Subject Code CS 301	Database Systems (DS)	Credits: 4 (3-1-0) Total hours: 56
Course Objectives	This course covers the relational database systems RDBS - the predominant system for business, scientific and engineering applications at present. The topics are reinforced using tools such as Oracle server in labs. The course includes entity-relation model, normalization, relational model, relational algebra, and data access queries as well as an introduction to SQL.	

Module 1	**12 Hours**
Introduction: An overview of database management system, database system vs file system, database system concept and architecture, data model schema and instances, data independence and database language and interfaces,(DDL,DML,DCL), overall database structure, database users. Data modeling using the Entity Relationship model: ER model concepts, notation for ER diagram, mapping constraints, keys, specialization, generalization, aggregation, reduction of an ER diagrams to tables, extended ER model, relationship of higher degree.	
Module 2	**14 Hours**
Relational data Model and Language: Relational data model concepts, integrity constraints, entity integrity, referential integrity, key constraints, domain constraints, relational algebra, relational calculus, tuple and domain calculus.Introduction on SQL: Characteristics of SQL, advantage of SQL, SQL data type and literals, types of SQL commands, SQL operators and their procedure, tables, views and indexes, queries and sub queries, aggregate functions, insert, update and delete operations, joins, unions, intersection, minus, cursors, triggers, procedures in SQL/PL SQL.	
Module 3	**18 Hours**
Data Base Design & Normalization: Functional dependencies, primary key, foreign key, candidate key, super key, normal forms, first, second, third normal forms, BCNF, 4th Normal form,5th normal form, loss less join decompositions, canonical cover, redundant cover, synthesis the set of relation , MVD, and JDs,inclusion dependence, transaction processing concept, transaction system, testing ofserializability, serializability of schedules, conflict & view serializable schedule, recoverability, Recovery from transaction failures, log based recovery, deadlock handling.	
Module 4	**12 Hours**
Concurrency Control Techniques: Concurrency control, locking techniques for concurrency control, 2PL, time stamping protocols for concurrency control, validation based protocol, multiple granularity, multi version schemes and recovery with concurrent transaction. Storage: Introduction, secondary storage devices, tertiary storage, buffering of blocks, structure of files, file organization, indexing and hashing, types of single level ordered indexes, multilevel indexes, dynamics multilevel indexes using B-trees and B+- Trees, database security.	

Reference books	(1) Korth, Silberschatz, "Database System Concepts", 4th ed., TMH, 2003. (2) Elmsari and Navathe, "Fundamentals of Database Systems", 4th ed., A. Wesley, 2004 (3) Raghu Ramakrishnan , Johannes Gehrke, " Database Management Systems", 3rd Edition, McGraw- Hill, 2003. (4) J D Ullman, "Principles of database systems", Computer Science Press, 2001.

Database Systems

- **Data:**

  Data are recorded representation of physical objects, abstract things, events and facts which affects in decision making.

- **Data Structure:**

  Data structure is a particular way of organizing data in a computer so that it can be used efficiently.

  Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.

- **Database:**

  Database is an interrelated data which related to a particular enterprise.

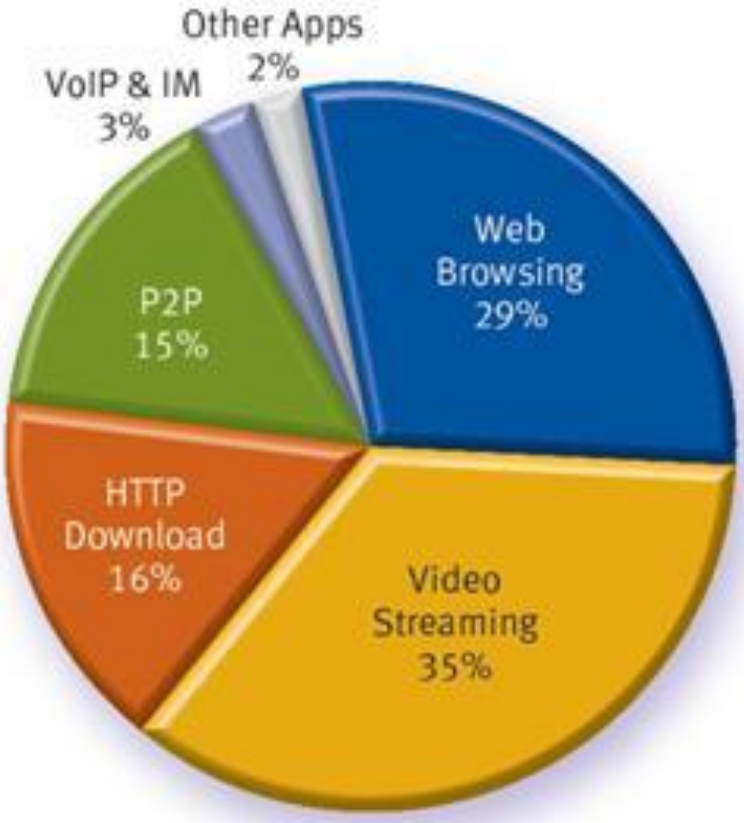
  or

  Database is an collection of data organized in a particular way.

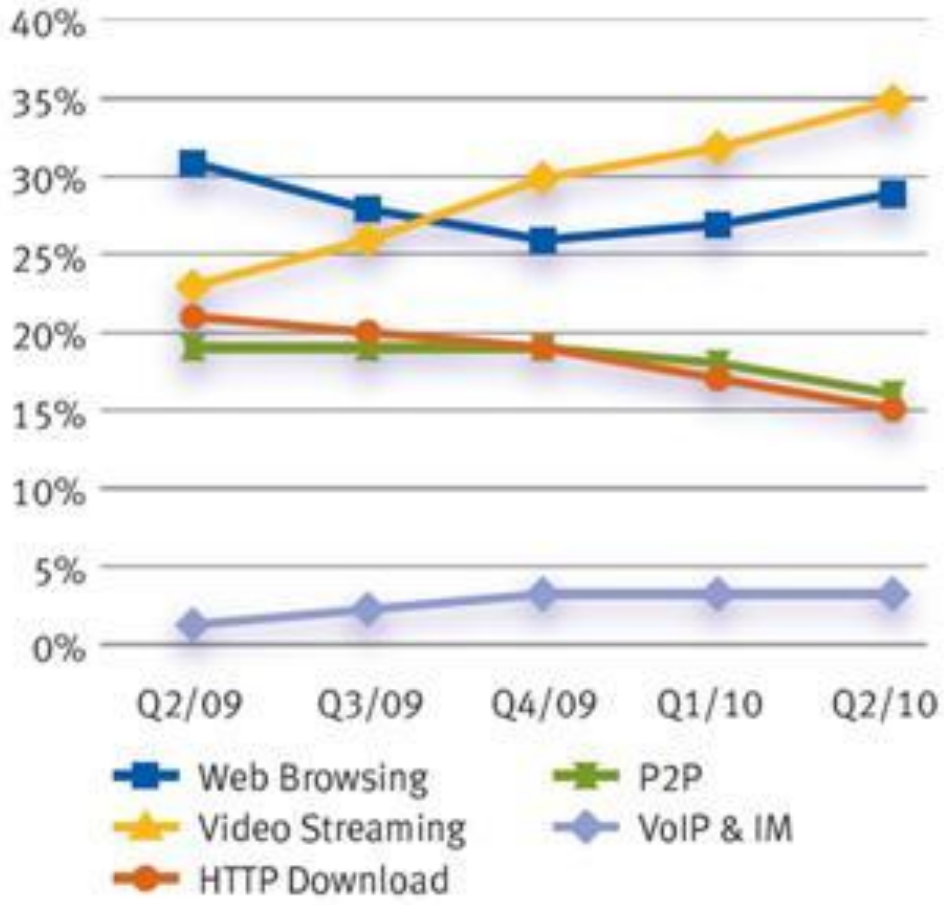
  or

  In simple terms we can say database is an interrelated data.

Application Breakdown



Mobile data usage broken down by top applications, H1/10



Mobile data usage trends broken down by top applications, Q2/09-Q2/10

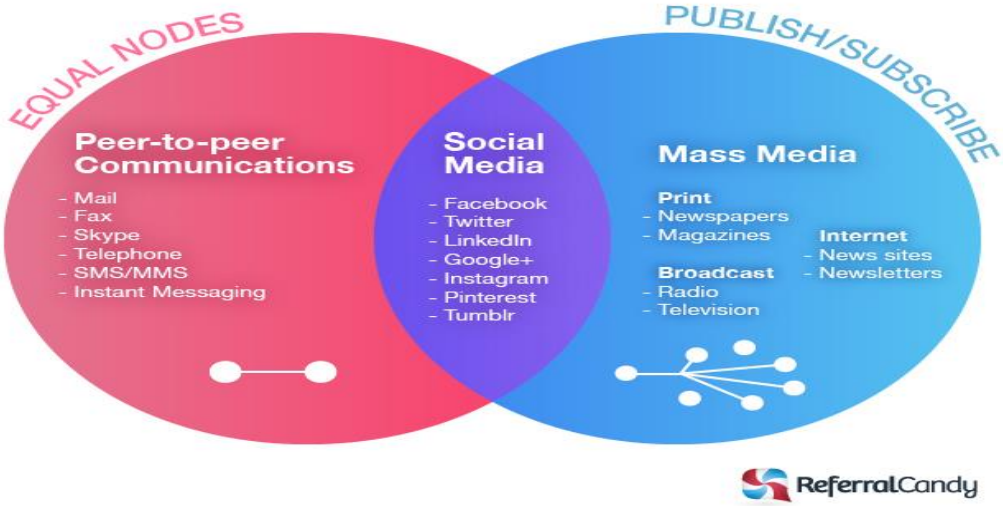

DOMO
DATA NEVER SLEEPS
How Much Data Is Generated Every Minute?
EVERY MINUTE of the DAY
THE MOBILE WEB RECEIVES 217 NEW USERS.
YOUTUBE USERS UPLOAD 48 HOURS OF NEW VIDEO
EMAIL USERS SEND 204,166,667 MESSAGES.
GOOGLE RECEIVES OVER 2,000,000 SEARCH QUERIES.
FACEBOOK USERS SHARE 684,478 PIECES OF CONTENT.
WORDPRESS USERS PUBLISH 347 NEW BLOG POSTS.
571 NEW WEBSITES ARE CREATED.
CONSUMERS SPEND $272,070 ON WEB SHOPPING.
PERFORM 2,083 CHECK-INS.
TWITTER USERS SEND OVER 100,000 TWEETS.
FLICKR USERS ADD 3,125 NEW PHOTOS.
INSTAGRAM USERS SHARE 3,600 NEW PHOTOS.
TUMBLR BLOG OWNERS PUBLISH 27,778 NEW POSTS.
BRANDS & ORGANIZATIONS ON FACEBOOK RECEIVE 34,722 "LIKES."
APPLE RECEIVES ABOUT 47,000 APP DOWNLOADS.
WITH NO SIGNS OF SLOWING, THE DATA KEEPS GROWING
2.1 MILLION PEOPLE

SOCIAL MEDIA

Social media is defined as a group of Internet-based applications that allow the creation and exchanges of user-generated content.

Social media gives users an easy-to-use way to communicate and network with each other on an unprecedented scale.

Facebook, the social networking site, recorded more than 845 million active users as of December 2011. In the third quarter of 2012, the number of active Facebook users was 1 billion. As of the first quarter of 2015, Facebook had 1.44 billion monthly active users..



Classification of Social Media

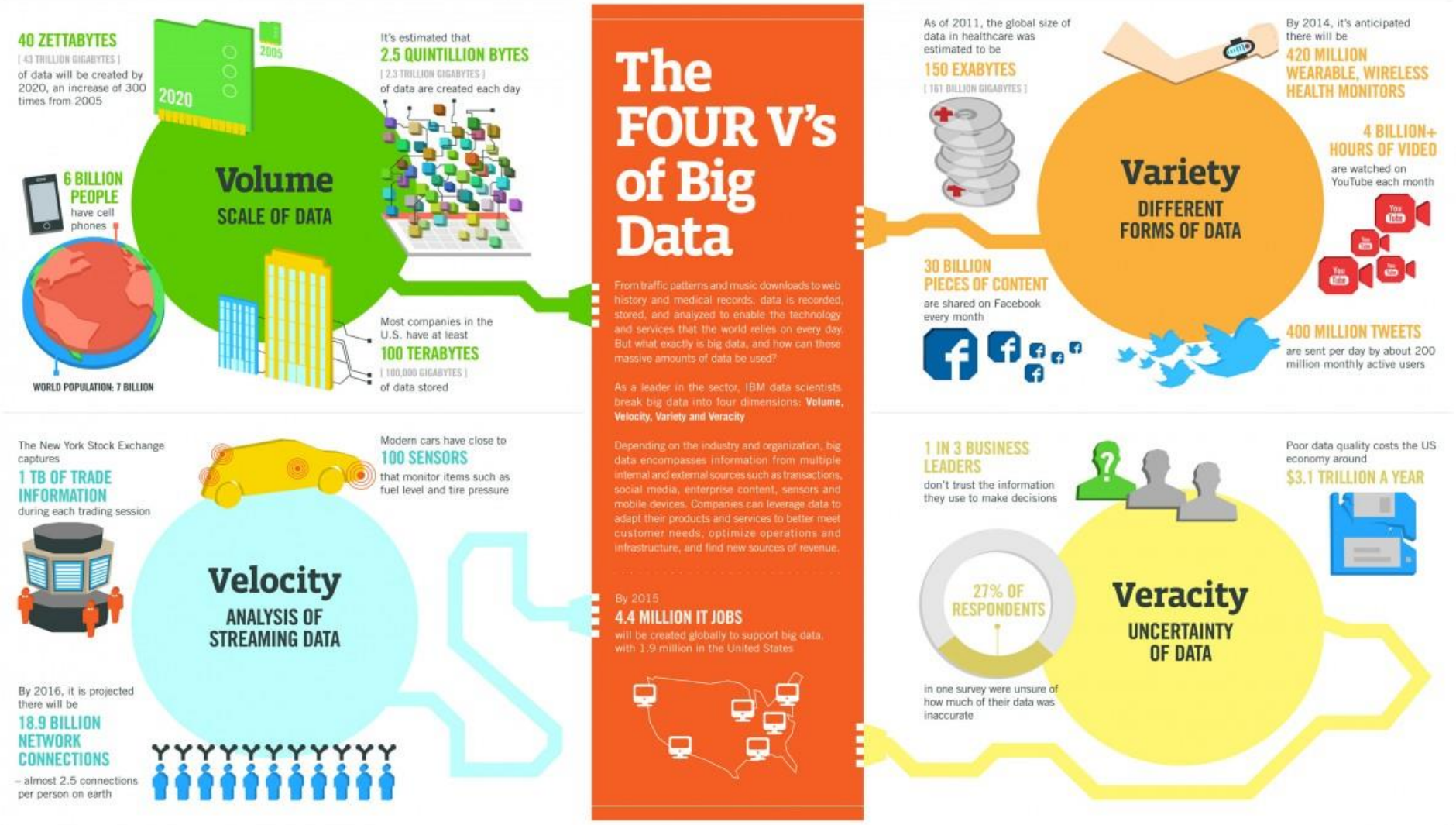
Nine different types of social media:

- Online social networking
- Blogging
- Micro-Blogging
- Wikis
- Social News
- Social Bookmarking
- Media Sharing
- Opinion, Reviews, and Rating
- Answers

Wikis

Introduction…contd.

- **Mining big data streams faces four principal challenges: volume, velocity, variety and veracity.**



Database System

According to Gartner, in 2008, the percentage of database sites using any given technology were (a given site may deploy multiple technologies):[2]

- **Oracle Database - 70%**
- **Microsoft SQL Server - 68%**
- **MySQL (Oracle Corporation) - 50%**
- **IBM DB2 - 39%**
- **IBM Informix - 18%**
- **SAP Sybase Adaptive Server Enterprise - 15%**
- **SAP Sybase IQ - 14%**
- **Teradata - 11%**

Database Systems...contd.

- DBMS is a collection of interrelated data and set of programs to access those data.
- The primary goal of DBMS is to provide a way to store and retrieve database formation that is both convenient and efficient.
- The database system must ensure the safety of information stored, despite system crashes or attempts unauthorized access.

An environment that is both convenient and efficient to use Database Applications:

- **Banking:** All transactions
- **Airlines**: Reservations, schedules
- **Universities**: Registration, grades
- **Sales:** Customers, products, purchases
- **Online retailers**: Order tracking, customized recommendations
- **Manufacturing**: Production, inventory, orders, supply chain
- **Human resources:** Employee records, salaries, tax deductions
- **Databases touch all aspects of our lives**

Database Systems...contd.

- **Database:**

  Database is an interrelated data which related to a particular enterprise.

  or

  Database is an collection of data organized in a particular way.

  or

  In simple terms we can say database is an interrelated data.

- **DBMS is a collection of interrelated data and set of programs to access those data.**

- **The primary goal of DBMS is to provide a way to store and retrieve database formation that is both convenient and efficient.**

- **The database system must ensure the safety of information stored, despite system crashes or attempts unauthorized access.**

Database Systems...contd.

- A database represents some aspect of the real world, sometimes called the mini world or the universe of discourse (UoD). Changes to the mini world are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.
- Example: Banking, finance, student, etc.

- **Database Management System(DBMS):**

DBMS contains information about a particular enterprise

- **Collection of interrelated data**
- **Set of programs to access the data**

Database Systems...contd.

- **The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications.**

- **Defining:**

- Defining a database involves specifying the data types, structures, and constraints of the data to be stored in the database.

- The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called meta-data.

- **Constructing**

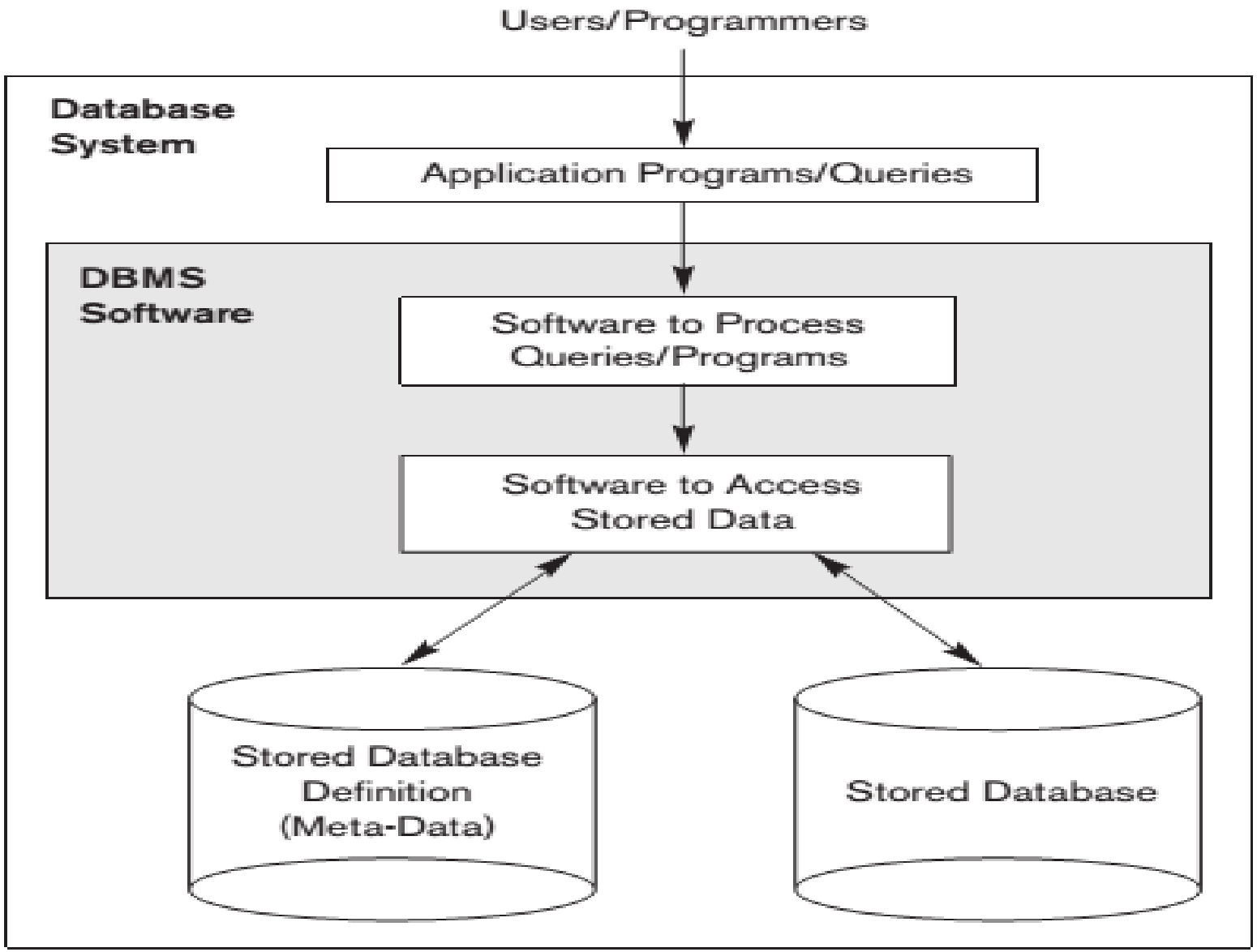
Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS.

- **Manipulating:**

Manipulating a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld.

Database Systems...contd.

Database System Environment



Database Systems...contd.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Student Database

Database Systems...contd.

- In the early days, database applications were built directly on top of file systems

- Drawbacks of using file systems to store data

  - Data redundancy and inconsistency
    - Multiple file formats, duplication of information in different files

  - Difficulty in accessing data
    - Need to write a new program to carry out each new task

  - Data isolation -Multiple files and formats

  - Integrity problems

    - Integrity constraints (e.g. account balance > 0) become “buried” in program code rather than being stated explicitly
    - Hard to add new constraints or change existing ones

Database Systems...contd.

- **Atomicity of updates**

  - **Failures may leave database in an inconsistent state with partial updates carried out**
  - **Example:**

    Transfer of funds from one account to another should either complete or not happen at all

- **Concurrent access by multiple users**

  -Concurrent accessed needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
  -Example:

  Two people reading a balance and updating it at the same time

- **Security problems**

  - Hard to provide user access to some, but not all data
  - Database systems offer solutions to all the above problems

Characteristics of the Database Approach

- **Self-describing nature of a database system**

  Database system contains not only the database itself but also a complete definition or description of the database structure and constraints (Meta Data).

- **Insulation between programs and data, and data abstraction**

  The structure of data files is stored in the DBMS catalog separately from the access programs.

- **Support of multiple views of the data**

  - **A database typically has many users, each of whom may require a different perspective or view of the database.**

  - **A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored.**

- **Sharing of data and multiuser transaction processing**

  - **Data for multiple applications (online transaction processing (OLTP)) is to be integrated and maintained in a single database.**

  -The DBMS must include concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct.

Database Systems...contd.

