Lecture Notes on Analysis & Design of Accounting Databases

Jagdish S. Gangolly[1]
Department of Accounting & Law
State University of New York at Albany

September 24, 2003

PREFACE

> The main object of teaching is not to give explanations, but to knock at the doors of the mind. If any boy is asked to give an account of what is awakened in him by such knocking, he will probably say something silly. For what happens within is much bigger than what comes out in words. Those who pin their faith on university examinations as the test of education take no account of this.
>
> Rabindranath Tagore

These notes are prepared exclusively for the benefit of the students in the course *Acc 682 Analysis & Design of Accounting Databases* in the Department of Accounting & Law at the State University of New York at Albany, and are not to be used by others for **any** purpose without the express permission of the author.

In these notes, I consider only Relational and Object-Relational Database Management Systems, and therefore do not deal with other DBMSes such as Hierarchical, Network, or pure Object databases. This should not be a major drawback in as much as the bulk of DBMSes used for accounting in the real world today are relational.

I make use of much of the materials in the text for the course without explicit reference. The text is,

- *A First Course in Database Systems, 2nd. ed* by Jeffrey D. Ullman and Jennifer Widom (Prentice Hall, 2002)
- *Programming in Prolog, 4th ed.* by W.F. Clocksin and C.S. Mellish (Springer-Verlag, 1994)

I shall be adding to these notes as we go along. You can download the file and print the pages that you need. You will find the instructions for viewing postscript files on the course homepage at

http://www.albany.edu/acc/courses/acc682.fall2003/

Jagdish S. Gangolly
Albany, NY 12222

Contents

Chapter 1

Introduction

Endless invention, endless experiment,
Brings knowledge of motion, but not of stillness;
Knowledge of speech, but not of silence;
Knowledge of words, and the ignorance of the word.
All our knowledge brings us nearer to our ignorance,
All our ignorance brings us nearer to death,
But nearness to death no nearer to God.
Where is the life we have lost in living?
Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?
The cycles of heaven in twenty centuries
Bring us farther from God and nearer to the Dust.

From *Choruses from "The Rock"*, T.S.Eliot

In this note I shall describe the traditional file-oriented design of accounting systems, discuss their drawbacks & how the database systems alleviate the problems with such systems, describe two views of database systems (conceptual and architectural), and finally discuss the desirable properties of database systems.

1.1 File-Oriented Accounting Systems

Traditionally, accounting systems were organised around the transaction cycles, and so were comprised of subsystems such as Billing/Sales/Accounts Receivable, Purchase & Accounts Payable, Cash & Treasury, Conversion & Production, Budgeting & Standard Costing, Payroll, etc. Often, such subsystems were built without an overall architecture for either the system as a whole or the data in the system. Such systems, often called traditional file-oriented systems, are illustrated in Figure 1.1.

