

3. Relational Model

3.1 Objectives

After going through this unit, you should be able to:

- Describe relational model and its advantages
- Perform basic operations using relational algebra

3.2 Introduction

- A model in database system basically defines the structure or organization of data and a set of operation on that data.
- Relational model is a simple model in which database is represented as a collection of ***Relation*** where each relation is represented by a two dimensional table.
- The relational model was first proposed by E.F. Codd in 1970 and the first such systems (notably INGRES and System/R) was developed in 1970s.
- Because of its simplicity, The relational model is now the dominant model for commercial data processing applications.

RollNo	Name	Age	Address
1	Sanjay Paudel	25	New Baneshwor
2	Rita Shrestha	22	Kalanki
3	Bikash Sharma	24	Thamel

Figure 1: A Sample Student Relation

Salient features of a relational table

- Values are atomic
- Column values are of the same kind (Domain)
- Each Row is unique (Primary Key)
- Sequence of columns is insignificant
- Sequence of rows is insignificant
- Each column must have a unique name
- Relationships and Keys
 - Keys - Fundamental to the concept of relational databases
 - Relationship - An association between two or more tables defined by means of keys

Relational Model [Properties]

- Each relation (or table) in a database has a unique name
- An entry at the intersection of each row and column is atomic (or single-valued); there can be no multi-valued attributes in a relation
- Each row is unique; no two rows in a relation are identical
- Each attribute (or column) within a table has a unique name

Relational Model [Advantages]

Following are some of the advantages of relational model

- **Ease of use**

The simple tabular representation of database helps the user defined and query the database conveniently. For example, we can easily find out the age of student whose name are starts with character “B”

- **Flexibility**

Since the database is a collection of tables, new data can be added and deleted easily. Also, manipulation of data from various tables can be done easily using various basic operations. For example, we can add a telephone number fields in the table at Figure-1.

- **Accuracy**

In relational database the relational algebraic operations are used to manipulate database. These are mathematical operations and ensure accuracy (and less of ambiguity) as compared to other models.

3.2.1 Tuple, Attribute, Domains and Relation

Basic terminology used in relational model are:

- **Tuple**

Each row in a table represents a record and is called a tuple.

- **Attribute**

The name of each column in a table is used to interpret its meaning and is called an attribute. Each table is called relation.

For example, Figure-1 represents a relation Student. The columns Rollno, Name, Age and Address are the attributes of Student and each row in the table represents a separate tuple (record).

- **Domain**

A domain is a set of permissible values that can be given to an attribute. So every attribute in a table has a specific domain. Values to these attributes cannot be assigned outside their domains.

In the example above if domain of Rollno is a set of integer values from 1 to 1000 then a value outside this range will not be valid.

The domain can be defined by assigning a type or a format or a range to an attribute.

3.2.1 Tuple, Attribute, Domains and Relation (cont)

- **Relation**

A relation consists of:

1. Relational schema
2. Relation instance

1. Relational schema

A relational schema specifies the relation's name, its attributes and the domain of the each attribute. If R is the name of relation and A_1, A_2, \dots, A_n is a list of attributes representing R then $R(A_1, A_2, \dots, A_n)$ is called a relational schema. Each attribute in this relational schema takes a value from some specific domain called Domain (A_i)

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

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

E.g. *customer (Customer-schema)*

Relation Instance

2. Relation Instances or Relation state

A relation instances denoted as r is a collection of tuples for a given relational schema at a specific point of time.

A relation state r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$ is a set of n -tuples

$r = \{t_1, t_2, \dots, t_m\}$ where each n -tuple is an ordered list of n values

$t = \langle v_1, v_2, \dots, v_n \rangle$ where each v_i belongs to domain (A_i) or contains null values.