The characteristic that allows program-data independence and program-operation independence is called data abstraction.

or

Generally refers to the suppression of details of the data organization and storage, and the highlighting of the essential features for improved understanding of data

- **Levels of abstraction:**
  - **Physical level: Describes how a record (e.g., customer) is stored.**
  - **Logical level: Describes data stored in database, and the relationships among the data.**

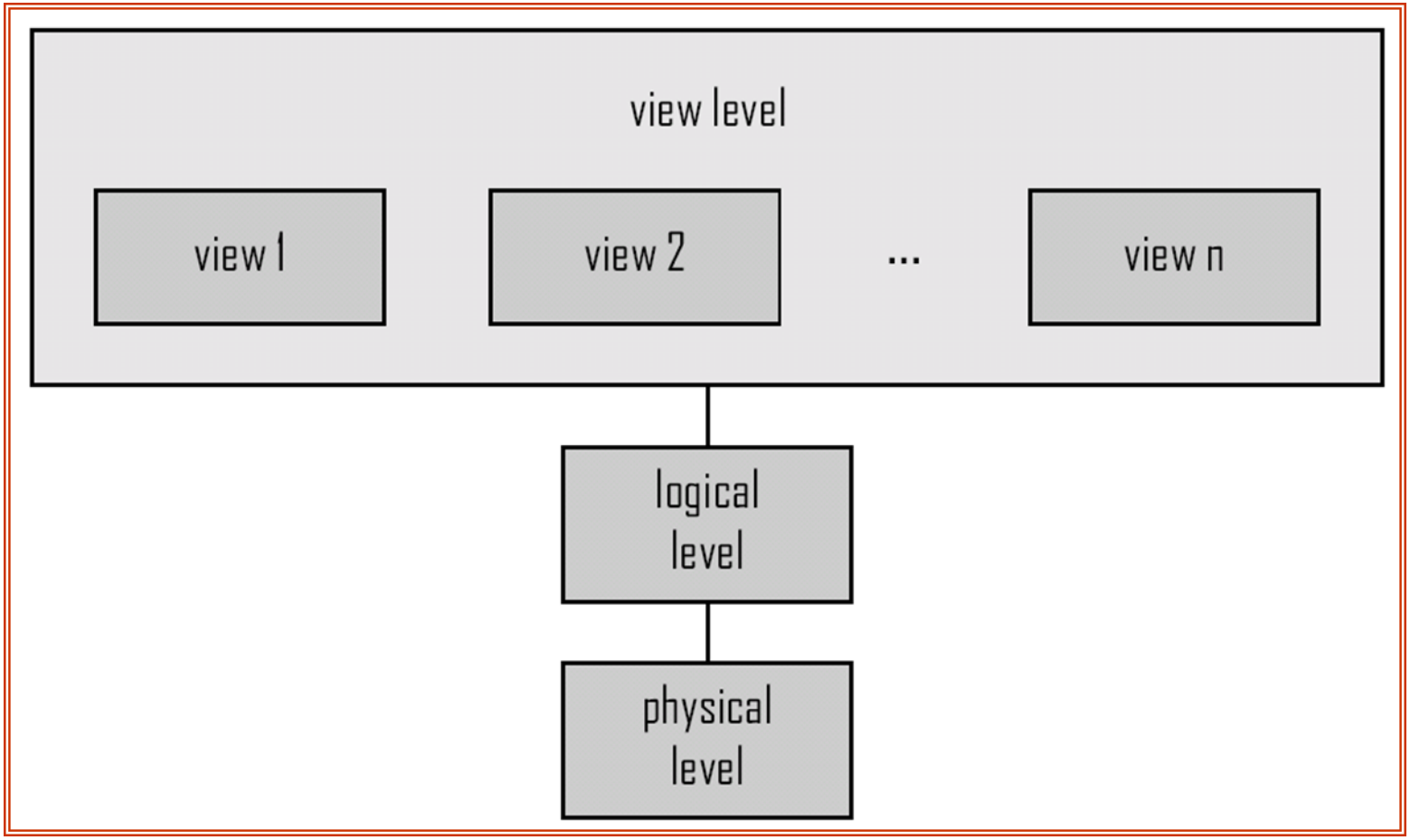
    Example:

```
type customer = record
                    customer_id:string;
                    customer_name:string;
                    customer_street:string;
                    customer_city : string;
                end;
```

  - **View level: Application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.**

View of Data

An architecture for a database system



Basic Terminologies

Schema: The logical structure of the database

- **Analogous to type information of a variable in a program**
- **Physical schema: Database design at the physical level**
- **Logical schema: Database design at the logical level**

Instance: The actual content of the database at a particular point in time

- **Analogous to the value of a variable**

A data model

- **A collection of concepts that can be used to describe the structure (data types, relationships, and constraints that apply to the data) of a database**

- **Most data models also include a set of basic operations for specifying retrievals and updates on the database**

Database Systems...contd.

customer_id	*customer_name*	*customer_street*	*customer_city*
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account_number	*balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

customer_id	*account_number*
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Instance of Database

Data Models

- **Relational Model**

- **The relational model uses a collection of tables to represent both data & the relationships among those data**
- **Each table has multiple columns, and column has a unique name**
- **It is also called as record based models-database is structured in fixed-format records of several types.**
- **E.g**

customer_id	*customer_name*	*customer_street*	*customer_city*
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account_number	*balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

customer_id	*account_number*
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

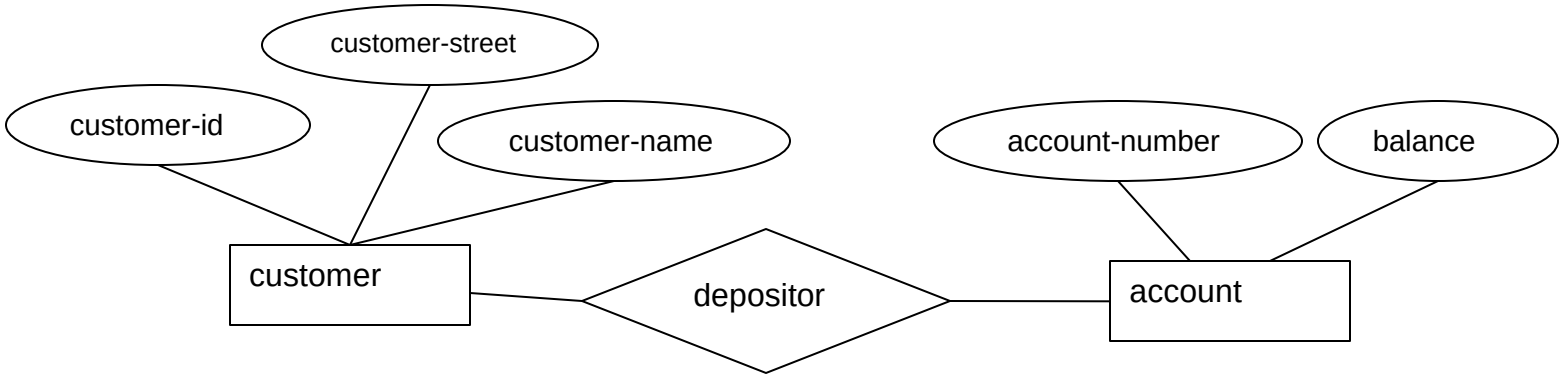
(c) The *depositor* table

The Entity–Relationship Model (ER Model)

- **It is based on the perception that real world consists of collection of basic objects called entity and of relationship among those objects**

- **Entity is an thing or object in the real world and it is distinguishable from other objects**

- **It is widely used in the database design**
- **E.g**



The Object-Based Model

- It combines the features of object-oriented data model and relational data model.

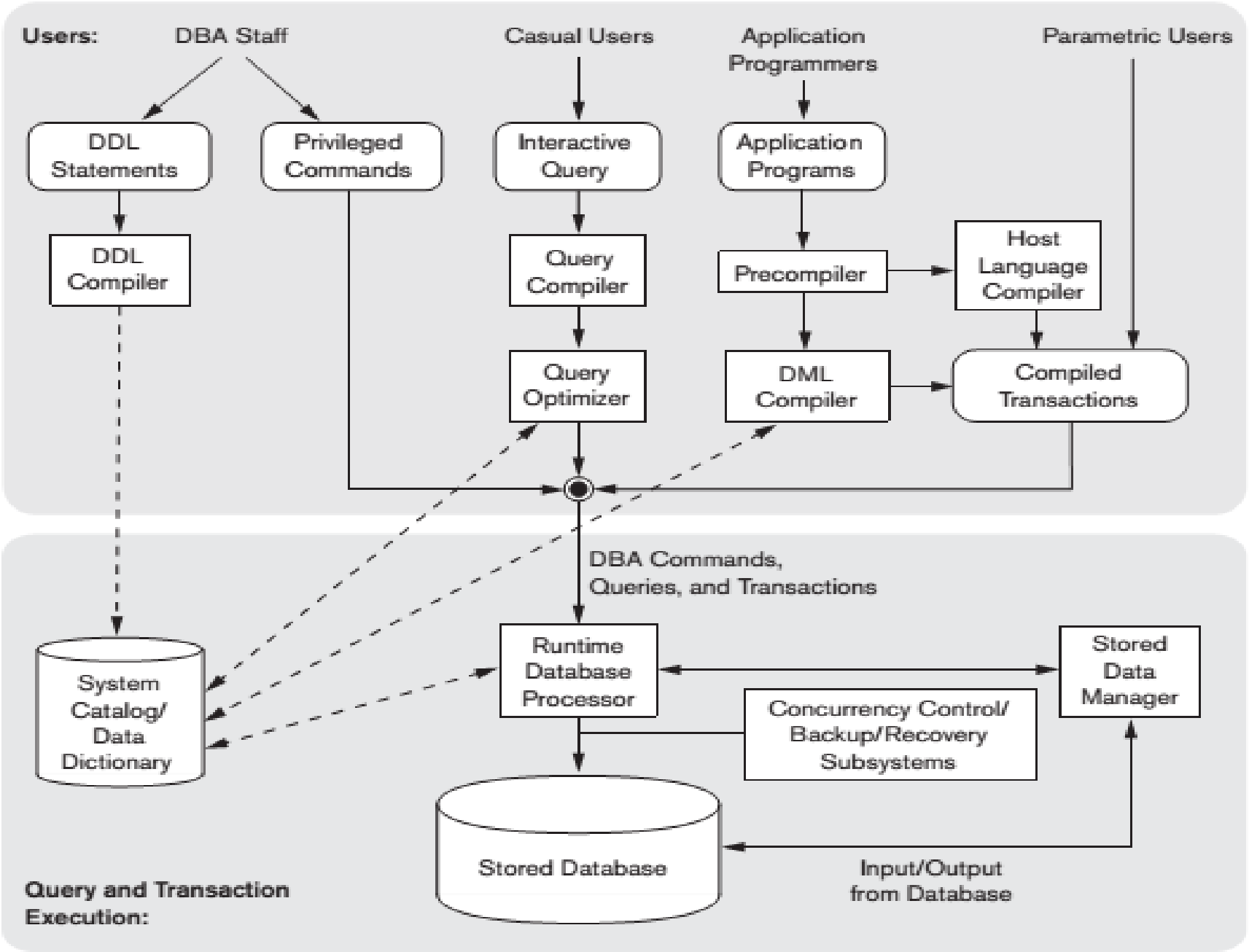
- The object-relational data model extends the relational data model by providing a richer type system including complex data types and object orientation oriented

- The database systems based on the object –relational model provide a convenient migration path for users of relational databases who wish use object-orientated features.

Semistructured Data model

- The semistructured data model permits the specification of data where individual data items of the same type may have different sets of attributes

-The extensible markup Language (XML) is widely used to represent semistructured data



Component Models of a DBMS and their interfaces

Different Actors of Database System

Database Designers:

- Database are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data.

- These tasks are mostly undertaken before the database is actually implemented and populated with data.

- It is the responsibility of database designers to communicate with all prospective database users in order to understand their requirements and to create a design that meets these requirements

Database Systems...contd.

End User: **End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use.**

- **Casual end users: Occasionally access the database, but they may need different information each time.**

- They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.

- **Naïve or parametric end users:**

- Unsophisticated user who interact with the system by invoking one of the application that have been written previously. Make up a sizable portion of database end users.

- Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called canned transactions—that have been carefully programmed and tested.

- The tasks that such users perform are varied:

- **Bank tellers check account balances and post withdrawals and deposits.**
- **Reservation agents for airlines, hotels, and car rental companies check a availability for a given request and make reservations**

Database Systems...contd.

Sophisticated end users: Include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.

Standalone users: Maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces.

- **An example is the user of a tax package that stores a variety of personal financial data for tax purposes.**

- **System Analysts and Application Programmer:**
  - **System analysts determine the requirements of end users, especially naive and parametric end users.**
  - **They develop specifications for standard canned transactions that meet these requirements.**
  - **Application programmers implement these specifications as programs; then they test, debug, document, and maintain these canned transactions.**

Different Actors of Database System

Database Administrators:

- **Schema definition: Executing Data Definition Language(DDL)**
- **Storage Structures and access method definition**
- **Schema and physical-organization modification**
- **Granting of authorization for data access**
- **Routine maintenance:**
  - **Periodically backing the database to prevent from loss of data**
  - **Monitoring jobs running on the database and ensuring performance**
  - **Ensuring free space availability for normal operations**

Database Management System Languages and Interfaces

Database Systems...contd.

Data Manipulation Language:

- **Language for accessing and manipulating the data organized by the appropriate data model. DML also known as query language**

-The types of access are:

- **Retrieval of information stored in the database**
- **Insertion of new information into the database**
- **Deletion of information from the database**
- **Modification of information stored in the database**

Two classes of languages

- **Procedural DML– User specifies what data is required and how to get those data**
- **Declarative (nonprocedural)DML–User specifies what data is required without specifying how to get those data**

Database Systems...contd.

- **General DML statements are:**

SELECT - retrieve data from the a database

INSERT - insert data into a table

UPDATE - updates existing data within a table

DELETE - deletes all records from a table, the space for the records remain

MERGE - UPSERT operation (insert or update)

CALL - call a PL/SQL or Java subprogram

EXPLAIN PLAN - explain access path to data

LOCK TABLE - control concurrency

Database Systems...contd.

- **Data Definition Language**
- **Specification notation for defining the database schema.**

- **DDL also used to specify:**
  - **Domain constraints: Domain of possible value must be associated with every attribute**
  - **Referential integrity: The value that appears in one relation for a given set of attributes should also appears in the another relation for a certain set of attributes**
  - **Assertion: Assertion is any condition that the database must always satisfy.**
  - **Authorization: Read authorization, write authorization, update authorization, delete authorization.**
  - **Example:**

```
create table account (
                    account_number    char(10),
                    branch_name       char(10),
                    balance           integer)
```

- **Referential integrity** – A foreign key must have a matching primary key or it must be null

  -This constraint is specified between two tables (parent and child); it maintains the correspondence between rows in these tables. It means the reference from a row in one table to other table must be valid.

- **Examples**

  - In the Customer/Order database: Customer(**custid**, custname)
    Order(**orderID**, custid, OrderDate)

  - To ensure that there are no orphan records, we need to enforce referential integrity.

  - An orphan record is one whose foreign key value is not found in the corresponding entity – the entity where the PK is located.

  - Recall that a typical join is between a PK and FK.

  - **The referential integrity constraint states that the CustID in the Order table must match a valid CustiD in the Customer table. Most relational databases have declarative referential integrity.**

- When the tables are created the referential integrity constraints are set up.

Database Management System Internals

- Storage management
- Query processing
- Transaction processing

Storage Management

- **The main components of storage manager are:**

- **Authorization and Integrity Manager: Checks integrity constraints & authority of user**
- **Transaction Manager: Ensures database remains in constant state despite system failure**
- **File Manager: Manages allocation of free space**
- **Buffer Manager: Responsible for fetching data from storage into main memory**

- **Storage manager implements several data structures as a part of physical implementation**
  - **Data Files: Stores database itself**
  - **Data Dictionary: Stores metadata**
  - **Indices: Which provides fast access to data items**

Query Processor: **Query processing components includes:**

- **DDL Interpreter:** **Which interprets DDL statements and records the definition in the data dictionary**

- **DML Compiler:** **Which translates DML statements in a query language into a evaluation plan consisting of low-level instructions that query evaluation engine understand.**

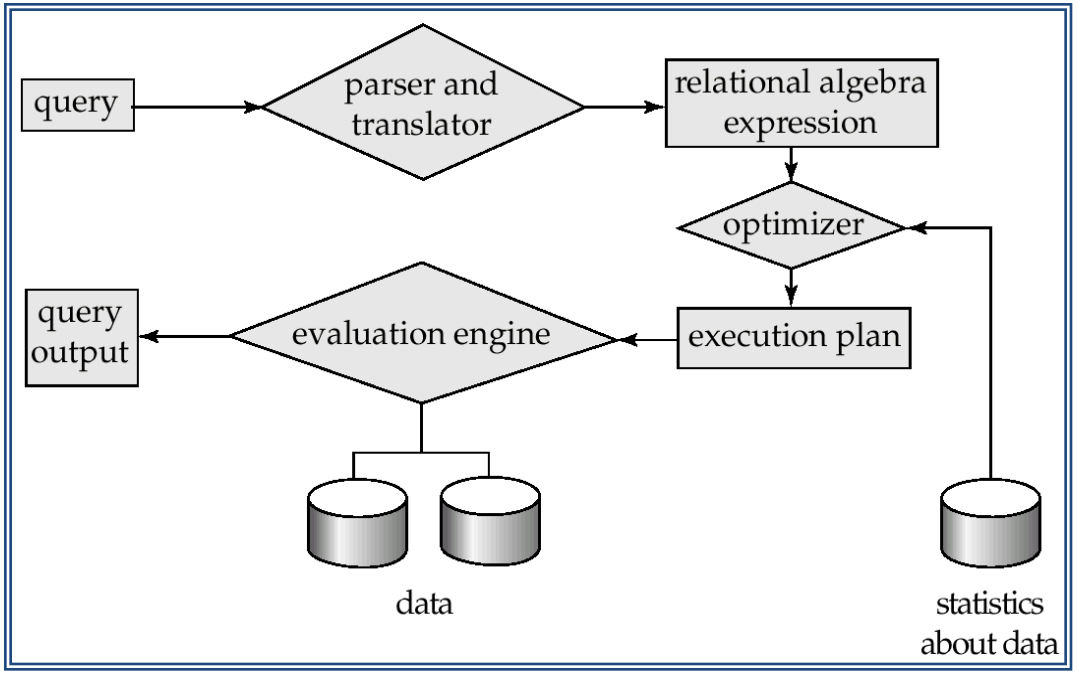
  -DML compiler can also perform query optimization

- **Query Evaluation Engine:** **Which executes low-level instructions generated by the DML compiler**

Query processing

It includes following steps:

1. **Storage Parsing and Translation**
2. **Optimization**
3. **Evaluation**



- **Alternative ways of evaluating a given query**
  - **Equivalent expressions**
  - **Different algorithms for each operation**
- **Cost difference between a good and a bad way of evaluating a query can be enormous**
- **Need to estimate the cost of operations**
  - **Depends critically on statistical information about relations which the database must maintain**
  - **Need to estimate statistics for intermediate results to compute cost of complex expressions**

Transaction Management

- A transaction is a collection of operations that performs a single logical function in a database application

- Transaction-management component ensures that the database remains in a consistent (correct) state despite system failures: power failures and operating system crashes) and transaction failures.

- Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

DBMS Interfaces

- Menu-Based Interfaces for web Clients or Browsing
- Forms-Based Interfaces for Naïve users
- Graphical User Interface
- Natural Language Interface
- Interface for DBA
- Interface for Parametric User

Database Systems...contd.

- **Main inhibitors (costs) of using a DBMS:**
  - **High initial investment and possible need for additional hardware.**
  - **Overhead for providing generality, security, concurrency control, recovery, and integrity functions.**

- **When a DBMS may be unnecessary:**
  - **If the database and applications are simple, well defined, and not expected to change.**
  - **If there are stringent real-time requirements that may not be met because of DBMS overhead.**
  - **If access to data by multiple users is not required.**

- **When no DBMS may suffice:**
  - **If the database system is not able to handle the complexity of data because of modeling limitations**
  - **If the database users need special operations not supported by the DBMS.**

Relational Model

- **The relational model uses a collection of tables to represent both data & the relationships among those data.**

- **Each table has multiple columns, and column has a unique name. 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 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**

Relational Model...contd.

Database schema: **The database schema is a logical design of 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**

E.g. account : information about accounts
depositor : which customer owns which account
customer : information about customers

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

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

Relational Model…contd.

Because tables are relations, we use mathematical terms relation and tuple in place of table and row

Comparison of database terms with programming language terms

- **The concept of relation corresponds to the programming language notion of the variable.**
- **The concept of relation schema corresponds to the programming language notion of the type definition.**
- **The concept of relation instance corresponds to the programming language notion of a value of the variable.**

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

Relational Model...contd.

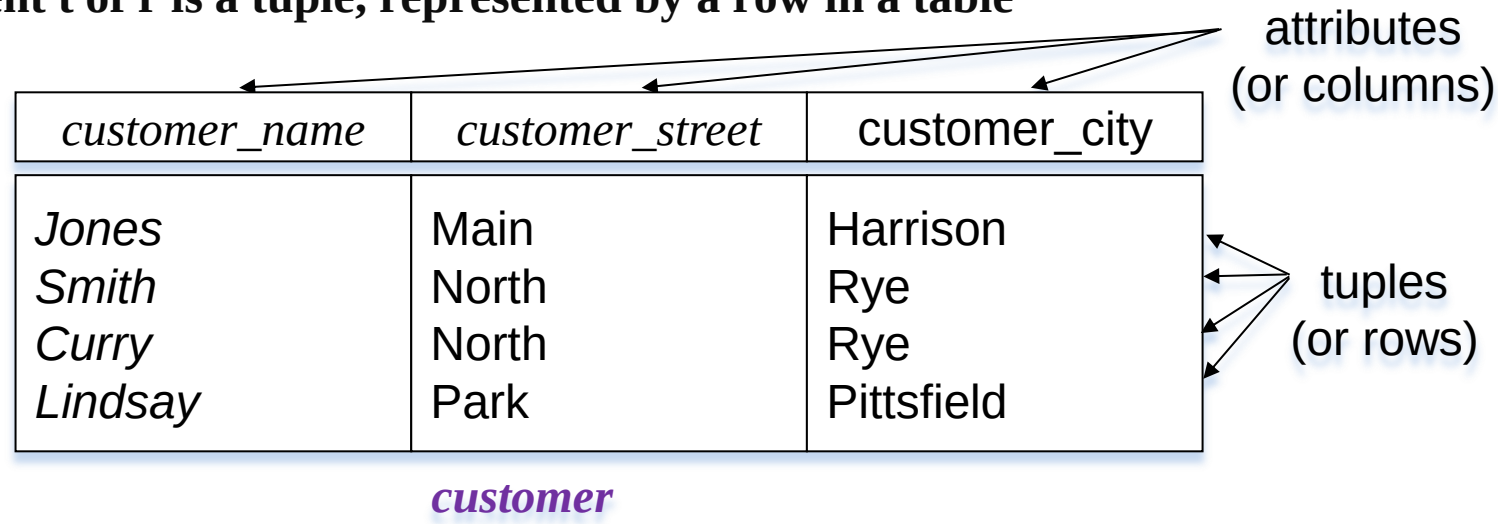
Relation Schema

- Formally, given domains D_1, D_2, …. D_n a relation r is a subset of D_1 x D_2 x … x D_n
  Thus, a relation is a set of n-tuples (a_1, a_2, …, a_n) where each $a_i \in D_i$
- A relation is a subset of cartesian product of a list of domains
- Schema of a relation consists of
  - attribute definitions
    - name
    - type/domain
  - integrity constraints

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



Relational Model...contd.

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

Relational Model...contd.

Why Split Information Across Relations

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 deals with how to design relational schemas

Relational Model...contd.