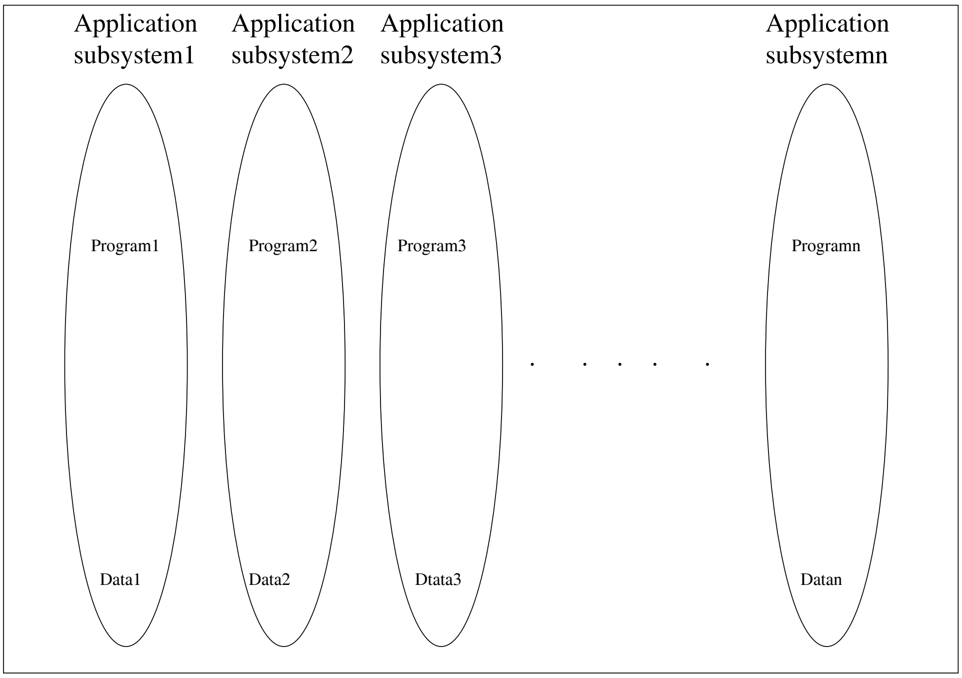


Figure 1.1: A Traditional File-Oriented Accounting System

In the traditional file-oriented system, each individual application "owns" its data, and this ownership relationship leads to certain anomalies.

- *data dependence*, ie., the data is application dependent. There are two consequences of this data dependence.
  - Since the data is owned by the applications in a traditional file-oriented accounting system, the specifications of the data will be embedded in the application programs. That being the case, *should an application change, so must the data.* Since accounting operates in a dynamic environment, such changes are needed often, and it can be quite costly to change data.

  - Since the data is "owned" by the application, it is likely to be defined to suit the needs of such an application. If the needs of the application diverges from those of other applications that might need the same data, in the absence of an enterprise-wide unified view of data, semantically equivalent data can be represented in different ways by the various applications, leading to problems in comprehending data between applications.

- *Data redundancies.* Since each application owns its data, when more than one application needs the same data, the the following anomalies arise in the absence of an enterprise-wide model for data:

  - *Lack of uniformity of meaning of data*, with the result semantically equivalent data can be represented in different ways leading to a lack of a unified view of data for the organisation as a whole. In the presence of diverse needs for data across applications, each application will represent its data in ways deemed optimal from its own point of view – a situation that can be sub-optimal from the point of view of the enterprise as a whole.
  - In the absence of uniformity of data, when the same data is needed by more than one application, applications must either obtain the data "owned" by other applications and translate it to suit its needs, or maintain duplicate/redundant data to avoid such translation. Neither solution is optimal; the former solution entails unnecessary programming efforts, the latter solution entails data redundancies and associated unnecessary data storage & data inconsistencies.
  - Since there is likely to be lack of a unified enterprise-wide view of meaning of data, *it is difficult to enforce standards, and hence the integrity of data is difficult to maintain.*

- *Difficulties in data access by the users.* Since the users must access the data through programs which have data structures embedded in them, it puts additional burden on the users in terms of the need for their understanding *how* the data is stored (in addition to their understanding of the *meaning* of such data). The problem is compounded by the absence of a unified enterprise-wide model for meaning of data and the consequent lack of standardisation of the semantics of data.

1.2 Databases

Databases were developed in order to alleviate some of the problems referred to above. In the design and use of databases, there is a presumption that the data is shared by all the applications in the enterprise, and therefore it is necessary to have a unified view of data from the point of view of its semantics as well as representation. This view leads to data independence because the concept of data "ownership" is replaced by one of data "sharing".

Databases reduce data redundancies, and it is possible to develop & enforce standards for data. This leads to enhanced integrity of data. Since there is one repository of data in the organisation, it is also easier to enforce security measures for corporate data.

Another advantage of databases is data abstraction which ensures that the users of the database do not need to be concerned with *how* the data is organised, but just *what* the data *means*. In the traditional file-oriented accounting systems, since the specification of data is embedded in the application programs, to extract useful information, the users need to know *how* the data is organised (data structures used) in addition to the *meaning* of data; in the database systems the users need to know just *what* the data means.

In summary, databases provide the following advantages over the traditional file-oriented accounting systems:

- *Data independence*
- *Reduced data redundancies*
- *uniformity of meaning of data*
- *Enforcement of standards & security over data, and enhanced integrity of data*
- *Data abstraction*

We can study two different aspects of database systems: conceptual view and the architectural view.

1.2.1 Conceptual view

In database systems, there is a conceptual schema which defines *what* the data is. Since ultimately all data is stored on data storage devices (such as hard disks), it is necessary to have a physical schema that describes *how* the data is to be stored on the storage devices. The conceptual schema is translated into the physical schema in order to facilitate storage of data.

