ANNAI WOMEN'S COLLEGE

(Arts & Science)

(Affiliated to Bharathidasan University, Tiruchirappalli – 620 024)

Aurobindo Nagar, TNPL Road, Punnamchatram, Karur – 639 136.

Course Material

Paper Name : DATA BASE SYSTEMS

Paper Code : 16SCCCA4

Subject Handler : S.Leelavathi

No.Of Units : 5 units

SUBJECT CODE: 16SCCCA4 SUBJECT NAME: DATA BASE SYSTEMS

DATA BASE SYSTEMS

Unit I

Introduction: Database-System Applications- Purpose of Database Systems - View of Data--Database Languages - Relational Databases - Database Design -Object-Based and Semi structured Databases - Data Storage and Querying Transaction Management -Data Mining and Analysis - Database Architecture - Database Users and Administrators - History of Database Systems.

Unit II

Relational Model: Structure of Relational Databases - Fundamental Relational-Algebra Operations Additional Relational-Algebra Operations- Extended Relational-Algebra Operations - Null Values - Modification of the Database.

Unit III

SQL: Data Definition - Basic Structure of SQL Queries - S e t O p e r a t i o n s - Ag g r e g a t e Fu n c t i o n s - N u l l V a l u e s - Nested Subqueries - Complex Queries - Views -Modification of the Database - Joined Relations - SQL Data Types and Schemas - Integrity Constraints -Authorization - Embedded SQL

Unit IV

Relational Languages: The Tuple Relational Calculus - The Domain Relational Calculus -Query-by- Example. Database Design and the E-R Model: Overview of the Design Process - The Entity-Relationship Model - 3 Constraints - Entity-Relationship Diagrams - Entity-Relationship Design Issues - Weak Entity Sets - Database Design for Banking Enterprise

Unit V

Relational Database Design: Features of Good Relational Designs - Atomic Domains and First Normal Form - Decomposition Using Functional Dependencies - Functional-Dependency Theory - Decomposition Using Functional Dependencies - Decomposition Using Multivalued Dependencies-More Normal Forms - Database-Design Process

Text Book:

1. Database System Concepts, Fifth edition, Abraham Silberschatz , Henry F. Korth, S. Sudarshan, McGraw-Hill-2005.

Unit-1

Database Management System (DBMS)

- DBMS contains information about a particular enterprise
  - Collection of interrelated data
  - Set of programs to access the data
  - 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

Purpose of Database Systems

- 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
    - 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

Levels of Abstraction

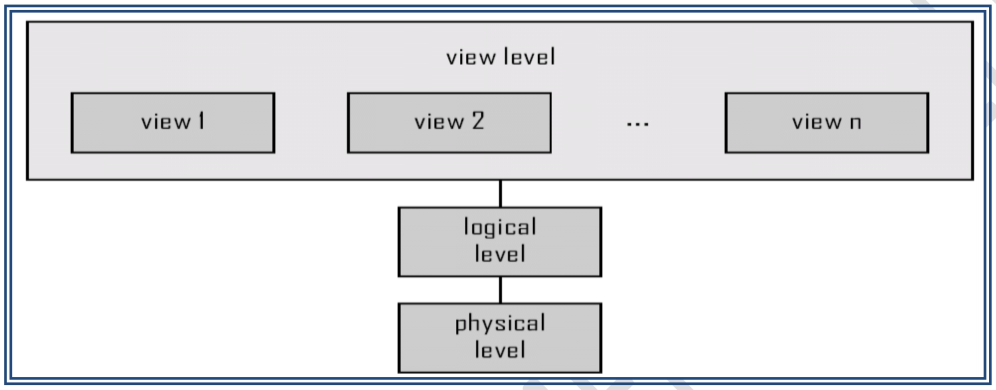
- Physical level: describes how a record (e.g., customer) is stored.
- Logical level: describes data stored in database, and the relationships among the data.
  type *customer* = record

customer_id : string;
customer_name : string;
customer_street : string;
customer_city : integer;
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



Instances and Schemas

- Similar to types and variables in programming languages
- Schema – the logical structure of the database
  - Example: The database consists of information about a set of customers and accounts and the relationship between them)
  - 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
- Physical Data Independence – the ability to modify the physical schema without changing the logical schema
  - Applications depend on the logical schema
  - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

Data Models

- A collection of tools for describing
  - Data
  - Data relationships
  - Data semantics

    - Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semistructured data model (XML)
- Other older models:
    - Network model
    - Hierarchical model

Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
    - DML also known as query language
- Two classes of languages
    - Procedural – user specifies what data is required and how to get those data
    - Declarative (nonprocedural) – user specifies what data is required without specifying how to get those data
- SQL is the most widely used query language

Data Definition Language (DDL)

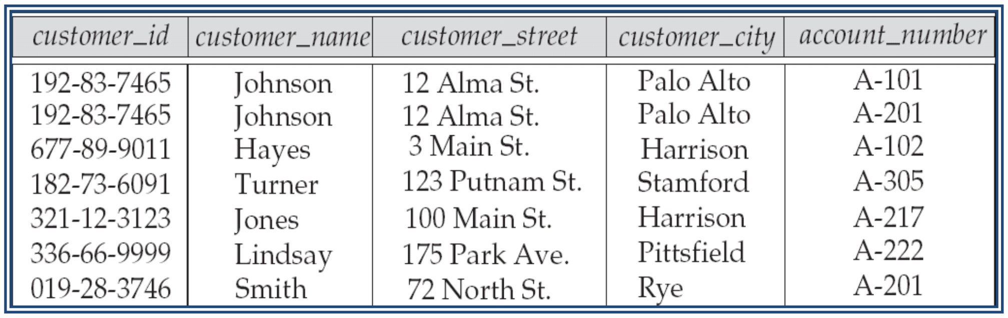
- Specification notation for defining the database schema

Example:

```
create table account (
    account-number    char(10),
    balance           integer)
```

