CSC 214: INTRODUCTION TO DATA MANAGEMENT
Introduction to DBMS and Relational Theory

Course

Course code: CSC 214
Course title: Data Management I
Credit unit: 2
Course status: Compulsory

INTRODUCTION TO THE COURSE

COURSE DESCRIPTION
CSC 214 is a comprehensive introduction to theories and practices in Database Design and Management. Database Management I concentrates on principles, design, implementation and application of Database Management Systems. Students are introduced to the fundamental theories, concepts and techniques needed to properly understand and implement the relational database model which is the bedrock of today's mainstream database products.

COURSE JUSTIFICATION
An in-depth understanding of the principles and application of database systems is a critical success factor for information professionals taking leadership roles in future information systems initiatives. This course offers students the opportunity of rigorous study of the traditional principles of database design, implementation and usage.

COURSE OBJECTIVES
Upon successful completion, students should be able to:

- Demonstrate good knowledge of basic database concepts, including the structure and operation of the relational data model.
- Understand and successfully apply logical database design principles, including E-R diagrams and database normalization.
- Assess the quality and ease of use of data modelling and diagramming tools.
- Design and implement a small database project.
- Describe and discuss selected advanced database topics, such as distributed database systems and the data warehouse.

COURSE CONTENTS
Information storage & retrieval, information management applications. Information capture and representation, analysis & indexing, search, retrieval, information privacy; integrity, security; scalability, efficiency and effectiveness.
Introduction to database system:
Components of database systems DBMS functions, Database architecture and data independence. Data modelling, Entity-relationship model, Database, design using entity-relationship and semantic object models, Relational data model, process of database design.

COURSE REQUIREMENT
There are no formal prerequisites for this course.

METHOD OF GRADING
Method of grading

S/N	GRADING	SCORE(%)
1.	Continuous Assessments • C.AI	7%

	• C.AII (Mid-Semester Test) • C.AIII	15% 8%
2.	Assignment	
3.	Practical (Laboratory work)/ Case Studies	
4.	Final Examination	70%
5.	Total	100

Course Delivery Strategies:
Lectures are delivered via electronic media (e-learning platform and power point presentations) and other available multimedia resources. Students are also encouraged to work with our programmers and avail themselves of laboratory facilities for practical work. Students are expected to demonstrate their understanding of concepts by completing given tasks in class and submitting assignments as at when due.

Resources used/Reading Material:

Books

- Database Management System (2Ed) by Raghu Ramakrishtan and Johannes Gehrke
- Database Systems: Design, Implementation, and Management (10Ed) by Carlos Coronel, Steven Morris, and Peter Rob (2012). Cengage Learning, Boston. ISBN-13: 978-1-111-96960-8
- Database principles and design (3Ed) by Colin Ritchie (2008). Cengage Learning, London. ISBN-13: 978-1-84480-540-2.
- Database System, the complete book (2Ed) by Hector G. M., Jeffrey D. U., Jennifer W. (2009). Pearson Education Inc. New Jersey. ISBN 0-13-606701-8
- Relational Theory for Computer Professionals by C. J. Date (2013). O'Reilly Media, Inc. Sebastopol. ISBN: 978-1-449-36943-9

Online resources

- Database Management Systems Relational, Object-Relational and Object-Oriented Data Models. Center for Objekt Teknology. Available online: http://www.cit.dk/COT/reports/reports/Case4/05-v1.1/cot-4-05-1.1.pdf
- http://www.help2engg.com/dbms/dbms-languages
- Database Management System by tutorialpoint. Available online https://www.tutorialspoint.com/dbms/dbms_tutorial.pdf

Data: raw representation of unprocessed facts, figures, concepts or instruction. It can exist in any form, usable or not. Data are facts presented without relation to other things. E.g. It is raining

Information: information is data that has been given meaning by way of relational connection. This "meaning" can be useful, but does not have to be. In computer parlance, a relational database makes information from the data stored within it. Information embodies the understanding of some sort. E.g. the temperature dropped to 15 degrees and then it started raining.

DATABASE
A Database is a shared, integrated computer structure that is repository to:
-End-user data, that is, raw facts of interest to the end user
-Metadata, or data about data describes of the data characteristics and the set of relationships that link the data found within the db.

A database is a collection of data, typically describing the activities of one or more related organizations. For example, a university database might contain information about the following:

- Entities such as students, faculty, courses, and classrooms.
- Relationships between entities, such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses.

Proper storage of data in a database will enhance efficient

- Data Management
- Data processing
- Data retrieval

Database System

Refers to an organization of components that define and regulate the collection, storage, management from general management point of view, the DB system is composed of

- Hardware
- Software
- People –system administrators: database systems operations
    - DB administrators: manage the DBMS and ensure the DB is
    - functioning properly
    - DB designers
    - System analysts and programmers design and implement the application programs
    - end user
- Procedures
- Data

DBMS: A database management system (DBMS) is a collection of programs that manages the database structure and controls access to the data stored in the database. In a sense, a database resembles a very well-organized electronic filing cabinet in which powerful software (the DBMS) helps manage the cabinet's contents

ADVANTAGES OF A DBMS

- **Data independence:** This is the technique that allow data to be changed without affecting the applications that process it. We can change the way the database is physically stored and accessed without having to make corresponding changes to the way the database is perceived by the user. Changing the way the database is physically stored and accessed is almost always to improve performance; and the fact that we can make such changes without having to change the way the database looks to the user means that existing application programs, queries, and the like can all still work after the change. Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.
- **Efficient data access:** A DBMS deploys sophisticated techniques to store and retrieve data efficiently.
- **Data integrity control:** the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, updating the status for supplier S1 to 200 will rejected, if status values are supposed never to exceed 100
- **Security Control:** the DBMS can enforce access controls that govern what data is visible to different classes of users. Users are only allowed to perform an operation he or she is allowed to carry out on data.
- **Concurrent access and crash recovery:** A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.
- **Reduced application development time:** Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because

many important tasks are handled by the DBMS instead of being implemented by the application.

DBMS Architecture

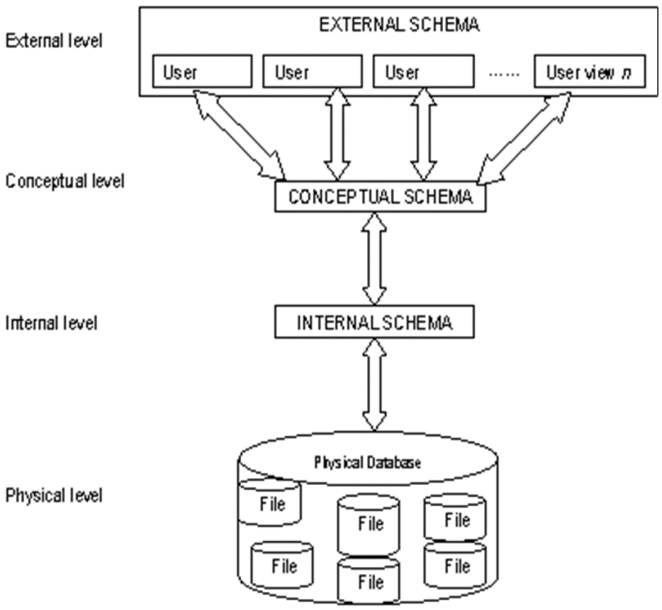
The DBMS provides users with an abstract view of the data in it i.e. the system hides certain details of how the data is stored and maintained from users. A DBMS can be viewed as divided into levels of abstraction. A common architecture generally used is the ANSI/SPARC (American National Standards Institute - Standards Planning and Requirements Committee) model.

The ANSI/SPARC model abstracts the DBMS into a 3-tier architecture as follows:

External level

Conceptual level

Internal level

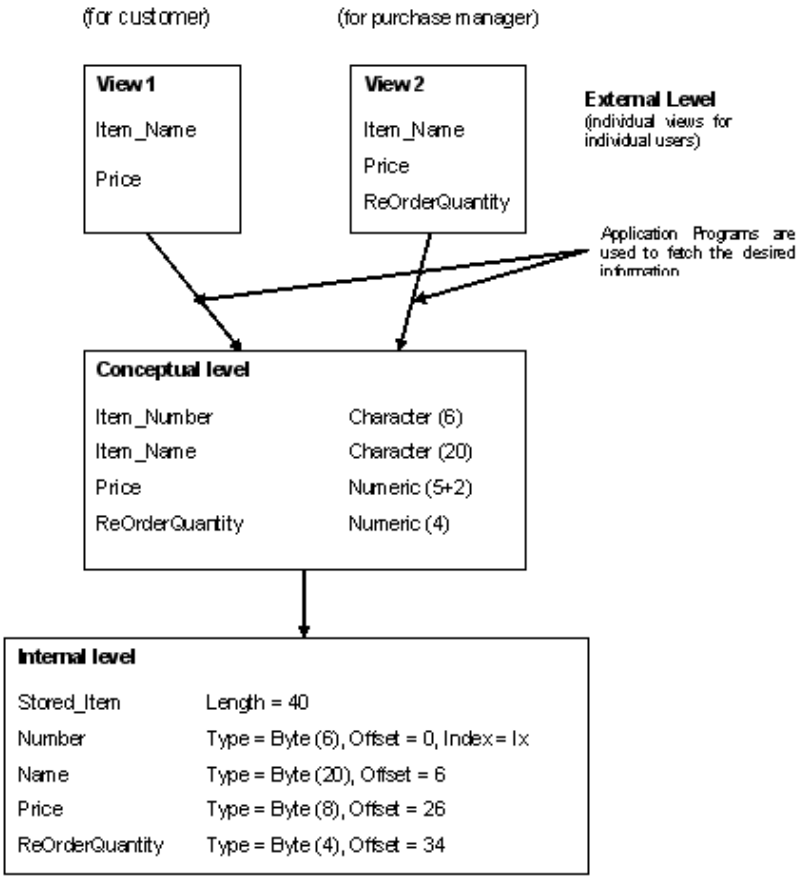


ANSI/SPARC 3-tier DBMS architecture

i. **External level:** The external level is the user's view of the database and closest to the users. It presents only the relevant part of the DBMS to the user. E.g. A bank database stores a lot more information but an account holder is only interested in his/her account details such as the current account balance, transaction history etc. An external schema describes each external view. The external schema consists of the definition of the logical records and the relationships in the external view. In the external level, the different views may have different representations of the same data.

ii. **Conceptual level:** At this level of database abstraction, all the database entities and relationships among them are included. Conceptual level provides the community view of the database and describes what data is stored in the database and the relationships among the data. In other words, the conceptual view represents the entire database of an organization. It is a complete view of the data requirements of the organization that is independent of any storage consideration. The conceptual schema defines conceptual view. It is also called the logical schema. There is only one conceptual schema per database. The figure shows the conceptual view record of a data base.

iii. **Internal level or physical level:** The lowest level of abstraction is the internal level. It is the one closest to physical storage device. This level is also termed as physical level, because it describes how data are actually stored on the storage medium such as hard disk, magnetic tape etc. This level indicates how the data will be stored in the database and describe the data structures, file structures and access methods to be used by the database. The internal schema defines the internal level. The internal schema contains the definition of the stored record, the methods of representing the data fields and accessed methods used. The figure shows the internal view record of a database.



DBMS LANGUAGES

THE WORKINGS OF A DBMS IS CONTROLLED BY FOUR DIFFERENT LANGUAGES, THEY ARE

- **DATA DEFINITION LANGUAGE (DDL):** USED BY THE DBA AND DATABASE DESIGNERS TO SPECIFY THE CONCEPTUAL SCHEMA OF A DATABASE. IN MANY DBMSS, THE DDL IS ALSO USED TO DEFINE INTERNAL AND EXTERNAL SCHEMAS (VIEWS). IN SOME DBMSS, SEPARATE STORAGE DEFINITION LANGUAGE (SDL) AND VIEW DEFINITION LANGUAGE (VDL) ARE USED TO DEFINE INTERNAL AND EXTERNAL SCHEMAS. SDL IS TYPICALLY REALIZED VIA DBMS COMMANDS PROVIDED TO THE DBA AND DATABASE DESIGNERS. SOME EXAMPLES INCLUDE:
  - **CREATE** - TO CREATE OBJECTS IN THE DATABASE
  - **ALTER** - ALTERS THE STRUCTURE OF THE DATABASE
  - **DROP** - DELETE OBJECTS FROM THE DATABASE
  - **TRUNCATE** - REMOVE ALL RECORDS FROM A TABLE, INCLUDING ALL SPACES ALLOCATED FOR THE RECORDS ARE REMOVED
  - **COMMENT** - ADD COMMENTS TO THE DATA DICTIONARY
  - **RENAME** - RENAME AN OBJECT

- **DATA MANIPULATION LANGUAGE (DML):** THESE STATEMENTS MANAGING DATA WITHIN SCHEMA OBJECTS. THEY SPECIFY DATABASE RETRIEVALS AND UPDATES. DML COMMANDS (DATA SUBLANGUAGE) CAN BE EMBEDDED IN A GENERAL-PURPOSE PROGRAMMING LANGUAGE (HOST LANGUAGE), SUCH AS COBOL, C, C++, OR JAVA.
  - A LIBRARY OF FUNCTIONS CAN ALSO BE PROVIDED TO ACCESS THE DBMS FROM A
  - PROGRAMMING LANGUAGE
  - ALTERNATIVELY, STAND-ALONE DML COMMANDS CAN BE APPLIED DIRECTLY (CALLED A QUERY
  - LANGUAGE).

  SOME EXAMPLES IN SQL INCLUDE:
  - **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

- **Data Control Language (DCL):** used for granting and revoking user access on a database
  - To grant access to user – GRANT
  - To revoke access from user – REVOKE

- **Transaction Control (TCL):** Statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

  Some examples include:
  - COMMIT - save work done
  - SAVEPOINT - identify a point in a transaction to which you can later roll back
  - ROLLBACK - restore database to original since the last COMMIT
  - SET TRANSACTION - Change transaction options like isolation level and what rollback segment to useIn practical data definition language, data manipulation language and data control languages are not separate language; rather they are the parts of a single database language such as SQL.

Example

Write the SQL code that will create the table structure for a table named EMP_1. This table is a subset of the EMPLOYEE table. The basic EMP_1 table structure is summarized in the following table. EMP_NUM is the primary key and JOB_CODE is the FK to JOB.

Hint: Primary Key cannot contain null value

ATTRIBUTE (FIELD) NAME	DATA DECLARATION
EMP_NUM	CHAR(3)
EMP_LNAME	VARCHAR(15)
EMP_FNAME	VARCHAR(15)
EMP_INITIAL	CHAR(1)
EMP_HIREDATE	DATE
JOB_CODE	CHAR(3)

```
CREATE TABLE EMP_1(
EMP_NUM CHAR(6) NOT NULL,
EMP_LNAME VARCHAR(15),
EMP_FNAME VARCHAR(15),
EMP_INITIAL CHAR(1),
EMP_HIREDATE DATE,
JOB_CODE CHAR(3),
PRIMARY KEY (EMP_NUM),
FOREIGN KEY(JOB_CODE) REFERENCES JOB (JOB_CODE)
) ;
```

Having created the table structure in (a), write the SQL code to enter the first two rows for the table EMP_1 below:

EMP_NUM	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_HIREDATE	JOB_CODE
101	News	John	G	08-Nov-00	502
102	Senior	David	H	12-Jul-89	501
103	Arbough	June	E	01-Dec-96	500
104	Ramoras	Anne	K	15-Nov-87	501
105	Johnson	Alice	K	01-Feb-93	502
106	Smithfield	William		22-Jun-04	500
107	Alonzo	Maria	D	10-Oct-93	500
108	Washington	Ralph	B	22-Aug-91	501
109	Smith	Larry	W	18-Jul-97	501

```
INSERT INTO EMP_1
(EMP_NUM, EMP_LNAME, EMP_FNAME, EMP_INITIAL, EMP_HIREDATE, JOB_CODE)
VALUES
("101", "News", "John", "G", "08-Nov-00", "502"),
("102", "Senior", "David", "H", "12-Jul-89", "500");
```

Assuming the data shown in the EMP_1 table have been entered, write the SQL code that will list all attributes for a job code of 502.

```
SELECT * FROM EMP_1
WHERE JOB_CODE = '502';
```

Write the SQL code that will save the changes made to the EMP_1 table.

```
COMMIT WORK;
```
NB: WORK is optional.