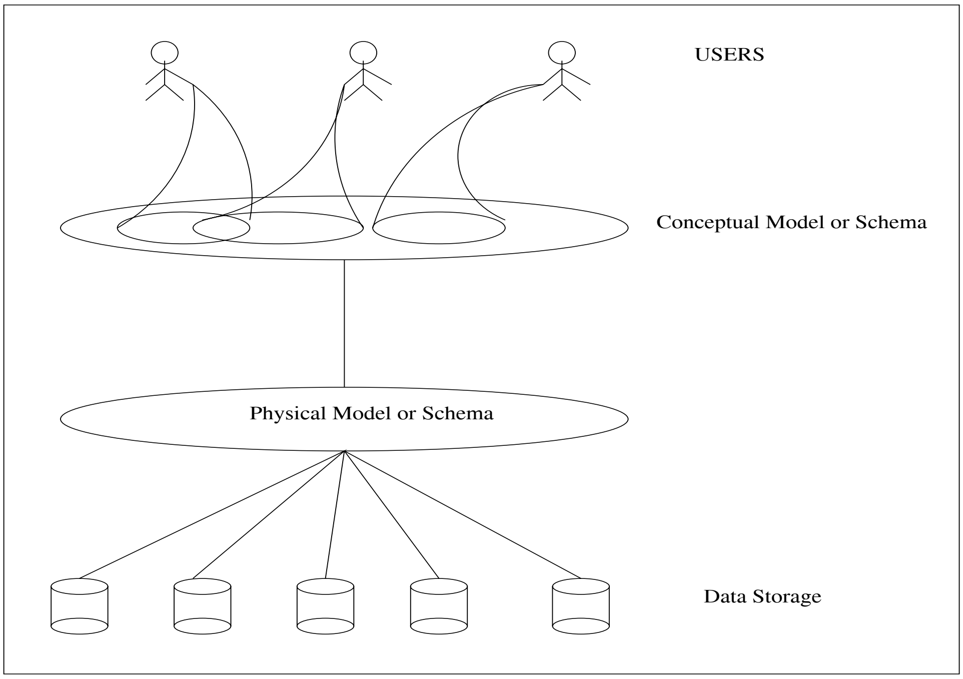


Figure 1.2: Conceptual View of a Database System for Accounting

The users view the subset of the database that is relevant to the queries that they need answered. While the subsets of the database that different users view are not mutually exclusive, the common data they view are not necessarily stored separately as can occur in the traditional file-oriented accounting systems. This is illustrated in Figure 1.2.

1.2.2 Architectural View

The four main components of a database management system are *Query Processor*, *Transaction Manager*, *Storage Manager*, and *Data & Metadata*. The Figure 1.3 shows the architectural view of a database system.