- DDL compiler generates a set of tables stored in a *data dictionary*
- Data dictionary contains metadata (i.e., data about data)
    - Database schema
    - Data *storage and definition* language
        - Specifies the storage structure and access methods used
    - Integrity constraints
        - Domain constraints
        - Referential integrity (references constraint in SQL)
        - Assertions
    - Authorization

Relational Model



customer_id	customer_name	customer_street	customer_city	account_number
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

SQL

- SQL: widely used non-procedural language
  - Example: Find the name of the customer with customer-id 192-83-7465

    select *customer.customer_name*
    from *customer*
    where *customer.customer_id* = '192-83-7465'

  - Example: Find the balances of all accounts held by the customer with customer-id 192-83-7465

    select *account.balance*
    from *depositor, account*
    where *depositor.customer_id* = '192-83-7465' and
    depositor.account_number = *account.account_number*

- Application programs generally access databases through one of
  - Language extensions to allow embedded SQL
  - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database

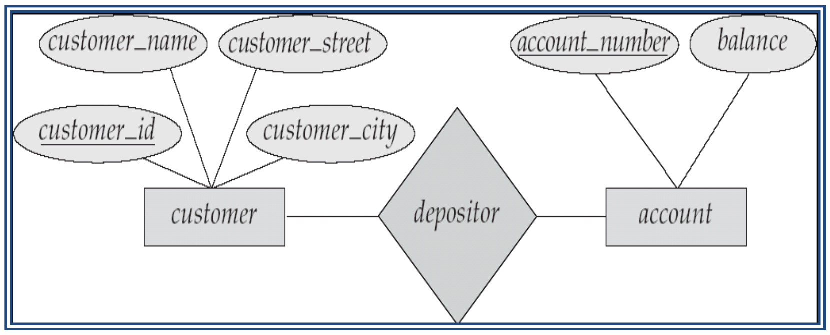
Database Design

The process of designing the general structure of the database:

- Logical Design – Deciding on the database schema. Database design requires that we find a "good" collection of relation schemas.
  - Business decision – What attributes should we record in the database?
  - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database

The Entity-Relationship Model

- Models an enterprise as a collection of *entities* and *relationships*
  - Entity: a "thing" or "object" in the enterprise that is distinguishable from other objects
    - Described by a set of *attributes*
  - Relationship: an association among several entities
- Represented diagrammatically by an *entity-relationship diagram:*
-



Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Provide upward compatibility with existing relational languages.

XML: Extensible Markup Language

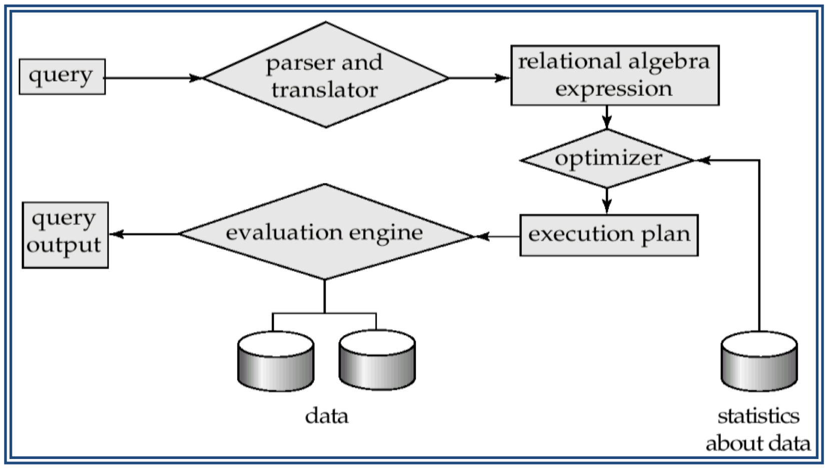
- Defined by the WWW Consortium (W3C)
- Originally intended as a document markup language not a database language
- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange data, not just documents
- XML has become the basis for all new generation data interchange formats.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Storage Management

- Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
  - Interaction with the file manager
  - Efficient storing, retrieving and updating of data
- Issues:
  - Storage access
  - File organization
  - Indexing and hashing