Keys

- Let K $\subseteq$ R ***K*** **is a superkey of** ***R*** **if values for** ***K*** **are sufficient to identify a unique tuple of each possible relation** ***r(R)*** **by "possible** ***r*** **" we mean a relation** ***r*** **that could exist in the enterprise we are modeling.**

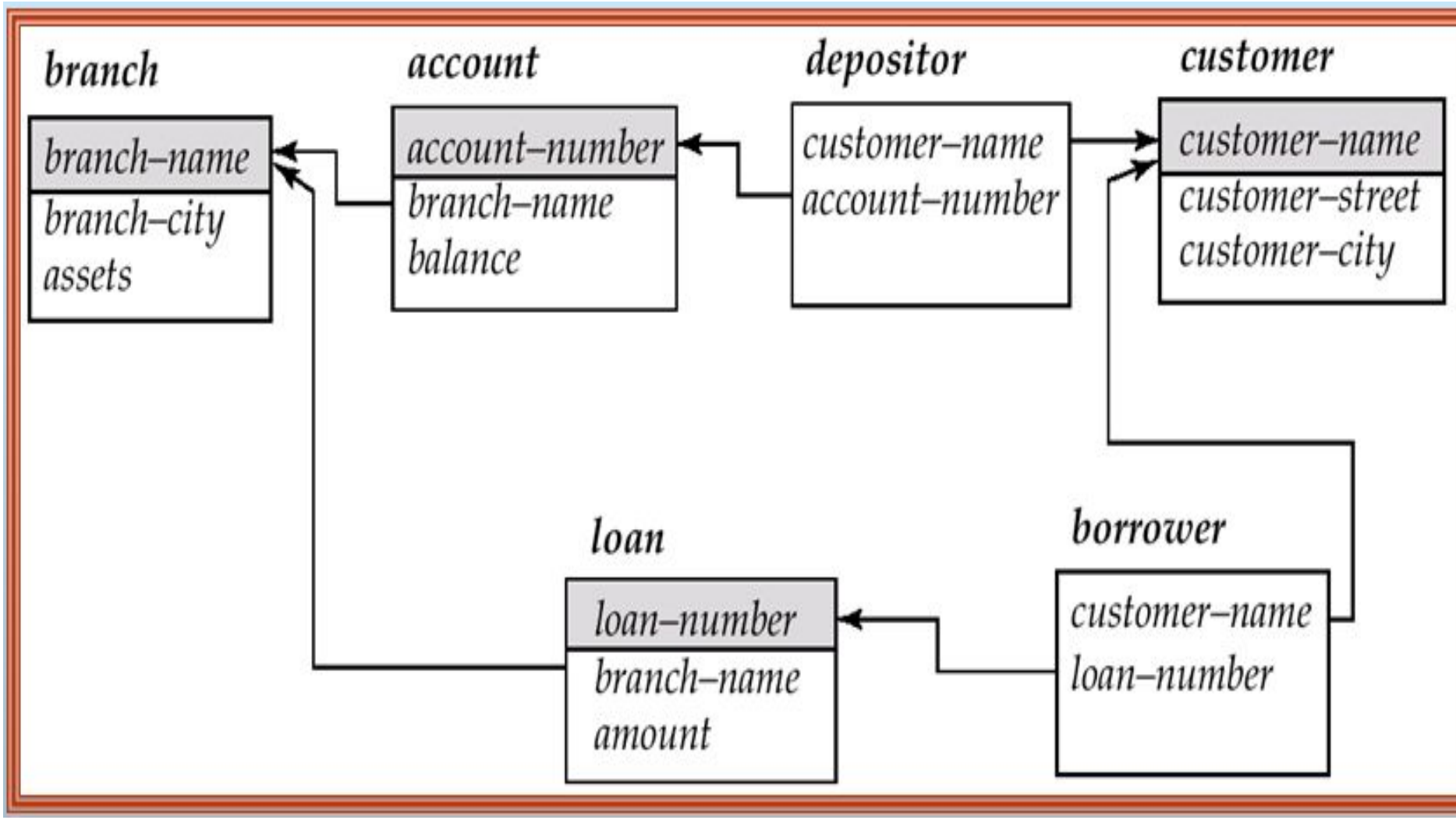
- **Example:{*customer_name,customer_street*} and**
  {*customer_name*}

  are both superkeys of *Customer*, if no two customers can possibly have the same name

- **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.**

- **A superkey is a set of one or more attributes that, taken collectively identify uniquely a tuple in the relation.**

Relational Model...contd.



Banking Enterprise

Relational Model...contd.

- **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**

- **Foreign Key: A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a foreign key.**

  E.g. *customer_name* and *account_number* attributes of *depositor* are foreign keys to *customer* and *account* respectively.

  - Only values occurring in the primary key attribute of the referenced relation may occur in the foreign key attribute of the referencing relation.

Relational Model...contd.

- **Query Languages**
- **A query language is a language in which a user requests information from the database.**
- **In procedural language the user interacts the system to perform a sequence of operations on the database to perform result.**
- **The relational algebra is a pure procedural language.**
- **Relational algebra gives formal foundation for relational model operations**
- **It is used as basis for implementing and optimizing queries in the query processing & optimization model**

Relational Model...contd.

- **Some of the relational algebra concepts are incorporated into the SQL standard query language for RDBMS.**
- **The basic set of operations for relational model is the relational algebra.**
- **A sequence of relation algebra operations forms a relational algebra expression**
- **The result of a retrieval is a new relation which may have been formed from one or more relations**
- **The fundamental operations in the relational algebra are Select, Project, Union, Setdifference, Cartesian product and Rename.**

Relational Model...contd.

Procedural language

Six basic operators

- **select: σ**
- **project: ∏**
- **union: ∪**
- **set difference: –**
- **Cartesian product: x**
- **rename: ρ**

The operators take one or two relations as inputs and produce a new relation as a result.

Select Operation

- Relation r

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

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

A	B	C	D
α	α	1	7
β	β	23	10

Project Operation

- 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

Union Operation

- **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
α	2
β	1
β	3

Set Difference Operation

- **Relations *r*, *s*:**

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- ***r* – *s*:**

A	B
α	1
β	1

Cartesian-Product Operation

- Relations *r*, *s*:

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- *r* x *s*:

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

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:**

$$\rho_x(E)$$

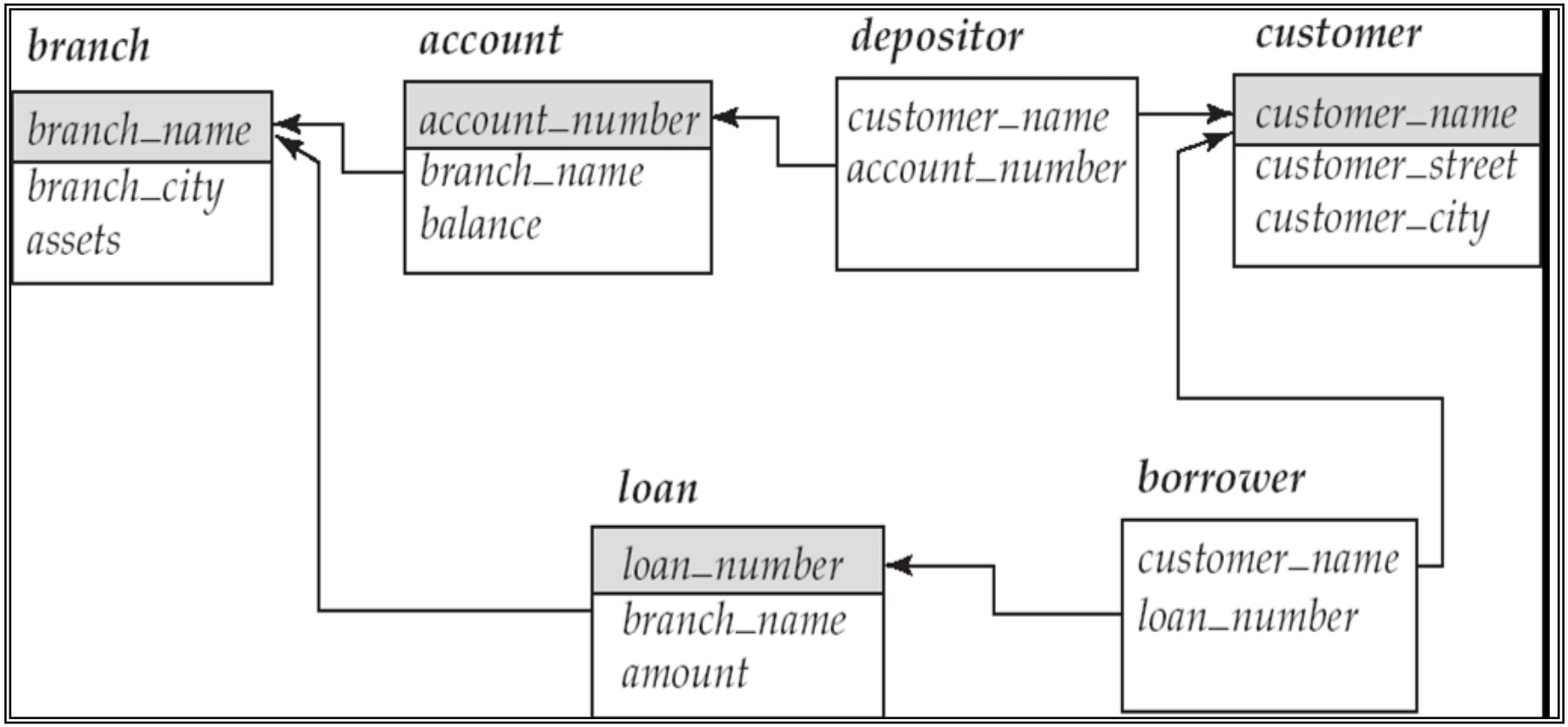
returns the expression *E* under the name *X*

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

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

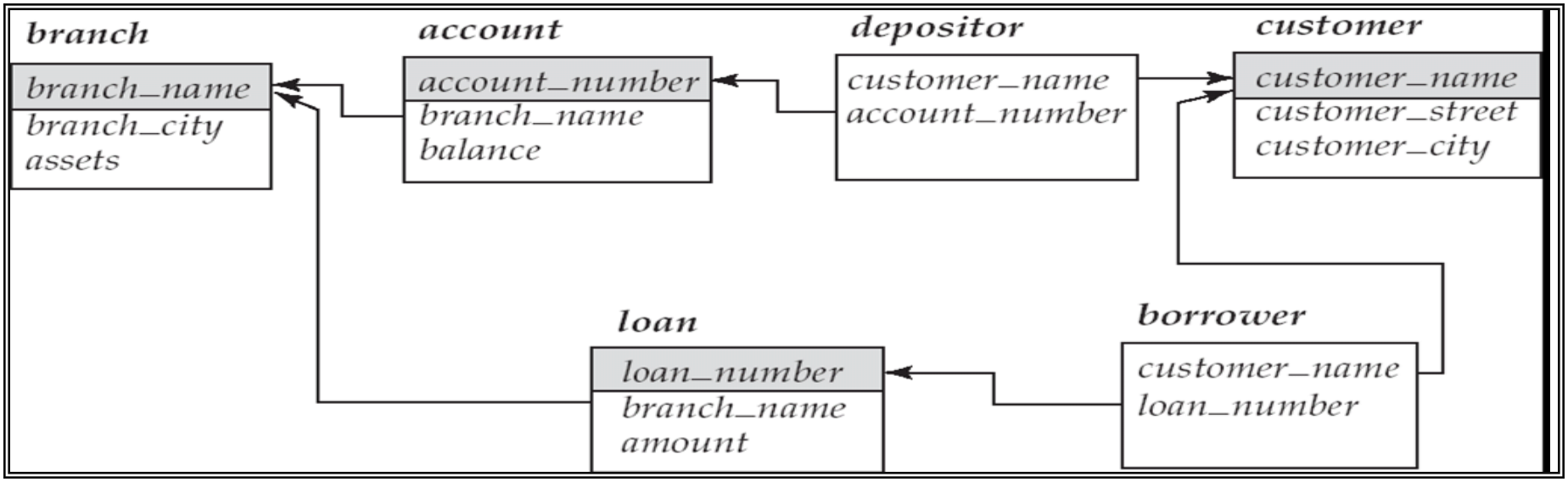
returns the result of expression *E* under the name *X*, and with the attributes renamed to A_1, A_2, …., A_n.

Relational Algebra Queries:



- **Find all loans of over Rs 1200**
- **Find the loan number for each loan of an amount greater than Rs1200**
- **Find the names of all customers who have a loan, an account, or both, from the bank**

Relational Algebra Queries:



- **Find all loans of over Rs1200**

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

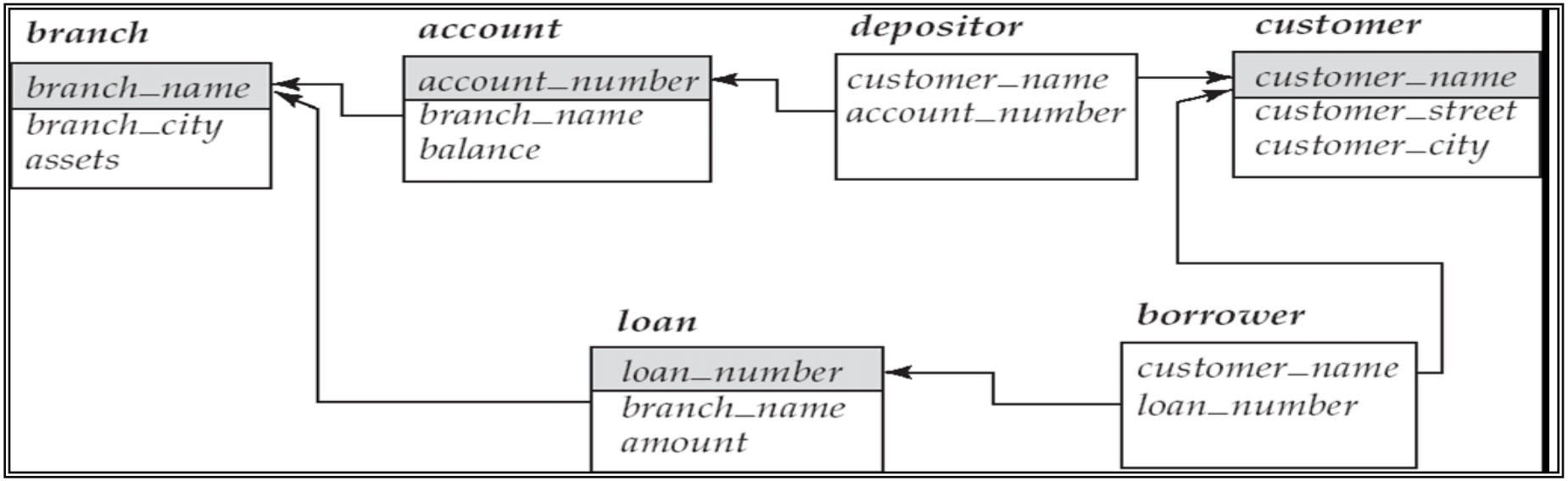
- **Find the loan number for each loan of an amount greater than Rs1200**

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

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

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

Project Operation



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

- **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.**

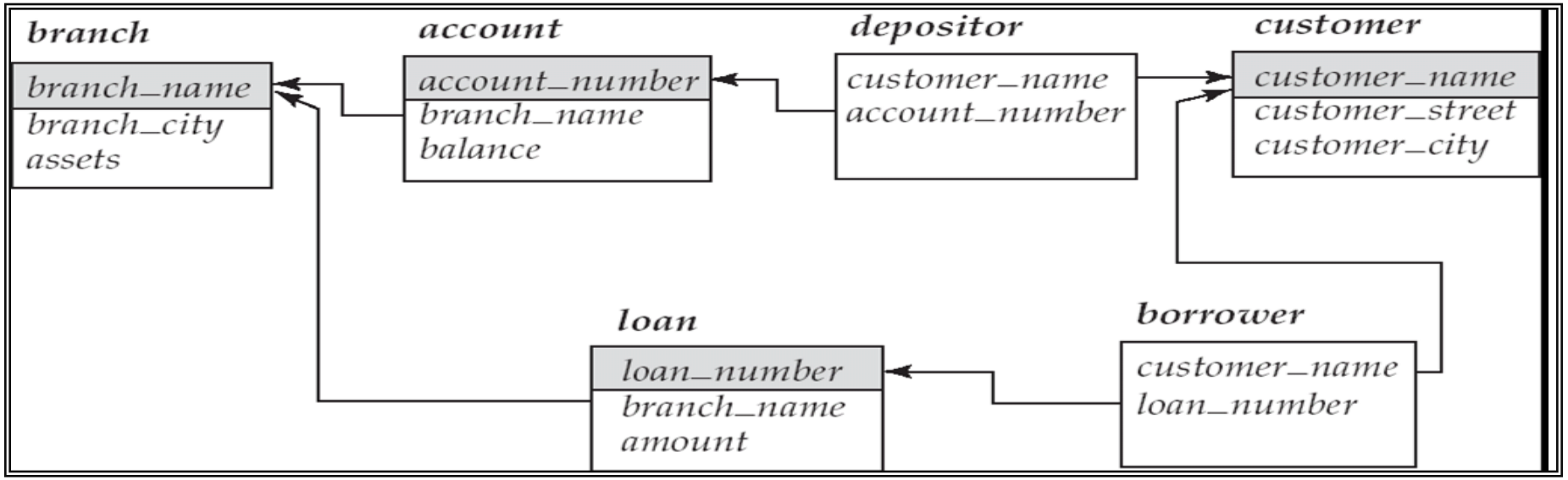
Project Operation

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

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

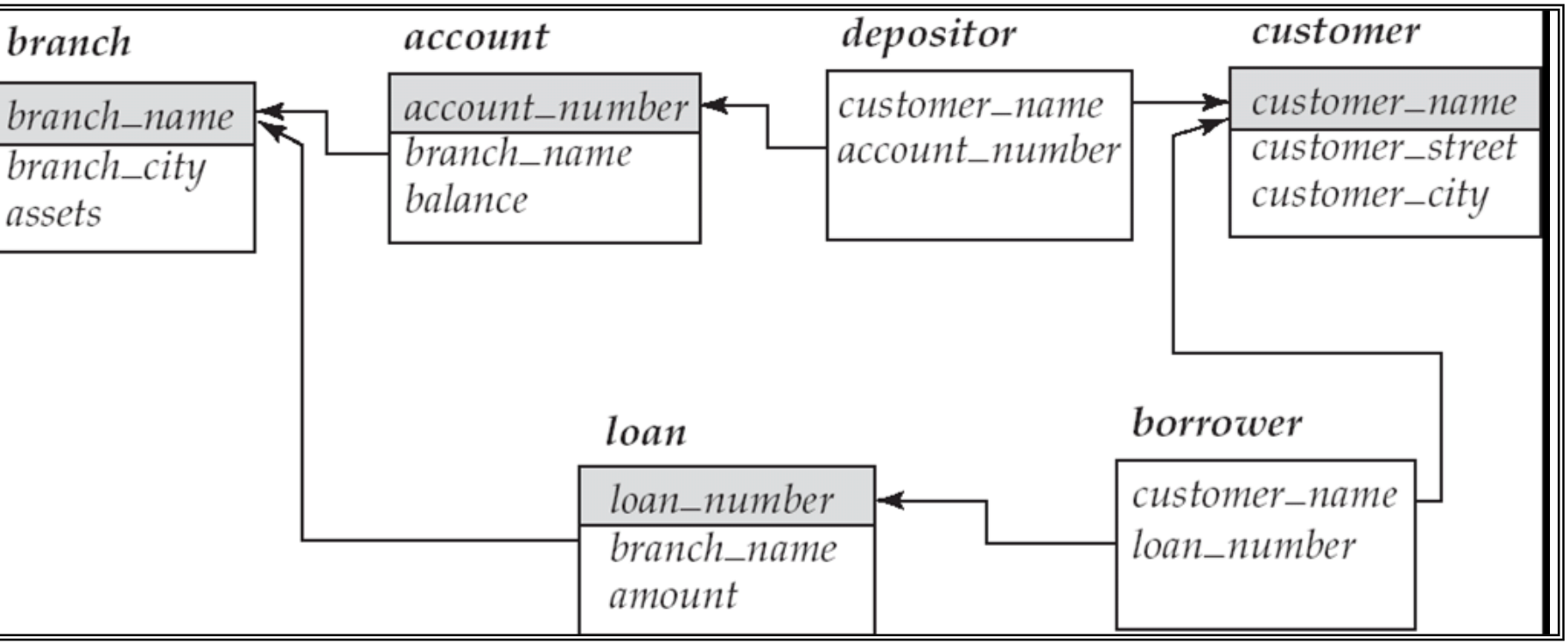
- **Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.**

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



Project Operation

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

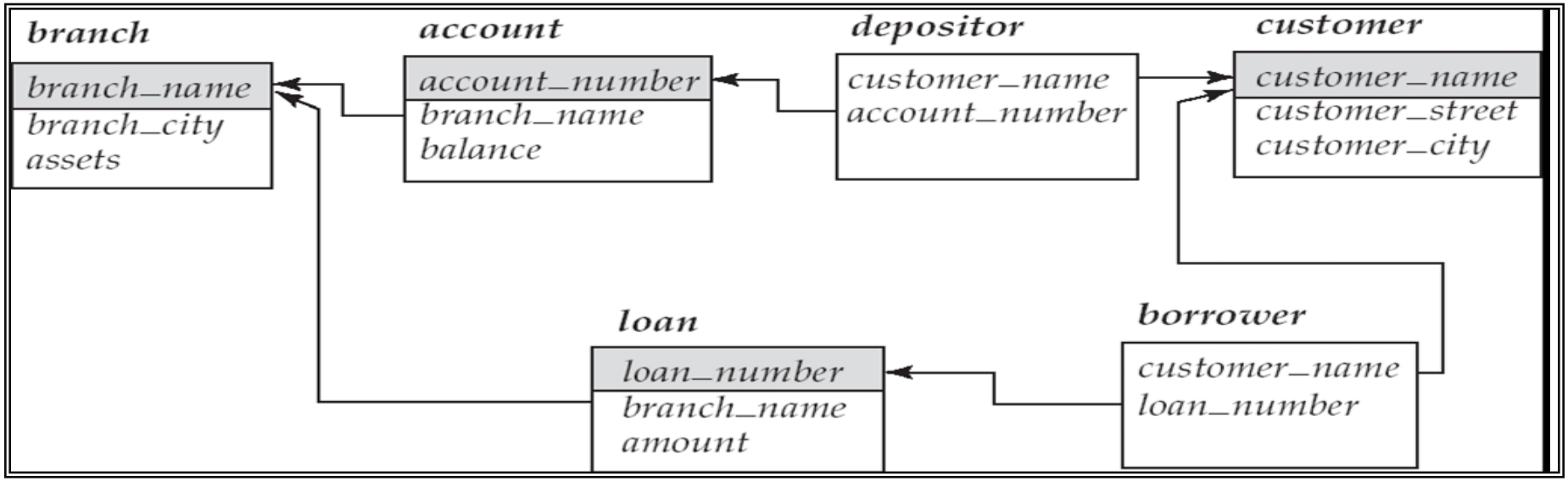


Project Operation

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

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

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



Set-Intersection Operation

- **Additional Operations**
  - **Set intersection (∩)**
  - **Natural join (⋈)**
  - **Aggregation** *(g)*
  - **Outer Join**
  - **Division(÷)**

Set-Intersection Operation

- **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

- 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 ⋈ 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

Natural-Join Operation

- **Notation: $r \bowtie s$**

- Let r and s be relations on schemas R and S respectively.
  Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from r and t_s from s.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
    - t has the same value as t_r on r
    - t has the same value as t_s on s