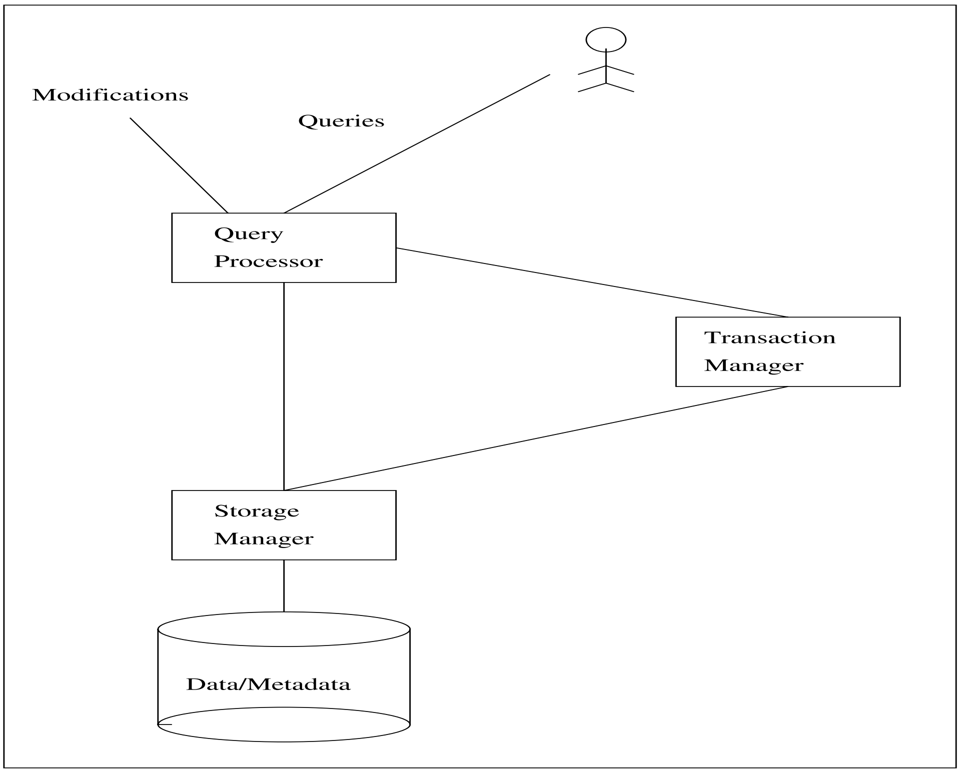


Figure 1.3: Architectural View of a Database System for Accounting

There are two possible inputs to a query processor: *queries* which are questions that the users need answered by the database management system, and *modifications* which are either modifications that need to be made to the schema or modifications that need to be made to the data. User queries can be input to the database system either through a *generic query interface* through SQL (Structured Query Language) queries, or through application programs written by programs that make calls to the Database system through the *Application Programming Interface (API)* provided by the Database system vendor.

The commands for schema modifications, since they modify the model of data (conceptual or physical), are usually issued only by database administrators. The commands for modification for data can be issued either through the generic query interface, or through an application program that makes calls to DBMS via the API.

The transaction Manager ensures that different queries are run simultaneously without compromising the integrity of the answers generated by the database system and that of the database itself. It also interfaces with the Query Manager and the Storage Manager to ensure that the queries are handled in such a way that there are no conflicts that can compromise the integrity. This is accomplished by locking of items requested by queries, managing the locks, and the resolution of deadlocks if and when they occur.

The storage manager manages the memory pertaininmg to the database on the disks *file manager* as wellas the main memory *buffer manager*. It manages the movement of data between the disks and the buffer in a way to ensure the integrity of the database.

The database itself consists of data itself, and *metadata*, ie., information about the data in the database. While the data itself consists of information pertaining to transactions, metadata consists of names of relations & attributes, the relational schema (the *signature* of relations in the database that specify the attributes of each relation), the datatypes for each attribute, and the indexes that are maintained in the database in order to facilitate efficient retrieval of information from the database.

1.3 Database Integrity & ACID Properties

Database systems are often a mission-critical applications for large corporations, and in fact large corporations could not survive for long if there were sustained interruptions to their database systems, or if the integrity of their databases is compromised. In order to assure that the database systems maintain the integrity of corporate databases, the transaction managers must have the properties of *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

Atomicity requires that a transaction is recorded in its entirety or not at all. Failure of a database system to enforce this property could leed to trial balances that do not balance, or subsidiary ledger balances that do not add up to the control account balances, if they were reconstructed using the database.

Consistency requires that after each transaction is recorded, the database state is consistent. For example, the duality (debit/credit) aspect of double-entry is not compromised so that if the books were closed at any time they would balance (both in terms of debits & credits and the reconciliation of subsidiary books and control ledgers, if the books wewre reconstructed using the database).

Isolation requires that when more than one transaction is executed simultaneously, the effects of the transactions are isolated from each other and effect is as though the transactions are run sequentially in the sequence consistent with the business operations that the database supports.

Durability requires that once the *database transaction* is ready to *complete* (*committed*) and recorded in the *log*, the changes should not be lost even if there is a system failure.

The above properties are implemented in database management systems by *Locking*, *Logging*, and *Transaction commit*.

Chapter 2

Modeling of Databases

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
>
> Alan Perlis

2.1 Introduction

In this chapter, I provide a description of terminology for modeling databases, discuss the *Object Definition Language (ODL)* in which object-oriented databases can be specified, discuss the *Entity-Relationship Diagrams (ERD)* in which relational databases are specified, and finally discuss the design principles for data models.

2.2 Database Modeling

Design of databases involves identification of the various objects of interest (called *Entities*), the characteristics that describe those objects (called *attributes*), the associations between those objects (called *relationships*), determining the structure of those attributes & relationships and ways to represent them. The relational database model, as we shall see, requires us to make compromises since it permits only certain representations, whereas the Object Definition Language allows us to represent the databases with all the richness that we see in real world business practice. Therefore, we will first study the ODL, then the ERD, and finally see how we can translate ODL specifications into relational specifications so that all the decisions involving compromises made are explicitly.

The starting point for the design of an accounting database is usually a clear description of *what* the database is expected to contain and not what is *done* to the contents of such database. One sometimes finds elements of this description in the audit workpapers prepared for the purpose of studying the internal control structure of a client corporation.

Consider the following description. I will illustrate the basic terminology of database modeling using this example, and then proceed to illustrate how the database is specified in ODL and ERD.