DBMS Data Model

A data model is a notation for describing data or information. The description generally consists of three parts:

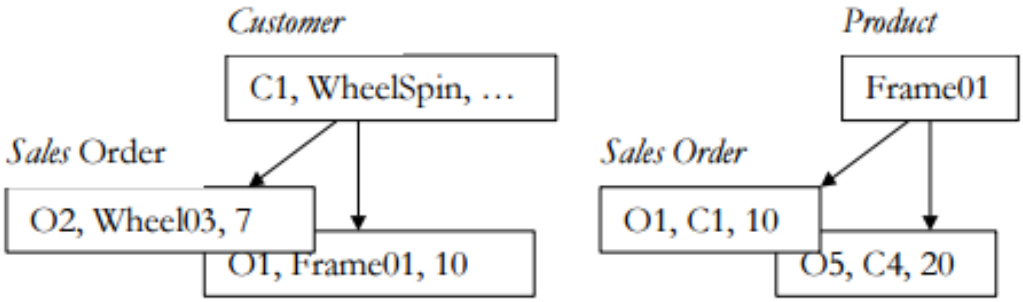
i. Structure of the data: The data structures used to implement data in the computer are sometimes referred to, in discussions of database systems, as a physical data model.
ii. Operations on the data: In database data models, there is usually a limited set of operations that can be performed but DBA can describe database operations at a very high level, yet have the database management system implement the operations efficiently.
iii. Constraints on the data. Database data models usually have a way to describe limitations on what the data can be. These constraints can range from the simple (e.g., "a day of the week is an integer between 1 and 7" or "a movie has at most one title") to some very complex limitations.

Traditionally, there are four DBMS. These four data models also represent the historical developments of the DBMS:

Hierarchical Database Model

This is the oldest DBMS data model. In this model, information is organized as a collection of inverted trees of records. The record at the root of a tree has zero or more child records; the child records, in turn, serve as parent records for their immediate descendants. This parent-child relationship recursively continues down the tree. The records consist of fields, where each field may contain simple data values (e.g. integer, real, text)., or a pointer to a record. The pointer graph is not allowed to contain cycles. Some combinations of fields may form the key for a record relative to its parent. Only a few hierarchical DBMSs support null values or

variable-length fields.



Example of Hierarchical data model

Applications can navigate a hierarchical database by starting at a root and successively navigate downward from parent to children until the desired record is found.

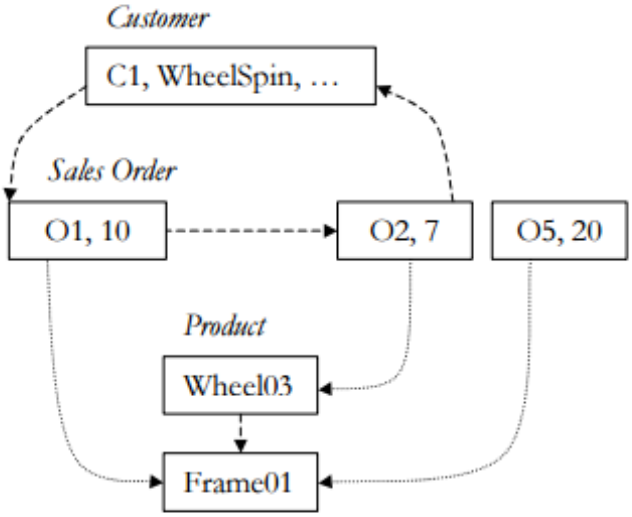
Searching down a hierarchical tree is very fast since the storage layer for hierarchical databases use contiguous storage for hierarchical structures. All other types of queries require sequential search techniques. A DDL for hierarchical data model must allow the definition of record types, fields types, pointers, and parent-child relationships. And the DML must support direct navigation using the parent-child relationships and through pointers.

Limitations

- Hierarchical model only permits one to many relationship. The concept of Logical relationship is often used to circumvent this limitation. Logical relationship superimpose another set of connection between data items separate from the physical tree structure. This of course increases its complexity
- Often a natural hierarchy does not exist and it is awkward to impose a parent-child relationship. Pointers partially compensate for this weakness, but it is still difficult to specify suitable hierarchical schemas for large models and this means Programs have to navigate very close to the physical data structure level, implying that the hierarchical data model offers only very limited data independence.
- Lack of ad hoc query capability placed burden on programmers to generate code for reports

Network model:

It represents complex data relationships more effectively than the hierarchical model. The major improvement is that the one-to-many limitation was removed; the models still views data in a hierarchical one-to-many structure but now record may have more than one parent. Network data models represent data in a symmetric manner, unlike the hierarchical data model (distinction between a parent and a child). Information is organized as a collection of graphs of record that are related with pointers. More flexible than a hierarchical data model and still permits efficient navigation.



Example of network data model

The records consist of lists of fields (fixed or variable length with maximum length), where each field contains a simple value (fixed or variable size). The network data model also introduces the notion of indexes of fields and records, sets of pointers, and physical placement of records. A DDL for network data models must allow the definition of record types, fields types, pointers and indexes. And the DML must allow navigation through the graphs through the pointers and indexes. Programs also navigates closely to the physical
storage structures, implying that the network data model only supports limited data independence, and are therefore difficult to maintain as the data models evolve over time.

Concepts introduced under the network model include:

- Schema: conceptual design of the entire database usually managed by the DBA
- Sub-schema: virtual view of portions of the database visible to application programmers
- Data management language: enables definition of and access to the schema and sub-schema. It consist of DDL to construct the schema and DML to develop programs
- Data Definition Language

Limitations

- Cumbersome
- Lack of ad hoc query capability placed burden on programmers to generate code for reports
- Structural change in the database could produce havoc in all application programs

The relational database Model

Developed by E.F. Codd (IBM) in 1970, the relational data model has a mathematical foundation in relational algebra. The model is based on first-order predicate logic and defines a table as an n-ary relation. Data is organized in relations (two-dimensional tables). Each relation contains a set of tuples (records). Each tuple contains a number of fields. A field may contain a simple value (fixed or variable size) from some domain (e.g. integer, real, text, etc.).

Advantages of relational model

- Built-in multilevel integrity: Data integrity is built into the model at the field level to ensure the accuracy of the data; at the table level to ensure that records are not duplicated and to detect missing primary key values; at the relationship level to ensure that the relationship between a pair of tables is valid; and at the business level to ensure that the data is accurate in terms of the business itself. (Integrity is discussed in detail as the design process unfolds.)

- Logical and physical data independence from database applications: Neither changes a user makes to the logical design of the database, nor changes a database software vendor makes to the physical implementation of the database, will adversely affect the applications built upon it.

- Guaranteed data consistency and accuracy: Data is consistent and accurate due to the various levels of integrity you can impose within the database. (This will become quite clear as you work through the design process.)

- Easy data retrieval: At the user's command, data can be retrieved either from a particular table or from any number of related tables within the database. This enables a user to view information in an almost unlimited number of ways.

One commonly perceived disadvantage of the relational database was that software programs based on it ran very slowly.

Some definitions

RELATION: a relation, as defined by E. F. Codd, is a set of tuples (d1, d2, ..., dn), where each element dj is a member of Dj, a data domain, for each j=1, 2, ..., n. A **data domain** is simply a data type. It specifies a data abstraction: the possible values for the data and the operations available on the data. For example, a String can have zero or more characters in it, and has operations for comparing strings, concatenating string, and creating strings. A relation is a truth predicate. It defines what attributes are involved in the predicate and what the meaning of the predicate is. In relational data model, relations are represented in the table format. This format stores the relation among entities. A table has rows and columns, where rows represent records and columns represent the attributes. E.g.

title	*year*	*length*	*genre*
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

TUPLE: A single row of a table, which contains a single record for that relation is called a tuple. A tuple has attribute values which match the required attributes in the relation. The ordering of attribute values is immaterial. Every tuple in the body of a given relation is required to conform to the heading (attribute) of that relation, i.e. it contains exactly one value, of the applicable type, for each attribute, and nothing else besides

ATTRIBUTE: The columns of a relation are named by attributes. Attributes appear at the tops of the columns. Usually, an attribute describes the meaning of entries in the column below. For instance, the column with attribute length holds the length, in minutes, of each movie.

ATTRIBUTE DOMAIN: Every attribute has some predefined value scope, known as attribute domain

ATTRIBUTE VALUE/INSTANCE: An attribute value is the value for an attribute in a particular tuple. An attribute value must come from the domain that the attribute specifies. Most relational **DBMS** allows **NULL** attribute values. Each attribute value in a relational model must be atomic i.e. it must be of some elementary type such as integer or string. It is not permitted for a value to be a record structure, set, list, array, or any other type that reasonably can have its values broken into smaller components.

SCHEMAS: The name of a relation and the set of attributes for a relation is called the schema for that relation. The schema is depicted by the relation name followed

by a parenthesized list of its attributes. Thus, the schema for relation Movies above is

Movies (title , year, length, genre)

In the relational model, a database consists of one or more relations. The set of schemas for the relations of a database is called a relational database schema, or just a database schema.

Data Types: All attributes must have a data type. The following are the primitive data types that are supported by SQL (Structured Query Language) systems.

i. Character strings of fixed or varying length. The type CHAR(n) denotes a fixed-length string of up to n characters. VARCHAR(n) also denotes a string of up to n characters. The difference is implementation-dependent; typically CHAR implies that short strings are padded to make n characters, while VARCHAR implies that an endmarker or string-length is used. Normally, a string is padded by trailing blanks if it becomes the value of a component that is a fixed-length string of greater length. For example, the string 'foo' if it became the value of a component for an attribute of type CHAR(5), would assume the value 'foo ' (with two blanks following the second o).

ii. Bit strings of fixed or varying length. These strings are analogous to fixed and varying-length character strings, but their values are strings of bits rather than characters. The type BIT (n) denotes bit strings of length n, while BIT VARYING (n) denotes bit strings of length up to n.

iii. The type BOOLEAN denotes an attribute whose value is logical. The possible values of such an attribute are TRUE, FALSE.

iv. The type INT or INTEGER (these names are synonyms) denotes typical integer values. The type SHORTINT also denotes integers, but the number of bits permitted may be less, depending on the implementation (as with the types int and short int in C).

v. Floating-point numbers can be represented in a variety of ways. We may use the type FLOAT or REAL (these are synonyms) for typical floating point numbers. A higher precision can be obtained with the type DOUBLE PRECISION. We can also specify real numbers with a fixed decimal point. For example, DECIMAL(n,d) allows values that consist of n decimal digits, with the decimal point assumed to be d positions from the right. Thus, 0123.45 is a possible value of type DECIMAL(6,2). NUMERIC is almost a synonym for DECIMAL, although there are possible implementation-dependent differences.

vi. Dates and times can be represented by the data types DATE and TIME, respectively. These values are essentially character strings of a special form. We may, in fact, coerce dates and times to string types, and we may do the reverse if the string "makes sense" as a date or time.

A relational database Schema is depicted by stating both the attributes and their datatype:

```
Movies (
title CHAR(100),
year INT,
length INT,
genre CHAR(10),
studioName CHAR(30),
producer INT
)
```

Relation instance: A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

```
{
        <Person SSN# = "123-45-6789" Name = "Art Larsson" City = "San Francisco">,
        <Person SSN# = "231-45-6789" Name = "Lino Buchanan" City = "Philadelphia">,
        <Person SSN# = "321-45-6789" Name = "Diego Jablonski" City = "Chicago">
}
```

It is more common and concise to show a relation value as a table. All ordering within the table is artificial and meaningless.

Design theory for Relational Database

A common problem with schema design involve trying to combine too much into one relation thus leading to redundancy. Thus, improvements to relational schemas pay close attention to eliminating redundancy. The theory of "dependences" is a well-developed theory for relational databases providing guidelines on how to develop good schema and eliminate flaws if any. The first concept we need to consider is Functional Dependency (FD).

Functional Dependency (FD)

FUNCTIONAL DEPENDENCY: the term functional dependence can be defined most easily this way:

Definition:

Let A and B be subsets of the attribute of a relation R. Then the functional dependency (FD)

A → B

holds in R if and only if, whenever two tuples of R have the same value for A, they also have the same value for B. A and B are the determinant and the dependent, respectively, and the FD overall can be read as "A functionally determines B" or "B is functionally dependent on A," or more simply just as

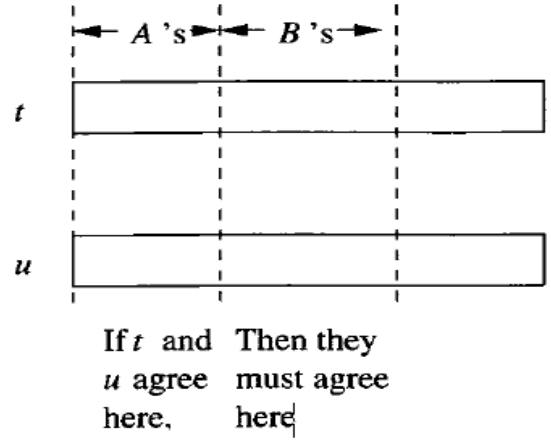
A → B

If A and B are composite, then we have

A1, A2, ..., An → B1, B2, ..., Bm

This is also equivalent to

A1, A2, ..., An → B1, A1, A2, ..., An → B2, ..., A1, A2, ..., An → Bm



The attribute(s) B is functionally dependent on attributes(s)A, if A determines B.

e.g. STU_PHONE is functionally dependent on STU_NUM.

STU_NUM is not functionally dependent on STU_PHONE because the STU_PHONE value 2267 is associated with two STU_NUM values: 324274 and 324291. (This could happen when roommates share a single land line phone number.)

The functional dependence definition can be generalized to cover the case in which the determining attribute values occur more than once in a table.

Functional dependence can then be defined this way:

Attribute B is functionally dependent on A if all of the rows in the table that agree in value for attribute A also agree in value for attribute B.

RELATION KEYS: The key's role is based on a concept known as **determination**. I.e. the statement "A determines B" indicates that if you know the value of attribute A, you can look up (determine) the value of attribute B. E.g.:

an invoice number identifies all of the invoice attributes such as invoice date and the customer name.
if we know **STU_NUM** in a **STUDENT** table we can look up (determine) student's last name, grade point average, phone number, etc.

STU_NUM	STU_LNAME	STU_FNAME	STU_INIT	STU_DOB	STU_HRS	STU_CLASS	STU_GPA	STU_TRANSFER	DEPT_CODE	STU_PHONE	PROF_NUM
321452	Bowser	William	C	12-Feb-1975	42	So	2.84	No	BIOL	2134	205
324257	Smithson	Anne	K	15-Nov-1981	81	Jr	3.27	Yes	CIS	2256	222
324258	Brewer	Juliette		23-Aug-1969	36	So	2.26	Yes	ACCT	2256	228
324269	Oblonski	Walter	H	16-Sep-1976	66	Jr	3.09	No	CIS	2114	222
324273	Smith	John	D	30-Dec-1958	102	Sr	2.11	Yes	ENGL	2231	199
324274	Katinga	Raphael	P	21-Oct-1979	114	Sr	3.15	No	ACCT	2267	228
324291	Robertson	Gerald	T	08-Apr-1973	120	Sr	3.87	No	EDU	2267	311
324299	Smith	John	B	30-Nov-1986	15	Fr	2.92	No	ACCT	2315	230

Table name: Student

The shorthand notation for "A determines B" is

A → B.

If A determines B, C, and D, we write

A → B, C, D.