- 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

The diagram shows a table representing a relation instance. The table has three columns and four rows. Arrows point from the text 'attributes (or columns)' to the column headers. Another set of arrows points from the text 'tuples (or rows)' to the data rows. The table content is as follows:

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

customer

Key constraints

Primary Key

- Column or a set of columns that uniquely identify a row in a table
- Must be unique and must have a value

Foreign Key

- Column or set of columns which references the primary key or a unique key of another table
- Rows in two tables are linked by matching the values of the foreign key in one table with the values of the primary key in another

- EMP_ID in table EMPLOYEE is the *primary key*
- DEPT_NO in table DEPARTMENT is the *primary key*
- DEPT_NO in table EMPLOYEE is a *foreign key*



Examples

Key constraints (cont)

Data Integrity

- Ensures correct and consistent navigation and manipulation of relational tables
- Two types of integrity rules
 - Entity integrity
 - Referential integrity
- The entity integrity rule states that the value of the primary key can never be a null value
- The referential integrity rule states that if a relational table has a foreign key, then every value of the foreign key must either be null or match the values in the relational table in which that foreign key is a primary key

Query Languages

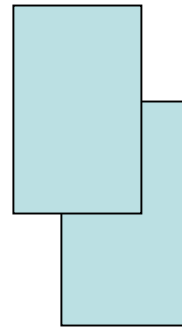
- Language in which user requests information from the database.
- Categories of languages
 - procedural
 - non-procedural
- “Pure” languages:
 - Relational Algebra
 - Tuple Relational Calculus
 - Domain Relational Calculus
- Pure languages form underlying basis of query languages that people use.

Relational Algebra

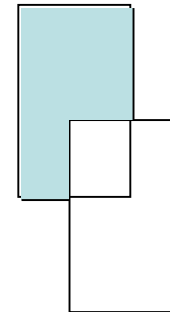
- Procedural language
- Relational tables are equivalent to sets.
- Operations that can be performed on sets can be performed on relational tables

- Relational Operations such as :

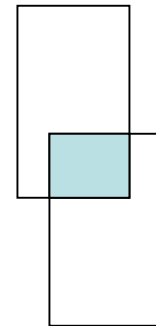
- Selection
- Projection
- Join
- Union
- Intersection
- Difference
- Product
- Division



UNION



DIFFERENCE



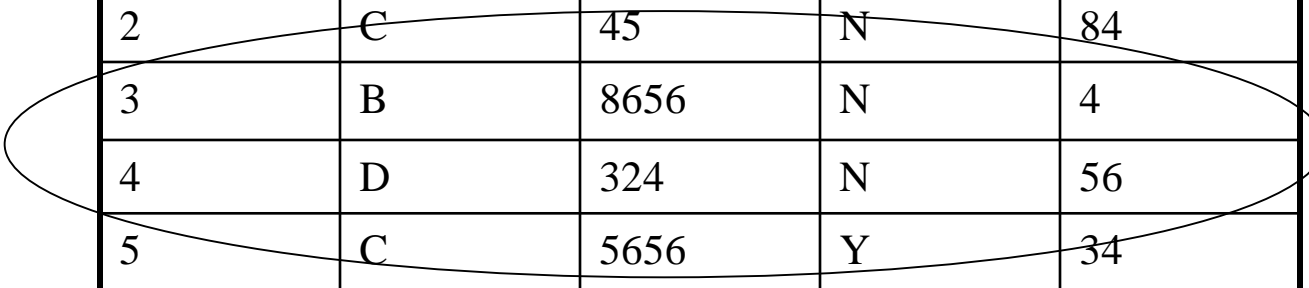
INTERSECTION

- The operators take two or more relations as inputs and give a new relation as a result.

