Unit II

Relational Model: Structure of Relational Databases, Relational Algebra, Relational Calculus, Extended Relational Algebra Operations, Views, Modifications of the Database. Domains, Tuples, Attributes, Relations, Characteristics of Relations, Joins and its type. Keys, Key Attributes of Relation, Relational database, Schemas, Integrity Constraints. Referential Integrity, Intension and Extension.

Structure of Relational Database

- A relational database consists of a collection of tables, each of which is assigned a unique name.

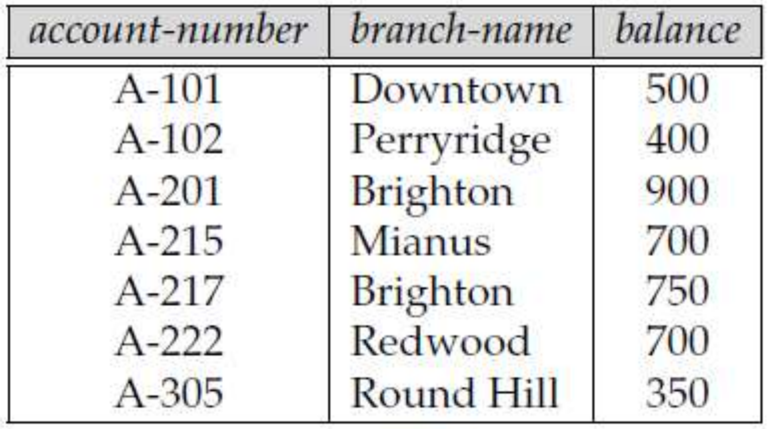
- Each table has a structure.

- we represented E-R databases by tables.

- A row in a table represents a *relationship* among a set of values.

Basic Structure

- Consider the account table. It has three column headers: account-number, branch-name, and balance.



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

The *account* relation.

Basic Structure Cont…

- Following the terminology of the relational model, we refer to these headers as attributes. For each attribute, there is a set of permitted values, called the domain of that attribute.
- For example, for the attribute branch-name, the domain is the set of all branch names.
- Let D1 denote the set of all account numbers, D2 the set of all branch names, and D3 the set of all balances.
- Any row of account must consist of a v1, v2, v3, where v1 is an account number (that is, v1 is in domain D1), v2 is a branch name (that is, v2 is in domain D2), and v3 is a balance (that is, v3 is in domain D3).
- In general, account will contain only a subset of the set of all possible rows. Therefore, account is a subset of D1 × D2 × D3

Database Schema

- When we talk about a database, we must differentiate between the database schema, which is the logical design of the database, and a database instance, which is a snapshot of the data in the database at a given instant in time.
- We adopt the convention of using lowercase names for relations, and names beginning with an uppercase letter for relation schemas.
- Following this notation, we use *Account-schema to denote the relation* schema for relation *account. Thus,*

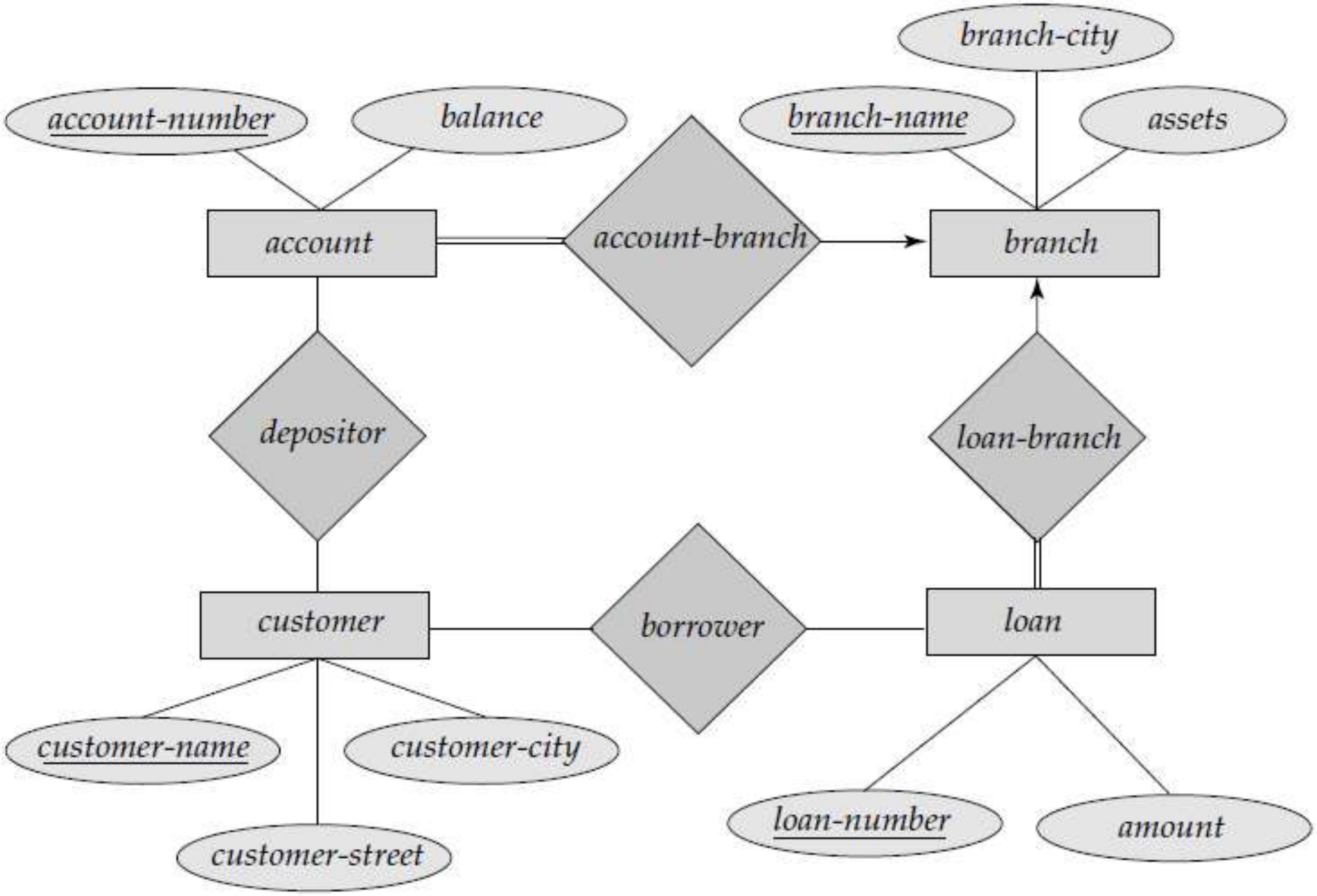
  Account-schema = (account_number, branch_name, balance)
- We denote the fact that *account is a relation on Account-schema by*

  account(Account-schema)
- In general, a relation schema consists of a list of attributes and their corresponding domains.