For the student example we can write:

STU_NUM → STU_LNAME, STU_FNAME, STU_INIT, STU_DOB, STU_TRANSFER

In contrast, **STU_NUM** is not determined by **STU_LNAME** because it is quite possible for several students to have the last name Smith.
Proper understanding of the principle of determination is vital to the understanding of a central relational database concept known as **functional dependence (FD)**.

Definitions
Key Attribute(s): We say a set of one or more attributes {A1, A2, ..., An} is a **key for a relation** R if:

i. Those attributes functionally determine all other attributes of the relation. That is, it is impossible for two distinct tuples of R to agree on all of A1, A2, ..., An (uniqueness).
ii. No proper subset of {A1, A2, ..., An} functionally determines all other attributes of R; i.e., a key must be minimal.

When a key consists of a single attribute A, we often say that A (rather than {A}) is a key. An attribute that is part of a key is called **key attribute.**
Consider the Relation Movies below:

title	*year*	*length*	*genre*	*studioName*	*starName*
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Attributes {title, year, starName} form a key for the relation Movies because it meets the two conditions:
Condition 1:

Do they functionally determine all the other attributes? Yes

Condition 2:

Do any proper subset of {title, year, starName} functionally determines all other attributes?

{title, year} do not determine starName thus {title, year} is not a key.
{year, starName} is not a key because we could have a star in two movies in the same year; therefore

{Year, starName} → title is not an FD.

{title, starName} is not a key, because two movies with the same title, made in different years, can have a star in common.

Therefore, no proper subset of {title, year, starName} functionally determines all other attributes

Super Key (shortened: super set of keys): An attribute or a combination of attributes that is used to identify the records uniquely is known as Super Key. It is to be noted that some superkeys are not (minimal) keys. Note that every superkey satisfies the first condition of a key: it functionally determines all other attributes of the relation. However, a superkey need not satisfy the second condition: minimality. A table can have many Super Keys. E.g. of Super Key

- ID
- ID, Name
- ID, Address
- ID, Department_ID
- ID, Salary
- Name, Address

Candidate Key: It can be defined as minimal Super Key or irreducible Super Key. In other words an attribute or a combination of attribute that identifies the record uniquely but none of its proper subsets can identify the records uniquely. E.g. of Candidate Key

Code

Name, Address

Primary Key: A Candidate Key that is used by the database designer for unique identification of each row in a table is known as Primary Key. A Primary Key can consist of one or more attributes of a table. E.g. of Primary Key - Database designer can use one of the Candidate Key as a Primary Key.

In this case we have "Code" and "Name, Address" as Candidate Key,

The designer may prefer "Code" as the Primary Key as the other key is the combination of more than one attribute.

Null values should never be part of a primary key, they should also be avoided to the greatest extent possible in other attributes too. A null is no value at all. It does not mean a zero or a space. There are rare cases in which nulls cannot be reasonably avoided when you are working with non-key attributes. For example, one of an EMPLOYEE table's attributes is likely to be the EMP_INITIAL. However, some employees do not have a middle initial. Therefore, some of the EMP_INITIAL values may be null. Null can also exist because of the nature of the relationship between two entities. Conventionally, the existence of nulls in a table is often an indication of poor database design. Nulls, if used improperly, can create problems because they have many different meanings. For example, a null can represent:

An unknown attribute value.

A known, but missing, attribute value.

A "not applicable" condition.

Foreign Key: A foreign key is an attribute or combination of attributes in one base table that points to the candidate key (generally it is the primary key) of another table. The purpose of the foreign key is to ensure referential integrity of the data i.e. only values that are supposed to appear in the database are permitted. E.g.

Consider two table

Employee (EmployeeID, EmployeeName, DOB, DOJ, SSN, DeptID, MgrID) and

DeptTbl (Dept_ID, Dept_Name, Manager_ID, Location_ID)

Dept_ID is the primary key in Table DeptTbl, the DeptID attribute of table Employee (dependent or child table) can be defined as the Foreign Key as it can reference to the Dept_ID attribute of the table DeptTbl (the referenced or parent table), a Foreign Key value must match an existing value in the parent table or be NULL.

Composite Key: If we use multiple attributes to create a Primary Key then that Primary Key is called Composite Key (also called a Compound Key or Concatenated Key).

Full functional dependency (FFD): If the attribute (B) is functionally dependent on a composite key (A) but not on any subset of that composite key, the attribute (B) is fully functionally dependent on (A).

Alternate Key: Alternate Key can be any of the Candidate Keys except for the Primary Key.

Secondary Key: The attributes that are not even the Super Key but can be still used for identification of records (not unique) are known as Secondary Key.
E.g. of Secondary Key can be Name, Address, Salary, Department_ID etc. as they can identify the records but they might not be unique.

Table name: PRODUCT
Primary key: PRODUCT
Foreign key: VEND_CODE

Database name: Ch03_SaleCo

PROD_CODE	PROD_DESCRIPT	PROD_PRICE	PROD_ON_HAND	VEND_CODE
001278-AB	Claw hammer	12.95	23	232
123-21UUY	Houselite chain saw, 16-in. bar	189.99	4	235
QER-34256	Sledge hammer, 16-lb. head	18.63	6	231
SRE-657UG	Rat-tail file	2.99	15	232
ZZX/3245Q	Steel tape, 12-ft. length	6.79	8	235

link

Table name: VENDOR
Primary key: VEND_CODE
Foreign key: none

VEND_CODE	VEND_CONTACT	VEND_AREACODE	VEND_PHONE
230	Shelly K. Smithson	608	555-1234
231	James Johnson	615	123-4536
232	Annelise Crystall	608	224-2134
233	Candice Wallace	904	342-6567
234	Arthur Jones	615	123-3324
235	Henry Ortozo	615	899-3425

An Example of relational DB with primary key and foreign key

Exercise
Suppose R is a relation with attributes A1, A2, ..., An. As a function of n, tell how many superkeys R has, if:
a) The only key is A1.
b) The only keys are A1 and A2
c) The only keys are {A1, A2} and {A3, A4}
d) The only keys are {A1, A2} and {A1, A3}

Rules About Functional Dependencies

These rules guide us on how we can infer a functional dependency from other given FD's.
E.g., given that a relation R (A, B, C) satisfies the FD's
A —> B and B —> C,
then we can deduce that R also satisfies the FD
A —> C.

Proof:
Consider two tuples of R that agree on A

Let the tuples agreeing on attribute A be (a, b1, c1) and (a, b2, c2)
Since R satisfies A → B, and these tuples agree on A, they must also agree on B. That is, b1 = b2

The tuples are now (a, b, c1) and (a, b, c2), where b is both b1 and b2.
Similarly, since R satisfies B → C, and the tuples agree on B, they agree also on C.
Thus, c1= c2; i.e., the tuples do agree on C.

We have proved that any two tuples of R that agree on A also agree on C, and that is the FD
A → C.
This rule is called the transitive rule

The Splitting/Combining Rule
Recall that the FD:
A1, A2, …, An → B1, B2, …, Bm
is equivalent to the set of FD's:
A1, A2, …, An → B1, A1, A2, …, An → B2, …, A1, A2, …, An → Bm

In other words, we may split attributes on the right side so that only one attribute appears on the right of each FD. Likewise, we can replace a collection of FD's having a common left side by a single FD with the same left side and all the right sides combined into one set of attributes. In either event, the new set of FD's is equivalent to the old. The equivalence noted above can be used in two ways.

- We can replace an FD
  A1, A2, …, An → B1, B2, …, Bm by a set of FD's
  A1, A2, …, An → Bi for i = 1, 2, …, m
  We call this transformation the splitting rule.
- We can replace a set of FD's
  A1, A2, …, An → Bi for i = 1, 2, …, m by the single FD
  A1, A2, …, An → B1, B2, …, Bm.
  We call this transformation the combining rule.

E.g. the set of FD's:
title year → length
title year → genre
title year → studioName
is equivalent to the single FD:
title year → length, genre, studioName

The splitting/ combining rule is stated as follows:
Suppose we have two tuples that agree in A1, A2, …, An. As a single FD, we would assert "then the tuples must agree in all of B1, B2, …, Bm." As individual FD's, we assert "then the tuples agree in B1, and they agree in B2, and, …, and they agree in Bm."

Trivial-dependency rule.

Trivial Functional Dependencies: If a functional dependency (FD) α → β holds in Relation R, then the term trivial is attached to the dependency if it is satisfied by all possible r(R)
i.e. α → β is trivial if β ⊆ α or β ∪ α = R
where β is a subset of α, then it is called a trivial FD.
e.g.
title, year → title
title → title
are both trivial FD

There is an intermediate situation in which some, but not all, of the attributes on the right side of an FD are also on the left. This FD is not trivial.

Non-trivial: If an FD X → Y holds, where Y is not a subset of X, then it is called a non-trivial FD.

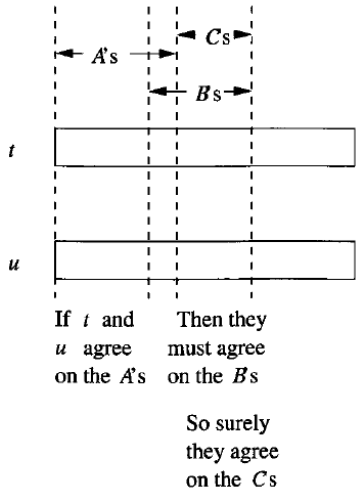
This can be simplified by removing from the right side of an FD those attributes that appear on the left. That is: The FD

A1, A2, ..., An → B1, B2, ..., Bm is equivalent to

A1, A2, ..., An → C1, C2, ..., Ck

where the C's are all those B's that are not also A's.

Completely non-trivial: If an FD X → Y holds, where x intersect Y = Φ, it is said to be a completely non-trivial FD.



Trivial dependency rule

Computing the Closure of Attributes

Given a set α = {A1, A2, ..., A n} of attributes of R and a set of functional dependencies FD, we need a way to find all of the attributes of R that are functionally determined by α. This set of attributes is called the closure of α under F and is denoted α^+. Finding α^+ is useful because:

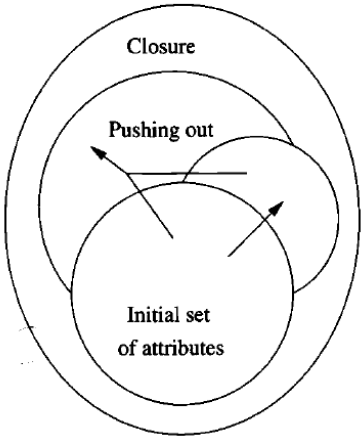
- if α^+ = R, then α is a superkey for R
- With closure we can find all FD's easily
  - To check if X → A
  - Compute X+
  - Check if A ∈ X
- if we find α^+ for all $\alpha \subseteq$ R, we've computed F^+ (except that we'd need to use decomposition to get all of it).

Formal definition of closure:

Suppose α = {A1, A2, ..., An} is a set of attributes and S is a set of FD's. The closure of α under the FD's in S is the set of attributes B such that every relation that satisfies all the FD's in set S also satisfies A1, A2, ..., An → B. That is, A1, A2, ..., An → B follows from the FD's of S.

We denote the closure of a set of attributes A1, A2, ..., An by
$\{A1, A2, ..., An\}^+$.

Note that A1, A2, ..., An are always in $\{A1, A2, ..., An\}^+$ because the FD A1, A2, ..., An → Ai is trivial when i is one of 1,2,... , n.



: Computing the closure of a set of attributes

THE FIGURE ABOVE ILLUSTRATES THE CLOSURE PROCESS:
STARTING WITH THE GIVEN SET OF ATTRIBUTES, WE REPEATEDLY EXPAND THE SET BY ADDING THE RIGHT SIDES OF FD'S AS SOON AS WE HAVE INCLUDED THEIR LEFT SIDES. EVENTUALLY, WE CANNOT EXPAND THE SET ANY FURTHER, AND THE RESULTING SET IS THE CLOSURE.
AN ALGORITHM FOR COMPUTING α^+:

```
result := α
repeat
        temp := result
        for each functional dependency  β → γ  in F do
        if  β ⊆ result then
                result := result ∪ γ
until temp = result
```

EXAMPLE:
CONSIDER A RELATION WITH ATTRIBUTES A, B, C, D, E, AND F. SUPPOSE THAT THIS RELATION HAS THE FD'S
AB → C, BC → AD, D → E, AND CF → B.
WHAT IS THE CLOSURE OF {A, B}?

SOLUTION
FIRST, SPLIT BC → AD INTO BC → A AND BC → D.

RESULT = {A, B}.
FOR AB → C
AB ⊆ RESULT, SO WE HAVE
RESULT = RESULT ∪ C I.E. RESULT = {A, B, C}.

FOR BC → C AND BC → D
BC ⊆ RESULT, SO WE HAVE
RESULT = RESULT ∪ A AND D I.E., RESULT = {A, B, C, D}

FOR D → E
D ⊆ RESULT, SO WE HAVE
RESULT = RESULT ∪ E I.E. RESULT = {A, B, C, D, E}

NO MORE CHANGES TO RESULT ARE POSSIBLE, THUS, $\{A, B\}^+ = \{A, B, C, D, E\}$.

BY COMPUTING THE CLOSURE OF ANY SET OF ATTRIBUTES, WE CAN TEST WHETHER ANY GIVEN FD A1, A2, ..., AN → B FOLLOWS FROM A SET OF FD'S S.
FIRST COMPUTE $\{A1, A2, ..., AN\}^+$ USING THE SET OF FD'S S. IF B IS IN $\{A1, A2, ..., AN\}^+$, THEN A1, A2, ..., AN → B DOES FOLLOW FROM S, AND IF B IS NOT IN $\{A1, A2, ..., AN\}^+$, THEN THIS FD DOES NOT FOLLOW FROM S.

More generally, A1, A2, ..., An → B1, B2, ..., Bm follows from set of FD's S if and only if all of B1, B2, ..., Bm are in $\{A1, A2, ..., An\}^+$

Example:
Consider the relation and FD's in the example above, Suppose we wish to test whether AB → D follows from these FD's. We compute $\{A, B\}^+$, which is {A, B, C, D, E}. Since D is a member of the closure, we conclude that AB → D does follow.
On the other hand, consider the FD
D → A. To test whether this FD follows from the given FD's, first compute $\{D\}^+$.
$\{D\}^+ = \{D, E\}$. Since A is not a member of {D, E}, we conclude that D → A does not follow.

Armstrong's Axioms

If F is a set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

- **Reflexivity / reflexive rule:** If $\{B1, B2, ..., Bm\} \subseteq \{A1, A2, ..., An\}$, then A1, A2, ..., An → B1, B2, ..., Bm. These are what we have called trivial FD's.
- **Augmentation rule:** If A1A2 ... An → B1B2 ... Bm, then
  A1A2 ... AnC1C2 ... Ck → B1B2, ... BmC1C2 ... Ck for any set of attributes C1, C2, ..., Ck
  Since some of the C 's may also be A's or B's or both, we should eliminate from the left side duplicate attributes and do the same for the right side.
- **Transitivity rule:** If A1, A2, ..., An → B1, B2, ..., Bm and B1, B2, ..., Bm → C1, C2, ..., Ck hold in relation R, then A1, A2, ..., An → C1, C2, ..., Ck also holds in R.

  If some of the C 's are among the A's, we may eliminate them from the right side by the trivial-dependencies rule

  To test whether A1, A2, ..., An → C1, C2, ..., Ck holds,
  we need to compute the closure
  $\{A1, A2, ..., An\}^+$ with respect to the two given FD's.
  The FD A1, A2, ..., An → B1, B2, ..., Bm tells us that all of B1, B2, ..., B m are in $\{A1, A2, ..., A n\}^+$.
  Then, we can use the FD B1, B2, ..., Bm → C1, C2, ..., Ck to add C1, C2, ..., Ck to $\{A1, A2, ..., An\}^+$.
  Since all the C's are in $\{A1, A2, ..., An\}^+$ we conclude that
  A1, A2, ..., An → C1, C2, ..., Ck holds for any relation that satisfies both
  A1, A2, ..., An → B1, B2, ..., Bm and B1, B2, ..., Bm → C1, C2, ..., Ck.