An Example: *Airline Reservation System*

Sununu Airlines flights never make intermediate stops. The `PASSENGER`s call the flight reservation system with information on their *name*, *address*, and *phoneNumber*. The *address* in turn consists of the *streetNumber*, *streetName*, *city*, *state*, and *zipCode*. The reservation assistants make the `RESERVATION` for the passengers. The reservation information includes the name of the passenger, *itinerarySource*, *itineraryDestination*, and the information on each `FLIGHT` on the itinerary. Each flight is assigned a *flightNumber*, and has scheduled *source*, *destination*, *departureTime*, and *arrivalTime*.

Passengers may make reservations on a number of flights, and a flight may contain many passengers depending on the capacity of the `AIRCRAFT` assigned to the flight. Sununu Airlines operates thropugh a fleet of aircraft, each of which has a *modelNumber*, *serialNumber*, and has a given passenger *capacity*. Sununu assigns one aircraft to each flight, but an aircraft can be assigned to many flights.

Sununu has organised its flight personnel consisting of `PILOT`s and `ATTENDANT`s into `CREW`s so that a pilot or an attendant is assigned to precisely one crew, but a crew can have many pilots and attendants. The crew operates as a team, and is assigned a *crewNumber*. Each pilot has a *pilotNumber*, *pilotName*, *pilotAddress*, and a *pilotDateOfHire*. Similarly each attendant has an *attendantNumber*, *attendantName*, *attendantAddress*, and an *attendantDateOfHire*.

The above description could have been derived from a study of the audit workpapers pertaining to the documentation of the auditor's understanding of the internal control structure, or could have been compiled on the basis of information gathered by questioning the employees, study of the client system documentation, the documentation provided by the client's repository, and similar sources.

In the above description we can see certain classes of objects, such as the `PASSENGER`, `FLIGHT`, `AIRCRAFT`, `PILOT`, `ATTENDANT`, `RESERVATION` and `CREW`. These are classes of interest about whom the database must maintain data. We can also see that each object belonging to a class can be described by certain characteristics or *attributes*. The PASSENGERs can be described by their *name*, *address*, *phoneNumber*, *itinerarySource*, and *itineraryDestination*; the FLIGHTs. can be described by their *flightNumber*, and has scheduled *source*, *destination*, *departureTime*, and *arrivalTime*; and so on. Both the objects and their attributes are *nouns*. While both objects and attributes are nouns, it is important to be able to distinguish between them; what is an attribute in one situation may be an object in another. For example, *color* may be an attribute from the point of view of the database designer of an automobile manufacturer, but an object from the point of view of a chemical company manufacturing paints. It is important to develop the facility to discriminate between objects and attributes in the design of a database.

You also will notice verbs or verb phrases such as such as `bookOn`, `assignedTo` for aircraft & flights, `assignedTo` for flights & crew, and `assignedTo` for pilots/attendants & crew. These are *relationships* between two or more objects. It is important to be cautioned that often, nouns are used as verbs (and vice versa).

For the rest of this chapter I shall discuss two ways of representing the database: ODL and ERD. I shall not explicitly discuss data structures here. The discussions in the class and those in Acc 681 should suffice.

2.3 Object Definition Language (ODL

Query languages for databases consist essentially of two parts. The first, usually called *Data Definition Language (DDL)* provides a vehicle for specifying the definition of data. The second, usually called *Data Manipulation Language (DML)* provides a vehicle for specifying the manipulation of data. ODL, a proposed standard language for specifying the definition of data in the object-oriented framework, is pretty close to the syntax of popular languages for developing information systems in general (such as C++, smalltalk, Java).

In ODL, we specify the structure of the databases in terms of the specifications of classes, attributes of objects in those classes, and the relationships between objects in different classees. We can diagrammatically represent them as in Figure 2.1.

