COMP2411

Tutorial 2 (Week 3)

Q1.

1. Explain the difference between physical and logical data independence.

Basic Concepts and Terminologies

- *Data Independence*
  - the ability to modify a schema definition in one level without affecting a schema in the next higher level
  - there are two kinds (a result of the 3-level architecture):
    - ***physical data independence***
      - *-- the ability to modify the physical schema without altering the conceptual schema and thus, without causing the application programs to be rewritten*
    - ***logical data independence***
      - *-- the ability to modify the conceptual schema without causing the application programs to be rewritten*

Example?

Page 14, Lecture Notes 1.

Q1.

2.2 Three-Schema Architecture and Data Independence

Three of the four important characteristics of the database approach, listed in Section 1.3, are (1) use of a catalog to store the database description (schema) so as to make it self-describing, (2) insulation of programs and data (program-data and program-operation independence), and (3) support of multiple user views. In this section we specify an architecture for database systems, called the **three-schema architecture**,[9] that was proposed to help achieve and visualize these characteristics. Then we discuss the concept of data independence further.

2.2.2 Data Independence

The three-schema architecture can be used to further explain the concept of **data independence**, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.5(a) should not be affected by changing the GRADE_REPORT file (or record type) shown in Figure 1.2 into the one shown in Figure 1.6(a). Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before.

   Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval speed of section records (Figure 1.2) by semester and year should not require a query such as *list all sections offered in fall 2008* to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

Elmasri, R. & Navathe, S.B. (2011). *Fundamentals of Database Systems: Chapter 2 Database System Concepts and Architecture* (6th ed.). Pearson.

Q1.

- **Logical level**. The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

1.3.2 Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

We study languages for describing schemas after introducing the notion of data models in the next section.

Silberschatz, A., Korth H.F. & Sudarshan, S. (2011). *Database System Concepts: Chapter1 Introduction* (6th ed.). McGraw-Hill.

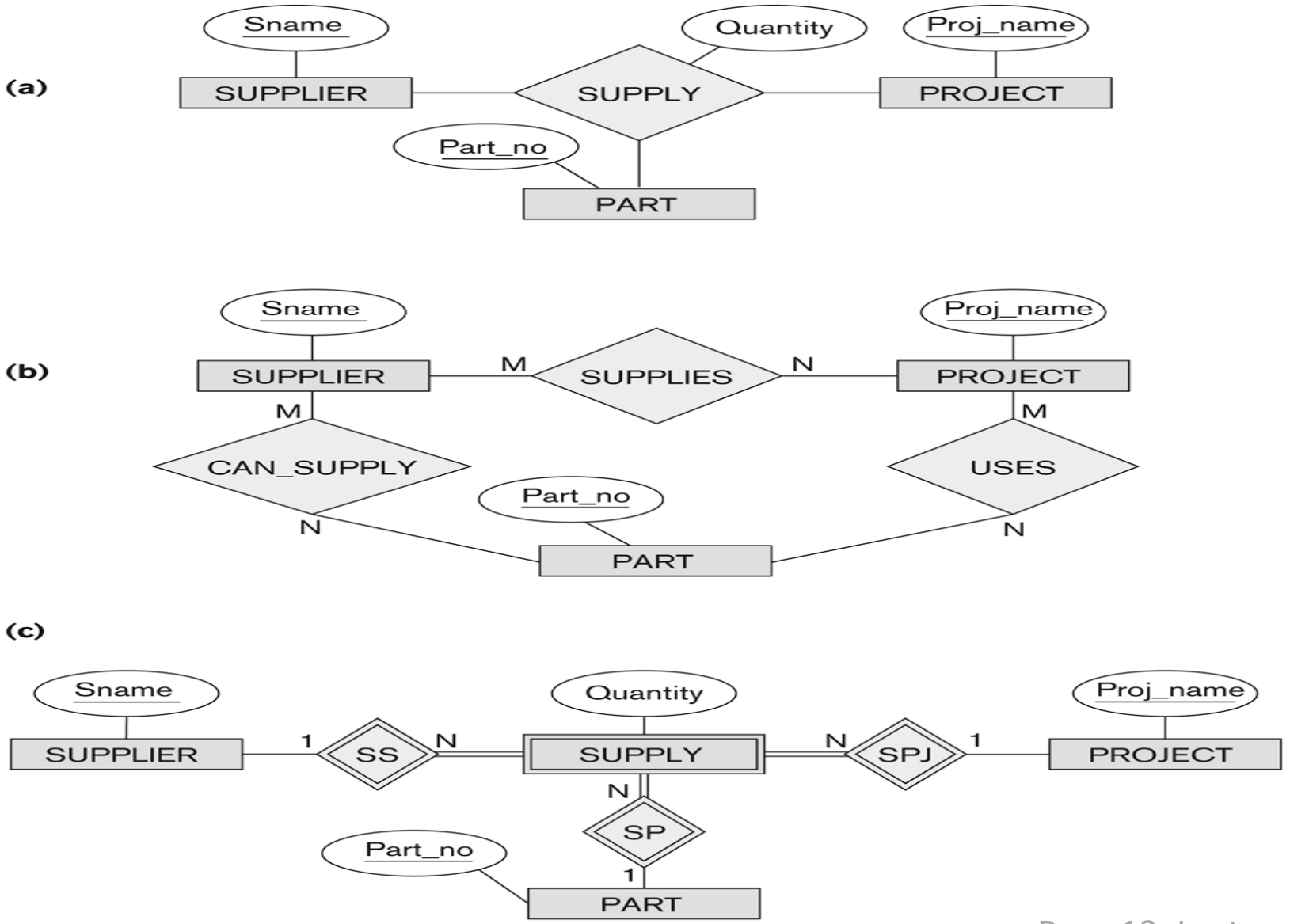
Q2.