- **Example:**

  $R = (A, B, C, D)$

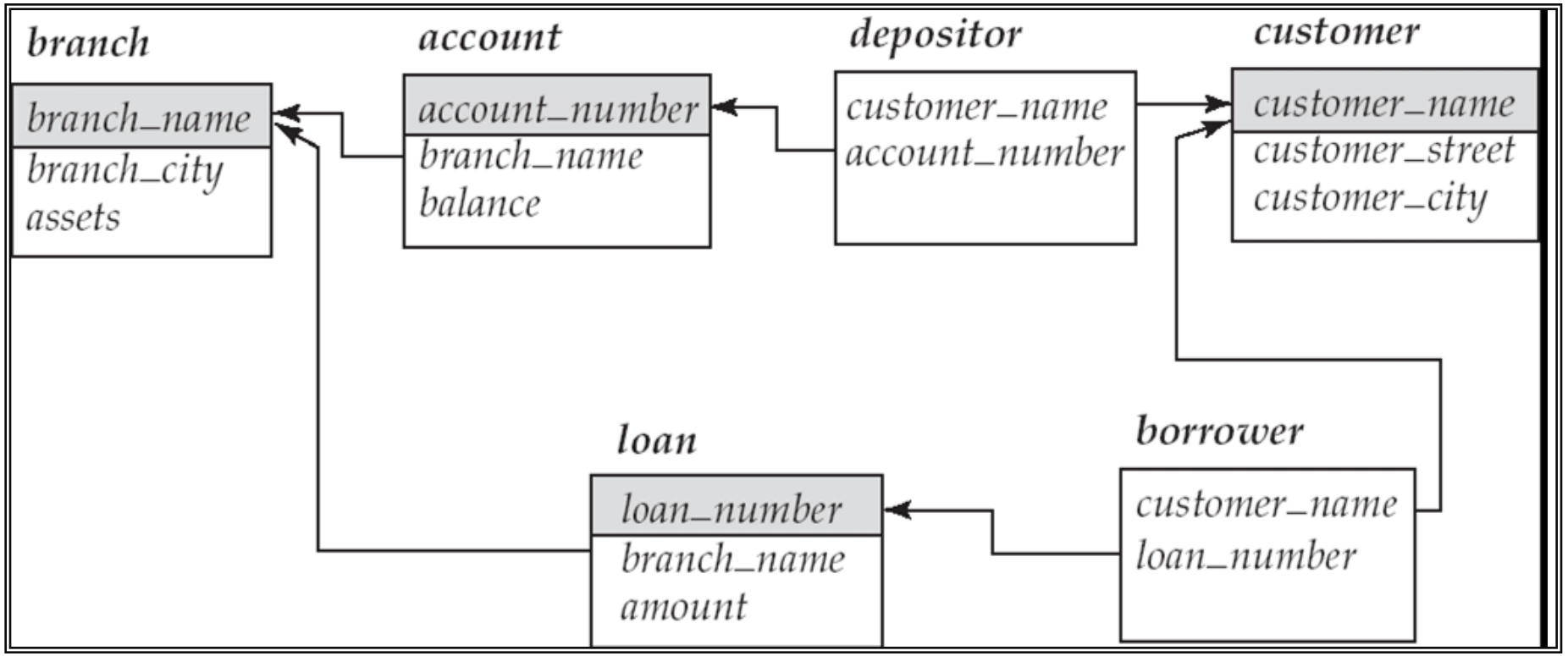
  $S = (E, B, D)$

  - Result schema = (A, B, C, D, E)
  - $r \bowtie s$ is defined as:

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

Natural Join

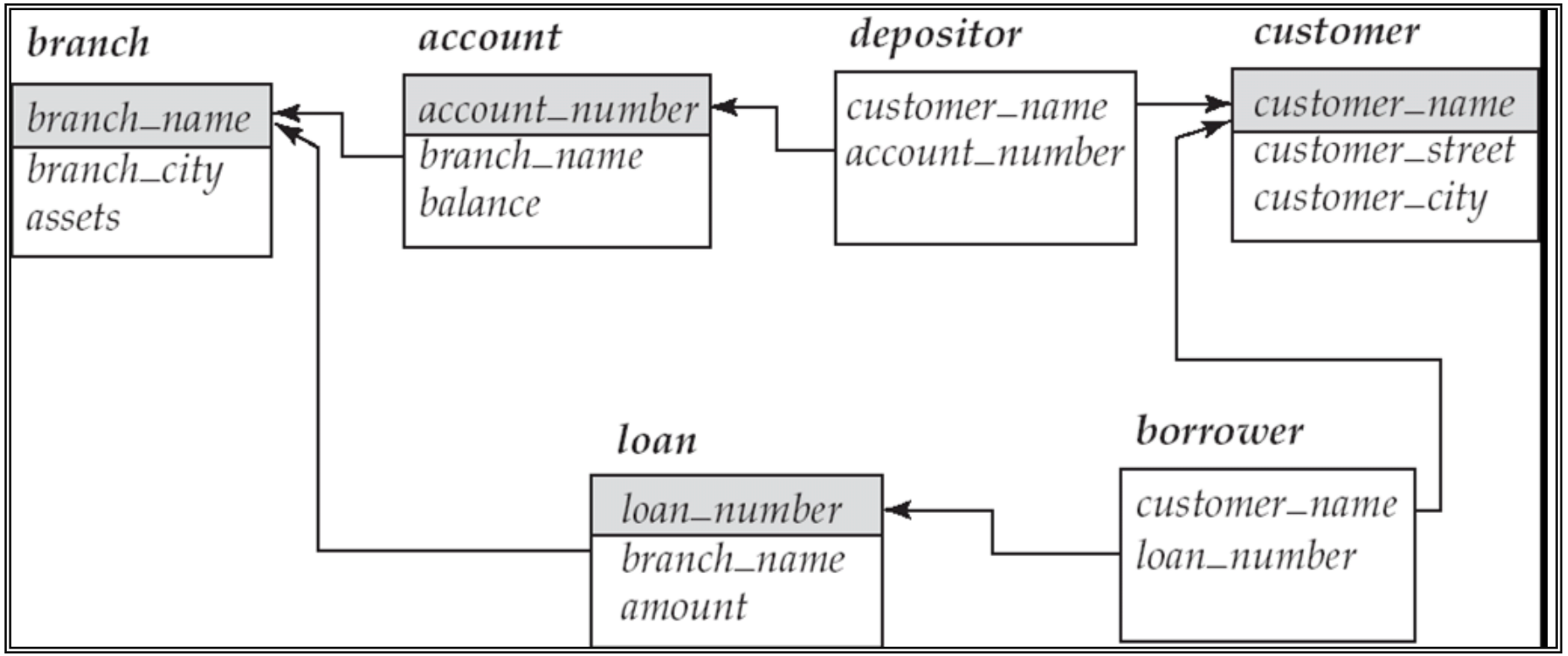
- **Find the name of all customers who have a loan at the bank and the loan amount**



Natural Join

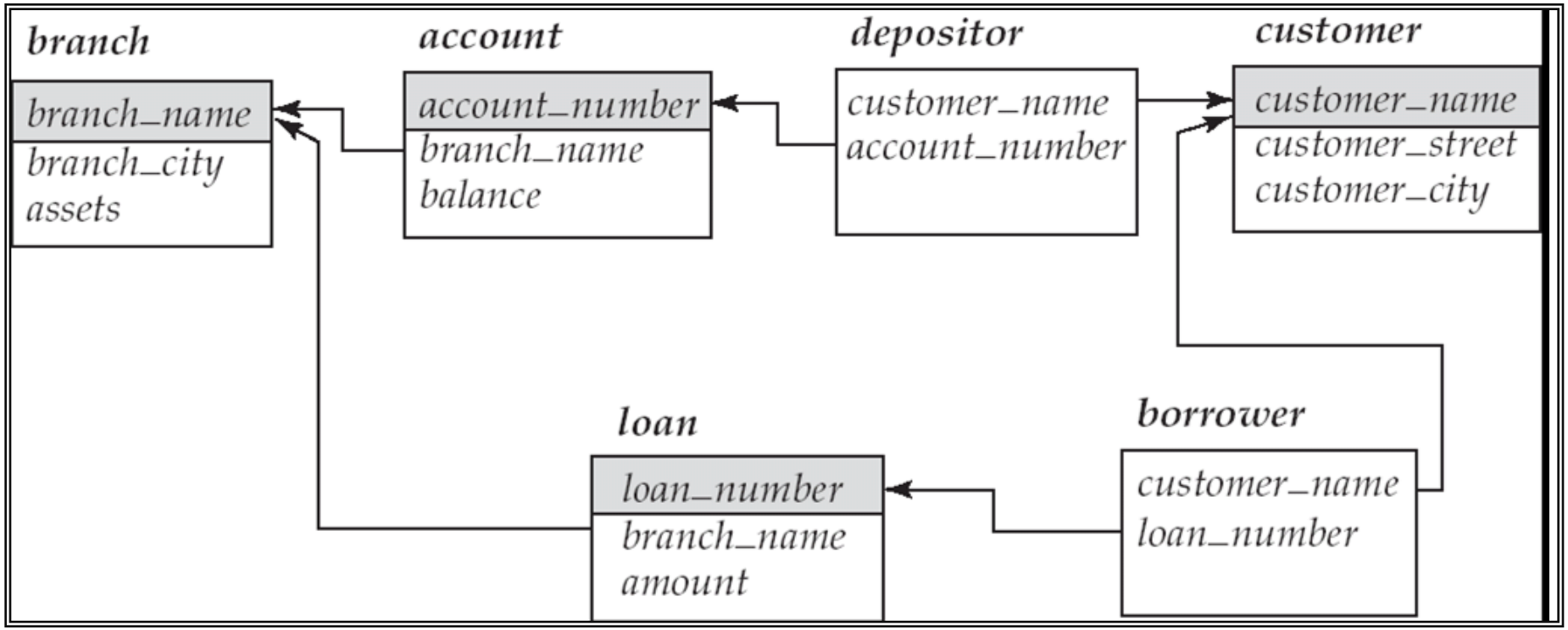
- **Find the name of all customers who have a loan at the bank and the loan amount**

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



Project Operation

- **Find all customers who have an account from at least the "Downtown" and the Uptown" branches.**

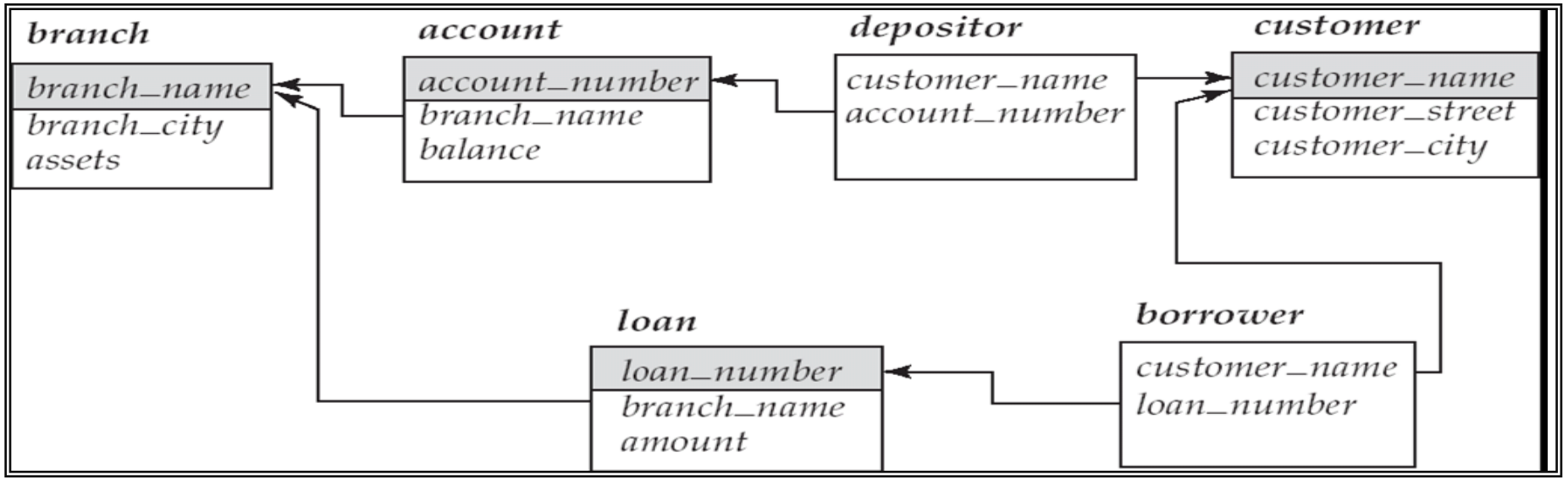


Project Operation

- **Find all customers who have an account from at least the "Downtown" and the Uptown" branches.**

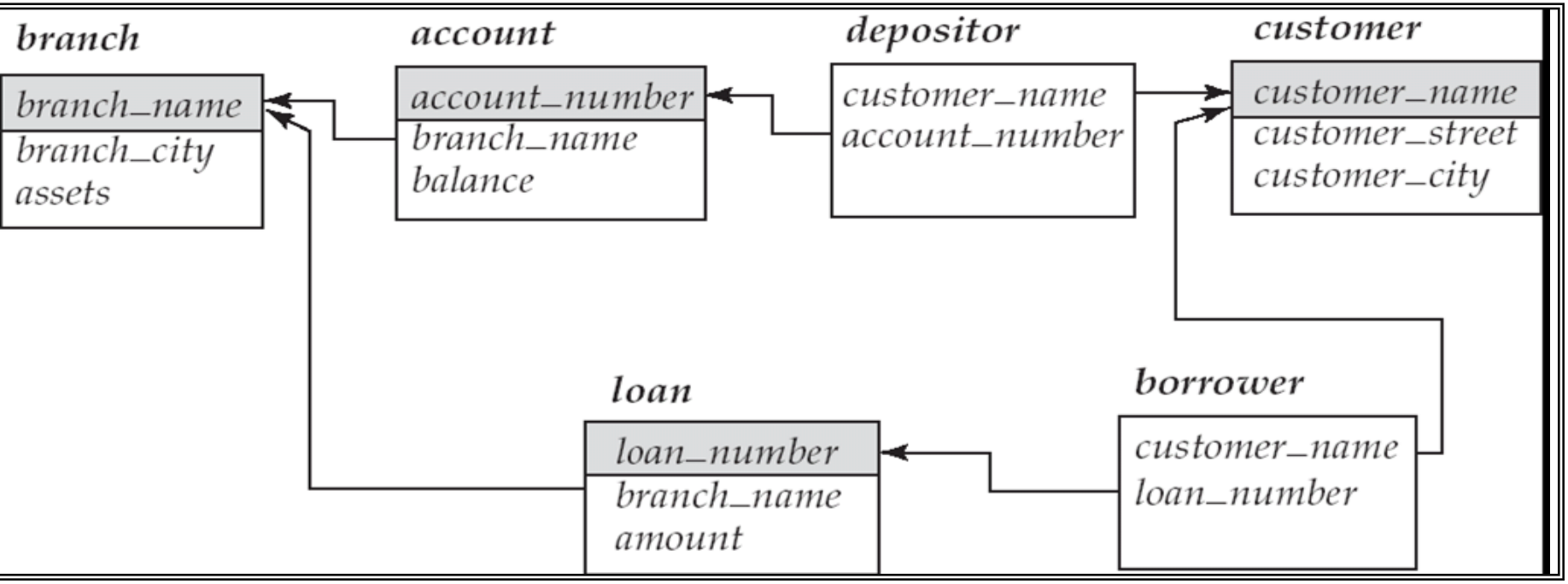
$$\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Downtown"}}(depositor \bowtie account)) \cap$$

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



Project Operation

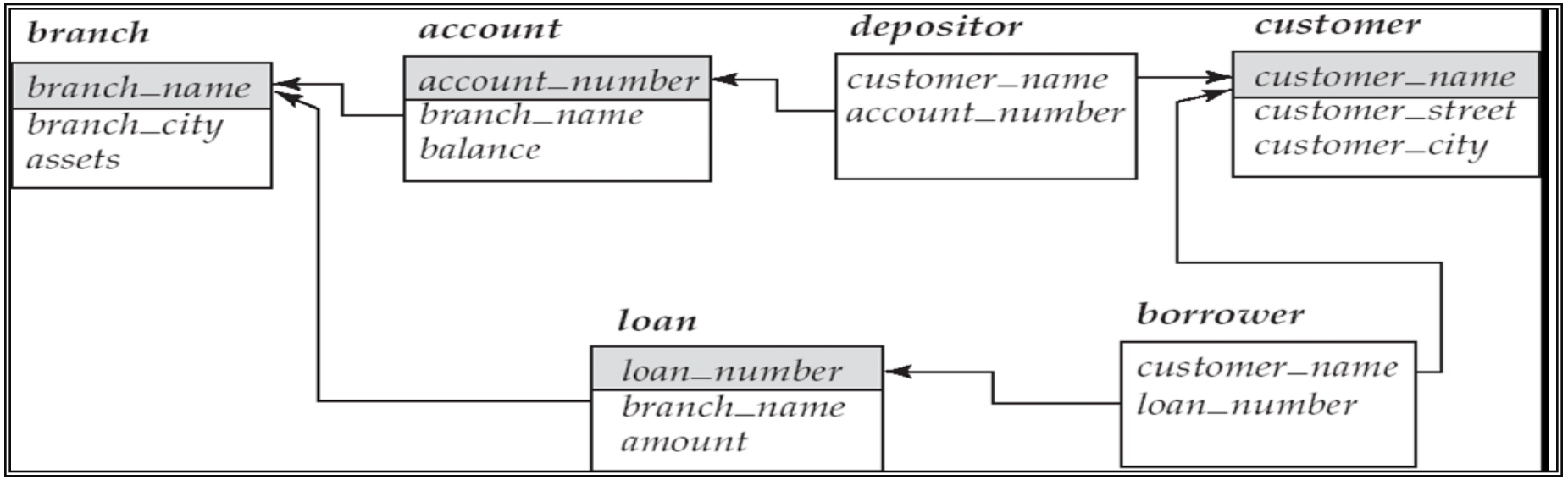
- **Find the largest account balance.**



Project Operation

- **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.**

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



Aggregate Functions and Operations

- **Aggregate functions that summarize data from tables**

- **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**

$$G_1, G_2, \ldots, G_n \; \vartheta_{F_1(A_1), F_2(A_2, \ldots, F_n(A_n))}(E)$$

E **is any relational-algebra expression**

- $G_1, G_2 \ldots, G_n$ **is a list of attributes on which to group**
- **Each** F_i **is an aggregate function**
- **Each** A_i **is an attribute name**

Aggregate Operation

- **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

- **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

Outer Join

- **An extension of the join operation that avoids loss of information.**

- **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.**

Example:

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

- **Join** $loan \bowtie borrower$

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

- **Left Outer Join** $loan \mathbin{⟕} borrower$

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

Outer Join

- Right Outer Join

  loan ⟖ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

- Full Outer Join

  loan ⟗ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

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 (as in SQL)**

- **For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL) Comparisons with null values return the special truth value: unknown**

- **If false was used instead of unknown, then not (A < 5)**
  would not be equivalent to A >= 5

- **Three-valued logic using the truth value unknown:**
  - **OR: (unknown or true) = true,**
    (unknown or false) = unknown
    (unknown or unknown) = unknown
  - **AND: (true and unknown) = unknown,**
    (false and unknown) = false,
    (unknown and unknown) = unknown
  - **NOT: (not unknown) = unknown**

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 ÷ 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

- Relations *r, s*:

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

r

B
1
2

s

- *r* ÷ *s*:

A
α
β

Division Operation

- Relations *r, s*:

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

r

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

s

- *r* ÷ *s*:

A	*B*	*C*
α	a	γ
γ	a	γ

Division Operation

- **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 $S \subseteq R$

$$r \div s = \prod_{R-S}(r) - \prod_{R-S}((\prod_{R-S}(r) \times s) - \prod_{R-S,S}(r))$$

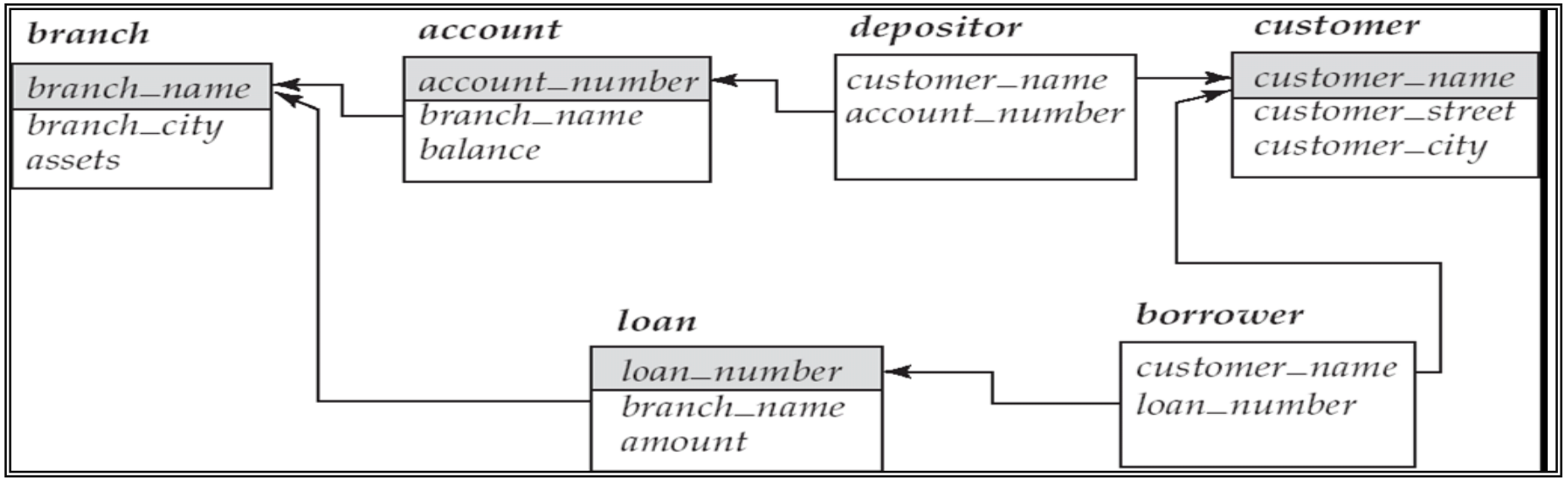
To see why

$\prod_{R-S,S}(r)$ **simply reorders attributes of** r

$\prod_{R-S}(\prod_{R-S}(r) \times s) - \prod_{R-S,S}(r))$ **gives those tuples t in**

$\prod_{R-S}(r)$ **such that for some tuple** $u \in s$, $tu \notin r$.