Database Schema Cont…

- Branch-schema = (branch_name, branch_city, branch_code)
- Customer-schema = (customer_name, customer_street, customer_city)
- Depositor -schema = (customer_name, account_number)
- Loan-schema = (loan_number, branch_name, amount)
- Borrower-schema = (customer_name, loan_number)

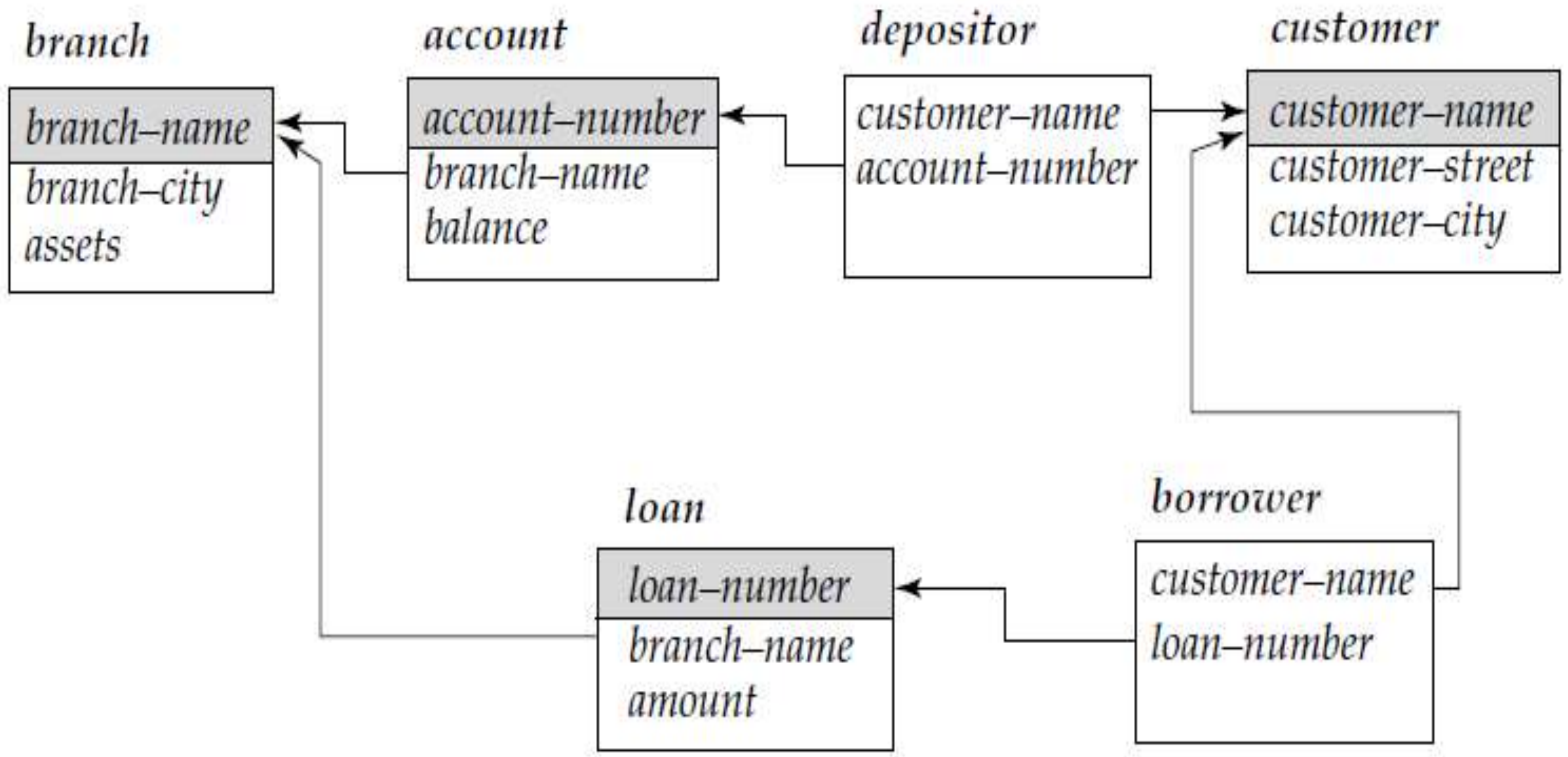
E-R diagram for the Banking Enterprise



Schema Diagram

- A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by schema diagrams.
- Figure (in next slide) shows the schema diagram for our banking enterprise.
- Each relation appears as a box, with the attributes listed inside it and the relation name above it.
- If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line.
- Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

Schema Diagram for the Banking Enterprise



Query Languages

- A query language is a language in which a user requests information from the database.
- Query languages can be categorized as either procedural or nonprocedural.
- In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result.
- In a nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information.
- The relational algebra is procedural, whereas the tuple relational calculus and domain relational calculus are nonprocedural.

The Relational Algebra

- The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their output.
- The fundamental operations in the relational algebra are:
- Select
- Project
- Union
- Set difference
- Cartesian product and
- Rename

Fundamental Operations

- The select, project, and rename operations are called unary operations, because they operate on one relation.
- The other three operations operate on pairs of relations and are, therefore, called binary operations.
- In addition to the fundamental operations, there are several other operations—namely:
- Set intersection
- Natural join
- Division and
- Assignment

Select Operation

- The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) *to* denote selection. The predicate appears as a subscript to σ.
- The argument relation is in parentheses after the σ. Thus, to select those tuples of the loan relation where the branch is “Perryridge,” we write

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

- We can find all tuples in which the loan amount is more than 1200Rs. by writing

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

Select Operation

- In general, we allow comparisons using $=, \neq, <, \leq, >, \geq$ in the selection predicate.

- Furthermore, we can combine several predicates into a larger predicate by using the connectives and ($\wedge$), or ($\vee$) and not ($\neg$).

- Find those tuples pertaining to loans of more than 1200Rs. made by the Perryridge branch, we write

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

Project Operation

- The project operation is a unary operation that returns its argument relation, with certain attributes left out.
- In this operation, any duplicate rows are eliminated.
- Projection is denoted by the uppercase Greek letter pi (Π).
- We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses.
- Write the query to list all loan numbers and the amount of the loan.

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

Project Operation

- Find the names of all customers with a loan in the bank.

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

- Find the names of all customers with an account in the bank.

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

Composition of Relational Operations

- The result of a relational operation is itself a relation.

- Since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into a relational-algebra expression.

- For Ex: Find those customers who live in Indore.

$$\Pi_{\text{customer\_name}}(\sigma_{\text{customer-city} = \text{“Indore”}}(\text{customer}))$$

Union Operation

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

- To answer the query, we need the union of two sets; that is, we need all customer names that appear in either or both of the two relations.

- We find these data by the binary operation union, denoted, as in set theory, by ∪.

- So the expression needed is

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

Union Operation

- For a union operation r ∪ s to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.

2. The domains of the i^{th} attribute of r and the i^{th} attribute of s must be the same, for all i.

- Note that r and s can be, in general, temporary relations that are the result of relational algebra expressions.

Set Difference Operation

- The set-difference operation, denoted by $-$, allows us to find tuples that are in one relation but are not in another.

- The expression $r - s$ produces a relation containing those tuples in r but not in s.

- We can find all customers of the bank who have an account but not a loan.