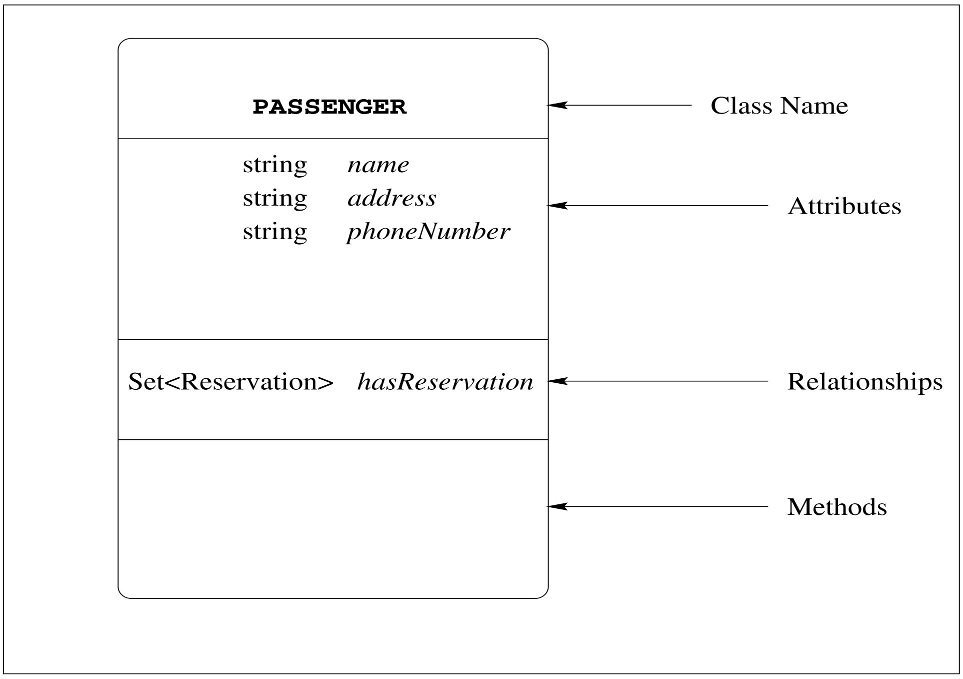


Figure 2.1: `PASSENGER`*Class*

Objects are classified into classes such that objects in any class share the same *properties*. The properties of objects consist of their attributes, relationships with other objects, and methods. The attributes describe the objects. For example, a `PASSENGER` can be described by the attributes *name*, *address*, and *phoneNumber*. Relationships are associations between objects. For example, There is a relationship names *hasReservation* between a `PASSENGER` and a *set* of `RESERVATION` objects, ie., there are reservations on a set of flights (presumably on flights that collectively let the passenger travel from the *itinerarySource* to the *itineraryDestination*).

Class is a set concept, and so the definition of a class must be unambiguous, ie., an object is either a member of the class or it is not. Since no two points in a set can be identical, each object has an identity, and so even if two objects are identical in terms of all properties, their identities are separate. For example, the `PASSENGER` class consists of all the passengers in the database. By the mere fact of any one being a passenger, (s)he shares all the properties of any other passenger (for example, has a name, address, phone number, has a relationship with a set of reservations, etc.)

```
interface <class name> {
<list of properties>
}
```

For example, we can write the ODL statement for the class `PASSENGER` as below

```
interface PASSENGER {
    attribute string name;
    attribute string address;
    attribute string phoneNumber;

    relationship Set<RESERVATION> hasReservation
        inverse RESERVATION::reservedBy;

}
```

`interface` is a keyword in ODL language for the declaration of a class. In the example above we declare a class whose name is `PASSENGER`. Every object belonging to this class has attributes `name`, `address` and `phoneNumber`. Also each such object has a relationship named `hasReservation` with a *set* of objects belonging to the class `RESERVATION`

Unlike in ERDs, as we shall see shortly, in ODL, both relationships and their inverses need to be specified in the declarations. For example, the relationship between `PASSENGER` and `RESERVATION` is given by the fact that corresponding to any `PASSENGER` there may be a *set* of `RESERVATION`s. The inverse relationship between `RESERVATION` and `PASSENGER` is given by the fact that corresponding to a RESERVATION, there is precisely one `PASSENGER` that the reservation pertains to. The distinction between a relationship and its inverse becomes apparent if one states the relationship in English language in two sentences in active and passive voice respectively.

To indicate that corresponding to the relationship `hasReservation` there is an inverse relationship named `reservedBy` specified in the declaration of the class `RESERVATION`, we use the class-scope. operator symbol `::`.