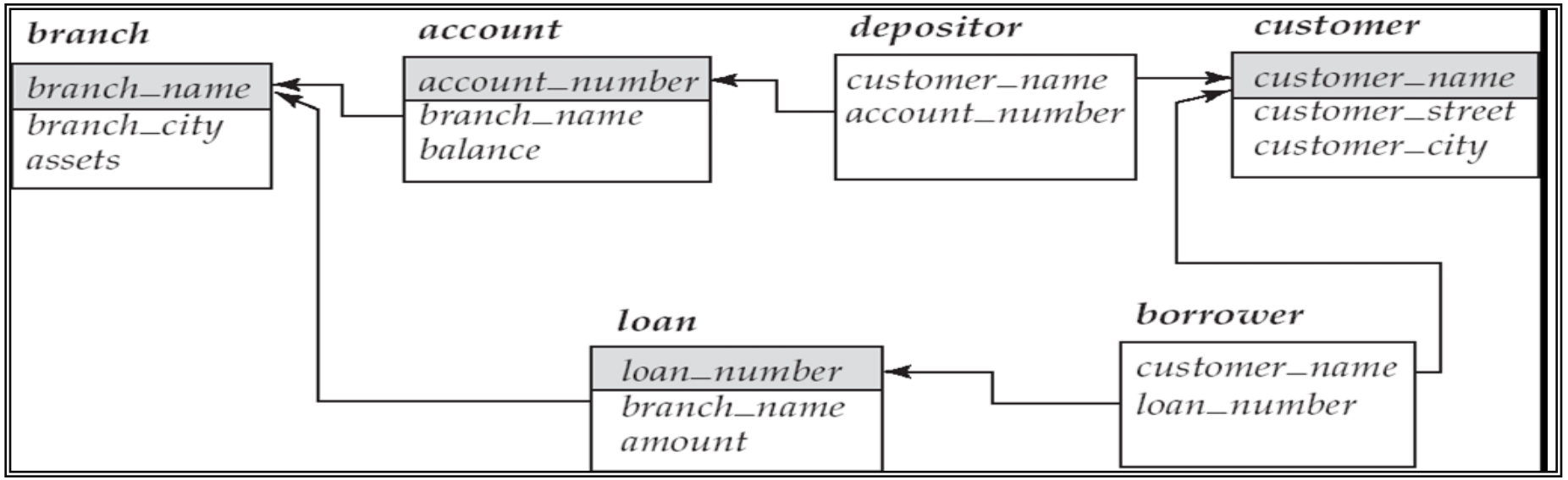
- **Find all customers who have an account at all branches located in Brooklyn city.**



Natural Join and Division

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

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

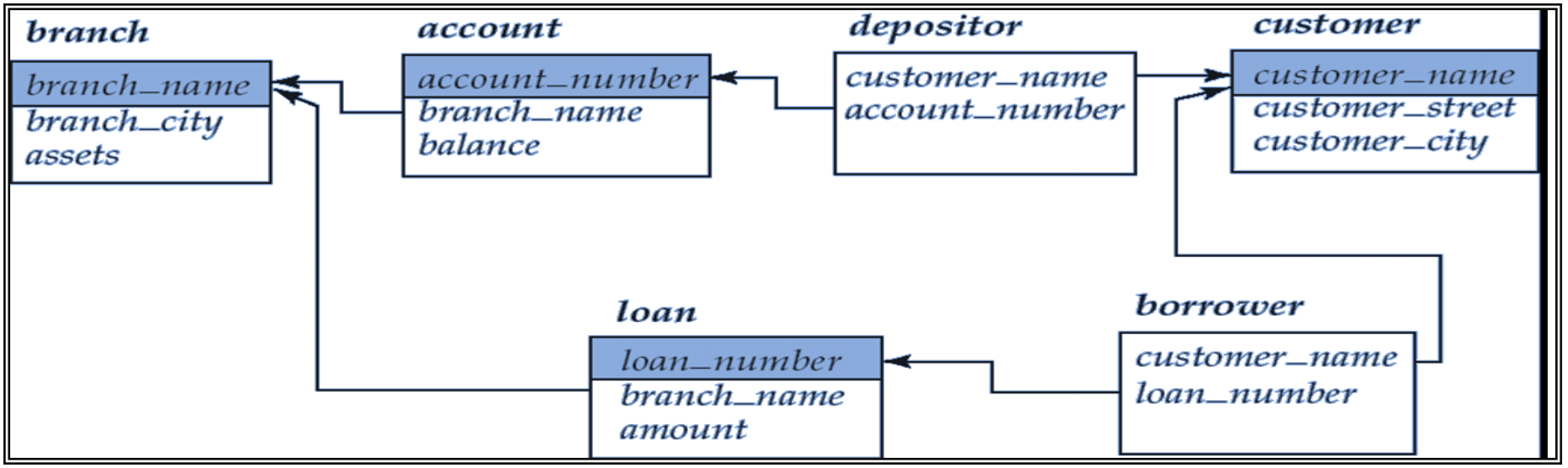


Natural Join

- **Find all customers who have an account from at least the "Downtown" and the Uptown" branches.**
  - $\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Downtown"}}(depositor \bowtie account)) \cap$

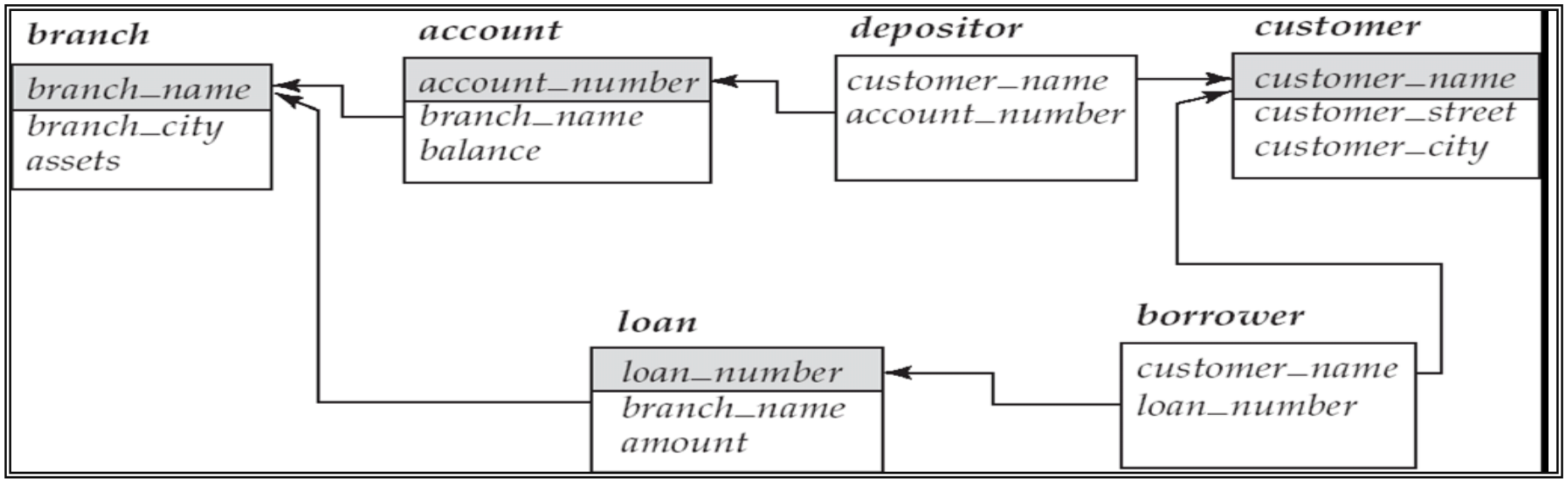
    $\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Uptown"}}(depositor \bowtie account))$

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



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

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



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 x E_2
  - $\sigma_p (E_1)$, P is a predicate on attributes in E_1
  - $\prod_s (E_1)$, S is a list consisting of some of the attributes in E_1
  - $\rho_x (E_1)$, x is the new name for the result of E_1

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:

<attribute> *op* <attribute> or <constant>

where ***op*** is one of: =, ≠, >, ≥. <. ≤

- **Example of selection:**

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

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

- **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

- **Notation:** $r - s$
- **Defined as:**

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

- **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

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

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

- **Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \varnothing$).**
- **If attributes of *r* and *s* are not disjoint, then renaming must be used.**

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.

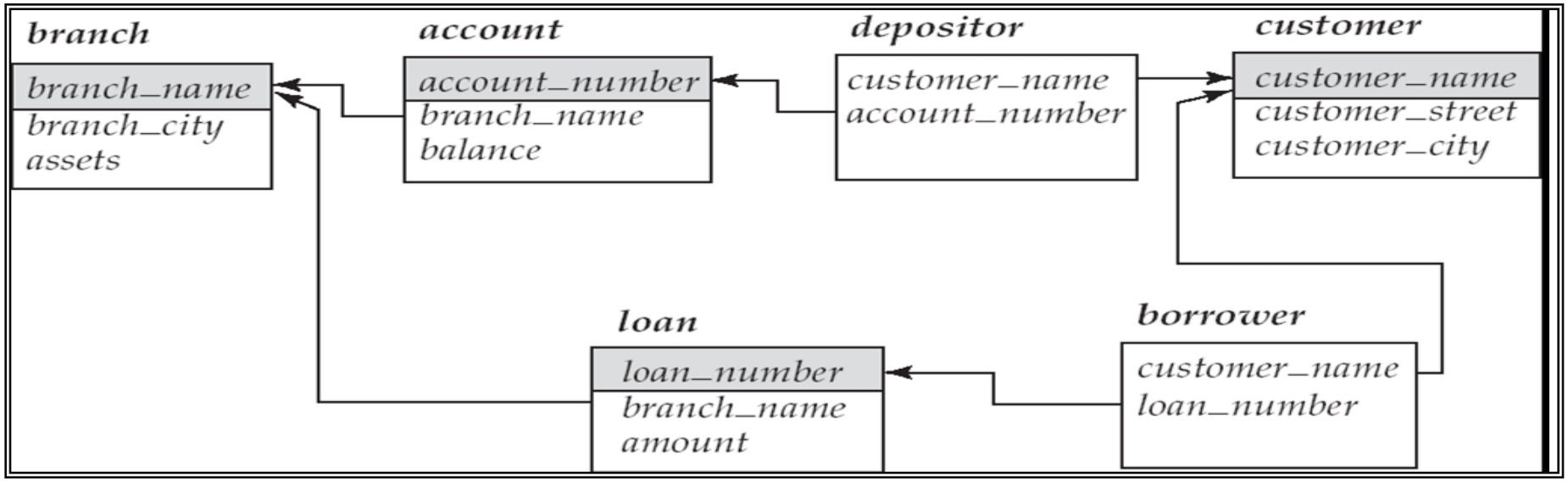
Deletion …contd.

- 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**

- **Delete all accounts at branches located in Needham.**



Deletion …contd.

- 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.**

$$r_1 \leftarrow \sigma_{branch\_city = \text{“Needham”}} (account \bowtie branch)$$
$$r_2 \leftarrow \prod_{account\_number,\ branch\_name,\ balance} (r_1)$$
$$r_3 \leftarrow \prod_{customer\_name,\ account\_number} (r_2 \bowtie depositor)$$
$$account \leftarrow account - r_2$$
$$depositor \leftarrow depositor - r_3$$

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.**

- **Example:**

- Insert information in the database specifying that Smith has Rs1200 in account A-973 at the Perryridge branch.

- Provide as a gift for all loan customers in the Perryridge branch, a Rs200 savings account. Let the loan number serve as the account number for the new savings account.

- **Insert information in the database specifying that Smith has Rs1200 in account A-973 at the Perryridge branch.**

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

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

- **Provide as a gift for all loan customers in the Perryridge branch, a Rs200 savings account. Let the loan number serve as the account number for the new savings account.**

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

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

$$depositor \leftarrow depositor \cup \prod_{customer\_name,\ loan\_number}(r_1)$$

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**

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

- **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…contd.

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

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

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

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

Structured Query Language (SQL)

History

- **IBM SEQUEL language developed as part of System R project at the IBM San Jose Research Laboratory**
- **Renamed Structured Query Language (SQL)**

- **ANSI and ISO standard SQL:**
  - **SQL-86**
  - **SQL-89**
  - **SQL-92**
  - **SQL:1999 (language name became Y2K compliant)**
  - **8.0 for SQL Server 2000.**
  - **SQL:2003**
  - **9.0 for SQL Server 2005.**
  - **10.0 for SQL Server 2008.**
  - **10.5 for SQL Server 2008 R2.**
  - **11.0 for SQL Server 2012.**
  - **12.0 for SQL Server 2014**
  - **13.0 for SQL Server 2016**
- **Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.**

Data Definition Language

- **Allows the specification of:**
  - **The schema for each relation including attribute types**
  - **Integrity constraints**
  - **Authorization information for each relation.**
- **Non-standard SQL extensions also allow specification of:**
  - **The set of indices to be maintained for each relations.**
  - **The physical storage structure of each relation on disk.**

Create Table Construct

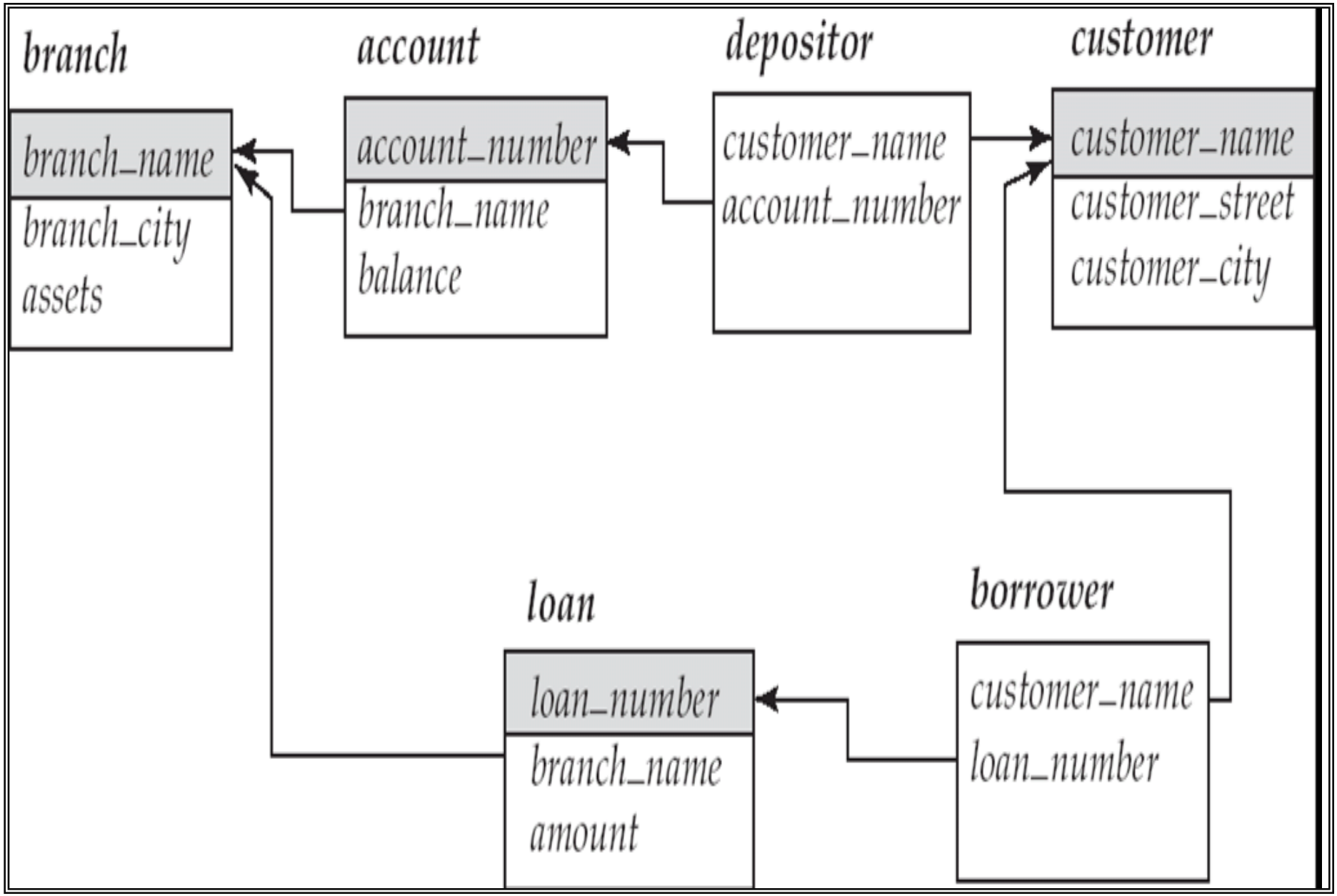
- **An SQL relation is defined using the create table command:**

  create table *r* (*A_1 D_1*, *A_2 D_2*, ..., *A_n D_n*,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - ***r* is the name of the relation**
  - **each *A_i* is an attribute name in the schema of relation *r***
  - ***D_i* is the data type of attribute *A_i***

- **Example:**

  create table *branch*
  (*branch_name* char(15),
  ***branch_city* char(30),**
  ***assets* integer)**


branch
branch_name
branch_city
assets
account
account_number
branch_name
balance
depositor
customer_name
account_number
customer
customer_name
customer_street
customer_city
loan
loan_number
branch_name
amount
borrower
customer_name
loan_number

Referential Integrity in SQL

create table *account*
(*account_number* **char**(10)**,**
branch_name **char**(15),
balance **integer**,
primary key (*account_number),*
foreign key (*branch_name*) **references** *branch*)

create table *depositor*
(*customer_name* **char**(20)**,**
account_number **char**(10)**,**
primary key (*customer_name*, *account_number),*
foreign key (*account_number*) **references** *account,*
foreign key (*customer_name*) **references** *customer*)

create table customer
(customer_name char(20),
customer_street char(30),
customer_city char(30),
primary key (customer_name))

create table branch
(branch_name char(15),
branch_city char(30),
assets numeric(12,2),
primary key (branch_name))

Domain Types in SQL

- **char(n):** Fixed length character string, with user-specified length *n*.
- **varchar(n):** Variable length character strings, with user-specified maximum length *n*.
- **Int:** Integer (a finite subset of the integers that is machine-dependent).
- **Smallint:** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d):** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision:** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n):** Floating point number, with user-specified precision of at least *n* digits.

Integrity Constraints on Tables

- **not null**
- **primary key** (A_1, ..., A_n)
- Example: Declare *branch name* as the primary key for *branch*

.

create table *branch*
(*branch_name* char(15)**,**
branch_city char(30) **not null**,
assets integer,
primary key (*branch_name*))

primary key declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89

Basic Insertion and Deletion of Tuples

- Newly created table is empty

- Add a new tuple to *account*

  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  – Insertion fails if any integrity constraint is violated

- Delete *all* tuples from *account*

  delete from *account*

Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped relation from the database.

  drop table

- The **alter table** command is used to add attributes to an existing relation:

  alter table *r* **add** *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.

  - All tuples in the relation are assigned *null* as the value for the new attribute.

- The **alter table** command can also be used to drop attributes of a relation:

  alter table *r* **drop** *A*

  where *A* is the name of an attribute of relation *r*

  - Dropping of attributes not supported by many databases

Basic Query Structure

- **General form of SQL query :**

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.

- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.

The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra

- Example: Find the names of all branches in the *loan* relation:

  select *branch_name*
  from *loan*

- In the relational algebra, the query would be:

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

- To force the elimination of duplicates, insert the keyword **distinct** after select.

- Find the names of all branches in the *loan* relations, and remove duplicates

  select distinct *branch_name*
  from *loan*

- The keyword all specifies that duplicates not be removed.

  select all *branch_name*
  from *loan*

The select Clause ….Contd.

- **An asterisk in the select clause denotes "all attributes"**

```
select *
from loan
```

- **The select clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.**

- **E.g.:**

```
select loan_number, branch_name, amount *100
from loan
```

The where Clause

- **The where clause specifies conditions that the result must satisfy**
  - **Corresponds to the selection predicate of the relational algebra.**

- **To find all loan number for loans made at the Perryridge branch with loan amounts greater than Rs1200.**

```
select loan_number
from loan
where branch_name = 'Perryridge'  and amount > 1200
```

- **Comparison results can be combined using the logical connectives and, or, and not.**

The from Clause

- **The from clause lists the relations involved in the query**
  - **Corresponds to the Cartesian product operation of the relational algebra.**

- **Find the Cartesian product of *borrower X loan***

  select *
  from *borrower, loan*

Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

select customer_name, borrower.loan_number, amount
from borrower, loan
where borrower.loan_number = loan.loan_number **and** branch_name = 'Perryridge'

Rename Operator

SQL allows renaming relations and attributes using the as clause:

old-name* as *new-name

Example: Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id*.

select *customer_name, borrower.loan_number* as *loan_id, amount*

from *borrower, loan*

where *borrower.loan_number = loan.loan_number*

Tuple Variables

- **Tuple variables are defined in the from clause via the use of the as clause**

- **Find the customer names and their loan numbers and amount for all customers having a loan at some branch.**

```
select customer_name, T.loan_number, S.amount
        from borrower as T, loan as S
        where  T.loan_number = S.loan_number
```

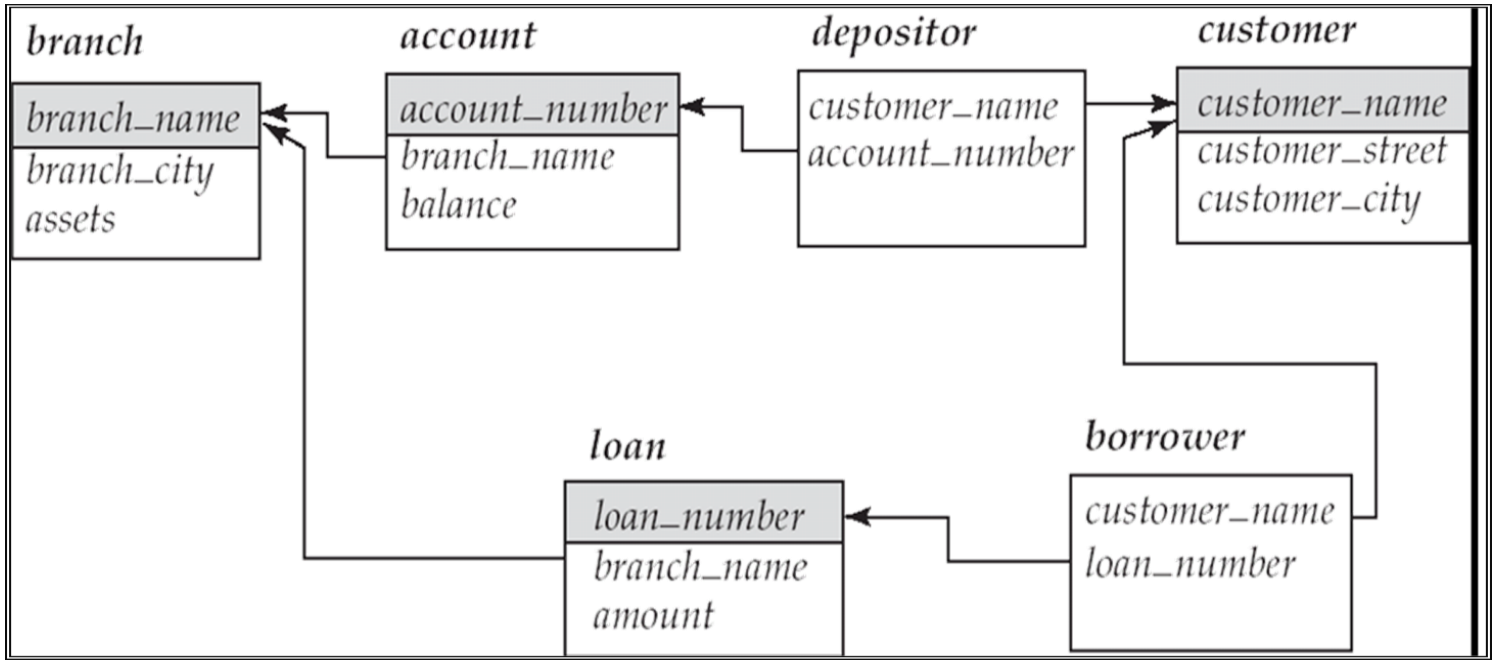
Find the names of all branches that have greater assets than some branch located in Brooklyn.

```
select distinct T.branch_name
from branch as T, branch as S
where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

Keyword **as** is optional and may be omitted

borrower **as** *T* ≡ *borrower T*

Some database such as Oracle *require* **as** to be omitted


branch
branch_name
branch_city
assets
account
account_number
branch_name
balance
depositor
customer_name
account_number
customer
customer_name
customer_street
customer_city
loan
loan_number
branch_name
amount
borrower
customer_name
loan_number

String Operations

- **SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that are described using two special characters:**
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.

- **Find the names of all customers whose street includes the substring "Main".**

```
select customer_name
from customer
where customer_street like '% Main%'
```

Match the name "Main%"

```
like 'Main\%' escape '\'
```

SQL supports a variety of string operations such as

concatenation (using "||")

converting from upper to lower case (and vice versa)

finding string length, extracting substrings, etc.

String Operations

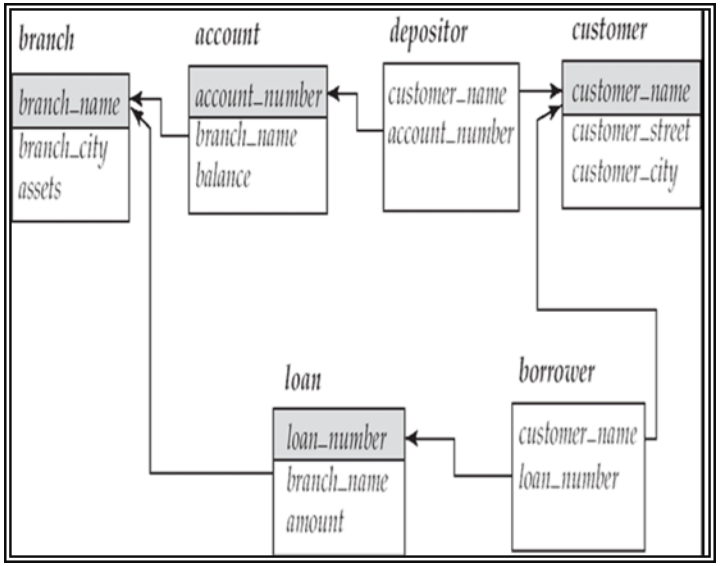
- **List in alphabetic order the names of all customers having a loan in Perryridge branch**