Query Processing

1. 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

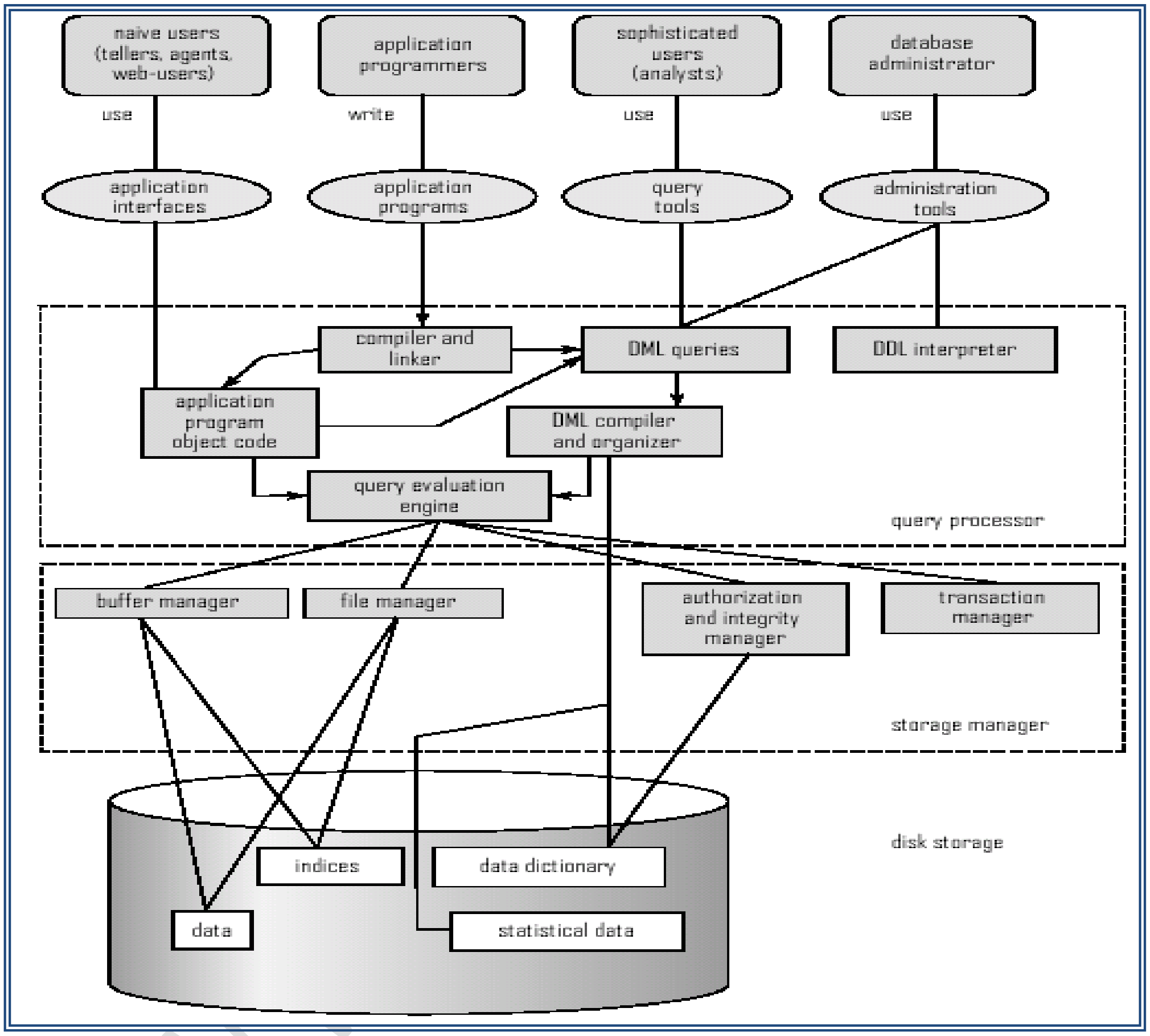
Transaction Management

- 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 (e.g., 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.

Database Architecture

The architecture of a database systems is greatly influenced by
the underlying computer system on which the database is running:

- Centralized
- Client-server
- Parallel (multi-processor)
- Distributed



Database Users

Users are differentiated by the way they expect to interact with
the system

- Application programmers – interact with system through DML calls
- Sophisticated users – form requests in a database query language
- Specialized users – write specialized database applications that do not fit into the traditional data processing framework
- Naïve users – invoke one of the permanent application programs that have been written previously
  - Examples, people accessing database over the web, bank tellers, clerical staff

Database Administrator

- Coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.
- Database administrator's duties include:

- Schema definition
- Storage structure and access method definition
- Schema and physical organization modification
- Granting user authority to access the database
- Specifying integrity constraints
- Acting as liaison with users
- Monitoring performance and responding to changes in requirements

History of Database Systems

- 1950s and early 1960s:
  - Data processing using magnetic tapes for storage
    - Tapes provide only sequential access
  - Punched cards for input
- Late 1960s and 1970s:
  - Hard disks allow direct access to data
  - Network and hierarchical data models in widespread use
  - Ted Codd defines the relational data model
    - Would win the ACM Turing Award for this work
    - IBM Research begins System R prototype
    - UC Berkeley begins Ingres prototype
  - High-performance (for the era) transaction processing
- 1980s:
  - Research relational prototypes evolve into commercial systems
    - SQL becomes industrial standard
  - Parallel and distributed database systems
  - Object-oriented database systems
- 1990s:
  - Large decision support and data-mining applications
  - Large multi-terabyte data warehouses
  - Emergence of Web commerce
- 2000s:
  - XML and XQuery standards
  - Automated database administration

Unit-2

Relational Model

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

Basic Structure

- Formally, given sets $D_1, D_2, \ldots. D_n$ a relation r is a subset of
  $D_1 \times D_2 \times \ldots \times D_n$
  Thus, a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$
- Example: If
  - *customer_name* = {Jones, Smith, Curry, Lindsay, …}
    /* Set of all customer names */
  - *customer_street* = {Main, North, Park, …} /* set of all street names*/
  - *customer_city* = {Harrison, Rye, Pittsfield, …} /* set of all city names */

Then r = { (Jones, Main, Harrison),
(Smith, North, Rye),
(Curry, North, Rye),
(Lindsay, Park, Pittsfield) }
is a relation over
customer_name x *customer_street* x *customer_city*

1.Attribute Types

- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the domain of the attribute
- Attribute values are (normally) required to be atomic; that is, indivisible
  - E.g. the value of an attribute can be an account number, but cannot be a set of account numbers
- Domain is said to be atomic if all its members are atomic
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations
  - We shall ignore the effect of null values in our main presentation and consider their effect later

2.Relation Schema

- $A_1, A_2, \ldots, A_n$ are *attributes*
- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*
  Example:
  Customer_schema = (*customer_name, customer_street, customer_city*)
- $r(R)$ denotes a *relation r* on the *relation schema R*

Example:

customer (Customer_schema)

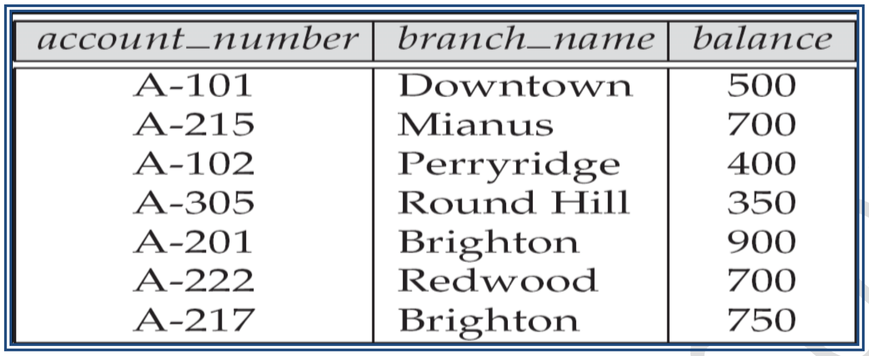
3.Relation Instance

- 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

Relations are Unordered

n Order of tuples is irrelevant (tuples may be stored in an arbitrary order)

n Example: *account* relation with unordered tuples



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

Database

- A database consists of multiple relations
- Information about an enterprise is broken up into parts, with each relation storing one part of the information