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

Set Difference Operation

- As with the union operation, we must ensure that set differences are taken between compatible relations. Therefore, for a set difference operation r − s to be valid, If,

- The relations r and s be of the same arity, and

- The domains of the i^{th} attribute of r and the i^{th} attribute of s be the same.

Cartesian-Product Operation

- The Cartesian-product operation, denoted by a cross (×), allows us to combine information from any two relations.
- We write the Cartesian product of relations r1 and r2 as

$$r1 \times r2$$

- However, since the same attribute name may appear in both r1 and r2, we need to devise a naming schema to distinguish between these attributes.
- We do so here by attaching to an attribute the name of the relation from which the attribute originally came.

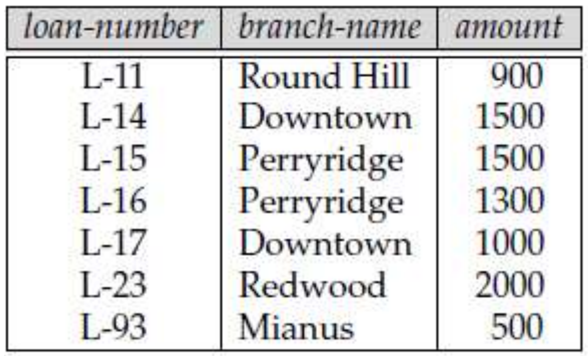
Cartesian-Product Operation

- For example, the relation schema for r = borrower × loan is (borrower.customer_name,borrower.loan_number, loan.loan_number, loan.branch_name, loan.amount)
- With this schema, we can distinguish borrower.loan-number from loan.loan-number.
- For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity.
- We can then write the relation schema for r as (customer_name, borrower.loan_number, loan.loan_number, branch_name, amount)

Example

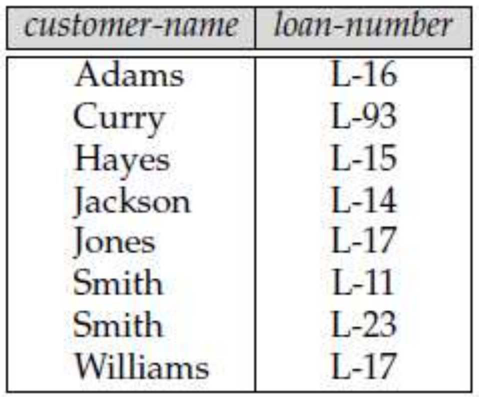
- Find the names of all customers who have a loan at the Perryridge branch. We write,

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



loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

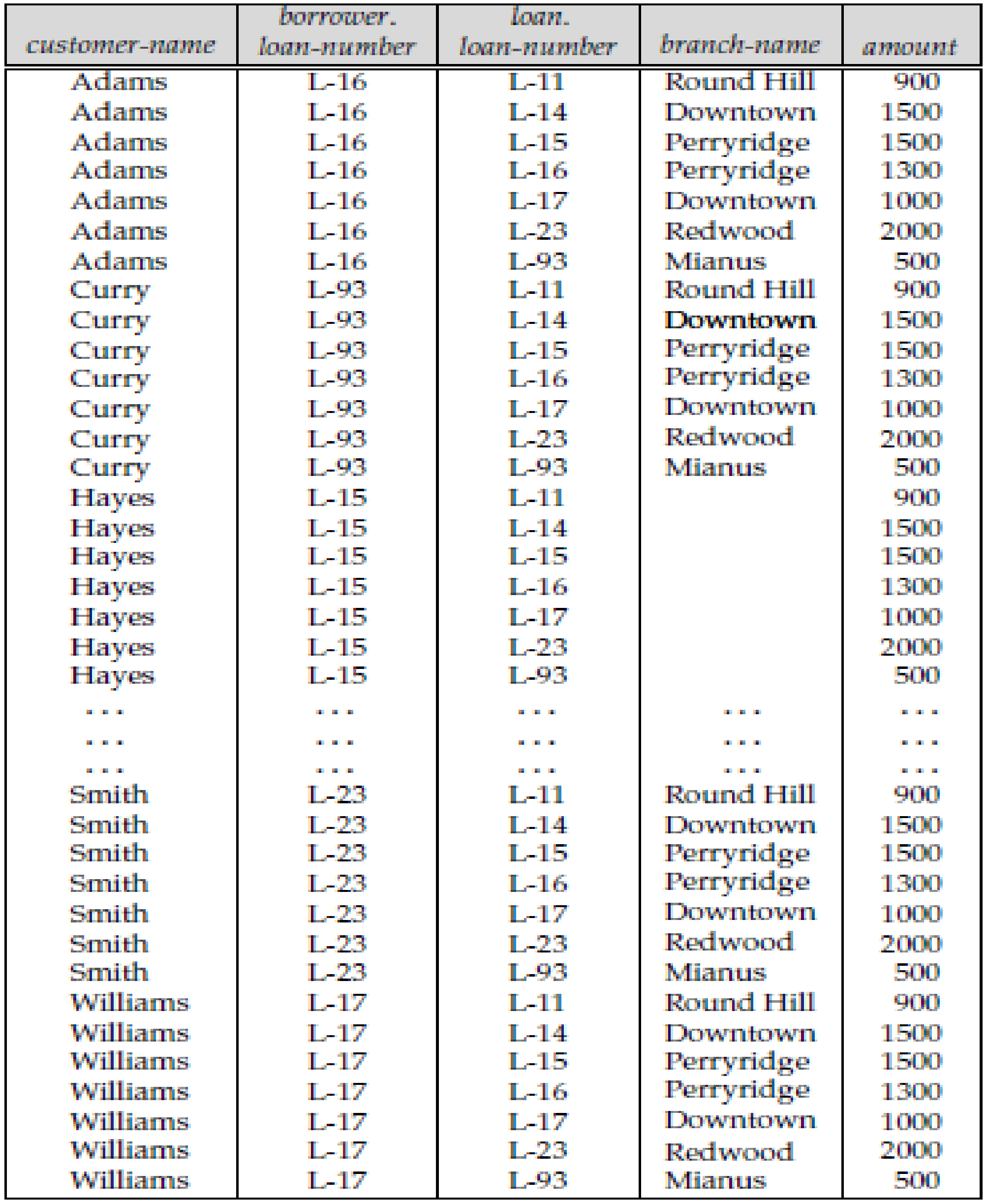
The *loan* relation.



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

The *borrower* relation

Result of borrower × loan



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

- The customer-name column may contain customers who do not have a loan at the Perryridge branch. (Because the Cartesian product takes all possible pairings of each tuple from borrower with each tuple of loan.)

- Since the Cartesian-product operation associates every tuple of loan with every tuple of borrower, we know that, if a customer has a loan in the Perryridge branch, then there is some tuple in borrower×loan that contains his name, and borrower.loan_number= loan.loan_number. So, if we write

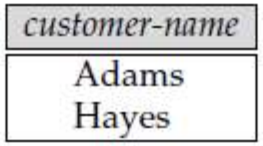
$\sigma_{\text{borrower.loan\_number=loan.loan\_number}}(\sigma_{\text{branch\_name=“Perryridge”}}(\text{borrower}\times\text{loan}))$

- we get only those tuples of borrower × loan that pertain to customers who have a loan at the Perryridge branch.