```
select distinct customer_name
from    borrower, loan
where borrower loan_number = loan.loan_number and
          branch_name = 'Perryridge'
order by customer_name
```

We may specify desc for descending order or asc for ascending order, for each attribute; ascending order is the default.

Example:

```
select distinct customer_name
    from    borrower, loan
where borrower loan_number = loan.loan_number
        and branch_name = 'Perryridge'
    order by customer_name decs
```



Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.

- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

Duplicates …Contd.

- SQL duplicate semantics:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations

- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.

  Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then, it occurs:

- $m + n$ times in *r* **union all** *s*
- $\min(m,n)$ times in *r* **intersect all** *s*
- $\max(0, m - n)$ times in *r* **except all** *s*

Set Operations

- **Find all customers who have a loan, an account, or both:**

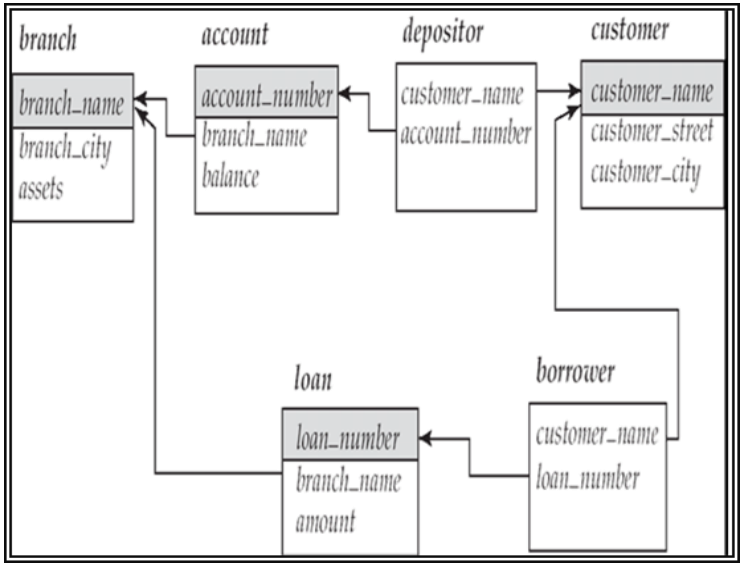
  (select customer_name **from** depositor)
  union
  (select customer_name **from** borrower)

- **Find all customers who have both a loan and an account.**

  (**select** customer_name **from** depositor)
  intersect
  (select customer_name **from** borrower)

- **Find all customers who have an account but no loan.**

  (**select** customer_name **from** depositor)
  except
  (select customer_name **from** borrower)



Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value.

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- **Find the average account balance at the Perryridge branch.**

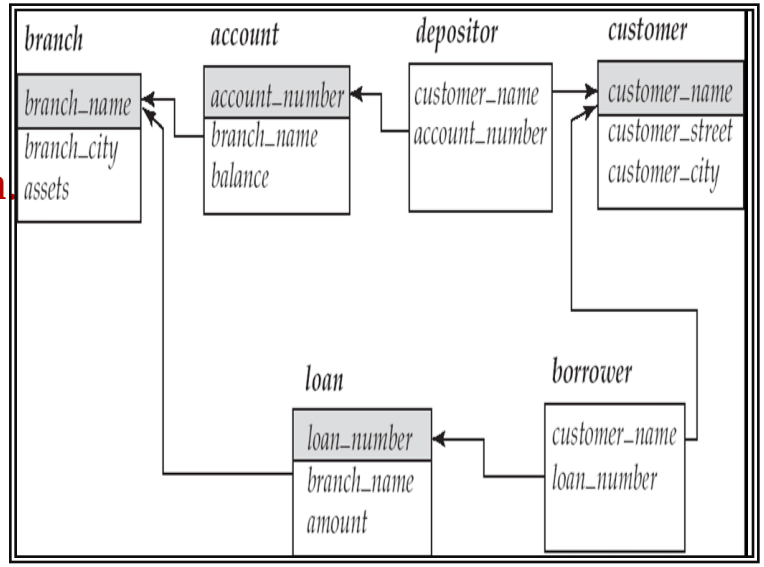
```
select avg (balance)
        from account
        where branch_name = 'Perryridge'
```

- **Find the number of tuples in the *customer* relation.**

```
select count (*)
        from customer
```

- **Find the number of depositors in the bank.**

```
select count (distinct customer_name)
        from depositor
```



Aggregate Functions – Group By

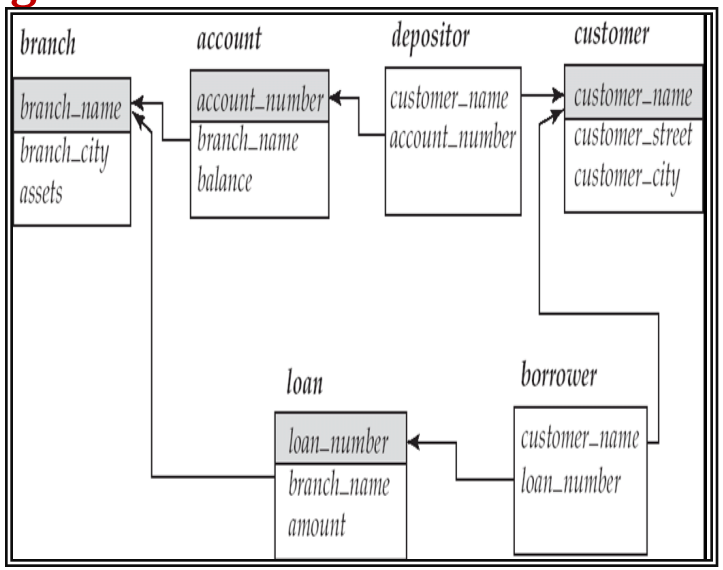
- **Find the number of depositors for each branch.**

```
select branch_name, count (distinct customer_name)
        from depositor, account
        where depositor.account_number = account.account_number
        group by branch_name
```

Aggregate Functions – Having Clause

- **Find the names of all branches where the average account balance is more than Rs 1,200.**

```
select branch_name, avg (balance)
        from account
        group by branch_name
        having avg (balance) > 1200
```



Quiz **Date: 20-9-19**

1. **What is database system?**
2. **Define the DBMS with necessary parameters.**
3. **Name the three schemas of database system.**
4. **List out any three major advantages of database systems over traditional file system.**
5. **Name the different types of DML**
6. **Define the following:**

   Primary key

   Foreign key

   Candidate Key

7. **Explain the following operation in relational algebra with necessary example**

   Natural join

   Cartesian product

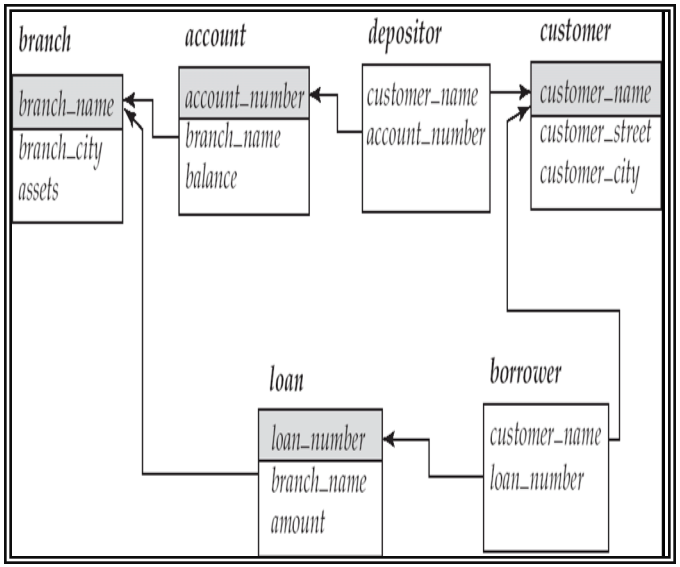
   Division

Nested Queries

- **SQL provides a mechanism for the nesting of subqueries.**
- **A subquery is a select-from-where expression that is nested within another query.**
- **A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.**

  - **Find all customers who have both an account and a loan at the bank.**

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                from depositor )
```



In Construct- Nested Queries

- **Find all customers who have a loan at the bank but do not have an account at the bank**

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer_name
                                    from depositor )
```

- **Find all customers who have both an account and a loan at the Perryridge branch**

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
          branch_name = 'Perryridge'  and
        (branch_name, customer_name ) in
                (select branch_name, customer_name
                 from depositor, account
                 where depositor.account_number =
                   account.account_number )
```

“Some” Construct

- **Find all branches that have greater assets than some branch located in Brooklyn.**

```
select distinct  T.branch_name
        from branch as T, branch as S
        where  T.assets > S.assets and
                S.branch_city = 'Brooklyn'
```

- Same query using > **some** clause

```
select branch_name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch_city = 'Brooklyn')
```

“All” Construct

- **Find the names of all branches that have greater assets than all branches located in Brooklyn.**

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

"Exists" Construct

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

```
select distinct S.customer_name
        from depositor as S
        where not exists (
                (select branch_name
                from branch
                where branch_city = 'Brooklyn')
                except
                (select R.branch_name
                from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name ))
```

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

```
select T.customer_name
 from depositor as T
 where unique (
        select R.customer_name
        from account, depositor as R
        where T.customer_name = R.customer_name
                and
                R.account_number = account.account_number and
                account.branch_name = 'Perryridge')
```

- **Find all customers who have at least two accounts at the Perryridge branch.**

```
select distinct T.customer_name
from depositor  as T
where not unique (
    select R.customer_name
    from account, depositor as R
    where T.customer_name = R.customer_name and
          R.account_number = account.account_number  and
          account.branch_name = 'Perryridge')
```

- Variable from outer level is known as a **correlation variable**

Modification of the Database – Deletion

- **Delete all account tuples at the Perryridge branch**

```
delete from account
where branch_name = 'Perryridge'
```

- **Delete all accounts at every branch located in the city ‘Needham’.**

```
delete from account
where branch_name in (select branch_name
                      from branch
                      where branch_city = 'Needham')
```

- **Delete the record of all accounts with balances below the average at the bank.**

delete from *account*
where *balance* < (**select avg** (*balance*)
from *account*)

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:
  1. First, compute **avg** balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- **Add a new tuple to *account***

  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)
  values ('Perryridge', 1200, 'A-9732')

- **Add a new tuple to *account* with *balance* set to null**

  insert into *account*
  values ('A-777','Perryridge', *null*)

Nested Queries

Rename Operator

SQL allows renaming relations and attributes using the as clause:

old-name* as *new-name

Example: Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id*.

select *customer_name, borrower.loan_number* as *loan_id, amount*

from *borrower, loan*

where *borrower.loan_number = loan.loan_number*

Tuple Variables

- **Tuple variables are defined in the from clause via the use of the as clause**

- **Find the customer names and their loan numbers and amount for all customers having a loan at some branch.**

```
select customer_name, T.loan_number, S.amount
        from borrower as T, loan as S
        where  T.loan_number = S.loan_number
```

- **Find the names of all branches that have greater assets than some branch located in Brooklyn.**

```
select distinct T.branch_name
from branch as T, branch as S
where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

Keyword **as** is optional and may be omitted

borrower **as** *T* ≡ *borrower T*

Some database such as Oracle *require* **as** to be omitted

String Operations

- **SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that are described using two special characters:**
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.

- **Find the names of all customers whose street includes the substring "Main".**

```
select customer_name
from customer
where customer_street like '% Main%'
```

Match the name "Main%"

```
like 'Main\%' escape '\'
```

SQL supports a variety of string operations such as

concatenation (using "||")

converting from upper to lower case (and vice versa)

finding string length, extracting substrings, etc.

String Operations

- **List in alphabetic order the names of all customers having a loan in Perryridge branch**

```
select distinct customer_name
from    borrower, loan
where borrower loan_number = loan.loan_number and
        branch_name = 'Perryridge'
order by customer_name
```

We may specify desc for descending order or asc for ascending order, for each attribute; ascending order is the default.

Example:

```
select distinct customer_name
from    borrower, loan
where borrower loan_number = loan.loan_number and
        branch_name = 'Perryridge'
order by customer_name decs
```

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.

- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

1. $\sigma_{\theta}(r_1)$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_{θ}, then there are c_1 copies of t_1 in $\sigma_{\theta}(r_1)$.

2. $\Pi_A(r)$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.

3. $r_1 \times r_2$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1 . t_2$ in $r_1 \times r_2$

Duplicates …Contd.

- Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

  $r_1 = \{(1, a)\ (2, a)\}$ $r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be

  $\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$

- SQL duplicate semantics:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  is equivalent to the *multiset* version of the expression:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations □□□□□□□□

- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all**.

Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then, it occurs:

- $m + n$ times in *r* **union all** *s*
- $\min(m,n)$ times in *r* **intersect all** *s*
- $\max(0, m - n)$ times in *r* **except all** *s*

Set Operations

- **Find all customers who have a loan, an account, or both:**

```
(select customer_name from depositor)
union
(select customer_name from borrower)
```

- **Find all customers who have both a loan and an account.**

```
(select customer_name from depositor)
intersect
(select customer_name from borrower)
```

- **Find all customers who have an account but no loan**.

```
(select customer_name from depositor)
except
(select customer_name from borrower)
```

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value.

  a**vg:** average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- **Find the average account balance at the Perryridge branch.**

```
select avg (balance)
        from account
        where branch_name = 'Perryridge'
```

- **Find the number of tuples in the *customer* relation.**

```
select count (*)
        from customer
```

- **Find the number of depositors in the bank.**

```
select count (distinct customer_name)
        from depositor
```

Aggregate Functions – Group By

- **Find the number of depositors for each branch.**

```
select branch_name, count (distinct customer_name)
        from depositor, account
        where depositor.account_number = account.account_number
        group by branch_name
```

Aggregate Functions – Having Clause

- **Find the names of all branches where the average account balance is more than Rs 1,200.**

```
select branch_name, avg (balance)
        from account
        group by branch_name
        having avg (balance) > 1200
```

Bank database

branch (branch_name, branch_city, assets)

customer (customer_name, customer_street, customer_city)

loan (loan_number, branch_name, amount)

borrower (customer_name, loan_number)

account (account_number, branch_name, balance)

depositor (customer_name, account_number)

Aggregate Functions – Group By

Find the number of depositors for each branch.

```
select branch_name, count (distinct customer_name)
        from depositor, account
        where depositor.account_number = account.account_number
        group by branch_name
```

Find the names of all branches where the average account balance is more than $1,200.

```
select branch_name, avg (balance)
        from account
        group by branch_name
        having avg (balance) > 1200
```

"In" construct: **It test for set membership, where the set is a collection of values produced by select clause**

- **Find all customers who have both an account and a loan at the bank.**

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                from depositor )
```

- **Find all customers who have a loan at the bank but do not have an account at the bank**

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer_name
                                    from depositor )
```

In construct…contd.

- **Find all customers who have both an account and a loan at the Perryridge branch**

```
select distinct customer_name
from borrower, loan
where borrower.loan_number = loan.loan_number and
  branch_name = 'Perryridge'  and
  (branch_name, customer_name ) in
            (select branch_name, customer_name
              from depositor, account
              where depositor.account_number =
                    account.account_number )
```

"Some" Construct:

>some: It test for set membership "greater than at least one", where the set is a collection of values produced by where clause.

- **Find all branches that have greater assets than some branch located in Brooklyn.**

```
select distinct  T.branch_name
        from branch as T, branch as S
        where  T.assets > S.assets and
                S.branch_city = 'Brooklyn'
```

- Same query using **> some** clause

```
select branch_name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch_city = 'Brooklyn')
```

SQL allows <some, <=some, >=some,=some and <>some

"All" Construct

- **> all It test for set membership "greater than all", where the set is a collection of values produced by where clause.**

- **Find the names of all branches that have greater assets than all branches located in Brooklyn.**

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

SQL also supports <all, <= all, >=all, =all, and <>all

"Exists" Construct

- **The "Exists" construct returns value true if the argument subquery has any tuples.**
- **not exists (B except A)**

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

```
select distinct S.customer_name
        from depositor as S
        where not exists (
                (select branch_name
                from branch
                where branch_city = 'Brooklyn')
                except
                (select R.branch_name
                from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name ))
```

Absence of Duplicate Tuples

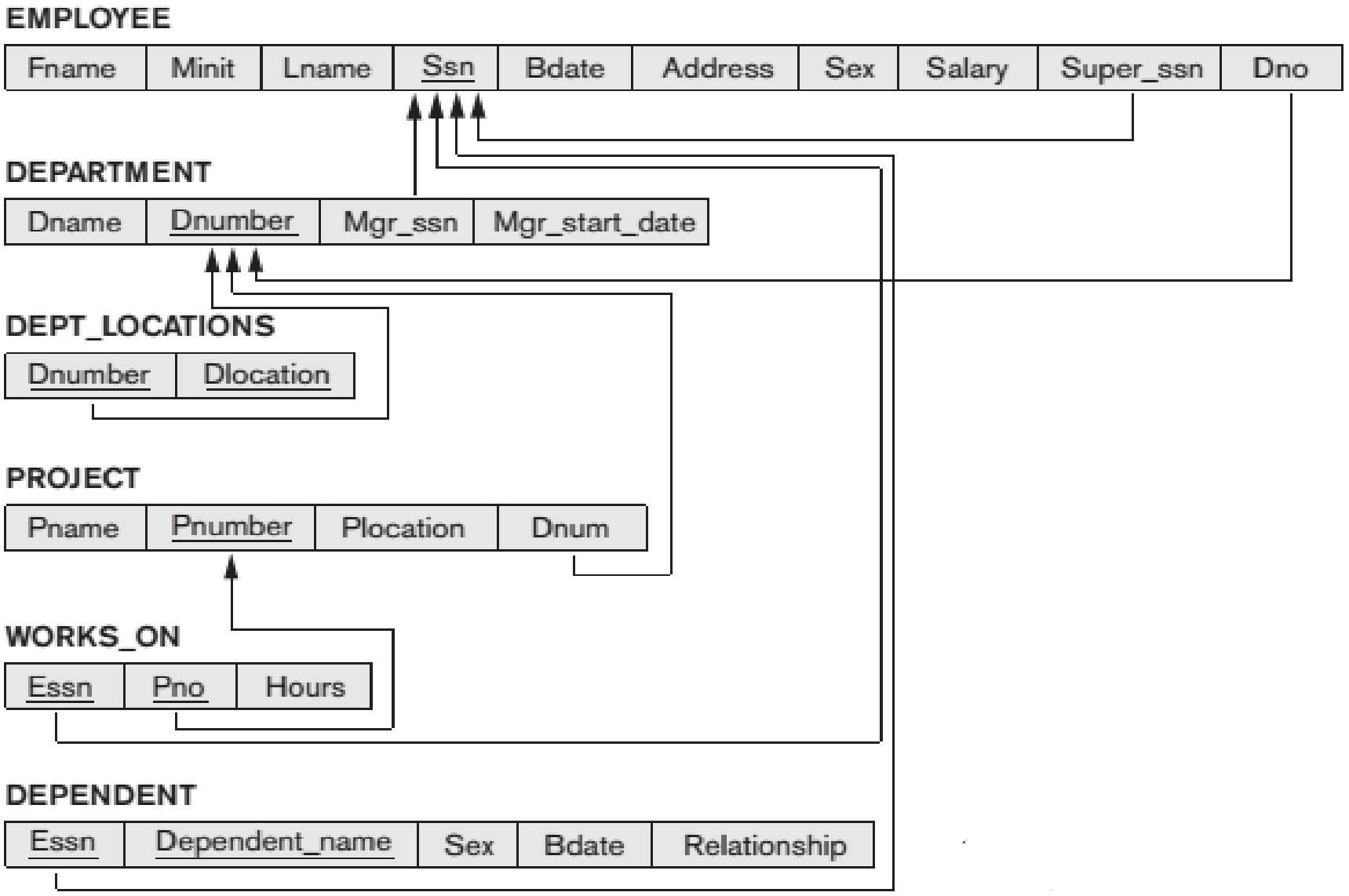
- The unique construct tests whether a subquery has "any duplicate tuples" in its result.
- Find all customers who have at most one account at the Perryridge branch.