Relational Algebra

Selection

The select operator, sometimes called restrict to prevent confusion with the SQL SELECT command, retrieves subsets of rows from a relational table based on a value(s) in a column or columns



A	B	C	D	E
1	A	212	Y	2
2	C	45	N	84
3	B	8656	N	4
4	D	324	N	56
5	C	5656	Y	34
6	A	445	N	4
7	B	546	Y	55

Relational Algebra

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:

$\langle \text{attribute} \rangle \text{ op } \langle \text{attribute} \rangle$ or $\langle \text{constant} \rangle$

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

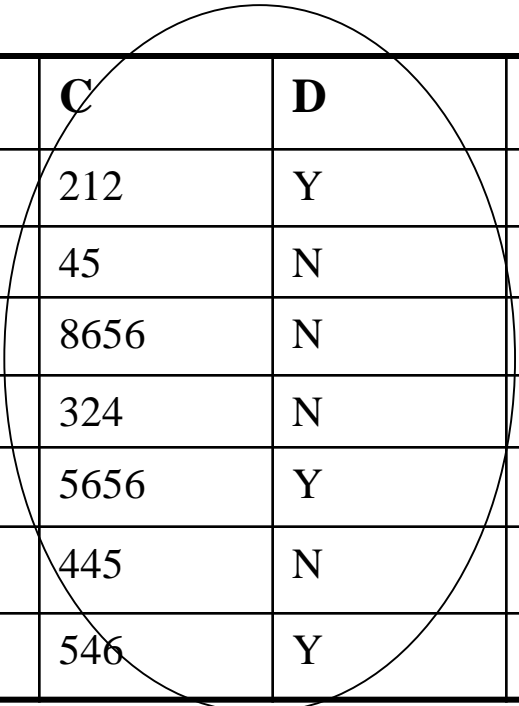
- Example of selection:

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

Relational Algebra

- Projection**

- The project operator retrieves subsets of columns from a relational table removing duplicate rows from the result



A	B	C	D	E
1	A	212	Y	2
2	C	45	N	84
3	B	8656	N	4
4	D	324	N	56
5	C	5656	Y	34
6	A	445	N	4
7	B	546	Y	55

Relational Algebra

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*

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

Relational Algebra

Cartesian-Product

- The product of two relational tables, also called the Cartesian Product, is the concatenation of every row in one table with every row in the second.
- The product of table A (having m rows) and table B (having n rows) is the table C (having m x n rows). The product is denoted as $A \times B$ or $A \text{ TIMES } B$

Table A

k	x	y
1	A	2
2	B	4
3	C	6

Table B

k	x	y
1	A	2
4	D	8
5	E	10

A TIMES B



ak	ax	ay	bk	bx	by
1	A	2	1	A	2
1	A	2	4	D	8
1	A	2	5	E	10
2	B	4	1	A	2
2	B	4	4	D	8
2	B	4	5	E	10
3	C	6	1	A	2
3	C	6	4	D	8
3	C	6	5	E	10

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.

Relational Algebra

Join

- Combines the product, selection and projection operations
- Combines (concatenates) data from one row of a table with rows from another or same table
- Criteria involve a relationship among the columns in the join relational table

If the join criterion is based on equality of column value, the result is called an equi join

A natural join is an equi join with redundant columns removed

Joins can also be done on criteria other than equality. Such joins are called non-equi (Theta) joins

k	a	b
1	A	2
2	B	4
3	C	6

Table A

k	c
1	aa
3	bb
5	cc

Table B

k	a	b	k	c
1	A	2	1	aa
3	C	6	3	bb

Equi-Join

k	a	b	c
1	A	2	aa
3	C	6	bb

Natural Join

Relational Algebra

Union

- The UNION operation of two tables is formed by appending rows from one table to those of a second to produce a third. Duplicate rows are eliminated
- Tables in an UNION operation must have the same number of columns and corresponding columns must come from the same domain

k	x	y
1	A	2
2	B	4
3	C	6

Table A

Table B

k	x	y
1	A	2
4	D	8
5	E	10

A Union B

k	x	y
1	A	2
2	B	4
3	C	6
4	D	8
5	E	10

Relational Algebra

Intersection

The intersection of two relational tables is a third table that contains common rows. Both tables must be union compatible. The notation for the intersection of A and B is $A \cap B = C$ or $A \text{ INTERSECT } B$

A Intersect B

k	x	y
1	A	2
2	B	4
3	C	6

Table A

k	x	y
1	A	2
4	D	8
5	E	10

k	x	y
1	A	2

Table B

Relational Algebra

Difference

- The difference of two relational tables is a third that contains those rows that occur in the first table but not in the second. The Difference operation requires that the tables be union compatible.