We can examine the relationships between class objects in terms of their *multiplicities* (sometimes referred to as *cardinalities*). Figure 2.2 illustrates the concept of multiplicities. In the figure, the sets representing the classes are shown as ovals with the class names inscribed in them. Subsets of the classes are shown as ovals inside the classes, and relationships are shown as lines connecting the subsets of classes and are labelled with relationship names. For example, the relationship between the classes PASSENGER and RESERVATION has multiplicity one-to-many as can be seen by describing the relationship in the following two sentences:

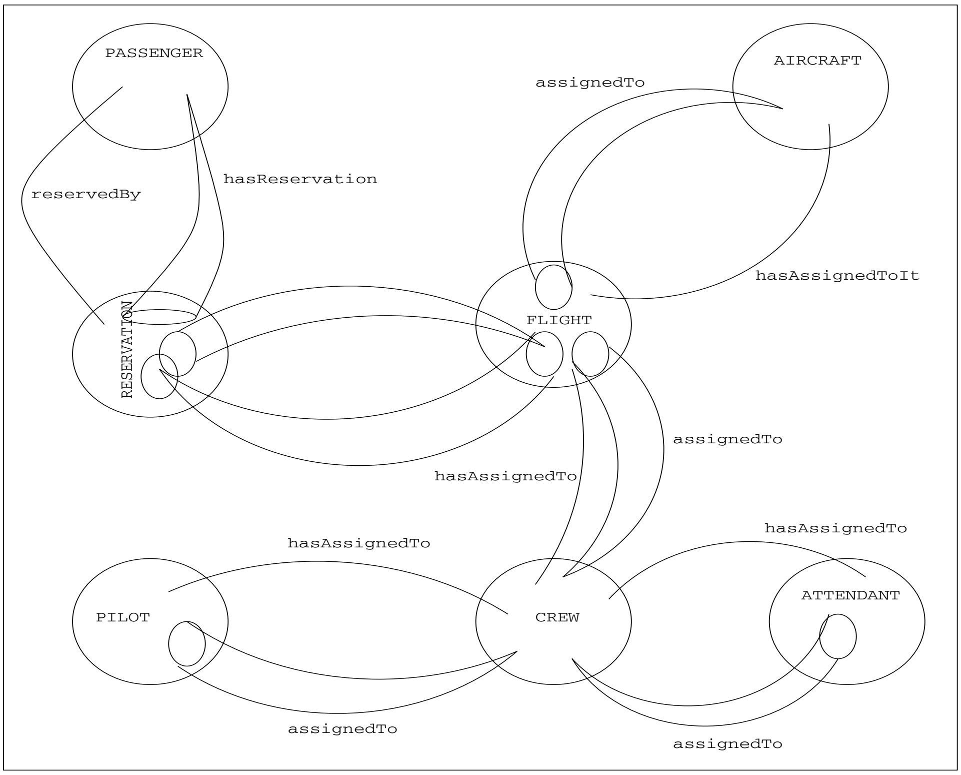


Figure 2.2: Multiplicities of relationships

A PASSENGER has reservation.
The itinarary in a RESERVATION is reserved by a PASSENGER.

Consider a particular PASSENGER. (S)he may have made a number of reservations (one for each itinarary), but if you consider a specific RESERVATION it could have been made only by a unique PASSENGER. This illustrated in the Figure 2.2. The multiplicity of the relationship between RESERVATION and FLIGHT, on the other hand, is many-to-many, as can also be seen by a similar interpretation of Figure 2.2.

2.3.1 Types in ODL

The type system in ODL is built from two basic types: *atomic types* (integer, float, character, character string, boolean, and enumerations), and *interface types* (such as the classes in our airline example). Structured types are built using these basic types by using the collection types (sets, bags, lists, and arrays) as type constructors.

Attribute types are built starting with atomic types (or structures whose fields are atomic types) and applying type constructors. Relationship types are built by applying type constructors to an interface type.

Interfaces can not appear in the type of an attribute, and atomic types can not appear in the type of a relationship.

2.4 Entity-Relationship Model

The Entity-Relationship model is *value oriented* unlike the *object oriented* model. In the object model, each object has an identity independent of the values taken by the attribute variables. On the other hand, in the entity-relationship model, you can not distinguish individual entities by appeal to their identities; such identities independent of the attribute values do not exist. Since entity set is a set concept, it is crucial that no two entities belonging to a set have identical values for *all* attribute variables. This means that there must be at least one attribute on whose values any two entities belonging to an entity set differ. The set of attributes on which any two objects in an entity set *must* differ is called the *key* of the entity set. The individual attributes that belong to the key are called *key attributes*. It should be obvious that the key uniquely identifies an entity. In fact, in the entity-relationship model, it is the value of the key that enables one to distinguish one entity from another.

While the Entity-Relationship model in some ways resembles the Object model, as we shall see, there are significant differences. In the Entity-Relationship model, we represent entities in rectangles, relationships in diamonds, and multiplicities of the relationships by lines or arrows that connect the relationships with the entities. The ERD for the airlines example is illustrated in Figure 2.3.