```
select T.customer_name
from depositor as T
where unique (
        select R.customer_name
        from account, depositor as R
        where T.customer_name = R.customer_name and
                R.account_number = account.account_number and
                account.branch_name = 'Perryridge')
```

- **Find all customers who have at least two accounts at the Perryridge branch.**

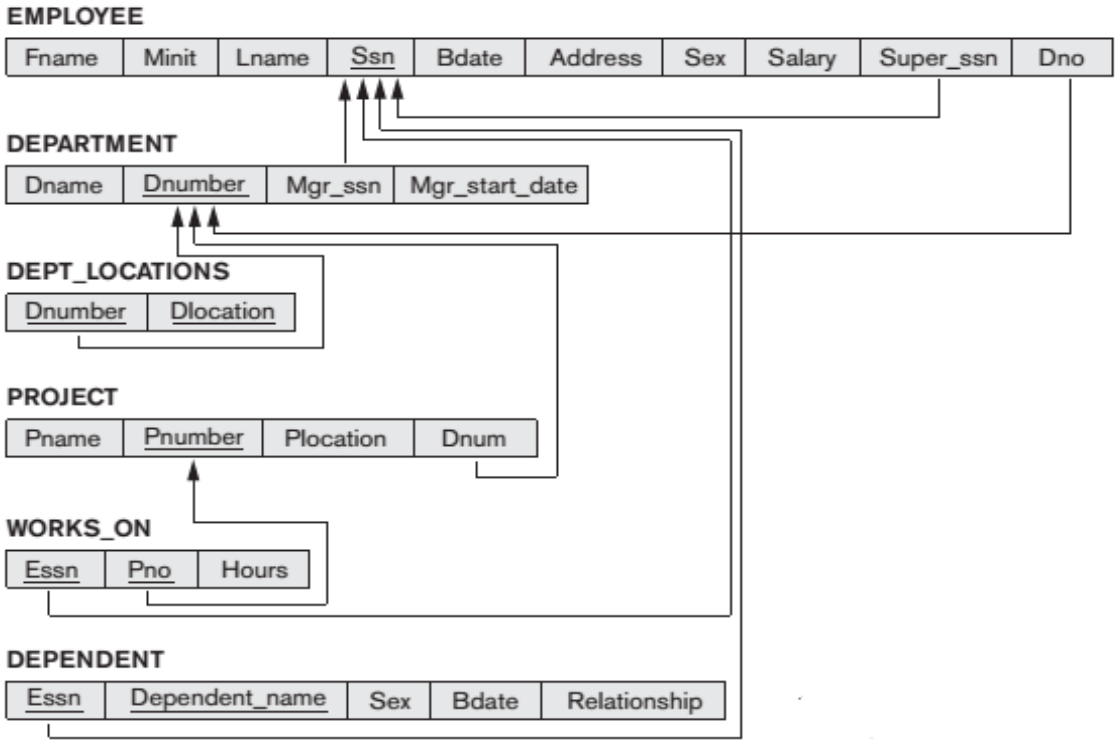
```
select distinct T.customer_name
from depositor  as T
where not unique (
    select R.customer_name
    from account, depositor as R
    where T.customer_name = R.customer_name and
            R.account_number = account.account_number  and
            account.branch_name = 'Perryridge')
```

```
select distinct T.customer_name
from depositor  as T
where not unique (
      select R.customer_name
      from account, depositor as R
      where T.customer_name = R.customer_name and
                  R.account_number = account.account_number  and
                  account.branch_name = 'Perryridge')
```


EMPLOYEE
Fname
Minit
Lname
Ssn
Bdate
Address
Sex
Salary
Super_ssn
Dno
DEPARTMENT
Dname
Dnumber
Mgr_ssn
Mgr_start_date
DEPT_LOCATIONS
Dnumber
Dlocation
PROJECT
Pname
Pnumber
Plocation
Dnum
WORKS_ON
Essn
Pno
Hours
DEPENDENT
Essn
Dependent_name
Sex
Bdate
Relationship

```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)             NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)             NOT NULL,
          Ssn                   CHAR(9)                 NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                     NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

- **Retrieve the birth date and address of the employee(s) whose name is ‘John B. Smith’.**
- **Retrieve the name and address of all employees who work for the ‘Research’ department.**
- **For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.**



- **Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.**

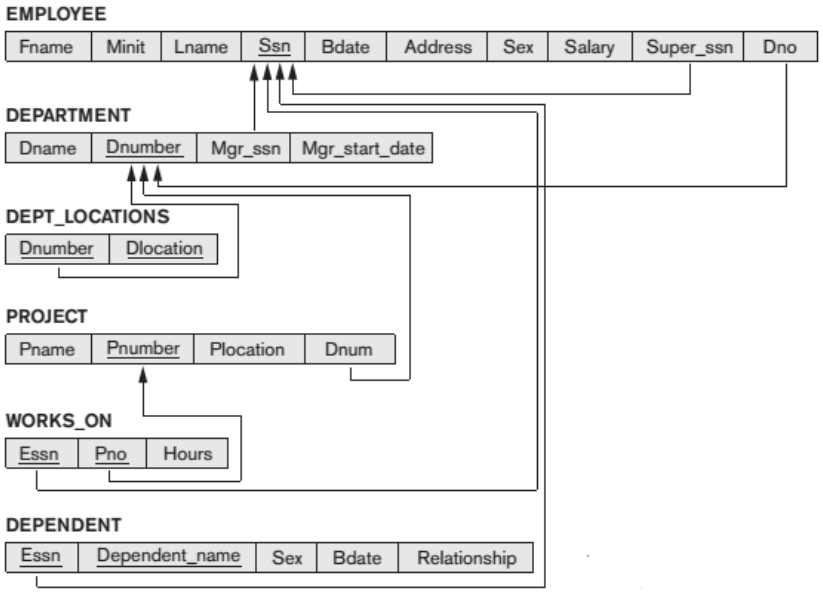
```
SELECT Bdate, Address
FROM EMPLOYEE
WHERE Fname='John'ANDMinit='B'ANDLname='Smith';
```

- **Retrieve the name and address of all employees who work for the 'Research' department.**

```
SELECT Fname, Lname, Address
FROM EMPLOYEE, DEPARTMENT
WHERE Dname='Research'ANDDnumber=Dno;
```

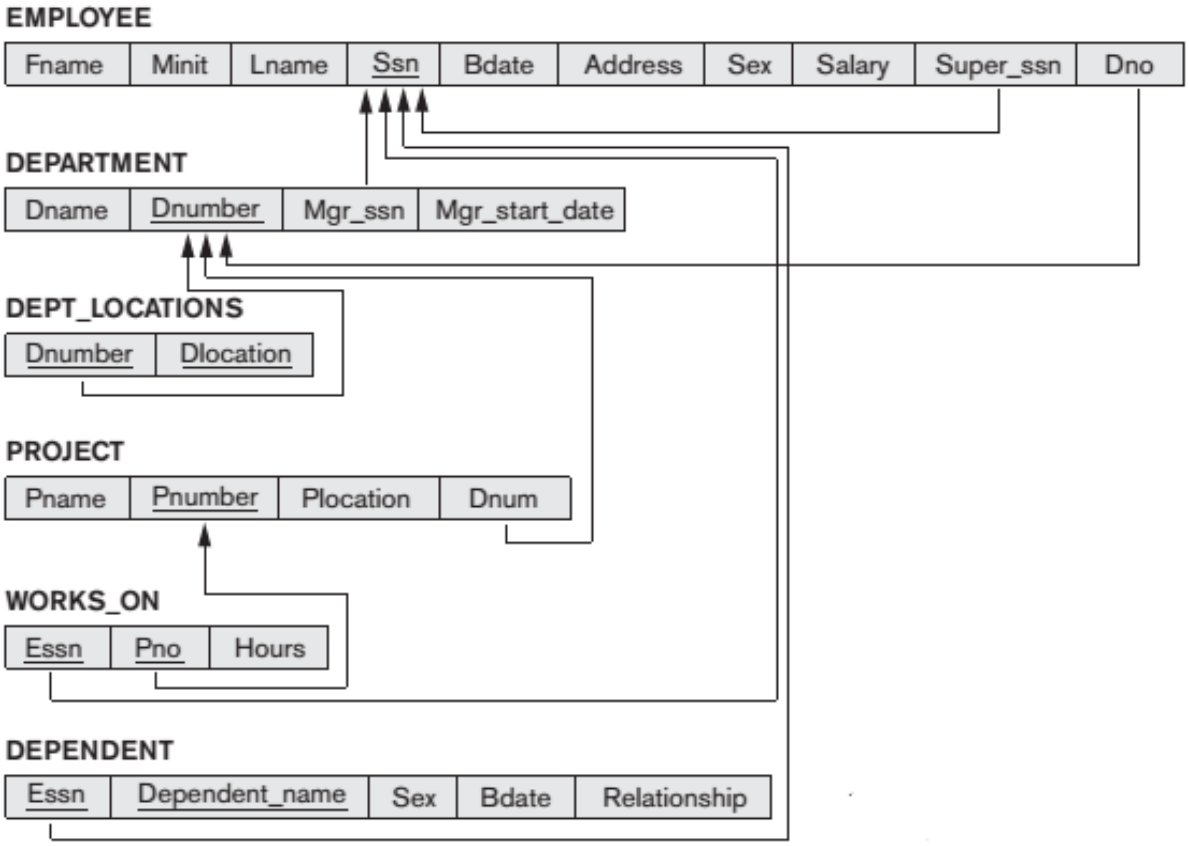
- **For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.**

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=DnumberANDMgr_ssn=SsnAND
Plocation='Stafford';
```



- **For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.**

- **Retrieve the salary of every employee and all distinct salary values**



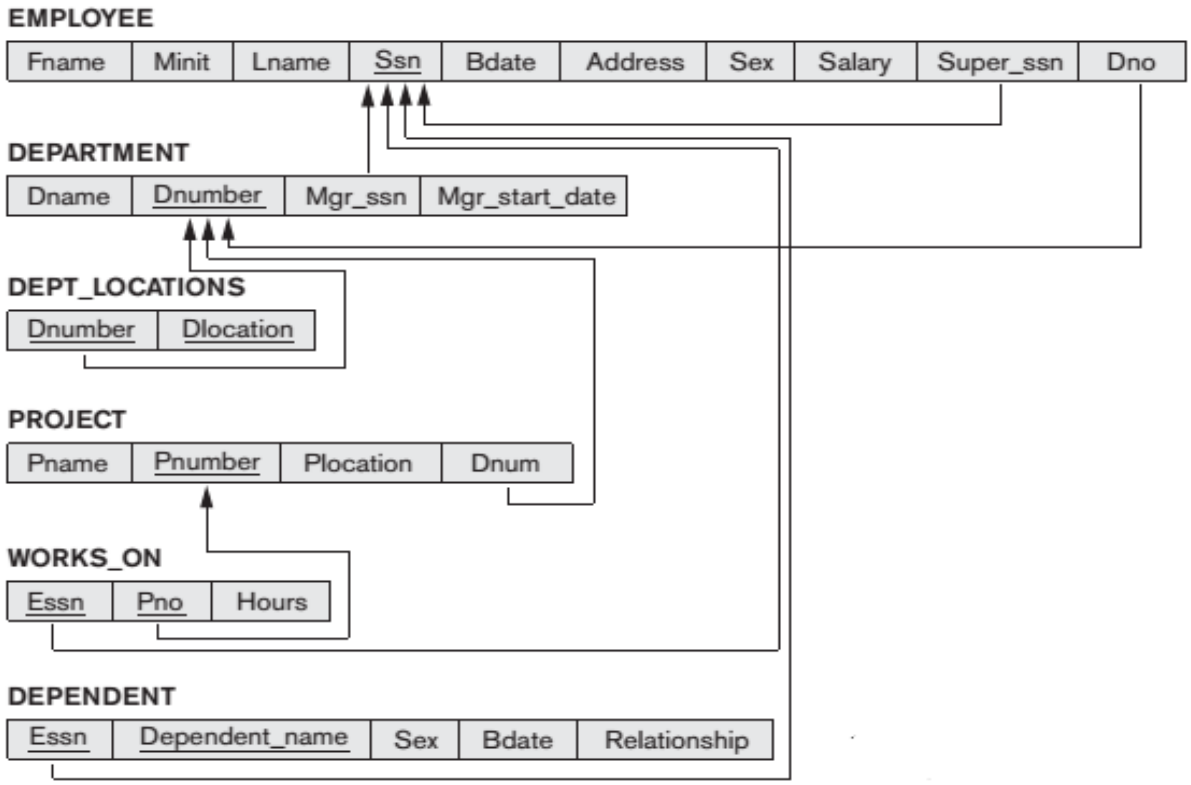
- **For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.**

```
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

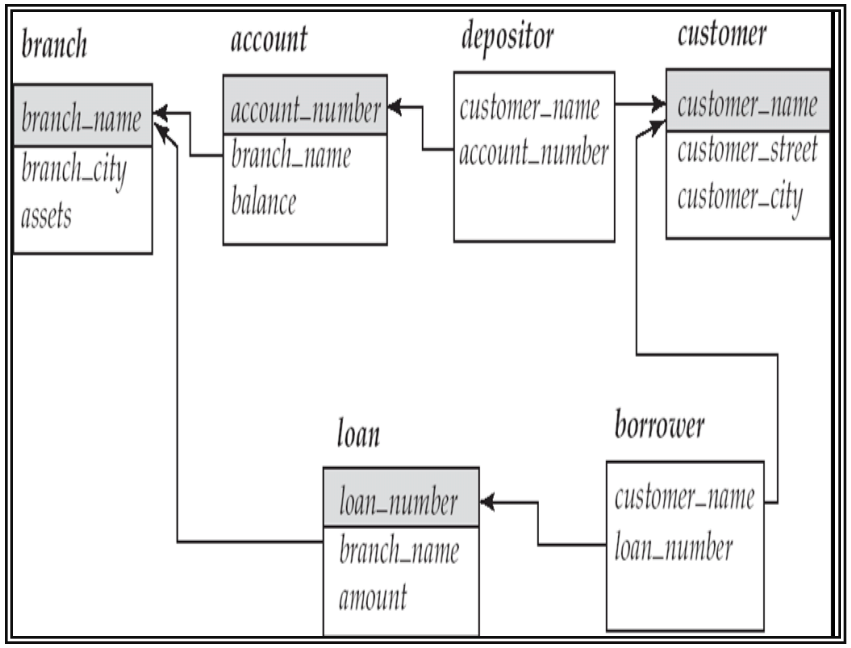
- **Retrieve the salary of every employee and all distinct salary values**

```
SELECT ALL Salary
FROM EMPLOYEE;

SELECT DISTINCT Salary
FROM EMPLOYEE;
```



Modification of the Database – Deletion



- **Delete all account tuples at the Perryridge branch**

```
delete from account
where branch_name = 'Perryridge'
```

- **Delete all accounts at every branch located in the city ‘Needham’.**

```
delete from account
where branch_name in (select branch_name
                      from branch
                      where branch_city = 'Needham')
```

- **Delete the record of all accounts with balances below the average at the bank.**

```
delete from account
        where balance < (select avg (balance )
                         from account )
```

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:
  1. First, compute **avg** balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- **Add a new tuple to *account***

```
insert into account
        values ('A-9732', 'Perryridge', 1200)
```

or equivalently

```
insert into account (branch_name, balance, account_number)
        values ('Perryridge',  1200, 'A-9732')
```

- **Add a new tuple to *account* with *balance* set to null**

```
insert into account
        values ('A-777','Perryridge',  null )
```

Modification of the Database – Insertion

- **Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account**

```
insert into account
    select loan_number, branch_name,  200
    from loan
    where branch_name = 'Perryridge'
insert into depositor
    select customer_name, loan_number
    from loan, borrower
    where branch_name = 'Perryridge'
            and loan.account_number = borrower.account_number
```

- **The select from where statement is evaluated fully before any of its results are inserted into the relation**
  - **Motivation: insert into table1 select * from table1**

Modification of the Database – Updates

- **Increase all accounts with balances over Rs10,000 by 6%, all other accounts receive 5%.**
  - **Write two update statements:**

```
update account
set balance = balance * 1.06
where balance > 10000

update account
set balance = balance * 1.05
where balance > 10000
```

  - **The order is important**
  - **Can be done better using the case statement**

Joined Relations – Datasets

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

loan

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

borrower

Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations

- loan **inner join** borrower **on**
  loan.loan_number = borrower.loan_number

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

- loan **left outer join** borrower **on**
  loan.loan_number = borrower.loan_number

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

Joined Relations

- *loan* **natural inner join** *borrower*

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

loan **natural right outer join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Find all customers who have either an account or a loan (but not both) at the bank.

```
select customer_name
        from (depositor natural full outer join borrower )
        where account_number is null or loan_number is null
```

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- **Find the average account balance of those branches where the average account balance is greater than Rs 1200.**

```
select branch_name, avg_balance
from (select branch_name, avg (balance)
        from account
        group by branch_name )
      as branch_avg ( branch_name, avg_balance )
where avg_balance > 1200
```

Note : We do not need to use the **having** clause, since we compute the temporary (view) relation branch_avg in the **from** clause, and the attributes of branch_avg can be used directly in the **where** clause.

View Definition

- A relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

- 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 SQL 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.

- **A view consisting of branches and their customers**

```
create view all_customer as
        (select branch_name, customer_name
         from depositor, account
        where depositor.account_number =
                account.account_number )
        union
        (select branch_name, customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number )
```

- **Find all customers of the Perryridge branch**

```
select customer_name
        from all_customer
        where branch_name = 'Perryridge'
```

Uses of Views

- Hiding some information from some users
  - Consider a user who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount.
  - Define a view

    (**create view** cust_loan_data **as**
    select customer_name, borrower.loan_number, branch_name
    from borrower, loan
    where borrower.loan_number = loan.loan_number)
  - Grant the user permission to **read cust_loan_data, but not borrower or loan**
  - Predefined queries to make writing of other queries easier
  - Common example: Aggregate queries used for statistical analysis of data

Processing of Views

- **When a view is created**
  - **Query expression is stored in the database along with the view name**
  - **Expression is substituted into any query using the view**
- **Views definitions containing 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

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

- **Find all accounts with the maximum balance**

with *max_balance* (*value*) **as**
select max (*balance*)
from *account*

select *account_number*
from *account*, *max_balance*
where *account.balance = max_balance.value*

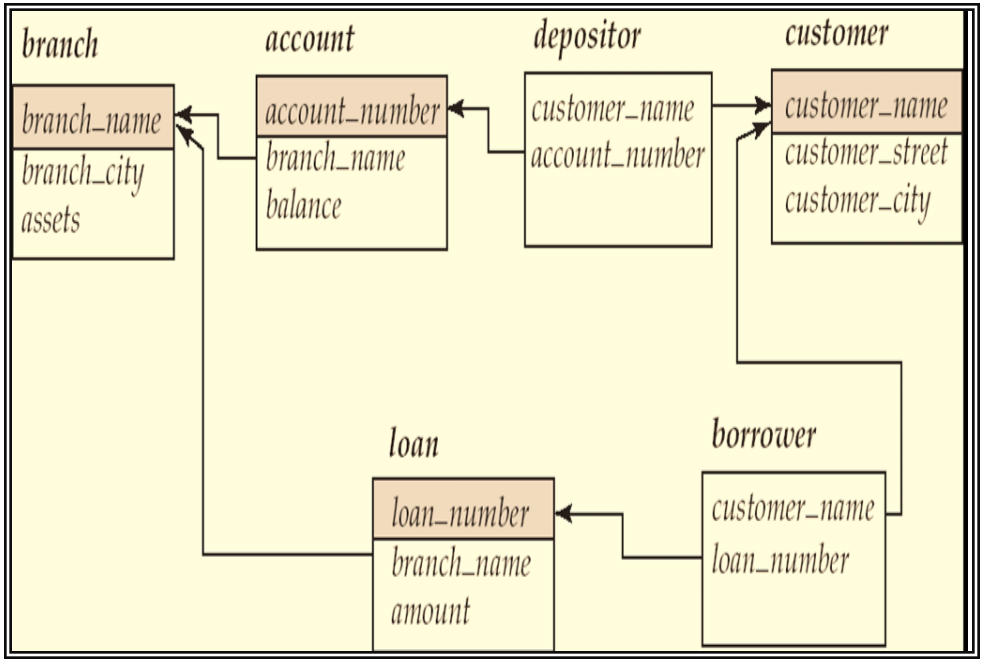
Complex Queries using With Clause

- **Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.**

```
with branch_total (branch_name, value) as
        select branch_name, sum (balance)
        from account
        group by branch_name

with branch_total_avg (value) as
        select avg (value)
        from branch_total

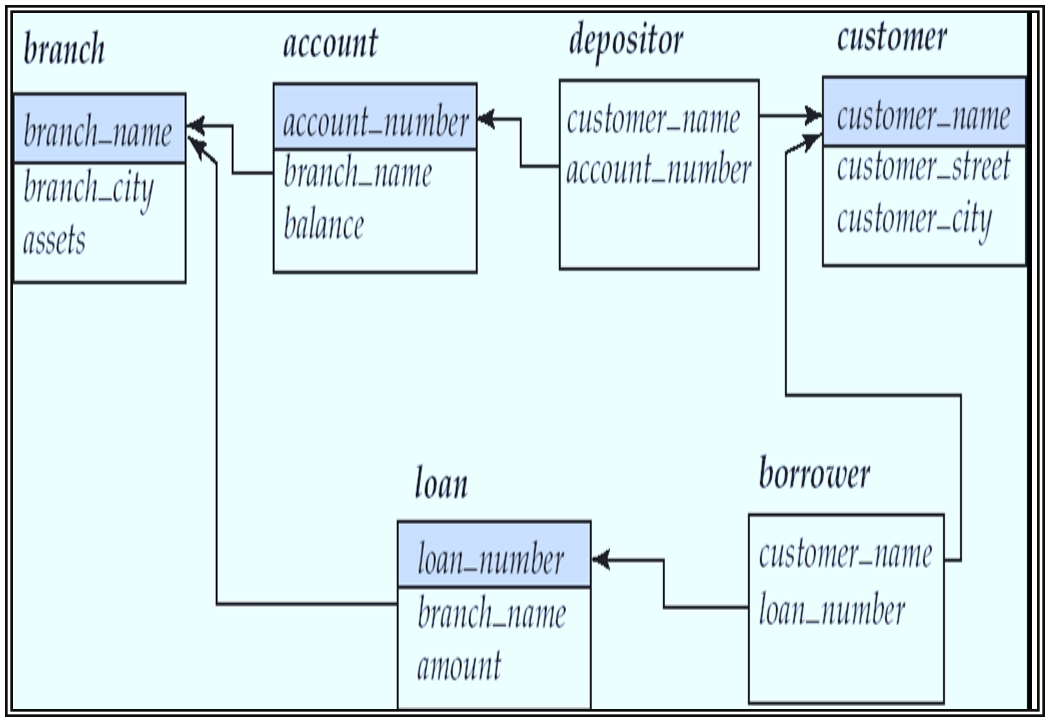
select branch_name
  from branch_total, branch_total_avg
  where branch_total.value >= branch_total_avg.value
```



Update of a View

- **Create a view of all loan data in the loan relation, hiding the amount attribute**

create view loan_branch **as**
select loan_number, branch_name
from loan



Add a new tuple to loan_branch

insert into loan_branch
values ('L-37‘, 'Perryridge‘)

This insertion must be represented by the insertion of the tuple

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

into the loan relation

Updates Through Views ..contd.

- **Some updates through views are impossible to translate into updates on the database relations**

  - **create view** v **as**
    select loan_number, branch_name, amount
    from loan
    where branch_name = ‘Perryridge’

    insert into v **values** **(** 'L-99','Downtown', '23'**)**

- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

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 predicate **is null** can be used to check for null values.

Find all loan number which appear in the loan relation with null values for amount.

```
select loan_number
from loan
where amount is null
```

The result of any arithmetic expression involving null is null
Example: 5 + null returns null

However, aggregate functions simply ignore nulls

Null Values and Three Valued Logic

- **Any comparison with null returns unknown**
  - Example: 5 < null or null <> null or null = null

- **Three-valued logic using the truth value unknown**:
  - OR: (unknown **or** true) = true,
    (unknown **or** false) = unknown
    (unknown **or** unknown) = unknown

  - AND: (true **and** unknown) = unknown,
    (false **and** unknown) = false,
    (unknown **and** unknown) = unknown

  - NOT: (**not** unknown) = unknown

  - "P **is unknown**" evaluates to true if predicate P evaluates to unknown

- **Result of where clause predicate is treated as false if it evaluates to unknown**

Null Values and Aggregates

- Total all loan amounts

  select sum (amount)
  from loan

  - Above statement ignores null amounts
  - Result is null if there is no non-null amount

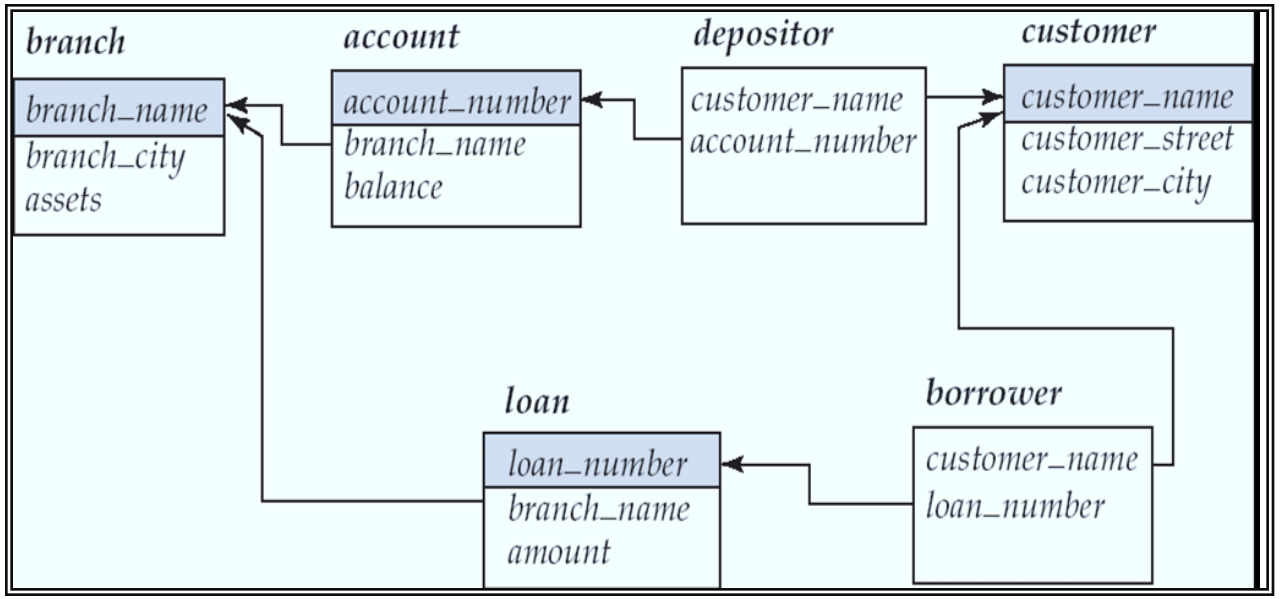
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

The where Clause . . . Contd.