Additional rules:

- **Union:** If X → Y and X → Z, then X → Y Z
- **Pseudotransitivity:** If X → Y and W Y → Z, then W X → Z
- **Composition:** If X → Y and Z → W, then XZ → Y W

Transitive dependence: an attribute Y is said to be transitively dependent on attribute X if Y is functionally dependent on another attribute Z which is functionally dependent on X.

Closure of FD's set
Given Relation R and a set of FD's F that holds in R:
The closure of F in R (denoted F^+) is the set of all FD's F in R that are logically implied by F i.e. S the set of all regular FDs that can be derived from F

```
algorithm (F)
 /* F is a set of FDs */
F+ = ∅
for each possible attribute set X
      Compute the closure X+ of X on F
      for each attribute A ∈ X+
            add to F+ the FD: X → A
return F+
```

Example:

Assume there are 4 attributes A, B, C, D and that F = {A → B, B → C}. To compute F+ we first get:

$A^+ = AB^+ = AC^+ = ABC^+ = \{A, B, C\}$
$B^+ = BC^+ = \{B, C\}$
$C^+ = \{C\}$
$D^+ = \{D\}$
$AD^+ = \{A, D\}$
$BC^+ = \{B, C\}$
$BD^+ = BCD^+ = \{B, C, D\}$
$ABD^+ = ABCD^+ = \{A, B, C, D\}$
$ACD^+ = \{A, C, D\}$

Exercise

Consider a relation with schema R (A, B, C, D) and FD's AB → C, D → D and D → A.

i. What are all the nontrivial FD's that follow from the given FD's? You should restrict yourself to FD's with single attributes on the right side.
ii. What are all the keys of R?
iii. What are all the superkeys for R that are not keys?

Relational Set Operators

The Relational Data Model is designed in such ways that data may be processed with mathematical
operations. Data in relational tables are of little use unless they are manipulated to yield meaningful information. Relational Algebra forms the theoretical basis for manipulating table content using eight operators; four relational operators and four set operators. Relational operators take one or two relations as inputs and return relations as the result while set operators take one or two sets as inputs and return sets as the result.

Four relational operations

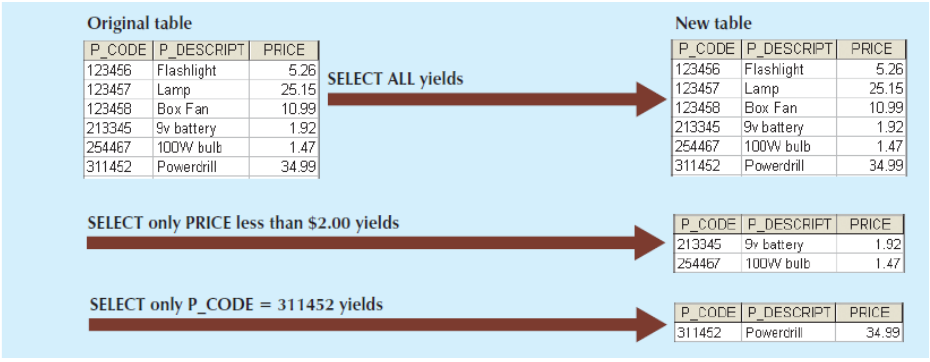
- Project
- Select
- Join
- Division

Four set operations

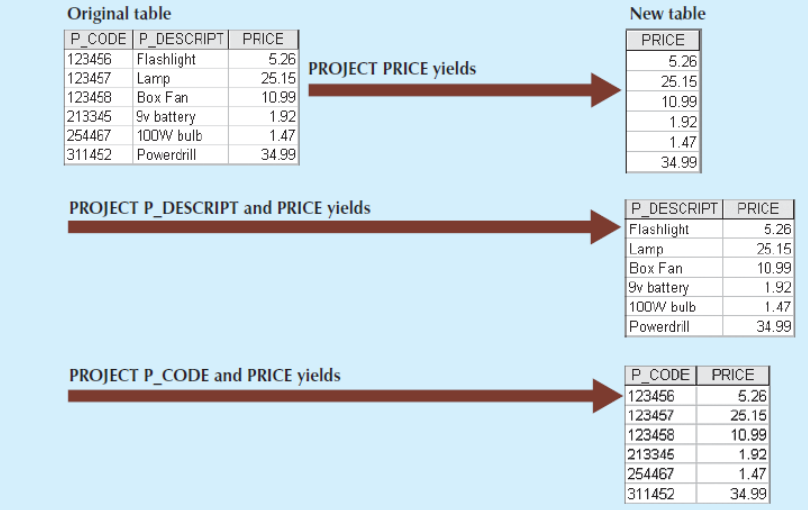
- Union
- Difference
- Intersection
- Cartesian Product

Very few DBMSs are capable of supporting all eight relational operators. To be considered minimally relational, the DBMS must support the key relational operators SELECT, PROJECT, and JOIN.

1. SELECT, also known as RESTRICT, yields values for all the rows found in a table that satisfy a given condition. SELECT yields a horizontal subset of a table.

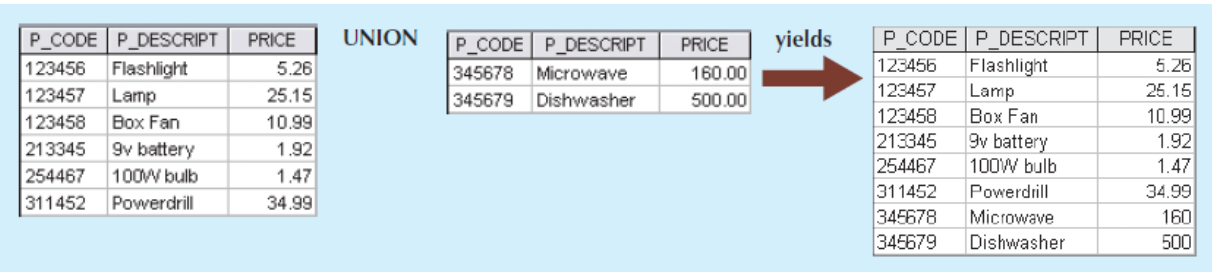


2. **PROJECT** yields all values for selected attributes. **PROJECT** yields a vertical subset of a table

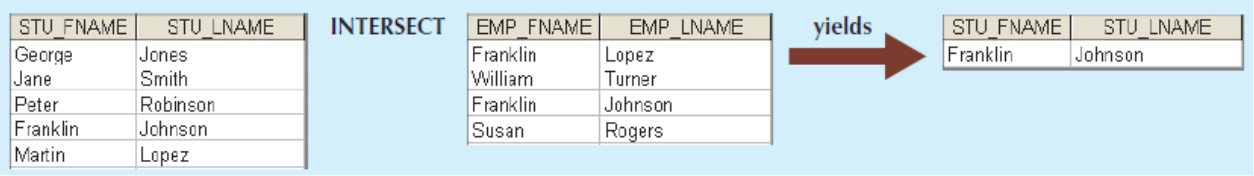


3. **UNION**: combines all rows from two or more tables, excluding duplicate rows. In order to be used in a **UNION**, the tables must be **UNION** compatible, that is:
   - The relations must all have the same number of attributes.
   - Corresponding columns must all have identical data types and lengths.

   When these criteria are met, the tables are said to be union compatible.



4. **INTERSECT**: yields only the rows that appear in both tables. As with **UNION**, the tables must be union-compatible to yield valid results.



5. **DIFFERENCE**: yields all rows in one table that are not found in the other table. As with the **UNION**, the tables must be **UNION**-compatible to yield valid results.

STU_FNAME	STU_LNAME
George	Jones
Jane	Smith
Peter	Robinson
Franklin	Johnson
Martin	Lopez

DIFFERENCE

EMP_FNAME	EMP_LNAME
Franklin	Lopez
William	Turner
Franklin	Johnson
Susan	Rogers

yields

STU_FNAME	STU_LNAME
George	Jones
Jane	Smith
Peter	Robinson
Martin	Lopez

6. PRODUCT: YIELDS ALL POSSIBLE PAIRS OF ROWS FROM TWO TABLES- ALSO KNOWN AS CARTESIAN PRODUCT. THEREFORE, IF ONE TABLE HAS SIX ROWS AND THE OTHER TABLE HAS THREE, THE PRODUCT YIELDS A LIST COMPOSED OF 6 X 3= 18 ROWS.

P_CODE	P_DESCRIPT	PRICE
123456	Flashlight	5.26
123457	Lamp	25.15
123458	Box Fan	10.99
213345	9v battery	1.92
254467	100W bulb	1.47
311452	Powerdrill	34.99

PRODUCT

STORE	AISLE	SHELF
23	W	5
24	K	9
25	Z	6

yields

P_CODE	P_DESCRIPT	PRICE	STORE	AISLE	SHELF
123456	Flashlight	5.26	23	W	5
123456	Flashlight	5.26	24	K	9
123456	Flashlight	5.26	25	Z	6
123457	Lamp	25.15	23	W	5
123457	Lamp	25.15	24	K	9
123457	Lamp	25.15	25	Z	6
123458	Box Fan	10.99	23	W	5
123458	Box Fan	10.99	24	K	9
123458	Box Fan	10.99	25	Z	6
213345	9v battery	1.92	23	W	5
213345	9v battery	1.92	24	K	9
213345	9v battery	1.92	25	Z	6
311452	Powerdrill	34.99	23	W	5
311452	Powerdrill	34.99	24	K	9
311452	Powerdrill	34.99	25	Z	6
254467	100W bulb	1.47	23	W	5
254467	100W bulb	1.47	24	K	9
254467	100W bulb	1.47	25	Z	6

7. JOIN: JOINS TWO TABLES TOGETHER USING A SHARED KEY USUALLY EITHER THE PRIMARY KEY OR FOREIGN KEY. JOIN ALLOWS THE USE OF INDEPENDENT TABLES LINKED BY COMMON ATTRIBUTES. JOIN IS A FUNDAMENTAL CONCEPT IN RELATIONAL DATABASE. A JOIN CAN EITHER BE INNER JOIN OR OUTER JOIN. AN INNER JOIN IS A JOIN THAT ONLY RETURNS MATCHED RECORDS FROM THE TABLES THAT ARE BEING JOINED E.G. ***NATURAL JOIN, EQUIJOIN, THETA JOIN***. IN AN OUTER JOIN, THE MATCHED PAIRS WOULD BE RETAINED, AND ANY UNMATCHED VALUES IN THE OTHER TABLE WOULD BE LEFT NULL. WE LOOK AT TYPES OF JOIN BELOW:
   - NATURAL JOIN (INNER JOIN): A NATURAL JOIN LINKS TABLES BY SELECTING ONLY THE ROWS WITH COMMON VALUES IN THEIR COMMON ATTRIBUTE(S). A NATURAL JOIN IS THE RESULT OF A THREE-STAGE PROCESS:
     a. PRODUCT OF THE TABLES IS CREATED
     b. SELECT IS PERFORMED ON THE OUTPUT OF STEP A) TO YIELD ONLY THE ROWS WHOSE VALUES ARE EQUAL.
     c. A PROJECT IS PERFORMED ON THE RESULTS OF STEP B TO YIELD A SINGLE COPY OF EACH ATTRIBUTE, THEREBY ELIMINATING DUPLICATE COLUMNS. THE FINAL OUTCOME OF A NATURAL JOIN YIELDS A TABLE THAT DOES NOT INCLUDE UNMATCHED PAIRS AND PROVIDES ONLY COPIES OF THE MATCHES.

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE
1132445	Walker	32145	231
1217782	Adares	32145	125
1312243	Rakowski	34129	167
1321242	Rodriguez	37134	125
1542311	Smithson	37134	421
1657399	Vanloo	32145	231

Table name: AGENT

AGENT_CODE	AGENT_PHONE
125	6152439887
167	6153426778
231	6152431124
333	9041234445

THE TWO TABLES TO BE USED FOR JOIN. IN THE FOLLOWING EXAMPLE,

```
SELECT*
FROM Customer
NATURAL JOIN Agent
```

NOTE A FEW CRUCIAL FEATURES OF THE NATURAL JOIN OPERATION:

- IF NO MATCH IS MADE BETWEEN THE TABLE ROWS, THE NEW TABLE DOES NOT INCLUDE THE UNMATCHED ROW. IN THAT CASE, NEITHER AGENT_CODE 421 NOR THE CUSTOMER WHOSE LAST NAME IS SMITHSON IS INCLUDED. SMITHSON'S AGENT_CODE 421 DOES NOT MATCH ANY ENTRY IN THE AGENT TABLE.

- The column on which the join was made—that is, AGENT_CODE—occurs only once in the new table.
- If the same AGENT_CODE were to occur several times in the AGENT table, a customer would be listed for each match. For example, if the AGENT_CODE 167 were to occur three times in the AGENT table, the customer named Rakowski, who is associated with AGENT_CODE 167, would occur three times in the resulting table. (A good AGENT table cannot, of course, yield such a result because it would contain unique primary key values.)

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1132445	Walker	32145	231	125	6152439887
1132445	Walker	32145	231	167	6153426778
1132445	Walker	32145	231	231	6152431124
1132445	Walker	32145	231	333	9041234445
1217782	Adares	32145	125	125	6152439887
1217782	Adares	32145	125	167	6153426778
1217782	Adares	32145	125	231	6152431124
1217782	Adares	32145	125	333	9041234445
1312243	Rakowski	34129	167	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1312243	Rakowski	34129	167	231	6152431124
1312243	Rakowski	34129	167	333	9041234445
1321242	Rodriguez	37134	125	125	6152439887
1321242	Rodriguez	37134	125	167	6153426778
1321242	Rodriguez	37134	125	231	6152431124
1321242	Rodriguez	37134	125	333	9041234445
1542311	Smithson	37134	421	125	6152439887
1542311	Smithson	37134	421	167	6153426778
1542311	Smithson	37134	421	231	6152431124
1542311	Smithson	37134	421	333	9041234445
1657399	Vanloo	32145	231	125	6152439887
1657399	Vanloo	32145	231	167	6153426778
1657399	Vanloo	32145	231	231	6152431124
1657399	Vanloo	32145	231	333	9041234445

Step 1: Cartesian product of the 2 tables

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	125	6152439887
1321242	Rodriguez	37134	125	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1132445	Walker	32145	231	231	6152431124
1657399	Vanloo	32145	231	231	6152431124

Step 2: SELECT yield only the rows for which the AGENT_CODE values are equal. The common columns are referred to as the join columns

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124

Step 3: PROJECT eliminates duplicate columns to yield only AGENT_CODE

- Equijoin, links tables on the basis of an equality condition that compares specified columns of each table. The outcome of the equijoin does not eliminate duplicate columns, and the condition or criterion used to join the tables must be explicitly defined. The equijoin takes its name from the equality comparison operator (=) used in the condition. E.g.

```
SELECT*
FROM Customer
JOIN Agent on (AGENT_CODE = CUSTOMER.AGENT_CODE)
```

- Theta join: If any other comparison operator such as (<, >, ...) is used, the join is called a theta join.

```
SELECT*
FROM Customer
JOIN Agent on (AGENT_CODE > CUSTOMER.AGENT_CODE)
```

- Outer Join: In an outer join, the matched pairs would be retained, and any unmatched values in the other table would be left null. It is an easy mistake to think that an outer join is the opposite of an inner join. However, it is more accurate to think of an outer join as an "inner join plus." The outer join still returns all of the matched records that the inner join returns, plus it returns the unmatched records from one of the tables. The SQL OUTER JOIN operator (+) is used only on one side of the join condition only. The subtypes of OUTER JOIN are:
  - Left outer join or left join
  - Right outer join or right join
  - Full outer join

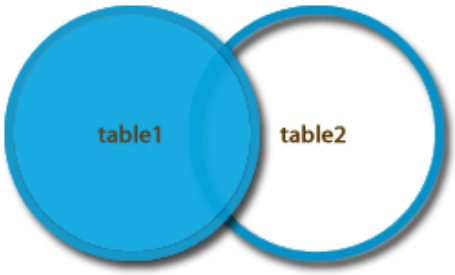
Syntax

```
Select *
FROM table1, table2
WHERE conditions [+];
```

- The LEFT JOIN (specified with the keywords LEFT JOIN and ON) joins two tables and fetches all matching rows of two tables for which the SQL-expression is true, plus rows from the first table that do not match any row in the second table.