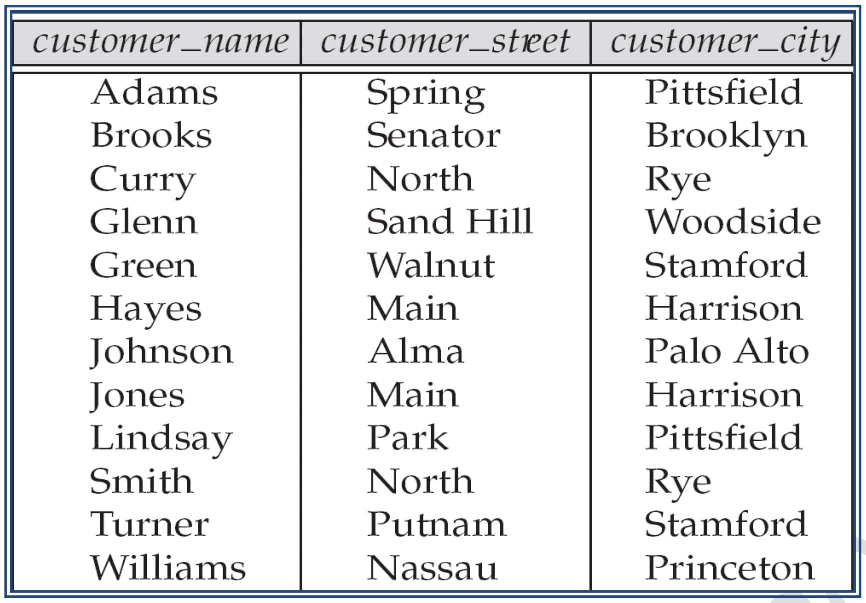
account : stores information about accounts

depositor : stores information about which customer owns which account

customer : stores information about customers

- Storing all information as a single relation such as
  bank(*account_number, balance, customer_name,* ..)
  results in
  - repetition of information
    - e.g.,if two customers own an account (What gets repeated?)
  - the need for null values
    - e.g., to represent a customer without an account
- Normalization theory (Chapter 7) deals with how to design relational schemas

The *customer* Relation



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

4.Keys

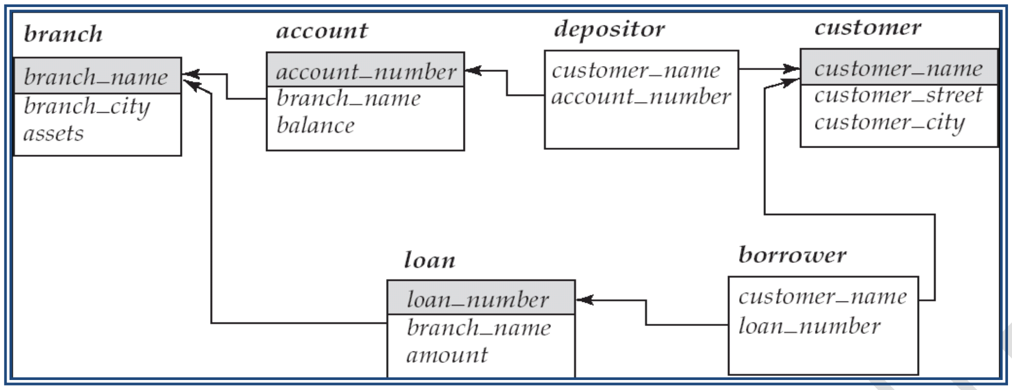
- Let K $\subseteq$ R
- *K* is a superkey of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
  - by "possible *r* " we mean a relation *r* that could exist in the enterprise we are modeling.
  - Example: {*customer_name, customer_street*} and
    {*customer_name*}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

5.Foreign Keys

- 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.

- Schema diagram

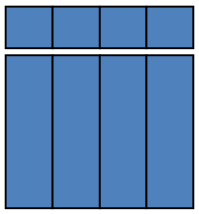


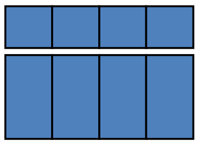
6.Query Languages

- Language in which user requests information from the database.
- Categories of languages
  - Procedural
  - Non-procedural, or declarative
- “Pure” languages:
  - Relational algebra
  - Tuple relational calculus
  - Domain relational calculus
- Pure languages form underlying basis of query languages that people use.

Fundamental Relational Algebra

- Procedural language
- Six basic operators
  - select: σ
  - project: $\prod$
  - union: $\cup$
  - 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

- 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)

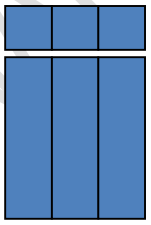
Each term is one of:

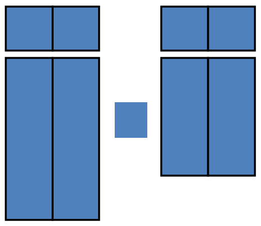
<attribute> *op* <attribute> or <constant>

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

- Example of selection:

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

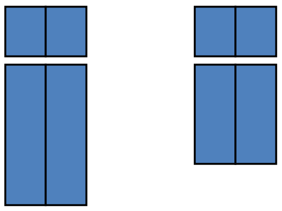


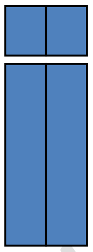


- Notation:

  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*

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



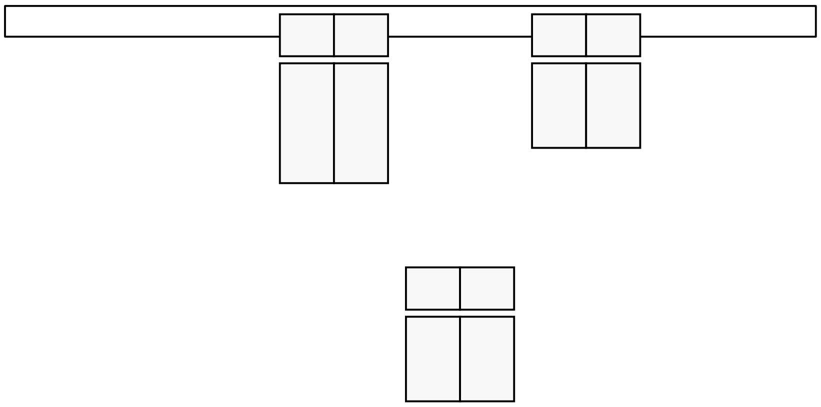


- Notation: $r \cup s$
- Defined as:

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

- 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

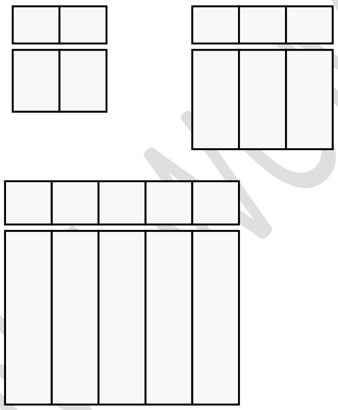
$$\prod_{customer\_name} (depositor) \cup \prod_{customer\_name} (borrower)$$



- 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



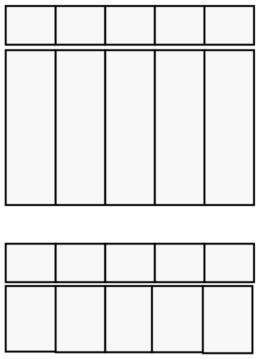
- Notation $r \times s$
- Defined as:

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

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

Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{A=C}(r \times s)$
- $r \times s$
- $\sigma_{A=C}(r \times s)$



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

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

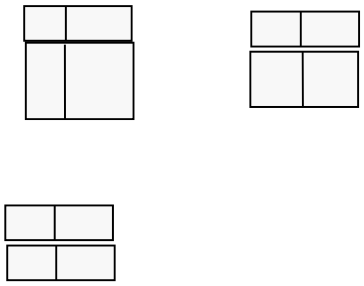
Additional Relational algebra Operations

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

- Set intersection
- Natural join
- Division
- Assignment

Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:
- $r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$
- Assume:
  - *r, s* have the *same arity*
  - attributes of *r* and *s* are compatible
- Note: $r \cap s = r - (r - s)$



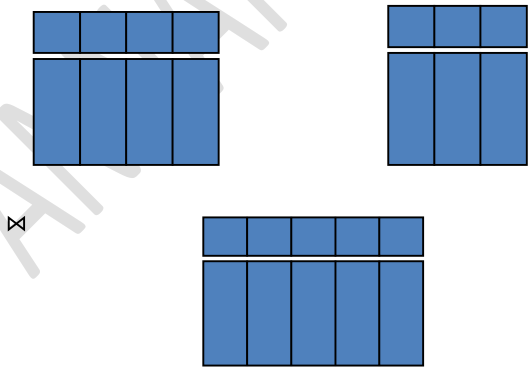
Natural-Join Operation

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

R = (*A, B, C, D*)
S = (*E, B, D*)

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

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



Division Operation

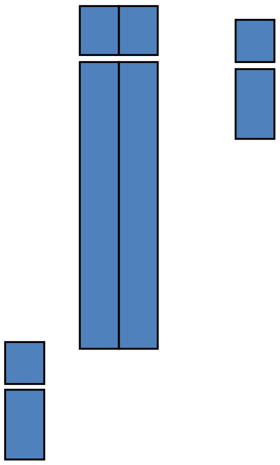
- Notation:
- Suited to queries that include the phrase “for all”.
- Let *r* and *s* be relations on schemas *R* and *S* respectively where

- $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
- $S = (B_1, \ldots, B_n)$

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



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

Assignment Operation

- The assignment operation (←) provides a convenient way to express complex queries.
  - Write query as a sequential program consisting of
    - a series of assignments
    - followed by an expression whose value is displayed as a result of the query.
  - Assignment must always be made to a temporary relation variable.
- Example: Write $r \div s$ as

$temp1 \leftarrow \prod_{R-S} (r)$

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

$result = temp1 - temp2$

- The result to the right of the ← is assigned to the relation variable on the left of the ←.
- May use variable in subsequent expressions.

Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions
- Outer Join

Generalized Projection

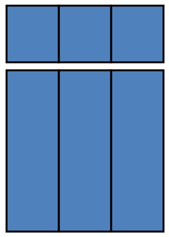
- Extends the projection operation by allowing arithmetic functions to be used in the projection list.
- *E* is any relational-algebra expression
- Each of $F_1, F_2, \ldots, F_n$ are are arithmetic expressions involving constants and attributes in the schema of *E*.
- Given relation *credit_info(customer_name, limit, credit_balance),* find how much more each person can spend:

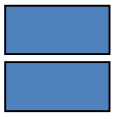
$$\prod_{customer\_name,\ limit - credit\_balance} (credit\_info)$$

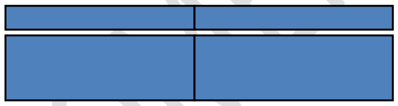
Aggregate Functions and Operations

- Aggregation function takes a collection of values and returns a single value as a result.

