loans of over $1200
  - $\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 1200\}$
- Find the names of all customers who have a loan of over $1200
  - $\{< c > \mid \exists\ l, b, a\ (< c, l > \in borrower \wedge < l, b, a > \in loan \wedge a > 1200)\}$
- Find the names of all customers who have a loan at the Perryridge branch and the loan amount:
  - $\{< c, a > \mid \exists\ l\ (< c, l > \in borrower \wedge \exists b\ (< l, b, a > \in loan \wedge$
    $b = \text{“Perryridge”}))\}$
  - $\{< c, a > \mid \exists\ l\ (< c, l > \in borrower \wedge < l, \text{“Perryridge”}, a > \in loan)\}$

Example Queries

- *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*)
- Find the names of all customers having a loan, an account, or both at the Perryridge branch:
  - $\{< c > \mid \exists\, l\, (< c, l > \in borrower$
    $\wedge \exists\, b,a\, (< l, b, a > \in loan \wedge b = \text{“Perryridge”}))$
    $\vee \exists\, a\, (< c, a > \in depositor$
    $\wedge \exists\, b,n\, (< a, b, n > \in account \wedge b = \text{“Perryridge”}))\}$
- Find the names of all customers who have an account at all branches located in Brooklyn:
  - $\{< c > \mid \exists\, \text{s},t\, (< c, s, t > \in customer) \wedge$
    $\forall\, x,y,z\, (< x, y, z > \in branch \wedge y = \text{“Brooklyn”}) \Rightarrow$
    $\exists\, a,b\, (< a, x, b > \in account \wedge < c,a > \in depositor)\}$

Safety of Expressions

The expression:

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

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 of *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).