Left Join: Syntax

```
SELECT *
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column_name=table2.column_name;
```



Pictorial representation of Left join

E.g.

```
SELECT *
FROM CUSTOMER
LEFT OUTER JOIN AGENT
ON CUSTOMER.AGENT_CODE = AGENT_CODE
```

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124
1542311	Smithson	37134	421	

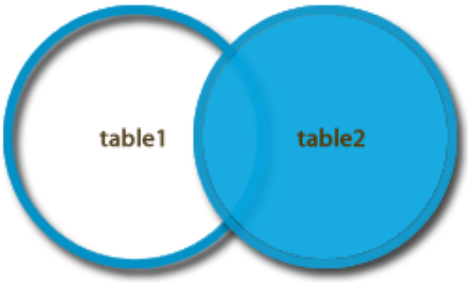
Left outer join or Left Join

- The RIGHT JOIN, joins two tables and fetches rows based on a condition, which are matching in both the tables (before and after the JOIN clause

mentioned in the syntax below), and the unmatched rows will also be available from the table written after the JOIN clause (mentioned in the syntax below).

Syntax

```
SELECT *
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column_name=table2.column_name;
```



Pictorial representation of Right Join

E.g.

```
SELECT *
FROM CUSTOMER
RIGHT OUTER JOIN AGENT
ON CUSTOMER.AGENT_CODE = AGENT_CODE
```

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124
			333	9041234445

Right Join

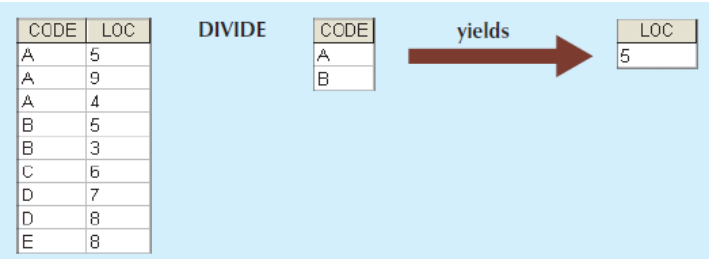
- Full outer join: the FULL OUTER JOIN combines the results of both left and right outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause.

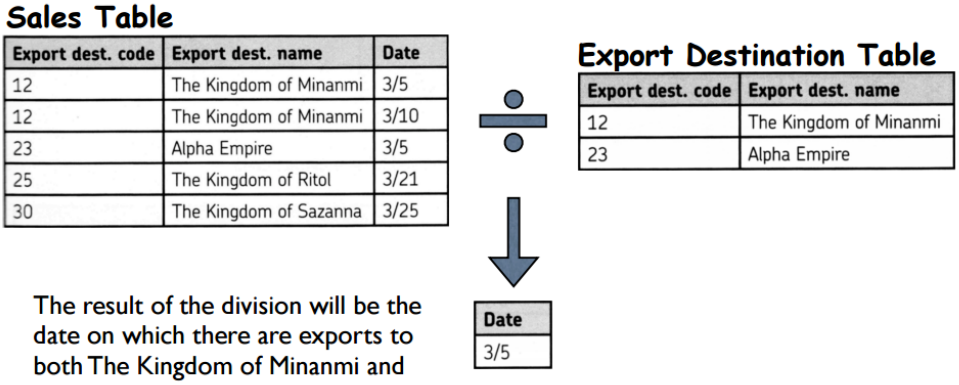
Syntax

```
SELECT *
FROM table1
FULL OUTER JOIN table2
ON table1.column_name=table2.column_name;
```

- ON table1.column_name=table2.column_name;More specifically, if an outer join is produced for tables CUSTOMER and AGENT, two scenarios are possible

8. The DIVIDE operation uses one single-column table (e.g., column "A") as the divisor and one 2-column table (i.e., columns "A" and "B") as the dividend. The tables must have a common column (e.g., column "A"). The output of the DIVIDE operation is a single column with the values of column "A" from the dividend table rows where the value of the common column (i.e., column "A") in both tables matches.





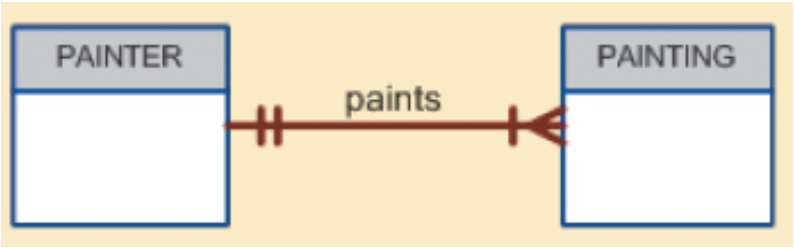
DIVIDE OPERATION

RELATIONSHIPS WITHIN THE RELATIONAL DATABASE

RELATIONSHIPS ARE CLASSIFIED AS: ONE-TO-ONE (1:1), ONE-TO-MANY (1:M), AND MANY-TO-MANY (M:N OR M:M). IN DEVELOPING A GOOD DATABASE DESIGNS, WE MUST FOCUS ON THE FOLLOWING POINTS:

- THE 1:M RELATIONSHIP IS THE RELATIONAL MODELING IDEAL. THEREFORE, THIS RELATIONSHIP TYPE SHOULD BE THE NORM IN ANY RELATIONAL DATABASE DESIGN.
- THE 1:1 RELATIONSHIP SHOULD BE RARE IN ANY RELATIONAL DATABASE DESIGN.
- M:N RELATIONSHIPS CANNOT BE IMPLEMENTED AS SUCH IN THE RELATIONAL MODEL. WE WILL LATER CONSIDER HOW ANY M:N RELATIONSHIP CAN BE CHANGED INTO TWO 1:M RELATIONSHIPS.

THE 1:M RELATIONSHIP



THE 1:M RELATIONSHIP BETWEEN PAINTER AND PAINTING

Table name: PAINTER
Primary key: PAINTER_NUM
Foreign key: none

Database

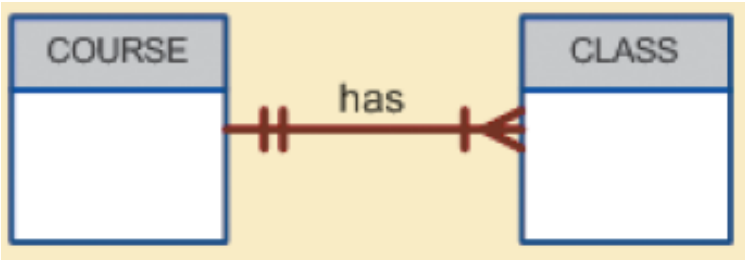
PAINTER_NUM	PAINTER_LNAME	PAINTER_FNAME	PAINTER_INITIAL
123	Ross	Georgette	P
126	Itero	Julio	G

Table name: PAINTING
Primary key: PAINTING_NUM
Foreign key: PAINTER_NUM

PAINTING_NUM	PAINTING_TITLE	PAINTER_NUM
1338	Dawn Thunder	123
1339	Vanilla Roses To Nowhere	123
1340	Tired Flounders	126
1341	Hasty Exit	123
1342	Plastic Paradise	126

THE IMPLEMENTED 1:M RELATIONSHIP BETWEEN PAINTER AND PAINTING

THE ONE-TO-MANY (1:M) RELATIONSHIP IS EASILY IMPLEMENTED IN THE RELATIONAL MODEL BY PUTTING THE PRIMARY KEY OF THE 1 SIDE IN THE TABLE OF THE MANY SIDE AS A FOREIGN KEY.



THE 1:M RELATIONSHIP BETWEEN COURSE AND CLASS

Table name: COURSE
Primary key: CRS_CODE
Foreign key: none

Database name: Ch03_TinyCollege

CRS_CODE	DEPT_CODE	CRS_DESCRIPTION	CRS_CREDIT
ACCT-211	ACCT	Accounting I	3
ACCT-212	ACCT	Accounting II	3
CIS-220	CIS	Intro. to Microcomputing	3
CIS-420	CIS	Database Design and Implementation	4
QM-261	CIS	Intro. to Statistics	3
QM-362	CIS	Statistical Applications	4

Table name: CLASS
Primary key: CLASS_CODE
Foreign key: CRS_CODE

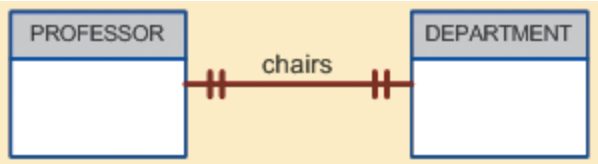
CLASS_CODE	CRS_CODE	CLASS_SECTION	CLASS_TIME	CLASS_ROOM	PROF_NUM
10012	ACCT-211	1	MWF 8:00-8:50 a.m.	BUS311	105
10013	ACCT-211	2	MWF 9:00-9:50 a.m.	BUS200	105
10014	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10015	ACCT-212	1	MWF 10:00-10:50 a.m.	BUS311	301
10016	ACCT-212	2	Th 6:00-8:40 p.m.	BUS252	301
10017	CIS-220	1	MWF 9:00-9:50 a.m.	KLR209	228
10018	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10019	CIS-220	3	MWF 10:00-10:50 a.m.	KLR209	228
10020	CIS-420	1	W 6:00-8:40 p.m.	KLR209	162
10021	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114
10022	QM-261	2	TTh 1:00-2:15 p.m.	KLR200	114
10023	QM-362	1	MWF 11:00-11:50 a.m.	KLR200	162
10024	QM-362	2	TTh 2:30-3:45 p.m.	KLR200	162

THE IMPLEMENTED 1:M RELATIONSHIP BETWEEN COURSE AND CLASS

THE 1:1 RELATIONSHIP: AS THE 1:1 LABEL IMPLIES, IN THIS RELATIONSHIP, ONE ENTITY CAN BE RELATED TO ONLY ONE OTHER ENTITY, AND VICE VERSA. FOR EXAMPLE, ONE DEPARTMENT CHAIR—

a professor—can chair only one department, and one department can have only one department chair.

The entities PROFESSOR and DEPARTMENT thus exhibit a 1:1 relationship.



The 1:M relationship between PROFRSSOR and DEPARTMENT

If we the examine the PROFESSOR and DEPARTMENT tables, we note some important features:

- Each professor is a College employee; thus, the professor identification is through the EMP_NUM. (However, note that not all employees are professors—there's another optional relationship.)
- The 1:1 PROFESSOR chairs DEPARTMENT relationship is implemented by having the EMP_NUM as foreign key in the DEPARTMENT table. Note that the 1:1 relationship is treated as a special case of the 1:M relationship in which the "many" side is restricted to a single occurrence. In this case, DEPARTMENT contains the EMP_NUM as a foreign key to indicate that it is the department that has a chair.
- Also, note that the PROFESSOR table contains the DEPT_CODE foreign key to implement the 1:M DEPARTMENT employs PROFESSOR relationship. This is a good example of how two entities can participate in two (or even more) relationships simultaneously. The preceding "PROFESSOR chairs DEPARTMENT" example illustrates a proper 1:1 relationship. In fact, the use of a 1:1 relationship ensures that two entity sets are not placed in the same table when they should not be. However, the existence of a 1:1 relationship sometimes means that the entity components were not defined properly. It could indicate that the two entities actually belong in the same table! As rare as 1:1 relationships should be, certain conditions absolutely require their use. One such condition is the concept called generalization hierarchy, which is a powerful tool for improving database designs under specific conditions to avoid a proliferation of nulls. One of the characteristics of generalization hierarchies is that they are implemented as 1:1 relationships.

  Table name: PROFESSOR
  Primary key: EMP_NUM
  Foreign key: DEPT_CODE

EMP_NUM	DEPT_CODE	PROF_OFFICE	PROF_EXTENSION	PROF_HIGH_DEGREE
103	HIST	DRE 155	6783	Ph.D.
104	ENG	DRE 102	5561	MA
105	ACCT	KLR 229D	8665	Ph.D.
106	MKT/MGT	KLR 125	3899	Ph.D.
110	BIOL	AAK 160	3412	Ph.D.
114	ACCT	KLR 211	4436	Ph.D.
155	MATH	AAK 201	4440	Ph.D.
160	ENG	DRE 102	2240	Ph.D.
162	CIS	KLR 209E	2359	Ph.D.
191	MKT/MGT	KLR 409B	4016	DBA
195	PSYCH	AAK 297	3550	Ph.D.
209	CIS	KLR 333	3421	Ph.D.
228	CIS	KLR 300	3000	Ph.D.
297	MATH	AAK 194	1145	Ph.D.
299	ECON/FIN	KLR 284	2851	Ph.D.
301	ACCT	KLR 244	4683	Ph.D.
335	ENG	DRE 208	2000	Ph.D.
342	SOC	BBG 208	5514	Ph.D.
387	BIOL	AAK 230	8665	Ph.D.
401	HIST	DRE 155	6783	MA
425	ECON/FIN	KLR 284	2851	MBA
435	ART	BBG 185	2278	Ph.D.

The 1:M DEPARTMENT employs PROFESSOR relationship is implemented through the placement of the DEPT_CODE foreign key in the PROFESSOR table.

The 1:1 PROFESSOR chairs DEPARTMENT relationship is implemented through the placement of the EMP_NUM foreign key in the DEPARTMENT table.

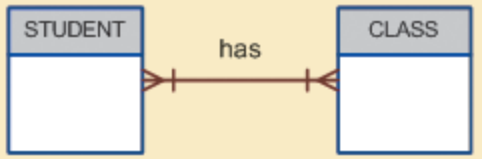
Table name: DEPARTMENT
Primary key: DEPT_CODE
Foreign key: EMP_NUM

DEPT_CODE	DEPT_NAME	SCHOOL_CODE	EMP_NUM	DEPT_ADDRESS	DEPT_EXTENSION
ACCT	Accounting	BUS	114	KLR 211, Box 52	3119
ART	Fine Arts	A&SCI	435	BBG 185, Box 128	2278
BIOL	Biology	A&SCI	387	AAK 230, Box 415	4117
CIS	Computer Info. Systems	BUS	209	KLR 333, Box 56	3245
ECON/FIN	Economics/Finance	BUS	299	KLR 284, Box 63	3126
ENG	English	A&SCI	160	DRE 102, Box 223	1004
HIST	History	A&SCI	103	DRE 156, Box 284	1867
MATH	Mathematics	A&SCI	297	AAK 194, Box 422	4234
MKT/MGT	Marketing/Management	BUS	106	KLR 126, Box 55	3342
PSYCH	Psychology	A&SCI	195	AAK 297, Box 438	4110
SOC	Sociology	A&SCI	342	BBG 208, Box 132	2008

The M:N Relationship: A many-to-many (M:N) relationship is not supported directly in the relational environment. However, M:N relationships can be implemented by creating a new entity in 1:M relationships with the original entities.

To explore the many-to-many (M:N) relationship, consider a rather typical college environment in which each **STUDENT** can take many **CLASS**es, and each **CLASS** can contain many **STUDENT**s. The ER model for this M:N relationship is below:

The ERM's M:N relationship between **STUDENT** and **CLASS**



Note the features of the ERM above:

- Each **CLASS** can have many **STUDENT**s, and each **STUDENT** can take many **CLASS**es.
- There can be many rows in the **CLASS** table for any given row in the **STUDENT** table, and there can be many rows in the **STUDENT** table for any given row in the **CLASS** table.