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

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

- The result of this expression, shown in Figure.



customer-name
Adams
Hayes

Rename Operation

- The rename operator, denoted by the lowercase Greek letter rho (ρ).
- For example, consider a relational-algebra expression E, the expression

$$\rho_x(E)$$

  returns the result of expression E under the name x.
- A relation r by itself is considered a relational-algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name.
- A second form of the rename operation is as follows.
- Assume that a relational algebra expression E has arity n. Then, the expression $\rho_{x(A1,A2,...,An)}(E)$ returns the result of expression E under the name x, and with the attributes renamed to A1,A2, . . .,An.

Example

- Find the names of all customers who live on the same street and in the same city as Smith.

$$\Pi_{\text{customer\_street, customer\_city}} (\sigma_{\text{customer\_name = “Smith”}}(\text{customer}))$$

- However, in order to find other customers with this street and city, we must reference the customer relation a second time. In the following query, we use the rename operation on the preceding expression to give its result the name smith-addr, and to rename its attributes to street and city, instead of customer-street and customer-city.

$$\Pi_{\text{customer.customer\_name}}(\sigma_{\text{customer.customer\_street=smith\_addr.street} \wedge \text{customer.customer\_city=smith\_addr.city}}(\text{customer} \times \rho_{\text{smith\_addr(street,city)}} (\Pi_{\text{customer\_street, customer\_city}} (\sigma_{\text{customer\_name = “Smith”}}(\text{customer})))))$$

Set-Intersection Operation

- Set intersection operation denoted by symbol (∩).
- Suppose that we wish to find all customers who have both a loan and an account.
- Using set intersection, we can write

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

Natural-Join Operation

- Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product.

- Consider the query “Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount.”

Natural-Join Operation

- To solve this query, we first form the Cartesian product of the borrower and loan relations. Then, we select those tuples that pertain to only the same loan-number, followed by the projection of the resulting customer-name, loan-number, and amount.

$$\Pi_{\text{customer\_name, loan.loan\_number, amount}} (\sigma_{\text{borrower.loan\_number =loan.loan\_number}}(\text{borrower} \times \text{loan}))$$

Natural-Join Operation

- The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.
- It is denoted by the “join” symbol $\bowtie$
- The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.
- Find the names of all customers who have a loan at the bank, and find the amount of the loan.

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

Example

- Find the names of all branches with customers who have an account in the bank and who live in Indore.

Example

- Find the names of all branches with customers who have an account in the bank and who live in Indore.

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

Extended Relational-Algebra Operations

Generalized Projection

- The generalized-projection operation extends the projection operation by allowing arithmetic functions to be used in the projection list.

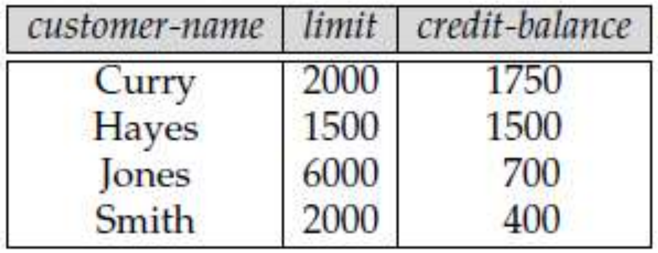
- The generalized projection operation has the form

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

- where E is any relational-algebra expression, and each of F1, F2, . . . , Fn is an arithmetic expression involving constants and attributes in the schema of E.

Generalized Projection

- For example, suppose we have a relation credit-info, which lists the credit limit and expenses so far (the credit-balance on the account).



customer-name	*limit*	*credit-balance*
Curry	2000	1750
Hayes	1500	1500
Jones	6000	700
Smith	2000	400

- If we want to find how much more each person can spend, we can write the following expression:

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

Generalized Projection

- The attribute resulting from the expression limit − credit_balance does not have a name. We can apply the rename operation to the result of generalized projection in order to give it a name. As a notational convenience, renaming of attributes can be combined with generalized projection as illustrated below:

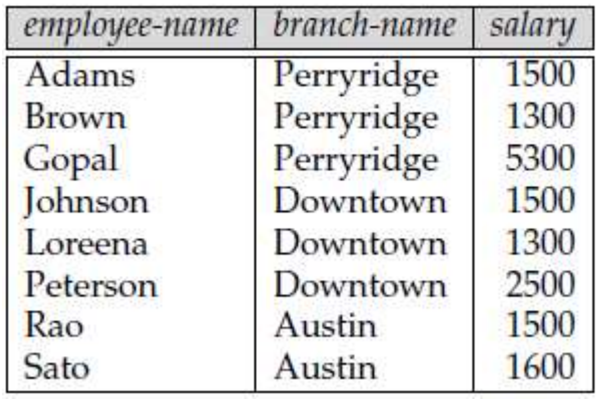
$$\Pi_{\text{customer\_name, (limit − credit\_balance)} \textbf{ as credit\_available}}(\text{credit\_info})$$

Aggregate Functions

- Aggregate functions take a collection of values and return a single value as a result.
- For example, the aggregate function sum takes a collection of values and returns the sum of the values.
- The aggregate function avg returns the average of the values.
- The aggregate function count returns the number of the elements in the collection.
- Aggregate functions min and max, return the minimum and maximum values in a collection.
- The relational-algebra operation $\mathcal{G}$ signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied.
- The symbol $\mathcal{G}$ is the letter G in calligraphic font; read it as "calligraphic G."

Aggregate Functions

- To illustrate the concept of aggregation, we shall use the pt-works relation for part-time employees.



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

The *pt-works* relation

- Suppose that we want to find out the total sum of salaries of all the part-time employees in the bank.
- The relational-algebra expression for this query is:

$$\mathcal{G}_{\text{sum(salary)}}(\text{pt-works})$$

Aggregate Functions

- Suppose we want to find the total salary sum of all part-time employees at each branch of the bank separately, rather than the sum for the entire bank.
- To do so, we need to partition the relation pt-works into groups based on the branch, and to apply the aggregate function on each group.
- The following expression using the aggregation operator G achieves the desired result:

$$_{\text{branch\_name}}\mathcal{G}_{\text{sum(salary)}}(\text{pt-works})$$

- In the expression, the attribute branch_name in the left-hand subscript of $\mathcal{G}$ indicates that the input relation pt-works must be divided into groups based on the value of branch-name.