The notation for difference is $A \text{ MINUS } B$ or $A - B$. As with arithmetic, the order of subtraction matters. That is, $A - B$ is not the same as $B - A$.

k	x	y
1	A	2
2	B	4
3	C	6

Table A

k	x	y
1	A	2
4	D	8
5	E	10

Table B

A MINUS B

k	x	y
2	B	4
3	C	6

B MINUS A

k	x	y
4	D	8
5	E	10

Relational Algebra

Division

The division operator results in columns values in one table for which there are other matching column values corresponding to every row in another table.

k	x	y
1	A	2
1	B	4
2	A	2
3	B	4
4	B	4
3	A	2

Table A

x	y
A	2
B	4

Table B

k
1
3

A DIV B

Composition of Operations

- Can build expressions using multiple operations

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

- $r \times s$

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
α	1	α	10	<i>a</i>
α	1	β	10	<i>a</i>
α	1	β	20	<i>b</i>
α	1	γ	10	<i>b</i>
β	2	α	10	<i>a</i>
β	2	β	10	<i>a</i>
β	2	β	20	<i>b</i>
β	2	γ	10	<i>b</i>

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

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
α	1	α	10	<i>a</i>
β	2	β	20	<i>a</i>
β	2	β	20	<i>b</i>

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

$$\rho_x(A1, A2, \dots, An)(E)$$

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

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*)

Relation: Branch

<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

Relation: Borrower

<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

Relation: Customer

<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

Relation: Depositor

<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

Relation: Account

<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

Relation: Loan

<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

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

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

– Query 2

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

Example Queries

Find the largest account balance

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

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

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

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

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
- 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 *credit-info(customer-name, limit, credit-balance)*, find how much more each person can spend:

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

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 algebra

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

- 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
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*:

<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 \mathcal{G} *sum(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 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.
 - 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 ⋈ *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 ⋈_L *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 ⋈_r *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 ⋈_f *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: $(\text{unknown} \textbf{ or } \text{true}) = \text{true}$,
 $(\text{unknown} \textbf{ or } \text{false}) = \text{unknown}$
 $(\text{unknown} \textbf{ or } \text{unknown}) = \text{unknown}$
 - AND: $(\text{true} \textbf{ and } \text{unknown}) = \text{unknown}$,
 $(\text{false} \textbf{ and } \text{unknown}) = \text{false}$,
 $(\text{unknown} \textbf{ and } \text{unknown}) = \text{unknown}$
 - NOT: $(\textbf{not } \text{unknown}) = \text{unknown}$
 - 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.

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

- 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.

$$r_1 \leftarrow \sigma_{branch-city = "Needham"}(account \bowtie branch)$$

$$r_2 \leftarrow \Pi_{branch-name, account-number, balance}(r_1)$$

$$r_3 \leftarrow \Pi_{customer-name, account-number}(r_2 \bowtie depositor)$$

$$account \leftarrow account - r_2$$

$$depositor \leftarrow depositor - r_3$$

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.

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

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

- 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.

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

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

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

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

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

- 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

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

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.)
- 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

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

- 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**

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

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

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

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

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

- 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 $\sigma_{branch-name = \text{“Perryridge”}}(account)$
 $v \leftarrow v \cup (L-99, Downtown, 23)$
- Others cannot be translated uniquely
 - $all-customer \leftarrow all-customer \cup \{(\text{“Perryridge”}, \text{“John”})\}$
 - 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 is 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

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

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

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

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

Example Queries

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

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

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

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

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. $\{t \mid t[A]=5 \vee \mathbf{true}\}$ 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:

$$\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$$

- 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

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

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

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

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

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

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

Example Queries

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

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

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

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

Safety of Expressions

$$\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$$

is safe if all of the following hold:

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