To examine the M:N relationship more closely, imagine a small college with two students, each of whom takes three classes. The table below shows the enrollment data for the two students.

Sample Student Enrollment Data

STUDENT'S LAST NAME	SELECTED CLASSES
Bowser	Accounting 1, ACCT-211, code 10014 Intro to Microcomputing, CIS-220, code 10018 Intro to Statistics, QM-261, code 10021
Smithson	Accounting 1, ACCT-211, code 10014 Intro to Microcomputing, CIS-220, code 10018 Intro to Statistics, QM-261, code 10021

Table name: STUDENT
Primary key: STU_NUM
Foreign key: none

STU_NUM	STU_LNAME	CLASS_CODE
321452	Bowser	10014
321452	Bowser	10018
321452	Bowser	10021
324257	Smithson	10014
324257	Smithson	10018
324257	Smithson	10021

Table name: CLASS
Primary key: CLASS_CODE
Foreign key: STU_NUM

CLASS_CODE	STU_NUM	CRS_CODE	CLASS_SECTION	CLASS_TIME	CLASS_ROOM	PROF_NUM
10014	321452	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10014	324257	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10018	321452	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10018	324257	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10021	321452	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114
10021	324257	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114

Given the data relationship and the sample data in the table above, it can be wrongly assumed that M:N relationship can be implemented by simply adding a foreign key in the many side of the relationship that points to the primary key of the related table. This not correct

- The tables will create many redundancies. For example, note that the STU_NUM values occur many times in the STUDENT table. In a real-world situation, additional student attributes such as address, classification, major, and home phone would also be contained in the STUDENT table, and each of those attribute values would be repeated in each of the records shown here. Similarly, the CLASS table contains many duplications: each student taking the class generates a CLASS record. The problem would be even worse if the CLASS table included such attributes as credit hours and course description.
- Given the structure and contents of the two tables, the relational operations become very complex and are likely to lead to system efficiency errors and output errors.

The problems inherent in the many-to-many (M:N) relationship can easily be avoided by creating a
composite entity (also referred to as a bridge entity or an associative entity). Because such a table is used to link the tables that were originally related in an M:N relationship, the composite entity structure includes—as foreign keys—at least the primary keys of the tables that are to be linked. The database designer can then define the composite table's primary key either by: using the combination of those foreign keys or create a new primary key. In the example above, we can create the composite ENROLL table CLASS and STUDENT. In this example, the ENROLL table's primary key is the combination of its foreign keys CLASS_CODE and STU_NUM. But the designer could have decided to create a single-attribute new primary key such as ENROLL_LINE, using a different line value to identify each ENROLL table row uniquely. (Microsoft Access users might use the Autonumber data type to generate such line values automatically).

Table name: STUDENT
Primary key: STU_NUM
Foreign key: none

STU_NUM	STU_LNAME
321452	Bowser
324257	Smithson

Table name: ENROLL
Primary key: CLASS_CODE + STU_NUM
Foreign key: CLASS_CODE, STU_NUM

CLASS_CODE	STU_NUM	ENROLL_GRADE
10014	321452	C
10014	324257	B
10018	321452	A
10018	324257	B
10021	321452	C
10021	324257	C

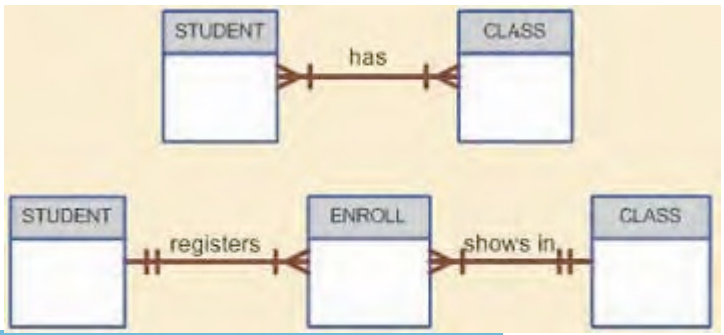
Table name: CLASS
Primary key: CLASS_CODE
Foreign key: CRS_CODE

CLASS_CODE	CRS_CODE	CLASS_SECTION	CLASS_TIME	CLASS_ROOM	PROF_NUM
10014	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10018	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10021	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114

Because the ENROLL table links two tables, STUDENT and CLASS, it is also called a linking table.
In other words, a linking table is the implementation of a composite entity.

The ENROLL table yields the required M:N to 1:M conversion. Observe that the composite entity represented by the ENROLL table must contain at least the primary keys of the CLASS and STUDENT tables (CLASS_CODE and STU_NUM, respectively) for which it serves as a connector. Also note that the STUDENT and CLASS tables now contain only one row per entity. The ENROLL table contains multiple occurrences of the foreign key values, but those controlled redundancies are incapable of producing anomalies as long as referential integrity is enforced. Additional attributes may be assigned as needed. In this case, ENROLL_GRADE is selected to satisfy a reporting requirement. Also note that the ENROLL table's primary key consists of the two attributes CLASS_CODE and STU_NUM because both the class code and the student number are needed to define a particular student's grade. Naturally, the conversion is reflected in the ERM, too. The revised relationship is shown below:

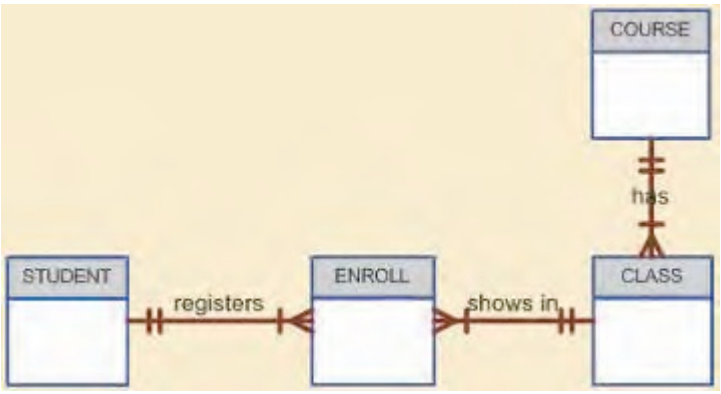
Changing the M:N relationship to two 1:M relationships



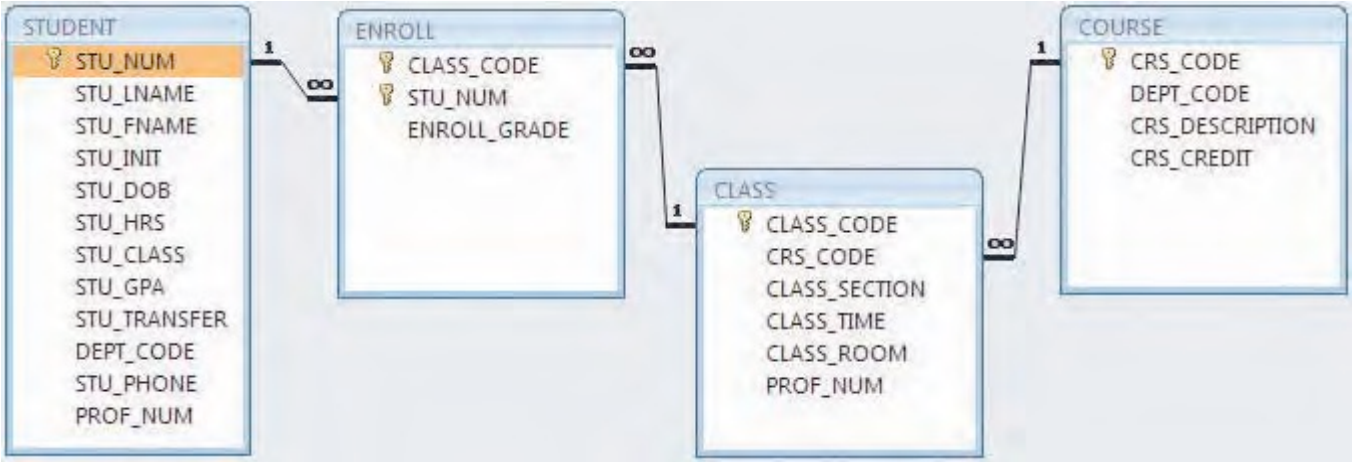
Note that the composite entity named ENROLL represents the linking table between STUDENT and CLASS. We can increase the amount of available information even as

we control the database's redundancies. Below is the expanded ERM, including the 1:M relationship between COURSE and CLASS. Note that the model is able to handle multiple sections of a CLASS while controlling redundancies by making sure that all of the COURSE data common to each CLASS are kept in the COURSE table.

expanded entity relationship model



The relationship diagram that corresponds to the ERM shown above is as below:



CODD'S RELATIONAL DATABASE RULES

In 1985, Dr. E. F. Codd published a list of 12 rules to define a relational database system. The reason Dr. Codd published the list was his concern that many vendors were marketing products as "relational" even though those products did not meet minimum relational standards. Dr. Codd's list, serves as a frame of reference for what a truly relational database should be. Note that even the dominant database vendors do not fully support all 12 rules.

Dr. Codd's 12 Relational Database Rules

RULE	RULE NAME	DESCRIPTION
1	**Information**	All information in a relational database must be logically represented as column values in rows within tables.
2	**Guaranteed Access**	Every value in a table is guaranteed to be accessible through a combination of table name, primary key value, and column name.
3	**Systematic Treatment of Nulls**	Nulls must be represented and treated in a systematic way, independent of data type.
4	**Dynamic Online Catalog Based on the Relational Model**	The metadata must be stored and managed as ordinary data, that is, in tables within the database. Such data must be available to authorized users using the standard database relational language.
5	**Comprehensive Data Sublanguage**	The relational database may support many languages. However, it must support one well-defined, declarative language with support for data definition, view definition, data manipulation (interactive and by program), integrity constraints, authorization, and transaction management (begin, commit, and rollback).
6	**View Updating**	Any view that is theoretically updatable must be updatable through the system.
7	**High-Level Insert, Update, and Delete**	The database must support set-level inserts, updates, and deletes.
8	**Physical Data Independence**	Application programs and ad hoc facilities are logically unaffected when physical access methods or storage structures are changed.
9	**Logical Data Independence**	Application programs and ad hoc facilities are logically unaffected when changes are made to the table structures that preserve the original table values (changing order of columns or inserting columns).
10	**Integrity Independence**	All relational integrity constraints must be definable in the relational language and stored in the system catalog, not at the application level.
11	**Distribution Independence**	The end users and application programs are unaware and unaffected by the data location (distributed vs. local databases).
12	**Nonsubversion**	If the system supports low-level access to the data, there must not be a way to bypass the integrity rules of the database.
	Rule Zero	All preceding rules are based on the notion that in order for a database to be considered relational, it must use its relational facilities exclusively to manage the database.

THE ENTITY RELATIONSHIP MODEL (ERM)

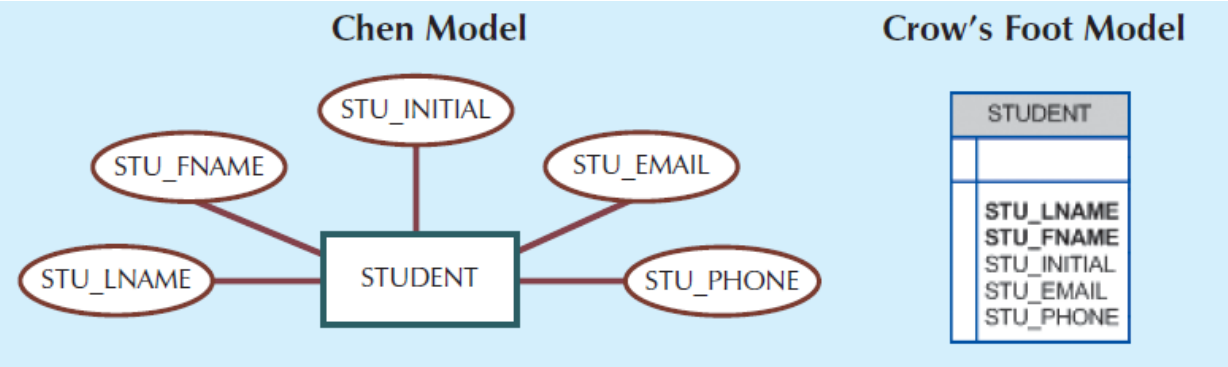
Peter Chen first introduced the ER data model in 1976; it was the graphical representation of entities and their relationships in a database structure that quickly became popular because it complemented the relational data model concepts. The relational data model and ERM combined to provide the foundation for tightly structured database design. ER models are normally represented in an entity relationship diagram (ERD), which uses graphical representations to model database components. The ERD represents the conceptual database as viewed by the end user. ERDs depict the database's main components: entities, attributes, and relationships. Because an entity represents a real-world object, the words entity and object are often used interchangeably. The notations used with ERDs are the original Chen notation and the newer Crow's Foot and UML notations. Some conceptual database modeling concepts can be expressed only using the Chen notation. Because of its implementation emphasis, the Crow's Foot notation can represent only what could be implemented. In summary:

- The Chen notation favors conceptual modeling.
- The Crow's Foot notation favors a more implementation-oriented approach.
- The UML notation can be used for both conceptual and implementation modeling.

The ER model is based on the following components:

- **Entity:** An entity is anything about which data are to be collected and stored. An entity is represented in the ERD by a rectangle, also known as an entity box. The name of the entity, a noun, is written in the center of the rectangle. The entity name is generally written in capital letters and is written in the singular form: PAINTER rather than PAINTERS, and EMPLOYEE rather than EMPLOYEES. Usually, when applying the ERD to the relational model, an entity is mapped to a relational table. Each row in the relational table is known as an entity instance or entity occurrence in the ER model. Each entity is described by a set of attributes that describes particular characteristics of the entity. For example, the entity EMPLOYEE will have attributes such as a Social Security number, a last name, and a first name. A collection of like entities is known as an entity set. The word entity in the ERM corresponds to a table—not to a row—in the relational environment. The ERM refers to a table row as an entity instance or entity occurrence.
- **Attributes:** Attributes are characteristics of entities. For example, the STUDENT entity includes, among many others, the attributes STU_LNAME, STU_FNAME, and STU_INITIAL. In the original Chen notation, attributes are represented by ovals and are connected to the entity rectangle with a line. Each oval contains the name of the attribute it represents. In the Crow's Foot notation, the attributes are written in the attribute box below the entity rectangle. Because the Chen representation is rather space-consuming, software vendors have adopted the Crow's Foot attribute display.

Attributes of the STUDENT entity: Chen and crow's foot



Required and Optional Attributes: A required attribute is an attribute that must have a value; in other words, it cannot be left empty. As shown above there are two boldfaced attributes in the Crow's Foot notation. This indicates that a data entry will be required. In this example, STU_LNAME and STU_FNAME require data entries because of the assumption that all students have a last name and a first name. But students might not have a middle name, and perhaps they do not (yet) have a phone number and an e-mail address. Therefore, those attributes are not presented in boldface in the entity box. An optional attribute is an attribute that does not require a value; therefore, it can be left empty.

Attribute domains: Attributes have a domain. A domain is the set of possible values for a given attribute. For example, the domain for the grade point average (GPA) attribute is written (0,4) because the lowest possible GPA value is 0 and the highest possible value is 4. The domain for the gender attribute consists of only two possibilities: M or F (or some other equivalent code). The domain for a company's date of hire attribute consists of all dates that fit in a range (for example, company startup date to current date). Attributes may share a domain. For instance, a student address and a professor address share the same domain of all possible addresses. In fact, the data dictionary may let a newly declared attribute inherit the characteristics of an existing attribute if the same attribute name is used. For example, the PROFESSOR and STUDENT entities may each have an attribute named ADDRESS and could therefore share a domain.