- **SQL includes a between comparison operator**

- **Find the loan number of those loans with loan amounts between Rs 90,000 and Rs 100,000 (that is, >= Rs 90,000 and <= Rs100,000)**

```
select loan_number
        from loan
        where amount between 90000 and 100000
```



Cursors

- **When ever a select statement is written in PL/SQL block the data returned by that select statement is stored in buffer in an area called context area.**

- **The rows that are stored in buffer in an area and are the result of the select statement are combined called as activeset.**

- **Cursor is a pointer point to the activeset returned by an SQL statement.**

- **Cursors are classified into two types**
  - **Implicit Cursors:**
    - **An implicit cursors are those that are automatically created and provided by oracle.**
    - **An implicit cursors is created automatically whenever a DML operation is performed or a select statement was returned.**
  - **Explicit Cursors:**
    - **Explicit cursors must be declared and write appropriate code by the user.**

Cursors

- **Steps to create and write code for an explicit cursors:**

  1.CURSOR DECLARATION

  2.OPENING CURSOR

  3.FETCHING DATA

  4.CLOSING CURSOR

1. Cursor Declaration: Before using an explicit cursor it must be declared within the declaration section. By declaration of cursor, cursor associated with an SQL statement to which the cursor has to point.

Syntax: CURSOR C_Name IS select statement;

2. OPENING CURSOR: Before accessing the data using a cursor, cursor must be opened during opening a cursor, the select statement associated with that cursor is executed and cursor points to the first row of the active set returned by that select statement.

Syntax: OPEN C_name;

3. FETCHING DATA: To access data by using cursor first we have to copy the values of a row to which cursor points into local variables. This process is called fetching data from the cursor.

Syntax: FETCHING DATA C_Name INTO local variable list;

4. CLOSING CURSOR: After completion of work with the cursor we close the cursor by which memory occupied by cursor points will be released.

Syntax: CLOSE C_Name

Cursors…contd.

Cursors Attributes:

Cursor attributes are used to find whether data is retrieved by the fetch statement or whether cursor is already opened or the number of rows fetched from the cursor until now.

1. **%FOUND: This attribute returns true if the previous fetch statement retrieves a row. Otherwise false.**

2. **%NOT FOUND: This attribute returns true if the previous fetch statement does not return any rows otherwise returns false.**

3. **%IS OPEN: This attribute returns true if the specified cursor is already open otherwise returns false.**

4. **%ROW COUNT: This attribute returns the number of rows fetched from the specified cursor until now.**

Cursors…contd.

- **Write a PL/SQL cursors to update the salaries of employees by using following criteria. Manager-1000, Analyst-750, any other -500. [USING WHILE]**

```
DECLARE
        CURSOR My_cur IS SELECT EMPNO, JOB, SAL FROM EMP;
         E_NO   EMP.EMPNO%TYPE;
         J      EMP.JOB%TYPE;
         S      EMP.SAL%TYPE
  BEGIN
        OPEN My_cur;
         FETCH My_cur INTO E_NO, J, S;
         WHILE My_cur %FOUND
          LOOP
            IF J:='MANAGER' THEN
                        S:=S+1000;
             ELSEIF J:= 'ANALYST' THEN
                        S := S+750;
             ELSE
                    S:=  S+500
            END IF;
            UPDATE EMP SET SAL=S WHERE EMPNO:=E_NO;
            FETCH My_cur INTO E_NO, J, S;
          END LOOP;
  CLSOE My_cur;
  END
```

Cursors…contd.

- **Write a PL/SQL cursors to update the salaries of employees by using following criteria. Manager-1000, Analyst-750, any other -500.** [USING LOOP]

```
DECLARE
        CURSOR My_cur IS SELECT EMPNO, JOB, SAL FROM EMP;
         E_NO   EMP.EMPNO%TYPE;
         J          EMP.JOB%TYPE;
         S          EMP.SAL%TYPE
BEGIN
        OPEN My_cur;
         LOOP
         FETCH My_cur INTO E_NO, J, S;
         EXIT WHEN My_cur % NOT FOUND
         IF J:='MANAGER' THEN
                  S:=S+1000;
                  ELSEIF J:= 'ANALYST' THEN
                  S := S+750;
                 ELSE
                S:=  S+500
         END IF;
          UPDATE EMP SET SAL=S WHERE EMPNO:=E_NO;
         END LOOP;
 CLSOE My_cur;
END
```

Cursors…contd.

- **Write a PL/SQL cursors to update the salaries of employees by using following criteria. Manager-1000, Analyst-750, any other -500.** [USING FOR LOOP]

```
DECLARE
        CURSOR My_cur IS SELECT *  FROM EMP;
         E_ROW EMP % ROW TYPE;
BEGIN
        OPEN My_cur;
        FOR E_ROW IN My_cur
        LOOP
        IF E_ROW.JOB:='MANAGER' THEN
                E_ROW.SAL:=E_ROW.SAL+1000;
                ELSEIF E_ROW. JOB:= 'ANALYST' THEN
                E_ROW.SAL := E_ROW.SAL+750;
                ELSE
                E_ROW.SAL := E_ROW.SAL+500
        END IF;
         UPDATE EMP SET SAL=E_ROW.SAL WHERE EMPNO:=E_ROW.EMPNO;
        END LOOP;
 CLSOE My_cur;
END
```

Cursors…contd.

Parameterized Cursors: We can pass arguments to a cursor like passing arguments to a function by defining parameters at the time of cursor declaration.

Cursor Declaration:

CURSOR My_cur (p1 datatype, p2 datatype,…,pn datatype) IS select statement;

Opening Cursor:

OPEN My_cur (arg1, arg2,…,argn);

```
DECLARE                    /* Print the details of student who doesn't paid fee  */
CURSOR My_cur(p_course  course%type ) IS select * from Student where course = p_course;
SROW   Student % ROW_TYPE;
C    Student. Course%type := '&course';
BEGIN
    OPEN My_cur;
    FETCH My_cur INTO SROW.Sno, SROW.Sname, SROW.course, SROW.Fee;
    LOOP
       IF SROW.Fee>0 THEN
         DBMS_OUPUT.PUT_LINE(SROW.Sno||"||SROW.Sname||''||SROW.COURSE||''||SROW.Fee);
        ENDIF
     FETCH My_cur INTO SROW.Sno, SROW.Sname, SROW.course,  SROW.Fee;
     END LOOP;
     CLOSE My_cur;
 END;
```

Procedures…contd.

Parameter modes:

Parameters can be passed to a procedure or function in three modes IN, OUT, INOUT. The behavior of parameters when they are passed to a function or procedure in any of the three modes is as follows:

IN: When a parameter is passed in IN mode then that parameter is read only within the procedure or function. This means we can read the value present in that parameter but it is not possible to change the value of that parameter.

OUT: When a parameter is passed as OUT parameter then that parameter is write only within the procedure or function. This means that we can change the value of that parameter but it is not possible to read the value of that parameter. The change made to the out parameter will reflect that change in its corresponding argument.

INOUT :When a parameter is passed in INOUT mode then that parameter is read/write within the procedure or function. This means that we can read the value of that parameter as well as we can change the value of that parameter. Any change made to the INOUT parameter within the procedure or function will reflect that changes in its corresponding arguments.

Subprograms

- **PL/SQL blocks that are created with a name and are stored within the database are called as subprograms.**
- **Unlike anonymous blocks subprograms can be used any number of times as they are stored within the database and have a name.**
- **Subprograms include functions, procedures and packages.**

Procedures: Procedures are the PL/SQL subprograms that performs a task and doesn't return any value to the place from which it is called.

- **Creating a procedure:**

```
CREATE OR REPLACE PROCEDURE P_name ( PARM1 IN/OUT/INOUT DATATYPE,
                                     PARM2 IN/OUT/INOUT DATATYPE,
                                     PARM3 IN/OUT/INOUT DATATYPE,
                                     ....
                                     PARMn IN/OUT/INOUT DATATYPE) IS
LOCAL VARIABLE DECLARATIONS
BEGIN
EXECUTABLE STAEMENTS
EXCEPTION
ERROR HANDLING ROUTINES
END P_name
```

Where: - P_name is the procedure name,
- prameter1 to parametern are the names of parameters that hold the arguments passed to the procedure.
- IN, OUT, INOUT are the modes of parameters. Procedure doesn't require the keyword "DECLARE" for declaring the local variables.

Cursor…contd.

Calling a procedure: There are two approaches

- We call procedure from the SQL prompt then execute command is used.

  General Syntax: EXECUTE P_name (Arg1,Arg2,…,ARGn);

  E.g.: EXECUTE ADDSTUDENT(1, 'THEORY', ORACLE,2000);

- If we are calling a procedure from another PL/SQL block then we call directly i.e., use procedure name without using any commands.

  General Syntax: P_Name(Arg1, Arg2,…Argn);

Procedure…contd.

Write a procedure that updates the commission of all employees on following criteria:

Salesman 40% of Salary

Clerk 20% of salary

Others 10% of salary

```
CREATE OR REPLACE PROCEDURE COMMISION IS
CURSOR My_cur IS SELECT E_No, Job, Sal, Comm FROM EMPLOYEE;
EROW  EMP%ROW TYPE;
BEGIN
OPEN My_cur;
FETCH My_cur INTO EROW.E_No, EORW.Job, EORW.Sal, EROW.Comm;
WHILE My_cur%FOUND
LOOP
      IF EROW.Job='SALESMAN' THEN
         EROW.Comm=Sal*40/100;
      ELSE IF EROW.Job='CLERK'  THEN
        EROW.Comm=EORW.Sal*20/100;
     ELSE EROW.Comm=EROW.Sal*10/100;
    ENDIF
 UPDATE EMP SET Comm=EROW.Comm WHERE E_No=EROW.E_No;
FETCH My_cur INTO EROW.E_No, EORW.Job, EORW.Sal, EROW.Comm;
END LOOP;
CLOSE My_cur;
END;
```

Subprograms…contd.

- **Function is a PL/SQL block that performs a given task and will return a value to the place which it is called.**

- **Creating a function:**

```
CREATE OR REPLACE FUNCTION F_name ( PARM1 IN/OUT/INOUT DATATYPE,
                                    PARM2 IN/OUT/INOUT DATATYPE,
                                    PARM3 IN/OUT/INOUT DATATYPE,
                                    ….
                                    PARMn IN/OUT/INOUT DATATYPE)
                                    RETURN DATATYPE IS/AS
LOCAL VARIABLE DECLARATIONS
BEGIN
EXECUTABLE STAEMENTS
EXCEPTION
ERROR HANDLING ROUTINES
END F_name
```

Functions

- **Write a function accepts three numeric values and returns the total of these values.**

```
CREATE  OR REPLACE FUNCTION TOTAL (P_M1 MARKS.M1%TYPE,
                                   P_M2 MARKS.M2%TYPE
                                   P_M3 MARKS.M3%TYPE) RETURN NUMBER IS
    BEGIN
    RETURN (P_M1+P_M2+P_M3);
    END TOTAL;
```

- **Write a function that accepts total marks of a student in three subjects and returns average.**

```
CREATE  OR REPLACE FUNCTION AVEG (P_TOT MARKS.TOT%TYPE) RETURN NUMBER IS
BEGIN
RETURN (P_TOT/3);
END  AVEG;
```

- **Write a function that accepts average marks of a student and returns his grade.**

```
CREATE  OR REPLACE FUNCTION Grade (P_AVG  MARKS.AVEG%TYPE) RETURN VARCHAR2 IS
G    MARKS.GRADE%TYPE;
BEGIN
IF P_AVG>= 90 THEN
   G:= 'DISTINCTION'
ELSEIF  P_AVEG>=65 THEN
   G:= 'FIRST CLASS'
ELSE
  G :=' PASS'
ENDIF
RETURN (G);
END Grade
```

Packages

- **Packages are another type of PL/SQL blocks that are used to group related functions and procedures.**

- **Packages are also used to declare global variables, curser and exceptions.**

- **Like subprograms packages are also stored in the database but unlike subprograms they cannot be executed and they doesn't receive any arguments.**

- **A package consists of two parts, package specification and package body.**

  -Package Specification contains variables, cursors, exception declarations, functions and procedure definitions.

  -Package Body consists of the code to be executed for the functions and procedures declared within the package specification.

-Package specification and package body must be created separately but both must have the same name.

Packages…contd.

Package specification:

Syntax:

```
CREATE OR REPLACE PACKAGE P_Name IS/AS
VAR DECLARATIONS
CURSOR DECLARATIONS
EXCEPTION DECLARATIONS
FUNCTION DECLARATIONS
PROCEDURE DECLARATIONS
END P-N ame
```

E.g.,

```
CREATE OR REPLACE PACKAGE Stu_Pack AS
    PROCEDURE ADD_STUDENT(P_SNO STUDENT.SNo%TYPE, P_SNAME STUDENT.SName%TYPE,
                          P_COURSE STUDENT.Course);
    FUNCTION TOTAL (P_M1 MARKS.M1%TYPE, P_M2 MARKS.M2%TYPE, P_M3 MARKS.M3%TYPE)
                          RETURN NUMBER;
    FUNCTION AVEG(P_TOT, MARKS.TOT%TYPE) RETURN NUMBER;
    FUNCTION GRADE(P_AVG MARKS.AVEG%TYPE) RETURN VARCHAR2;
    PROCEDURE ADD_MARKS(P_SNO MARKS.Sno%TYPE, P_M1   MARKS.M1%TYPE,
                          P_M2   MARKS.M2%TYPE, P_M3   MARKS.M3%TYPE);
END Stu_Pack;
```

Packages…contd.

Package body:

Syntax: CREATE OR REPLACE PACKAGE BODY PB_Name IS/AS
FUNCTION DEFINITIONS
PROCEDURE DEFINITIONS
END PB-N ame

```
CREATE OR REPLACE PACKAGE BODY  Stu_Pack AS
        PROCEDURE ADD_STUDENT(P_SNo STUDENT.SNo%TYPE, P_SName STUDENT.SName%TYPE,
                                        P_Course STUDENT.Course) IS
          BEGIN
             INSERT INTO STUDENT VALUES(P_Sno, P_SName, P_Course);
          END ADD_STUDENT


 FUNCTION TOTAL (P_M1 MARKS.M1%TYPE, P_M2 MARKS.M2%TYPE, P_M3 MARKS.M3%TYPE)
  BEGIN
     RETURN (P_M1+P_M2+P_M3);
  END TOTAL

        FUNCTION AVEG(P_TOT, MARKS.TOT%TYPE)
         BEGIN
            RETURN (P_TOT/3);
        END AVEG
```

Packages…contd.

```
FUNCTION GRADE(P_AVG MARKS.AVEG%TYPE) RETURN VARCHAR2;
        G MARKS.GRADE%TYPE
                IF P_AVG >=90 THEN
                G:= 'Distinction';
                ELSEIF P_AVG>= 65 THEN
                G:= 'First Class';
                ELSEIF P_AVG>= 55 THEN
                G:= 'Second Class';
                ELSEIF P_AVG>= 35 THEN
                G:= 'Pass Class';
                ELSE
                G:= 'Fail';
                ENDIF
        RETURN(G);
        END GRADE;
PROCEDURE ADD_MARKS(P_SNO MARKS.Sno%TYPE, P_M1   MARKS.M1%TYPE,
                                        P_M2   MARKS.M2%TYPE, P_M3   MARKS.M3%TYPE)IS
T   MARKS.TOT%TYPE:= Stu_Pack.TOTAL(P_M1,P_M2,P_M3);
A    MARKS.AGEV%TYPE:=Stu_Pack.AVEG(T);
G    MARKS.GRADE%TYPE:= Stu_Pack.GRADE(A);
BEGIN
INSERT INTO MARKS VALUES(P_Sno, P_M1, P_M2, P_M3,T, A, G);
END ADD_MARKS
```

Packages…contd.

Package Specification:

```
CREATE OR REPLACE PACKAGE c_package AS
        -- Adds a customer
  PROCEDURE addCustomer(c_id   customers.id%type,
                        c_name  customers.name%type,
                       c_age  customers.age%type,
                       c_addr customers.address%type,
                       c_sal  customers.salary%type);

        -- Removes a customer
   PROCEDURE delCustomer(c_id  customers.id%TYPE);

       --Lists all customers
   PROCEDURE listCustomer;

END c_package;
```

Packages…contd.

CREATING THE PACKAGE BODY:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
  PROCEDURE addCustomer(c_id  customers.id%type,
    c_name customers.name%type,
    c_age  customers.age%type,
    c_addr  customers.address%type,
    c_sal   customers.salary%type)
  IS
  BEGIN
    INSERT INTO customers (id,name,age,address,salary)
      VALUES(c_id, c_name, c_age, c_addr, c_sal);
  END addCustomer;

  PROCEDURE delCustomer(c_id   customers.id%type) IS
  BEGIN
     DELETE FROM customers
      WHERE id = c_id;
  END delCustomer;
```

```
 PROCEDURE listCustomer IS
   CURSOR c_customers is
     SELECT  name FROM customers;
   TYPE c_list is TABLE OF
customers.name%type;
   name_list c_list := c_list();
   counter integer :=0;
   BEGIN
     FOR n IN c_customers LOOP
     counter := counter +1;
     name_list.extend;
     name_list(counter)  := n.name;
     dbms_output.put_line('Customer('
||counter|| ')'||name_list(counter));
     END LOOP;
   END listCustomer;

END c_package;
```

Advanced SQL

Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  Example: **date** '2005-7-27'

- **time:** Time of day, in hours, minutes and seconds.
  Example: **time** '09:00:30' **time** '09:00:30.75'

- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'

- **interval:** period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

PL/SQL Dates

Function	Time Zone	Datatype Returned
CURRENT_DATE	Session	DATE
CURRENT_TIMESTAMP	Session	TIMESTAMP WITH TIME ZONE
LOCALTIMESTAMP	Session	TIMESTAMP
SYSDATE	Database server	DATE
SYSTIMESTAMP	Database server	TIMESTAMP WITH TIME ZONE

Data Types in SQL …contd.

- **Can extract values of individual fields from date/time/timestamp**

  Example: **extract** (**year from** r.starttime)

- **Can cast string types to date/time/timestamp**

  Example: **cast** <string-valued-expression> **as date**

  Example: **cast** <string-valued-expression> **as time**

User-Defined Types

- **create type construct in SQL creates user-defined type**

  create type *Rupees* **as numeric (12,2) final**

- **create domain construct in SQL-92 creates user-defined domain types**

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

Domain Constraints

- **Domain constraints are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.**

- **New domains can be created from existing data types**

  Example: **create domain** *Rupees* **numeric**(12, 2)
  create domain *Pounds* **numeric**(12,2)

- **We cannot assign or compare a value of type Pounds to a value of type Rupees.**

  However, we can convert type as below
  (**cast** *r.A* **as** *Rupees*)
  (Should also multiply by the Pounds-to-Rupees conversion-rate)

Large-Object Types

- **Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:**
  - **blob: binary large object - object is a large collection of un-interpreted binary data (whose interpretation is left to an application outside of the database system)**
  - **clob: character large object -- object is a large collection of character data**
  - **When a query returns a large object, a pointer is returned rather than the large object itself.**

Integrity Constraints

- **Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.**
  - **A checking account must have a balance greater than Rs10,000.00**
  - **A salary of a bank employee must be at least Rupees 1000.00 an month**
  - **A customer must have a (non-null) phone number**

- **Constraints on a Single Relation**
  - **not null**
  - **primary key**
  - **unique**
  - **check (P), where P is a predicate**

Constraints…contd

- **not null Constraints**
  - **Declare branch_name for branch is not null**

    Example: branch_name char(15) not null

  - **Declare the domain Dollars to be not null**

    Example: create domain Dollars numeric(12,2) not null

- **unique Constraints**
  - **unique (A1, A2, …, Am)**
  - **The unique specification states that the attributes A1, A2, … Am form a candidate key.**
  - **Candidate keys are permitted to be null (in contrast to primary keys).**

The check clause

- **check** (*P*), where *P* is a predicate

Example: **Declare branch_name as the primary key for branch and ensure that the values of assets are non-negative.**

```
create table branch
   (branch_name     char(15),
    branch_city       char(30),
    assets                integer,
    primary key (branch_name),
    check (assets >= 0))
```

- **Use check clause to ensure that an hourly_wage domain allows only values greater than a specified value.**

```
create domain hourly_wage numeric(5,2)
constraint value_test check(value > = 4.00)
```

- The domain has a constraint that ensures that the hourly_wage is greater than 4.00
- The clause constraint value_test is optional; useful to indicate which constraint an update violated.

Referential Integrity

- **Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.**
  - Example: If "Perryridge" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch "Perryridge".

- **Primary and candidate keys and foreign keys can be specified as part of the SQL create table statement:**
  - The **primary key** clause lists attributes that comprise the primary key.
  - The **unique key** clause lists attributes that comprise a candidate key.
  - The **foreign key** clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Referential Integrity in SQL …Contd.

create table *account*
(*account_number* **char**(10)**,**
branch_name **char**(15),
balance **integer**,
primary key (*account_number),*
foreign key (*branch_name*) **references** *branch*)

create table *depositor*
(*customer_name* **char**(20)**,**
account_number **char**(10)**,**
primary key (*customer_name, account_number),*
foreign key (*account_number*) **references** *account,*
foreign key (*customer_name*) **references** *customer*)

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.

- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>

- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion

  This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

- Asserting

  for all X, P(X)

  is achieved in a round-about fashion using

  not exists X such that not P(X)

Assertion…contd.

- **Every loan has at least one borrower who maintains an account with a minimum balance of Rs1000**

- ```
  create assertion balance_constraint check
   (not exists (
       select *
      from loan
       where not exists (
          select *
           from borrower, depositor, account
           where loan.loan_number = borrower.loan_number
               and borrower.customer_name = depositor.customer_name
               and depositor.account_number = account.account_number
               and account.balance >= 1000)))
  ```
  ```

Assertion…contd.

- **The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.**

- **create assertion** sum_constraint **check**
  (**not exists** (**select ***
  from branch
  where (**select sum**(amount)
  from loan
  where loan.branch_name =
  branch.branch_name)
  >= (**select sum** (amount)
  from account
  where loan.branch_name =
  branch.branch_name)))

Authorization

- **Forms of authorization on parts of the database:**

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

- **Forms of authorization to modify the database schema** :
- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> **to** <user list>

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role

- Granting a privilege on a view does not imply granting any privileges on the underlying relations.

- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view

  Example: grant users U_1, U_2, and U_3 **select** authorization on the branch relation:

  grant select on branch **to** U_1, U_2, U_3

- **insert**: the ability to insert tuples

- **update**: the ability to update using the SQL update statement

- **delete**: the ability to delete tuples.

- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>

  - Example: **revoke select on** *branch* **from** U_1, U_2, U_3

- All privileges that depend on the privilege being revoked are also revoked.

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.

- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

Embedded SQL

- The SQL standard defines embedding of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement > END_EXEC

- **From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.**

- Specify the query in SQL and declare a cursor for it

```
EXEC SQL
    declare c cursor for
    select depositor.customer_name, customer_city
    from depositor, customer, account
    where depositor.customer_name = customer.customer_name
        and depositor account_number = account.account_number
        and account.balance > :amount
END_EXEC
```

Embedded SQL ... Contd.

- The **open** statement causes the query to be evaluated

  EXEC SQL **open** *c* END_EXEC

- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** :*cn*, :*cc* END_EXEC

  Repeated calls to **fetch** get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* END_EXEC

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

J

J#	JNAME	CITY
J1	Sorter	Paris
J2	Display	Rome
J3	OCR	Athens
J4	Console	Athens
J5	RAID	London
J6	EDS	Oslo
J7	Tape	London

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

The suppliers-parts-projects database