Outer Join

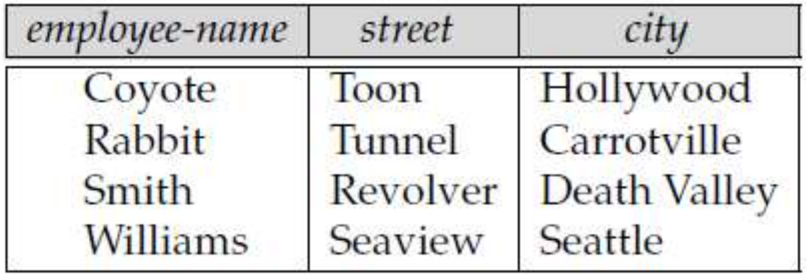
- The outer-join operation is an extension of the join operation to deal with missing information.
- Suppose that we have the relations with the following schemas, which contain data on full-time employees:

employee (employee-name, street, city)

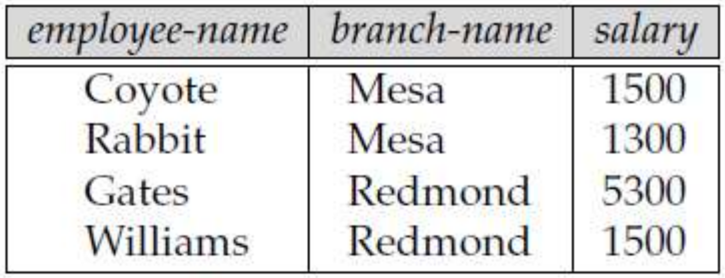
ft-works (employee-name, branch-name, salary)

- Consider the employee and ft-works relations in Figure(next slide).

Outer Join



employee-name	*street*	*city*
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

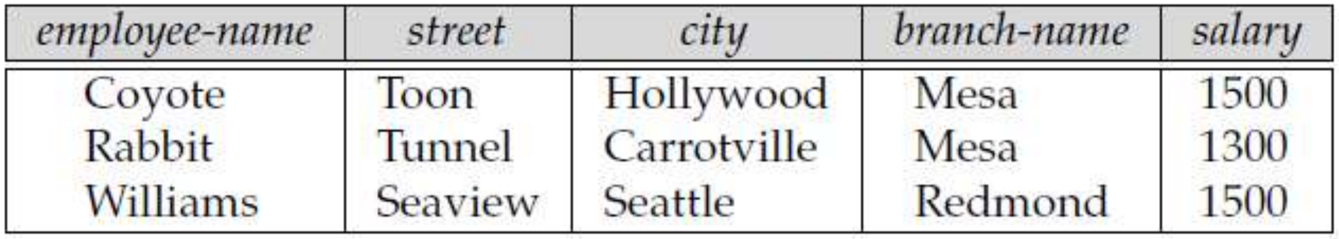


employee-name	*branch-name*	*salary*
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

Suppose that we want to generate a single relation with all the information (street, city, branch name, and salary) about full-time employees.

A possible approach would be to use the natural join operation as follows:

employee $\bowtie$ ft-works



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

Notice that we have lost the street and city information about Smith, since the tuple describing Smith is absent from the ft-works relation;

similarly, we have lost the branch name and salary information about Gates, since the tuple describing Gates is absent from the employee relation.

Outer Join

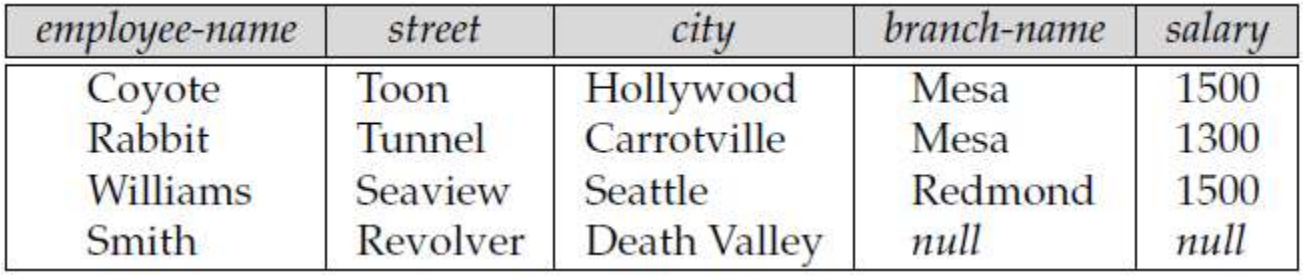
- We can use the outer-join operation to avoid this loss of information.

- There are actually three forms of the operation: left outer join, denoted by ⟕ , right outer join denoted by ⟖ and full outer join, denoted by ⟗.

- All three forms of outer join compute the join, and add extra tuples to the result of the join.

Left Outer Join

- The left outer join ⟕ takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation, and adds them to the result of the natural join.
- All information from the left relation is present in the result of the left outer join.

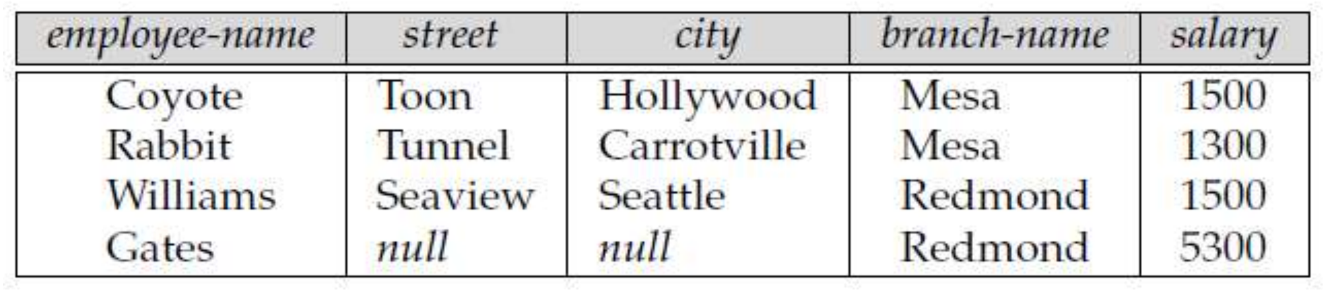


employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*

Result of *employee* ⟕ *ft-works*

Right Outer Join

- The right outer join ⟖ is symmetric with the left outer join: It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join.
- All information from the right relation is present in the result of the right outer join.

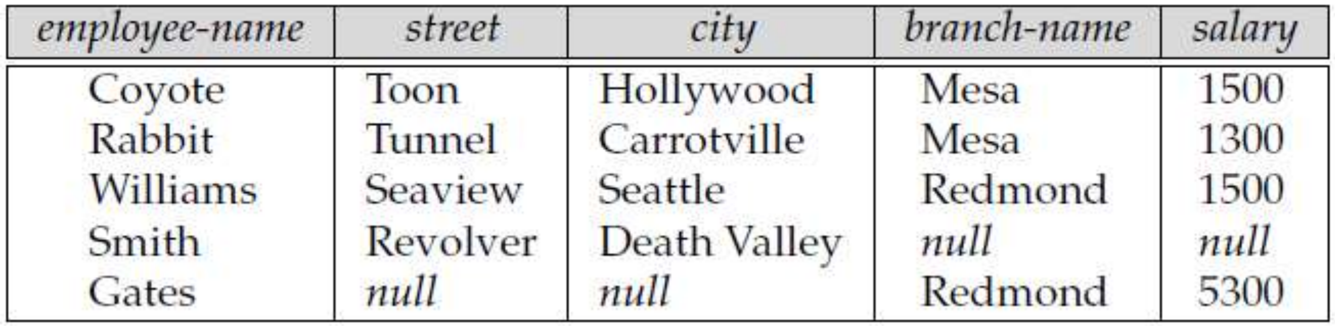


employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	*null*	*null*	Redmond	5300

Result of *employee* ⟖ *ft-works*

Full Outer Join

- The full outer join ⟗ does both of those operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join.