Identifiers (Primary Keys): The ERM uses identifiers, that is, one or more attributes that uniquely identify each entity instance. In the relational model,

such identifiers are mapped to primary keys (PKs) in tables. Identifiers are underlined in the ERD. Key attributes
are also underlined in a frequently used table structure shorthand notation using the format:

TABLE NAME (KEY_ATTRIBUTE 1, ATTRIBUTE 2, ATTRIBUTE 3, . . . ATTRIBUTE K)

For example, a CAR entity may be represented by:

CAR (CAR_VIN, MOD_CODE, CAR_YEAR, CAR_COLOR)

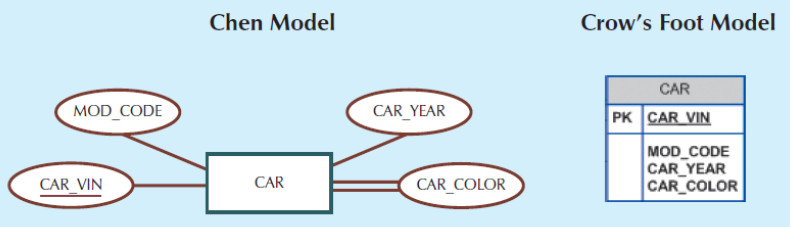
(Each car is identified by a unique vehicle identification number, or CAR_VIN.)

Composite Identifiers: Ideally, an entity identifier is composed of only a single attribute. However, it is possible to use a composite identifier, that is, a primary key composed of more than one attribute. E.g. CLASS entity of CRS_CODE and CLASS_SECTION instead of using CLASS_CODE. Either approach uniquely identifies each entity instance.

Composite and Simple Attributes: Attributes are classified as simple or composite. A composite attribute, not to be confused with a composite key, is an attribute that can be further subdivided to yield additional attributes. For example, the attribute ADDRESS can be subdivided into street, city, state, and zip code. Similarly, the attribute PHONE_NUMBER can be subdivided into area code and exchange number. A simple attribute is an attribute that cannot be subdivided. For example, age, sex and marital status would be classified as simple attributes. To facilitate detailed queries, it is wise to change composite attributes into a series of simple attributes.

Single-Valued Attributes: A single-valued attribute is an attribute that can have only a single value. For example, a person can have only one Social Security number, and a manufactured part can have only one serial number. Keep in mind that a single-valued attribute is not necessarily a simple attribute. For instance, a part's serial number, such as SE-08-02-189935, is single-valued, but it is a composite attribute because it can be subdivided into the region in which the part was produced (SE), the plant within that region (08), the shift within the plant (02), and the part number (189935).

Multivalued Attributes: Multivalued attributes are attributes that can have many values. For instance, a person may have several college degrees, and a household may have several different phones, each with its own number. Similarly, a car's color may be subdivided into many colors (that is, colors for the roof, body, and trim). In the Chen ERM, the multivalued attributes are shown by a double line connecting the attribute to the entity. The Crow's Foot notation does not identify multivalued attributes.



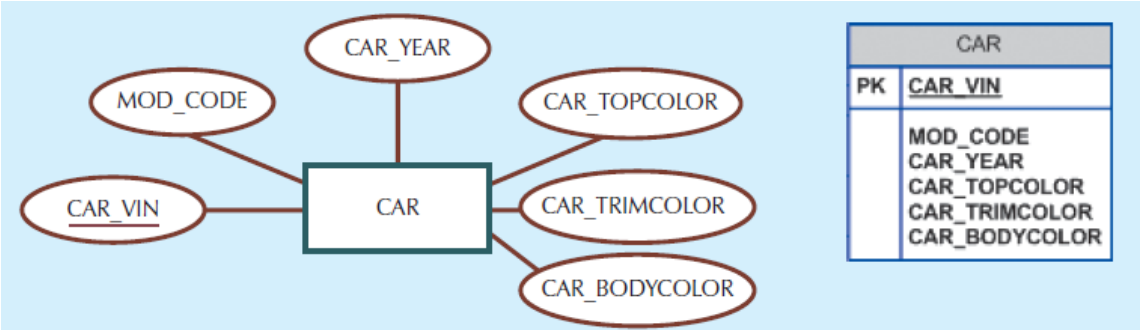
A multivalued attribute in an entity

The ERD above contains all of the components introduced thus far. Note that CAR_VIN is the primary key, and CAR_COLOR is a multivalued attribute of the CAR entity.

Implementing Multivalued Attributes

Although the conceptual model can handle M:N relationships and multivalued attributes, it poor practice to implement them in the RDBMS. In the relational table, each column/row intersection represents a single data value. The designer must decide on one of two possible courses of action to handle multivalued attributes:

i. Split the multivalued attribute to create several new attributes. For example, the CAR entity's attribute CAR_COLOR can be split to create the new attributes CAR_TOPCOLOR, CAR_BODYCOLOR, and CAR_TRIMCOLOR, which are then assigned to the CAR entity. Although this solution seems to work, its adoption can lead to major structural problems in the table. For example, if additional color components—such as a logo color—are added for some cars, the table structure must be modified to accommodate the new color section. In that case, cars that do not have such color sections generate nulls for the nonexisting components, or their color entries for those sections are entered as N/A to indicate "not applicable." Also consider the employee entity containing employee degrees and certifications. If some employees have 10 degrees and certifications while most have fewer or none, the number of degree/certification attributes would number 10, and most of those attribute values would be null for most of the employees.) In short, while solution 1 is practicable, it is not an acceptable solution.



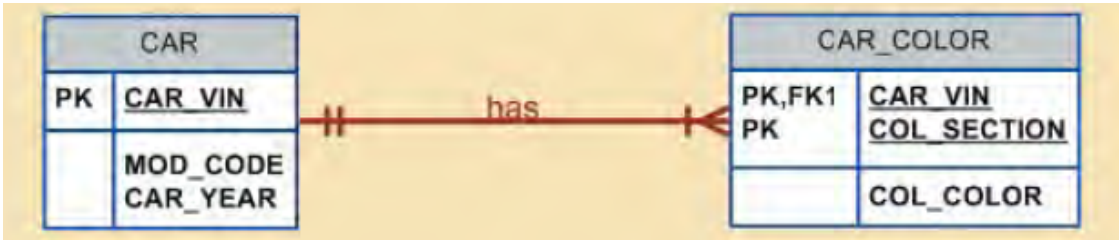
Splitting the multivalued attribute into new attributes

ii. Create a new entity composed of the original multivalued attribute's components. This new entity allows the designer to define color for different sections of the car. (See Table below).

Components of the Multivalued Attribute

SECTION	COLOR
Top	White
Body	Blue
Trim	Gold
Interior	Blue

Another benefit we can derive from this approach is that we are now able to assign as many colors as necessary without having to change the table structure.



A new entity set composed of a multivalued attribute's components

Note that the ERM shown in Figure above reflects the components listed in previous table. This is the preferred way to deal with multivalued attributes. Creating a new entity in a 1:M relationship with the original entity yields several benefits: it's a more flexible, expandable solution, and it is compatible with the relational model!

Derived Attributes: A derived attribute is an attribute whose value is calculated (derived) from other attributes. The derived attribute need not be physically stored within the database; instead, it can be derived by using an algorithm. For example, an employee's age, **EMP_AGE**, may be found by computing the integer value of the difference between the current date and the **EMP_DOB**. In Microsoft Access, we use:

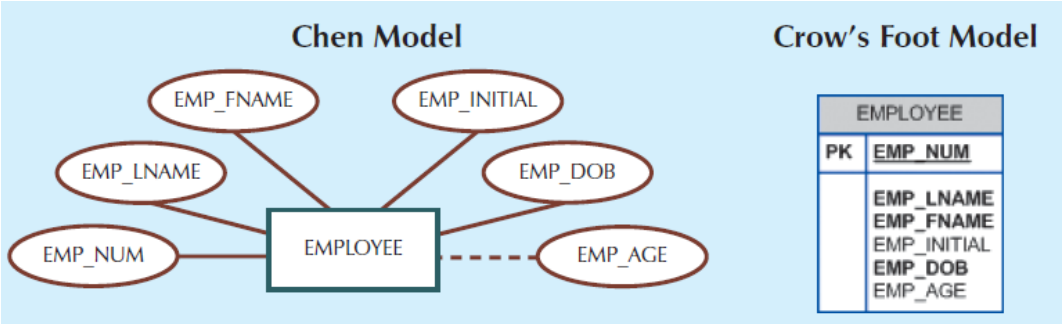
```
INT((DATE() - EMP_DOB)/365)
```

In Microsoft **SQL Server**, we use

```
SELECT DATEDIFF("YEAR", EMP_DOB, GETDATE()),
```

where **DATEDIFF** is a function that computes the difference between dates. The first parameter indicates the measurement, in this case, years. In Oracle, we use **SYSDATE** instead of **DATE()**.

A derived attribute is indicated in the Chen notation by a dashed line connecting the attribute and the entity. The Crow's Foot notation does not have a method for distinguishing the derived attribute from other attributes.



Depiction of a derived attribute

Derived attributes are sometimes referred to as computed attributes. A derived attribute computation can be as simple as adding two attribute values located on the same row, or it can be the result of aggregating the sum of values located on many table rows (from the same table or from a different table). The decision to store derived attributes in database tables depends on the processing requirements and the constraints placed on a particular application. The designer should be able to balance the design in accordance with such constraints.

Table below shows the advantages and disadvantages of storing (or not storing) derived attributes in the database.

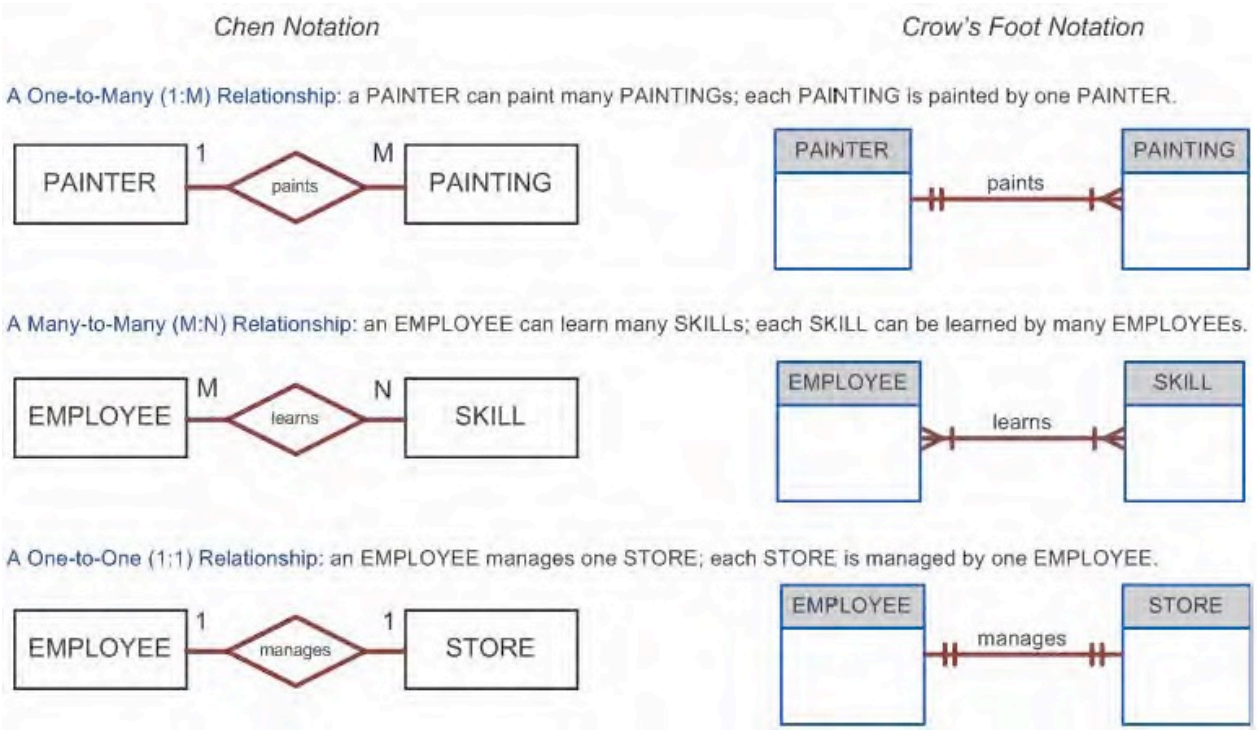
advantages and disadvantages of storing (or not storing) derived attributes in the database.

	DERIVED ATTRIBUTE	
	STORED	NOT STORED
Advantage	Saves CPU processing cycles Saves data access time Data value is readily available Can be used to keep track of historical data	Saves storage space Computation always yields current value
Disadvantage	Requires constant maintenance to ensure derived value is current, especially if any values used in the calculation change	Uses CPU processing cycles Increases data access time Adds coding complexity to queries

- Relationships. Relationships describe associations among data. Most relationships describe associations between two entities. The three types of relationships among data include:
  - one-to-many (1:M)
  - many-to-many (M:N)
  - and one-to-one (1:1).

The ER model uses the term connectivity to label the relationship types. The name of the relationship is usually an active or passive verb. For example, a **PAINTER** paints many **PAINTINGS**; an **EMPLOYEE** learns many **SKILLS**; an **EMPLOYEE** manages a **STORE**. Illustrated below are the different types of

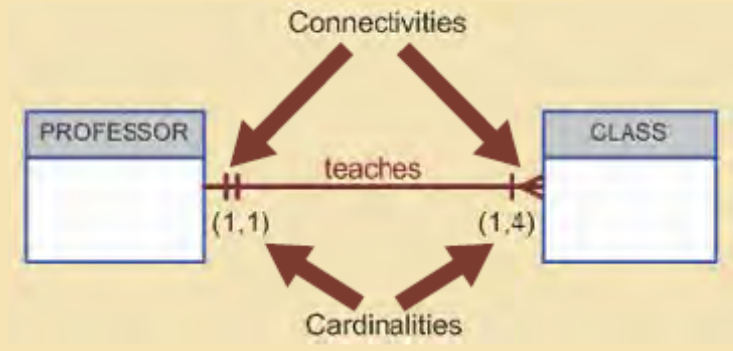
relationships using two ER notations: the original Chen notation and the more current Crow's Foot notation.



The left side of the ER diagram shows the Chen notation, based on Peter Chen's landmark paper. In this notation, the connectivities are written next to each entity box. Relationships are represented by a diamond connected to the related entities through a relationship line. The relationship name is written inside the diamond. The right side illustrates the Crow's Foot notation. The name "Crow's Foot" is derived from the three-pronged symbol used to represent the "many" side of the relationship. In the basic Crow's Foot ERD represented above, the connectivities are represented by symbols. For example, the "1" is represented by a short line segment, and the "M" is represented by the three-pronged "crow's foot." The relationship name is written above the relationship line. In Figure above, entities and relationships are shown in a horizontal format, but they may also be oriented vertically. The entity location and the order in which the entities are presented are immaterial; just remember to read a 1:M relationship from the "1" side to the "M" side.

Connectivity and Cardinality

As stated above, the term connectivity is used to describe the relationship classification. Cardinality expresses the minimum and maximum number of entity occurrences associated with one occurrence of the related entity. In the ERD, cardinality is indicated by placing the appropriate numbers beside the entities, using the format (x,y). The first value represents the minimum number of associated entities, while the second value represents the maximum number of associated entities. Many database designers who use Crow's Foot modeling notation do not depict the specific cardinalities on the ER diagram itself because the specific limits described by the cardinalities cannot be implemented directly through the database design. Correspondingly, some Crow's Foot ER modeling tools do not print the numeric cardinality range in the diagram; instead, you can add it as text if you want to have it shown.



Connectivity and Cardinality

Knowing the minimum and maximum number of entity occurrences is very useful at the application software level. A college might want to ensure that a class is not taught unless it has at least 10 students enrolled. Similarly, if the classroom can hold only 30 students, the application software should use that cardinality to limit enrollment in the class. However, keep in mind that the DBMS cannot handle the implementation of the cardinalities at the table level—that capability is provided by the application software or by triggers.