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values
- Aggregate operation in relational algebra
- *E* is any relational-algebra expression
  - $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name







- Result of aggregation does not have a name
  - Can use rename operation to give it a name
  - For convenience, we permit renaming as part of aggregate operation

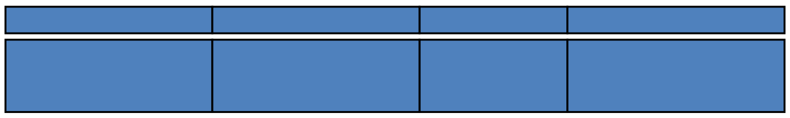
Outer Join

- 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.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) false by definition.
    - We shall study precise meaning of comparisons with nulls later

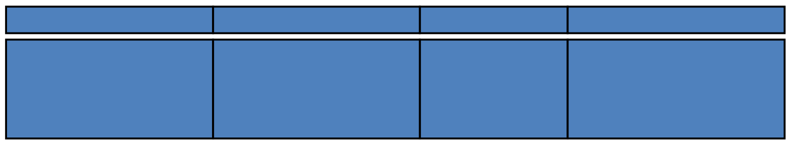
⋈

⟕

⟕



⟗



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*
    - In SQL "*P* is unknown" evaluates to true if predicate *P* evaluates to *unknown*
- Result of select predicate is treated as *false* if it evaluates to *unknown*

Modification of the Database

- The content of the database may be modified using the following operations:
    - Deletion
    - Insertion
    - Updating
- All these operations are expressed using the assignment operator.

Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

  $r \leftarrow r - E$

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

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.

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
- 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

Unit 3

SQL (structure query language)

Data Definition Language

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

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.
- More are covered in Chapter 4.

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 values in the domain of attribute A_i

- Example:

```
create table branch
(branch_name char(15) not null,
branch_city  char(30),
assets       integer)
```

Integrity Constraints in Create Table

- 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),
         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

Drop and Alter Table Constructs

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

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., 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.
- 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:

  Õ*branch_name* (*loan*)
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*
  - Some people use upper case wherever we use bold font.
- SQL allows duplicates in relations as well as in query results.
- 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*
- 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.
- The query:

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

  would return a relation that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100.

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 $1200.

  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.
- Comparisons can be applied to results of arithmetic expressions.
- SQL includes a between comparison operator
- Example: Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, ³ $90,000 and £ $100,000)

- select *loan_number*
  from *loan*
  where *amount* between 90000 and 100000

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 *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'

The Rename Operation

- The SQL allows renaming relations and attributes using the as clause:
  old-name as *new-name*
- 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 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*

n 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'

n Keyword as is optional and may be omitted
borrower as *T* ≡ *borrower T*

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.

Ordering the Display of Tuples

- 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: order by *customer_name* desc

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 $r_1 \times r_2$

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

  $r_1 = \{(1, a)\ (2,a)\}$ $\quad r_2 = \{(2), (3), (3)\}$
- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1) \times r_2$ 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:

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

Set Operations

- The set operations union, intersect, and except operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- 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*
- 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*

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.
  - Example: 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
  - More on next slide

Null Values and Three Valued Logic

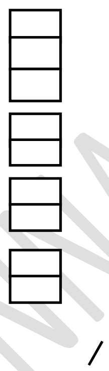
- Any comparison with *null* returns *unknown*
  -
  - Example*: 5 < null or null <> null or null = null*
- Thsree-valued logic using the truth value *unknown*:
  - OR: (*unknown* or *true*) = *true,*
    (*unknown* or *false*) = *unknown*
    (*unknown* or *unknown) = unknown*
  - AND: *(true* and *unknown) = 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.

Nested Subqueries

- 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.



Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)

- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

```
(select customer_name, borrower.loan_number, branch_name
     from borrower, loan
     where borrower.loan_number = loan.loan_number )
```

- A view provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view.
- 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.
- When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.

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')
```

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 )
```

- 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 (otherwise queries like

```
        insert into table1 select * from table1
```

  would cause problems)

Modification of the Database – Updates

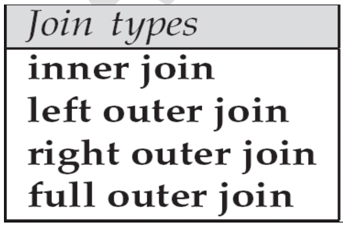
- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two update statements:

```
        update account
set balance = balance ∗ 1.06
where balance > 10000
        update account
set balance = balance ∗ 1.05
where balance ≤ 10000
```

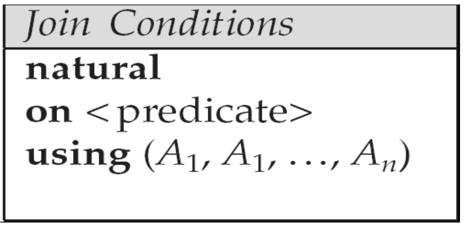
  - The order is important
  - Can be done better using the case statement (next slide)

Joined Relations

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the from clause
- Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.



Join types
inner join
left outer join
right outer join
full outer join



Join Conditions
natural
on < predicate>
using $(A_1, A_1, \ldots, A_n)$

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

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*

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

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

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

Unit 4

Relational Languages

- Tuple Relational Calculus
- Domain Relational Calculus
- Query-by-Example (QBE)
- Datalog

Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form

  $\{t \mid P(t)\}$
- It is the set of all tuples *t* such that predicate *P* is true for *t*
- *t* is a *tuple variable*, *t* [*A*] denotes the value of tuple *t* on attribute *A*
- $t \in r$ denotes that tuple *t* is in relation *r*
- *P* is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<, \leq, =, \neq, >, \geq$)
3. Set of connectives: and ($\wedge$), or (v), not ($\neg$)
4. Implication ($\Rightarrow$): x $\Rightarrow$ y, if x if true, then y is true

   $$x \Rightarrow y \equiv \neg x \vee y$$
5. Set of quantifiers:
   - $\exists\, t \in r\,(Q(t)) \equiv$ "there exists" a tuple in *t* in relation *r* such that predicate *Q* (*t*) is true
   - $\forall t \in r\,(Q(t)) \equiv Q$ is true "for all" tuples *t* in relation *r*

Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus

- Each query is an expression of the form:

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

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

Query-by-Example (QBE)

Basic Structure
Queries on One Relation
Queries on Several Relations
The Condition Box
The Result Relation
Ordering the Display of Tuples
Aggregate Operations
Modification of the Database