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

Result of *employee* ⟗ *ft-works*

Modifications of the Database

Deletion

- We can delete only whole tuples; we cannot delete values on only particular attributes.

- In relational algebra a deletion is expressed by

$$r \leftarrow r - E$$

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

Deletion Example

- Delete all of Smith's account records.

$$depositor \leftarrow depositor - \sigma_{customer\_name = \text{“Smith”}} (depositor)$$

- Delete all loans with amount in the range 0 to 50.

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

Insertion

- To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.
- The attribute values for inserted tuples must be members of the attribute's domain.
- Similarly, tuples inserted must be of the correct arity.
- The relational algebra expresses an insertion by

$$r \leftarrow r \cup E$$

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

View

- Any relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a view.
- We define a view by using the create view statement.
- To define a view, we must give the view a name, and must state the query that computes the view.
- The form of the create view statement is

create view v as <query expression>

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

View

- As an example, consider the view consisting of branches and their customers. This view to be called all_customer.
- We define this view as follows:

create view all_customer as

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

- Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates.
- Using the view all_customer, we can find all customers of the Perryridge branch by writing:

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

Relational Data Model

- The relational Model of Data is based on the concept of a Relation.
- A Relation is a mathematical concept based on the ideas of sets.
- The strength of the relational approach to data management comes from the formal foundation provided by the theory of relations.
- The model was first proposed by Dr. E.F. Codd of IBM in 1970 in the following paper: "A Relational Model for Large Shared Data Banks," Communications of the ACM, June 1970.
- The above paper caused a major revolution in the field of Database management and earned Ted Codd the coveted ACM Turing Award.

INFORMAL DEFINITIONS

- RELATION: A table of values
  - A relation may be thought of as a **set of rows**.
  - A relation may alternately be though of as a **set of columns**.
  - Each row represents a fact that corresponds to a real-world **entity** or **relationship**.
  - Each row has a value of an item or set of items that uniquely identifies that row in the table.
  - Sometimes row-ids or sequential numbers are assigned to identify the rows in the table.
  - Each column typically is called by its column name or column header or attribute name.

FORMAL DEFINITIONS

- A **Relation** may be defined in multiple ways.
- The **Schema** of a Relation: *R* (A1, A2, ...An)

  Relation schema *R* is defined over **attributes** A1, A2, ...An

  For Example -

  CUSTOMER (Cust-id, Cust-name, Address, Phone#)

- Here, CUSTOMER is a relation defined over the four attributes Cust-id, Cust-name, Address, Phone#
- Each of which has a **domain** or a set of valid values.
- For example, the domain of Cust-id is 6 digit numbers.

FORMAL DEFINITIONS

- A **tuple** is an ordered set of values
- Each value is derived from an appropriate domain.
- Each row in the CUSTOMER table may be referred to as a tuple in the table and would consist of four values.
- **<632895, "John Smith", "101 Main St. Atlanta, GA 30332","(404)894-2000">** is a tuple belonging to the CUSTOMER relation.
- A relation may be regarded as a **set of tuples** (rows).
- Columns in a table are also called attributes of the relation.

FORMAL DEFINITIONS

- A **domain** has a logical definition: e.g.“phone_numbers” are the set of 10 digit phone numbers valid in the country.

- A domain may have a data-type or a format defined for it. The phone_numbers may have a format: (ddd)-ddd-dddd where each d is a decimal digit. E.g., Dates have various formats such as month_name, date, year or yyyy-mm-dd, or dd mm,yyyy etc.

- An attribute designates the **role** played by the domain. E.g., the domain Date may be used to define attributes “Invoice-date” and “Payment-date”.

FORMAL DEFINITIONS

- The relation is formed over the Cartesian product of the sets; each set has values from a domain; that domain is used in a specific role which is conveyed by the attribute name.
- For example, attribute Cust-name is defined over the domain of strings of 25 characters. The role these strings play in the CUSTOMER relation is that of the name of customers.
- Formally,

  Given $R(A_1, A_2, ..., A_n)$

  $r(R) \subset dom\ (A_1)\ X\ dom\ (A_2)\ X\ ....X\ dom(A_n)$
- R: schema of the relation
- r of R: a specific "value" or population of R.
- R is also called the **intension** of a relation
- r is also called the **extension** of a relation

FORMAL DEFINITIONS

- Let S1 = {0,1}
- Let S2 = {a,b,c}

- Let R $\subset$ S1 X S2

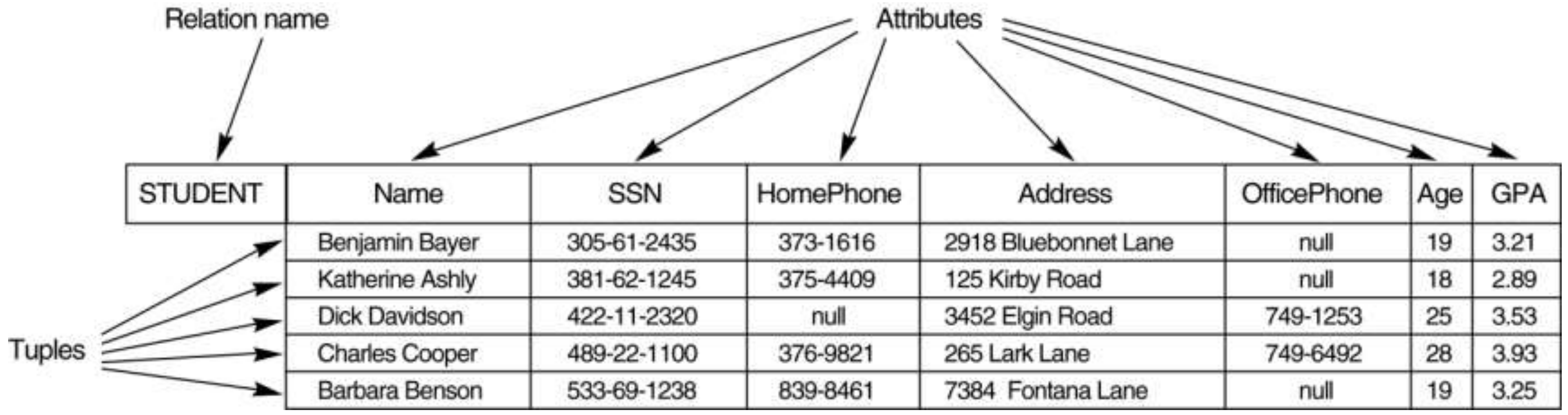
- Then for example: r(R) = {<0,a> , <0,b> , <1,c> }

  is one possible “state” or “population” or “extension” r of the relation R, defined over domains S1 and S2. It has three tuples.