2. Discuss the need to have "ternary" relationships in ER design:

   a) Is it possible to replace a ternary relationship by a set of binary ones? Why or why not?

   b) For a ternary relationship R, if one of the entity sets linked by R is a weak entity set, how do we convert R into a table?

Q2.



Page 13, Lecture Notes 2.

Figure 3.17
Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

Q2.

3.9 Relationship Types of Degree Higher than Two

In Section 3.4.2 we defined the **degree** of a relationship type as the number of participating entity types and called a relationship type of degree two *binary* and a relationship type of degree three *ternary*. In this section, we elaborate on the differences between binary and higher-degree relationships, when to choose higher-degree versus binary relationships, and how to specify constraints on higher-degree relationships.

3.9.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

The ER diagram notation for a ternary relationship type is shown in Figure 3.17(a), which displays the schema for the SUPPLY relationship type that was displayed at the instance level in Figure 3.10. Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where the meaning is that s is a SUPPLIER who is currently supplying a PART p to a PROJECT j. In general, a relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type.

Figure 3.17(b) shows an ER diagram for three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents different information than do three binary relationship types. Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (s, p) whenever supplier s *can supply* part p (to any project); USES, between PROJECT and PART, includes an instance (j, p) whenever project j uses part p; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (s, j) whenever supplier s supplies *some part* to project j. The existence of three relationship instances (s, p), (j, p), and (s, j) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply that an instance (s, j, p) exists in the ternary relationship SUPPLY, because the *meaning is different*. It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree n or should be

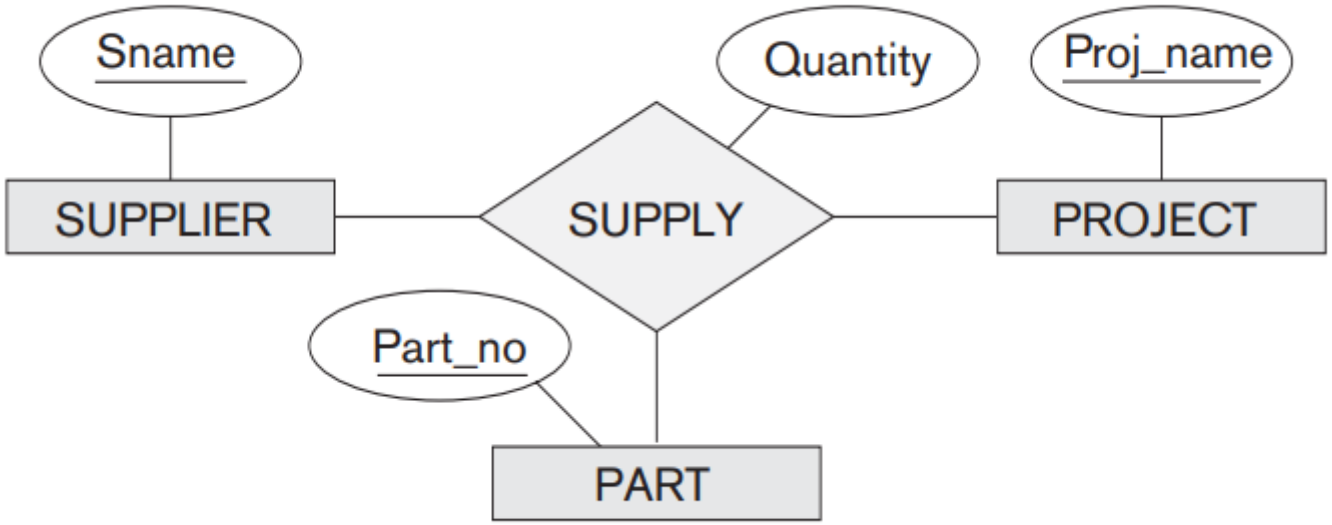
Elmasri, R. & Navathe, S.B. (2011). *Fundamentals of Database Systems: Chapter 2 Database System Concepts and Architecture* (6th ed.). Pearson

Q3.

3. Design an ER schema (diagram) for a database application that you are familiar with. Indicate all constraints that you know should hold on the database, either directly on the diagram if appropriate or in text.

   (Preferably, the schema is a reasonably rich one, e.g., it has at least 5 entity types, 4 relationship types, one weak entity type, and an n-ary (n > 2) relationship type. This diagram will be used in subsequent tutorials.)

Q3.



Could we add more entities in this diagram?