QBE — Basic Structure

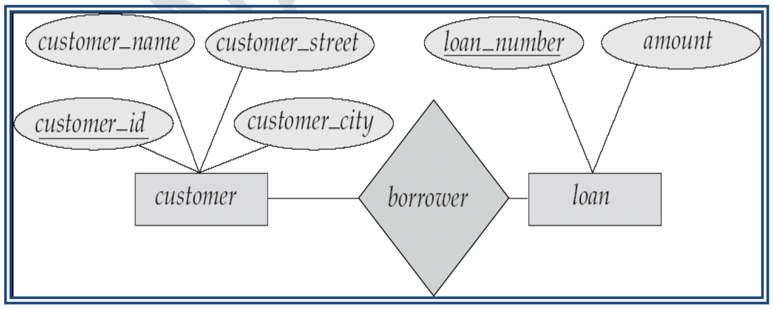
A graphical query language which is based (roughly) on the domain relational calculus
Two dimensional syntax – system creates templates of relations that are requested by users
Queries are expressed "by example"

Entity-Relationship Model

Modeling

A *database* can be modeled as:
a collection of entities,
relationship among entities.
An entity is an object that exists and is distinguishable from other objects.
Example: specific person, company, event, plant
Entities have *attributes*
Example: people have *names* and *addresses*
An entity set is a set of entities of the same type that share the same properties.
Example: set of all persons, companies, trees, holidays

E-R Diagrams



Rectangles represent entity sets.
Diamonds represent relationship sets.
Lines link attributes to entity sets and entity sets to relationship sets.

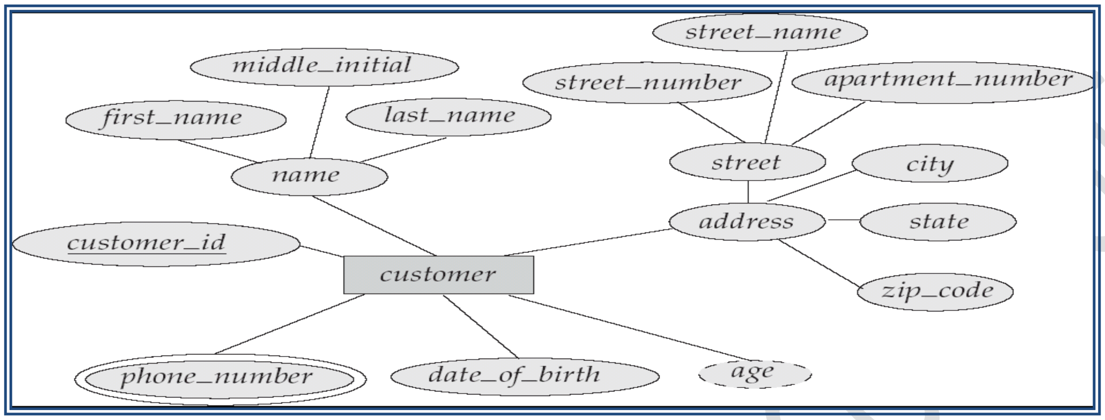
Ellipses represent attributes

Double ellipses represent multivalued attributes.

Dashed ellipses denote derived attributes.

Underline indicates primary key attributes (will study later)

E-R Diagram With Composite, Multivalued, and Derived Attributes



Constraints

We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (—), signifying "many," between the relationship set and the entity set.

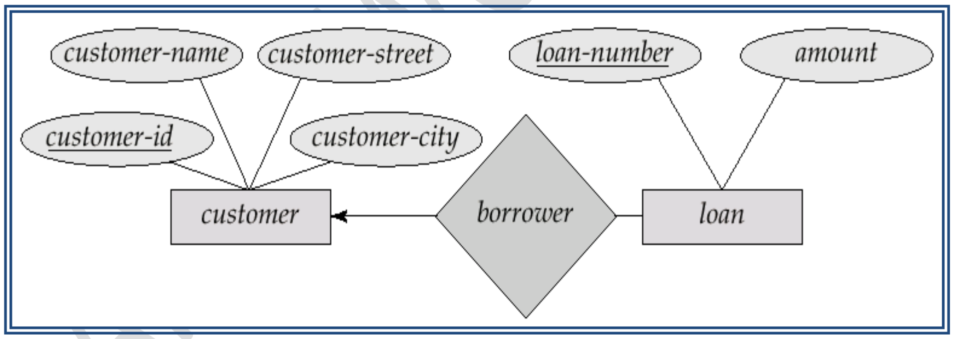
One-to-one relationship:

A customer is associated with at most one loan via the relationship *borrower*

A loan is associated with at most one customer via *borrower*

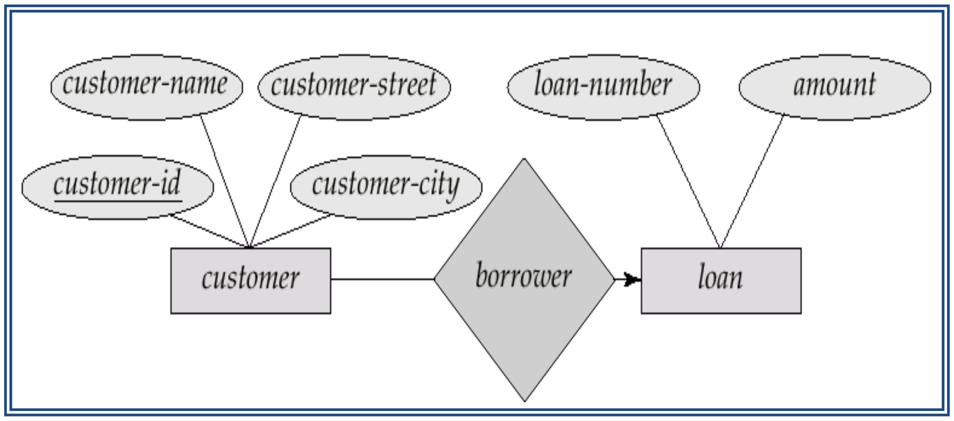
One-To-Many Relationship

In the one-to-many relationship a loan is associated with at most one customer via *borrower*, a customer is associated with several (including 0) loans via *borrower*