DEFINITION SUMMARY

<u>Informal Terms</u>		<u>Formal Terms</u>
Table		Relation
Column		Attribute/Domain
Row		Tuple
Values in a column		Domain
Table Definition		Schema of a Relation
Populated Table		Extension

Example



STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

CHARACTERISTICS OF RELATIONS

- **Ordering of tuples in a relation r(R)**: The tuples are *not* considered to be ordered, even though they appear to be in the tabular form.

- **Ordering of attributes in a relation schema R** (and of values within each tuple): We will consider the attributes in R(A_1, A_2, ..., A_n) and the values in t=<v_1, v_2, ..., v_n> to be *ordered* .

- **Values in a tuple**: All values are considered atomic (indivisible).
- A special **null** value is used to represent values that are unknown or inapplicable to certain tuples.

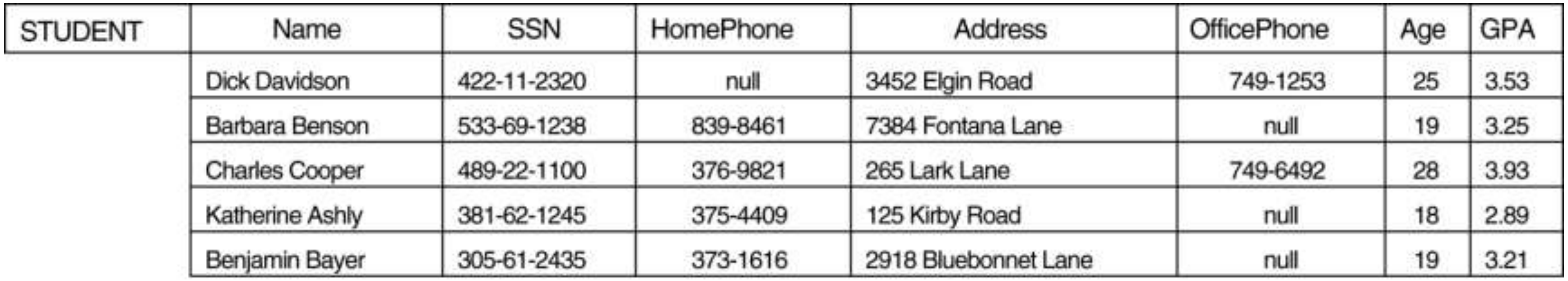
CHARACTERISTICS OF RELATIONS

- Notation:

- We refer to **component values** of a tuple t by $t[A_i] = v_i$ (the value of attribute A_i for tuple t).

  Similarly, $t[A_u, A_v, ..., A_w]$ refers to the subtuple of t containing the values of attributes $A_u, A_v, ..., A_w$, respectively.

CHARACTERISTICS OF RELATIONS



STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21

Relational Integrity Constraints

- Constraints are conditions that must hold on all valid relation instances.

- There are three main types of constraints:

1. **Key** constraints
2. **Entity integrity** constraints
3. **Referential integrity** constraints

Key Constraints

- **Superkey** of R: A set of attributes SK of R such that no two tuples in any valid relation instance r(R) will have the same value for SK. That is, for any distinct tuples t1 and t2 in r(R), t1[SK] $\neq$ t2[SK].
- **Key** of R: A "minimal" superkey; that is, a superkey K such that removal of any attribute from K results in a set of attributes that is not a superkey.

Example: The CAR relation schema:

CAR(State, Reg#, SerialNo, Make, Model, Year) has two keys

Key1 = {State, Reg#}, Key2 = {SerialNo}, which are also superkeys.

- If a relation has several **candidate keys**, one is chosen arbitrarily to be the **primary key**. The primary key attributes are underlined.

Entity Integrity

- **Relational Database Schema**: A set S of relation schemas that belong to the same database. S is the *name* of the database.

$$S = \{R_1, R_2, ..., R_n\}$$

- **Entity Integrity**: The primary key attributes(PK) of each relation schema R in S cannot have null values in any tuple of r(R). This is because primary key values are used to identify the individual tuples.

$$t[PK] \neq \text{null for any tuple } t \text{ in } r(R)$$

- Note: Other attributes of R may be similarly constrained to disallow null values, even though they are not members of the primary key.

Referential Integrity

- A constraint involving two relations (the previous constraints involve a single relation). Used to specify a relationship among tuples in two relations: the **referencing relation** and the **referenced relation**.

- Tuples in the referencing relation R_1 have attributes FK (called **foreign key** attributes) that reference the primary key attributes PK of the referenced relation R_2. A tuple t_1 in R_1 is said to **reference** a tuple t_2 in R_2 if $t_1[FK] = t_2[PK]$.

- A referential integrity constraint can be displayed in a relational database schema as a directed arc from R_1.FK to R_2.

Referential Integrity Constraint

Statement of the constraint

- The value in the foreign key column (or columns) FK of the the **referencing relation** R_1 can be either:

  (1) a value of an existing primary key value of the corresponding primary key PK in the **referenced relation** $R_{2,}$,

  or..

  (2) a null.

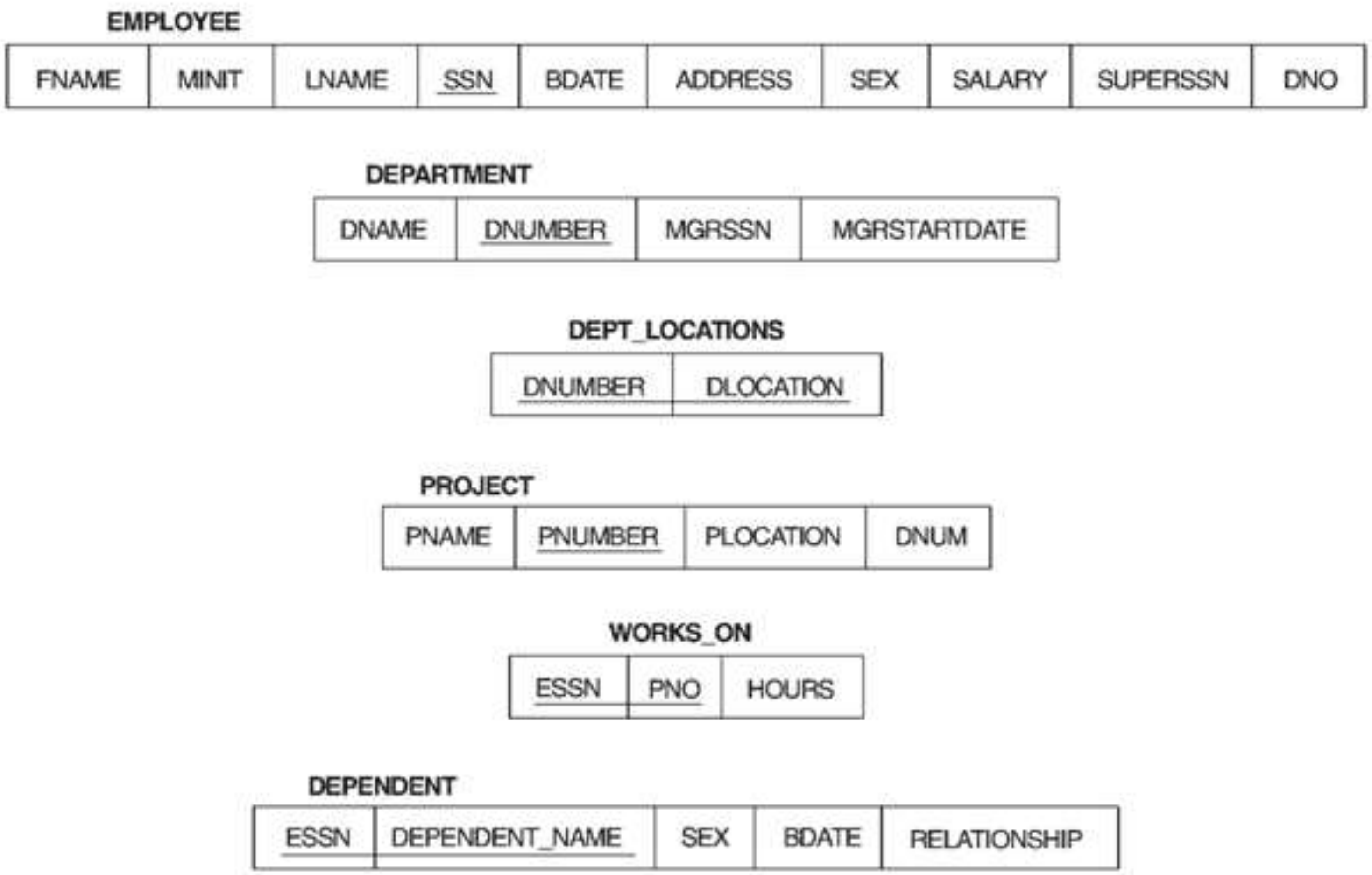
In case (2), the FK in R_1 should not be a part of its own primary key.