Existence Dependence: An entity is said to be existence-dependent if it can exist in the database only when it is associated with another related entity occurrence. In implementation terms, an entity is existence-dependent if it has a mandatory foreign key—that is, a foreign key attribute that cannot be null. For example, if an employee wants to claim one or more dependents for tax-withholding purposes, the relationship "EMPLOYEE claims DEPENDENT" would be appropriate. In that case, the DEPENDENT entity is clearly existence-dependent on the EMPLOYEE entity because it is impossible for the dependent to exist apart from the EMPLOYEE in the database. If an entity can exist apart from all of its related entities (it is existence-independent), then that entity is referred to as a strong entity or regular entity.

Relationship Strength: The concept of relationship strength is based on how the primary key of a related entity is defined. To implement a relationship, the primary key of one entity appears as a foreign key in the related entity. For example, the 1:M relationship between VENDOR and PRODUCT is implemented by using the VEND_CODE primary key in VENDOR as a foreign key in PRODUCT. There are times when the foreign key also is a primary key component in the related entity. Relationship strength decisions affect primary key arrangement in database design.

Weak (Non-identifying) Relationships: A weak relationship, also known as a non-identifying relationship, exists if the PK of the related entity does not contain a PK component of the parent entity. By default, relationships are established by having the PK of the parent entity appear as an FK on the related entity. For example, suppose that the COURSE and CLASS entities are defined as:

```
COURSE(CRS_CODE, DEPT_CODE, CRS_DESCRIPTION, CRS_CREDIT)
CLASS(CLASS_CODE, CRS_CODE, CLASS_SECTION, CLASS_TIME,
ROOM_CODE, PROF_NUM)
```

In this case, a weak relationship exists between COURSE and CLASS because the CLASS_CODE is the CLASS entity's PK, while the CRS_CODE in CLASS is only an FK. In this example, the CLASS PK did not inherit the PK component from the COURSE entity.

Table name: COURSE

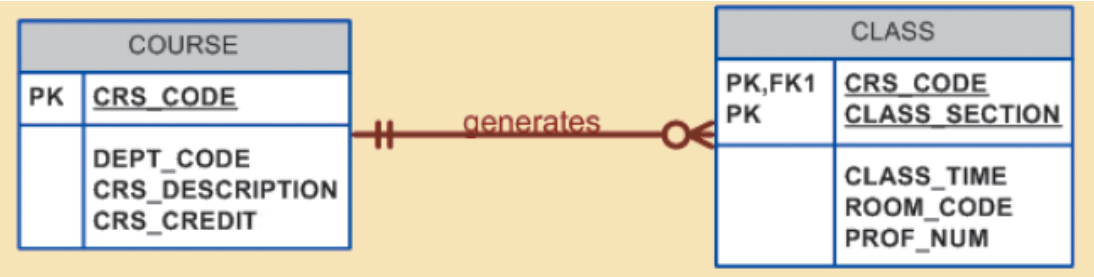
CRS_CODE	DEPT_CODE	CRS_DESCRIPTION	CRS_CREDIT
ACCT-211	ACCT	Accounting I	3
ACCT-212	ACCT	Accounting II	3
CIS-220	CIS	Intro. to Microcomputing	3
CIS-420	CIS	Database Design and Implementation	4
MATH-243	MATH	Mathematics for Managers	3
QM-261	CIS	Intro. to Statistics	3
QM-362	CIS	Statistical Applications	4

Table name: CLASS

CLASS_CODE	CRS_CODE	CLASS_SECTION	CLASS_TIME	ROOM_CODE	PROF_NUM
10012	ACCT-211	1	MWF 8:00-8:50 a.m.	BUS311	105
10013	ACCT-211	2	MWF 9:00-9:50 a.m.	BUS200	105
10014	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10015	ACCT-212	1	MWF 10:00-10:50 a.m.	BUS311	301
10016	ACCT-212	2	Th 6:00-8:40 p.m.	BUS252	301
10017	CIS-220	1	MWF 9:00-9:50 a.m.	KLR209	228
10018	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10019	CIS-220	3	MWF 10:00-10:50 a.m.	KLR209	228
10020	CIS-420	1	W 6:00-8:40 p.m.	KLR209	162
10021	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114
10022	QM-261	2	TTh 1:00-2:15 p.m.	KLR200	114
10023	QM-362	1	MWF 11:00-11:50 a.m.	KLR200	162
10024	QM-362	2	TTh 2:30-3:45 p.m.	KLR200	162
10025	MATH-243	1	Th 6:00-8:40 p.m.	DRE155	325

Crow's Foot notation depicts a strong relationship

Strong (Identifying) Relationships: A strong relationship, also known as an identifying relationship, exists when the PK of the related entity contains a PK component of the parent entity. For example, the definitions of the COURSE and CLASS entities COURSE(CRS_CODE, DEPT_CODE, CRS_DESCRIPTION, CRS_CREDIT)
CLASS(CRS_CODE, CLASS_SECTION , CLASS_TIME, ROOM_CODE, PROF_NUM)
indicate that a strong relationship exists between COURSE and CLASS, because the CLASS entity's composite PK is composed of CRS_CODE + CLASS_SECTION. (Note that the CRS_CODE in CLASS is also the FK to the COURSE entity.) The Crow's Foot notation depicts the strong (identifying) relationship with a solid line between the entities. Whether the relationship between COURSE and CLASS is strong or weak depends on how the CLASS entity's primary key is defined. Keep in mind that the order in which the tables are created and loaded is very important. For example, in the "COURSE generates CLASS" relationship, the COURSE table must be created before the CLASS table. After all, it would not be acceptable to have the CLASS table's foreign key reference a COURSE table that does not yet exist.



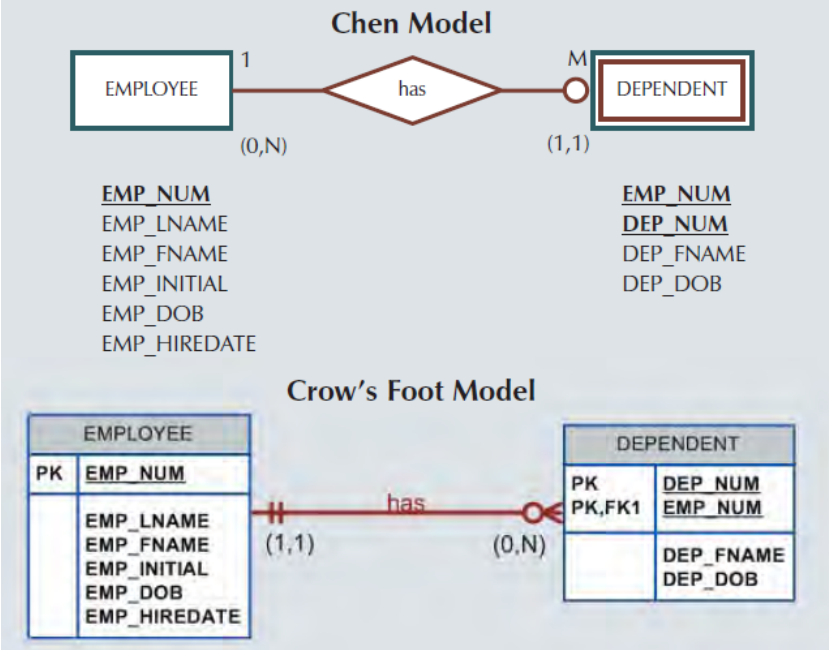
Crow's Foot notation depicts a strong relationship

Weak Entities: A weak entity is one that meets two conditions:

- The entity is existence-dependent; that is, it cannot exist without the entity with which it has a relationship.
- The entity has a primary key that is partially or totally derived from the parent entity in the relationship.

For example, a company insurance policy insures an employee and his/her dependents. For the purpose of describing an insurance policy, an

EMPLOYEE might or might not have a DEPENDENT, but the DEPENDENT must be associated with an EMPLOYEE. Moreover, the DEPENDENT cannot exist without the EMPLOYEE; that is, a person cannot get insurance coverage as a dependent unless s(he) happens to be a dependent of an employee. DEPENDENT is the weak entity in the relationship "EMPLOYEE has DEPENDENT."



Note that the Chen notation above identifies the weak entity by using a double-walled entity rectangle. The Crow's Foot notation generated by Visio Professional uses the relationship line and the PK/FK designation to indicate whether the related entity is weak.

A strong (identifying) relationship indicates that the related entity is weak. Such a relationship means that both conditions for the weak entity definition have been met—the related entity is existence-dependent, and the PK of the related entity contains a PK component of the parent entity. Remember that the weak entity inherits part of its primary key from its strong counterpart. For example, at least part of the DEPENDENT entity's key shown in Figure above was inherited from the EMPLOYEE entity:

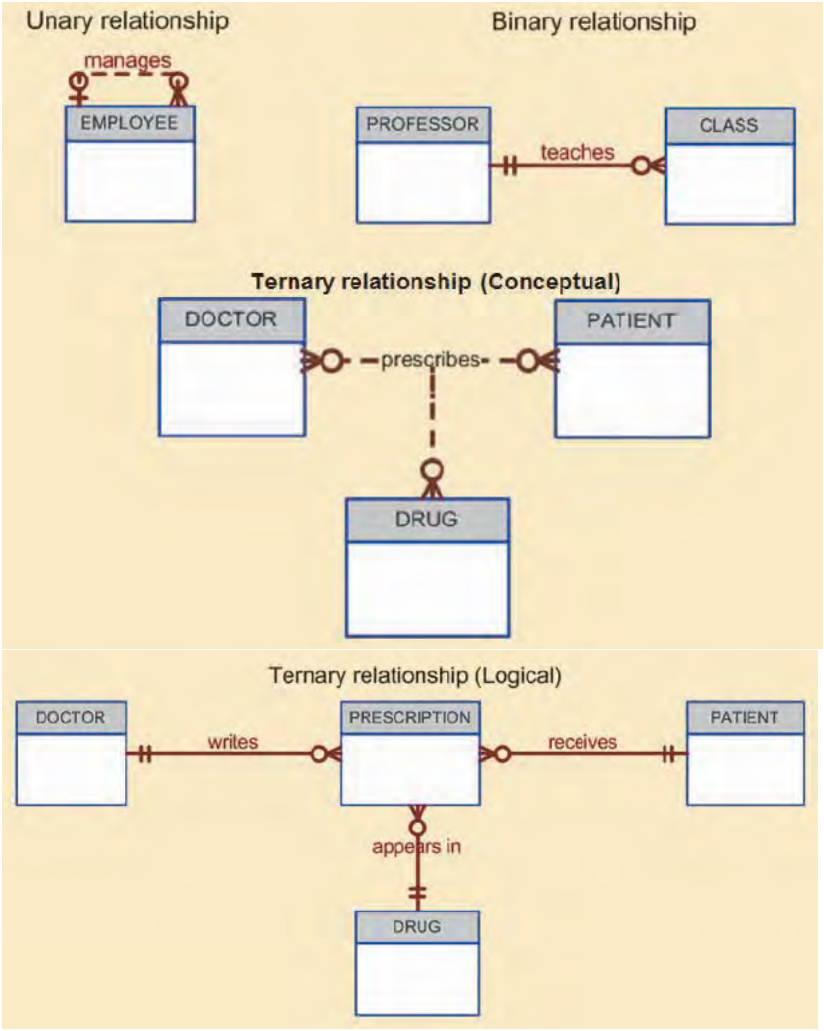
EMPLOYEE (EMP_NUM, EMP_LNAME, EMP_FNAME, EMP_INITIAL, EMP_DOB, EMP_HIREDATE)

DEPENDENT (EMP_NUM, DEP_NUM, DEP_FNAME, DEP_DOB)

CROW'S FOOT SYMBOL	CARDINALITY	COMMENT
	(0,N)	Zero or many. Many side is optional.
	(1,N)	One or many. Many side is mandatory.
	(1,1)	One and only one. 1 side is mandatory.
	(0,1)	Zero or one. 1 side is optional.

Crowfoot symbols

- **Relationship Degree:** A relationship degree indicates the number of entities or participants associated with a relationship. A unary relationship exists when an association is maintained within a single entity. A binary relationship exists when two entities are associated. A ternary relationship exists when three entities are associated. Although higher degrees exist, they are rare and are not specifically named. (For example, an association of four entities is described simply as a four-degree relationship.)



THREE TYPES OF RELATIONSHIP DEGREE

- UNARY RELATIONSHIPS: IN THE CASE OF THE UNARY RELATIONSHIP SHOWN ABOVE, AN EMPLOYEE WITHIN THE **EMPLOYEE** ENTITY IS THE MANAGER FOR ONE OR MORE EMPLOYEES WITHIN THAT ENTITY. IN THIS CASE, THE EXISTENCE OF THE "MANAGES" RELATIONSHIP MEANS THAT **EMPLOYEE** REQUIRES ANOTHER **EMPLOYEE** TO BE THE MANAGER—THAT IS, **EMPLOYEE** HAS A RELATIONSHIP WITH ITSELF. SUCH A RELATIONSHIP IS KNOWN AS A RECURSIVE RELATIONSHIP.
- BINARY RELATIONSHIPS A BINARY RELATIONSHIP EXISTS WHEN TWO ENTITIES ARE ASSOCIATED IN A RELATIONSHIP. BINARY RELATIONSHIPS ARE MOST COMMON. IN FACT, TO SIMPLIFY THE CONCEPTUAL DESIGN, WHENEVER POSSIBLE, MOST HIGHER-ORDER (TERNARY AND HIGHER) RELATIONSHIPS ARE DECOMPOSED INTO APPROPRIATE EQUIVALENT BINARY RELATIONSHIPS.
- TERNARY AND HIGHER-DEGREE RELATIONSHIPS: ALTHOUGH MOST RELATIONSHIPS ARE BINARY, THE USE OF TERNARY AND HIGHER-ORDER RELATIONSHIPS DOES ALLOW THE DESIGNER SOME LATITUDE REGARDING THE SEMANTICS OF A PROBLEM. A TERNARY RELATIONSHIP IMPLIES AN ASSOCIATION AMONG THREE DIFFERENT ENTITIES.

EXAMPLE

MR BRANDON'S THE OWNER OF **SPEED CAFÉ** HAS BEEN HAVING PROBLEMS WITH THE MANAGEMENT OF HIS CAFÉ. HAVING LEARNT THAT YOU ARE A **DB** DESIGNER, HE BELIEVES HE HAS FINALLY FOUND A SOLUTION. HE HAS ASKED YOU TO AUTOMATE THE MANAGEMENT OF HIS CAFÉ. SINCE THIS WILL INVOLVE A DATABASE BACKEND, YOU ARE SADDLED WITH THE TASK OF SHOWING HIM A GOOD DATABASE MODEL BASED ON THE FOLLOWING BUSINESS RULES:

- THE CAFÉ HAS SEVERAL EMPLOYEES EACH HAVING A UNIQUE IDENTIFICATION NUMBER, NAMES AND DATES OF BIRTH.
- AN EMPLOYEE IS EITHER A "TECHNICAL OFFICER" OR "CASUAL EMPLOYEE", BUT NOT BOTH. A TECHNICAL OFFICER HAS ACCESS TO ONE OR MORE COMPUTING FACILITIES IN

the Café and therefore has login usernames and passwords. Technical officers have varying salary rates based on their ranks. Casual Employee however, do not have access to computing facilities and their salaries are wages (i.e. based on the number of hours worked).

- All Computing facilities in the Café have names (e.g. computer, cable, printer etc.) and date of purchase (remember names are not unique, so you will have to choose a surrogate key).
- Access to Internet facilities in the Café (either by a staff or customer) is through a ticket. Each ticket has a unique ticket number, duration, time of production, period (number of days) of validity and price in Naira.

Draw an implementation oriented **ER** diagram for **SPEED CAFÉ** database indicating necessary connectivities, cardinalities and participation constraints (relationship strengths). You can state the necessary assumptions made.