Many-To-One Relationships

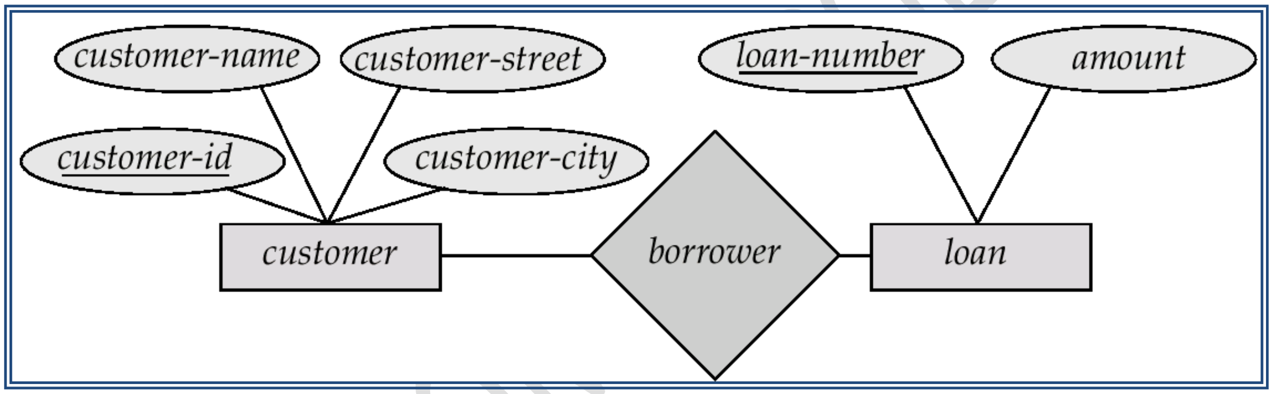
In a many-to-one relationship a loan is associated with several (including 0) customers via *borrower*, a customer is associated with at most one loan via *borrower*



Many-To-Many Relationship

A customer is associated with several (possibly 0) loans via borrower

A loan is associated with several (possibly 0) customers via borrower



Design Issues

1. Use of entity sets vs. attributes
2. Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question.
3. Use of entity sets vs. relationship sets
   Possible guideline is to designate a relationship set to describe an action that occurs between entities
4. Binary versus n-ary relationship sets
5. Although it is possible to replace any nonbinary (*n*-ary, for *n* > 2) relationship set by a number of distinct binary relationship sets, a *n*-ary relationship set shows more clearly that several entities participate in a single relationship.
6. Placement of relationship attributes

Weak Entity Sets

→An entity set that does not have a primary key is referred to as a weak entity set.

→The existence of a weak entity set depends on the existence of a identifying entity set

→ it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set

→Identifying relationship depicted using a double diamond

→The discriminator (*or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.

→The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

Unit5

Relational Database Design

Features of good relational designs

branch = (*branch_name*, *branch_city*, *assets*)
customer = (*customer_id*, *customer_name*, *customer_street*, *customer_city*)
loan = (*loan_number*, *amount*)
account = (*account_number*, *balance*)
employee = (*employee_id*. *employee_name*, *telephone_number*, *start_date*)
dependent_name = (*employee_id*, *dname*)
account_branch = (*account_number*, *branch_name*)
loan_branch = (*loan_number*, *branch_name*)
borrower = (*customer_id*, *loan_number*)
depositor = (*customer_id*, *account_number*)
cust_banker = (*customer_id*, *employee_id*, *type*)
works_for = (*worker_employee_id*, *manager_employee_id*)
payment = (*loan_number*, *payment_number*, *payment_date*, *payment_amount*)
savings_account = (*account_number*, *interest_rate*)
checking_account = (*account_number*, *overdraft_amount*)

Atomic domains and First Normal Form

Domain is atomic if its elements are considered to be indivisible units

l Examples of non-atomic domains:
- Set of names, composite attributes
- Identification numbers like CS101 that can be broken up into parts

A relational schema R is in first normal form if the domains of all attributes of R are atomic

Non-atomic values complicate storage and encourage redundant (repeated) storage of data

1. Example: Set of accounts stored with each customer, and set of owners stored with each account
2. We assume all relations are in first normal form (and revisit this in Chapter 9)

Atomicity is actually a property of how the elements of the domain are used.

l Example: Strings would normally be considered indivisible
2 Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
3 If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
4 Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

Decomposition Functional Dependencies

1. Constraints on the set of legal relations.
2. Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
3. A functional dependency is a generalization of the notion of a *key*.

Let R be a relation schema

$\alpha \subseteq R$ *and* $\beta \subseteq R$

The functional dependency

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations r(R), whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

Example: Consider r(A,B) with the following instance of r.

On this instance, $A \rightarrow B$ does NOT hold, but $B \rightarrow A$ does hold.

Use of

Functional Dependencies theory

We use functional dependencies to:

l test relations to see if they are legal under a given set of functional dependencies.

- If a relation r is legal under a set F of functional dependencies, we say that r satisfies F.

2 specify constraints on the set of legal relations

- We say that F holds on R if all legal relations on R satisfy the set of functional dependencies F.

Using Multivalued Dependencies (MVDs)

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The *multivalued dependency*

$$\alpha \rightarrow\rightarrow \beta$$

holds on R if in any legal relation *r(R)*, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

Use of Multivalued Dependencies

→We use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies

2. To specify constraints on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.

→If a relation *r* fails to satisfy a given multivalued dependency, we can construct a relations *r′* that does satisfy the multivalued dependency by adding tuples to *r*.

Database Design Process

→We have assumed schema *R* is given

1. *R* could have been generated when converting E-R diagram to a set of tables. *R* could have been a single relation containing *all* attributes that are of interest (called universal relation).

2. Normalization breaks *R* into smaller relations.
R could have been the result of some ad hoc design of relations, which we then test/convert to normal form.