Other Types of Constraints

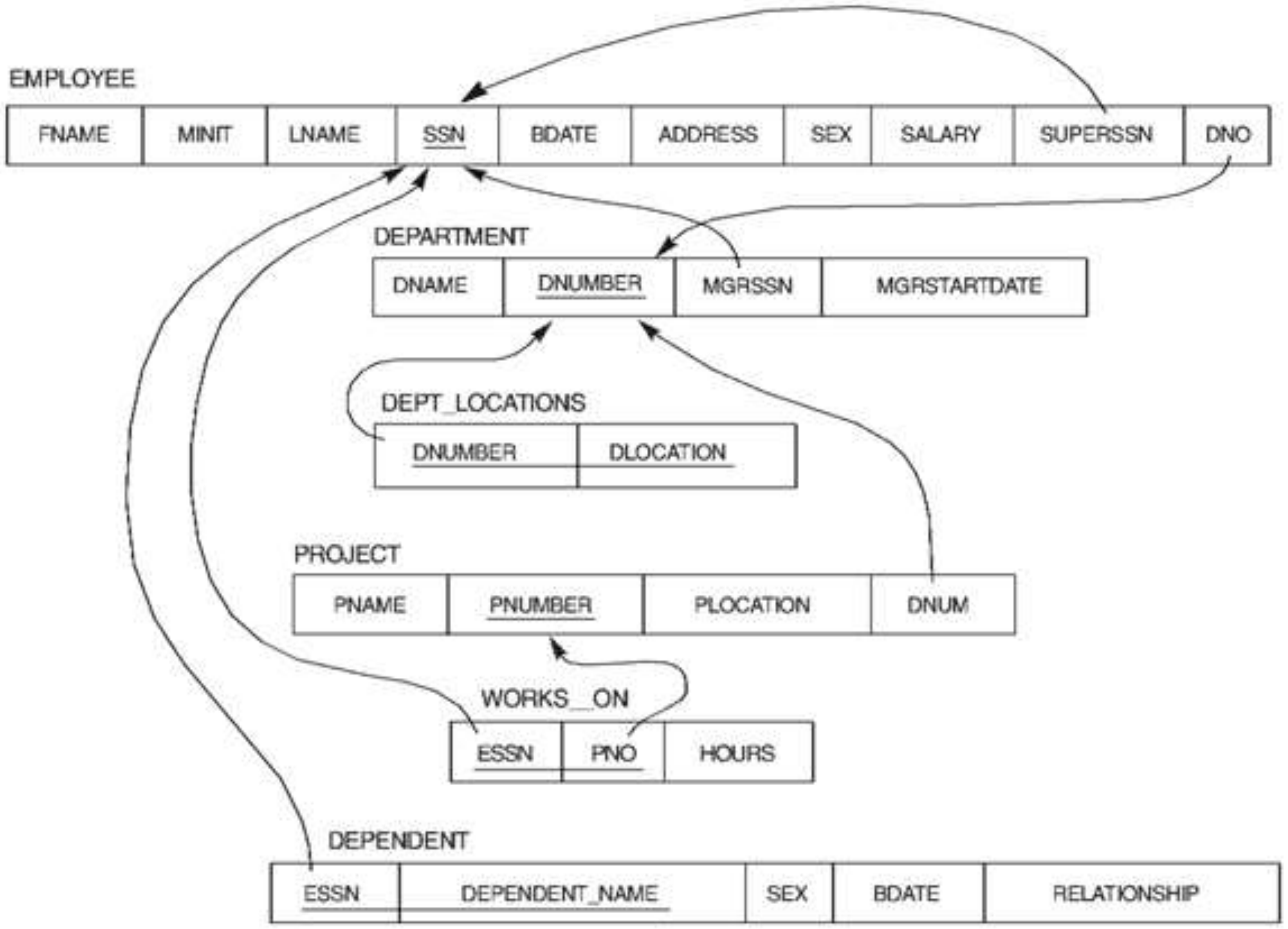
Semantic Integrity Constraints:

- based on application semantics and cannot be expressed by the model per se
- E.g., “the max. no. of hours per employee for all projects he or she works on is 56 hrs per week”
- A constraint specification language may have to be used to express these
- SQL-99 allows triggers and ASSERTIONS to allow for some of these

Schema diagram for the COMPANY relational database schema; the primary keys are underlined.


EMPLOYEE
FNAME | MINIT | LNAME | SSN | BDATE | ADDRESS | SEX | SALARY | SUPERSSN | DNO
DEPARTMENT
DNAME | DNUMBER | MGRSSN | MGRSTARTDATE
DEPT_LOCATIONS
DNUMBER | DLOCATION
PROJECT
PNAME | PNUMBER | PLOCATION | DNUM
WORKS_ON
ESSN | PNO | HOURS
DEPENDENT
ESSN | DEPENDENT_NAME | SEX | BDATE | RELATIONSHIP

Referential integrity constraints displayed on the COMPANY relational database schema diagram.



Exercise

- Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:
- STUDENT(SSN, Name, Bdate)
- COURSE(Course#, Cname, Dept)
- ENROLL(SSN, Course#, Quarter, Grade)
- BOOK_ADOPTION(Course#, Quarter, Book_ISBN)
- TEXT(Book_ISBN, Book_Title, Publisher, Author)
- **Draw a relational schema diagram specifying the foreign keys for this schema.**