Unlike in ODL, Relationships are represented in the ERDs one way only, ie., inverse relationships are not represented. Multiplicities of the many-to-one, one-to-many, and one-to-one kind are represented by the arrow as can be seen in the figure.

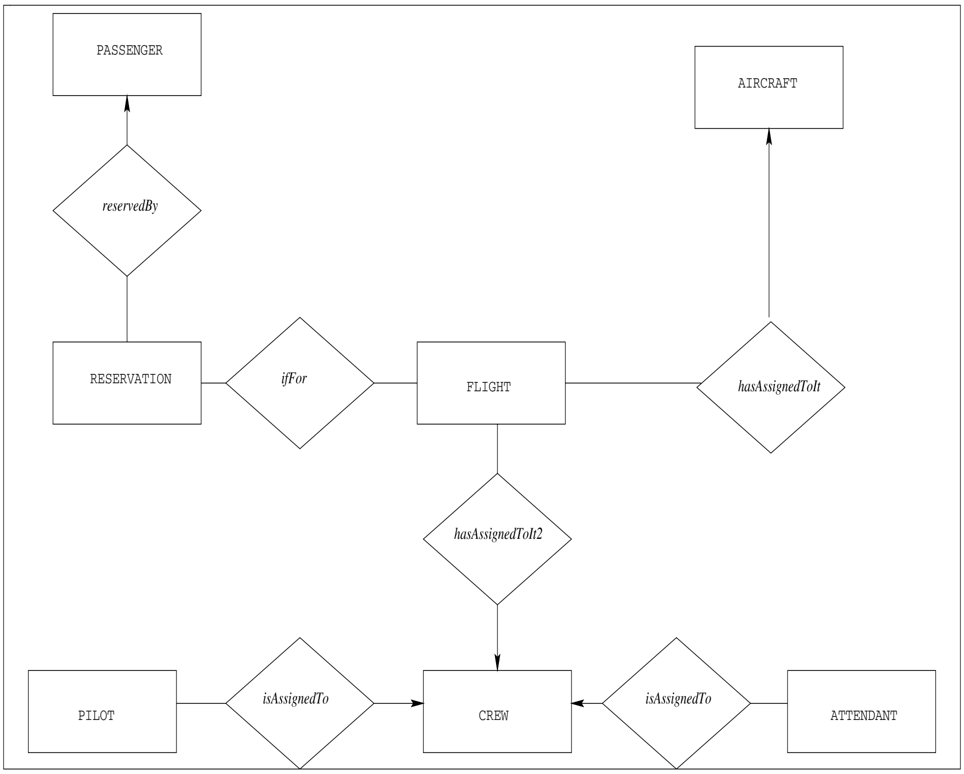


Figure 2.3: Entity-Relationship Diagram for the Airline Example

2.4.1 Binary and Multiway Relationships

Sometimes, when English language descriptions od databases are translated into ERDs, we can have relationships that relate to more than two entity sets. Such relationships are called *multiway relationships*. Such ERDs are difficult to interpret, and multiplicities can be ambiguous at best.

Consider the following example and the corresponding ERD in Figure 2.4.

Example:

> Drivers make delivery of products to customers using trucks.

ERDs in which every relationship is binary, ie., relationship that associates precisely two entity sets are relatively unambiguous, and their multiplicities are also unambiguous. Any ERD containing relationships which are multiway can be converted to ERDs containing only binary relationships by suitable conversion of multiway relationships into entity sets. For example, in the *Delivery example*, we have the ERD given in Figure 2.5.

2.4.2 Weak Entity Sets

The function of the entity set `DELIVERY` is to relate the four entity sets `CUSTOMER`, `TRUCK`, `DRIVER`, and `PRODUCT`. Individual entities belonging to the `DELIVERY` entity set do not have an existence independent of the entity sets that they relate to, and therefore are called *weak entities* shown in enclosed rectangles in the Figure 2.5. Since entity set is *set* concept and so members belonging to that set must differ on the value of at least one attribute, weak entity sets borrow the key attributes of the entity sets that help them establish their identity. The *many-to-one* relationships with such other entity sets (*of*, *to*, *on*, and *by* in Figure 2.5) are also enclosed in diamonds.

Weak entities are sometimes called *dependent entities*. They do not arise in ODL since entities there have identities independent of attribute values, and multiway relationships are not allowed in ODL.

2.4.3 A Sales Invoice Example

Consider a typical sales invoice in the revenue cycle given in Figure 2.6.

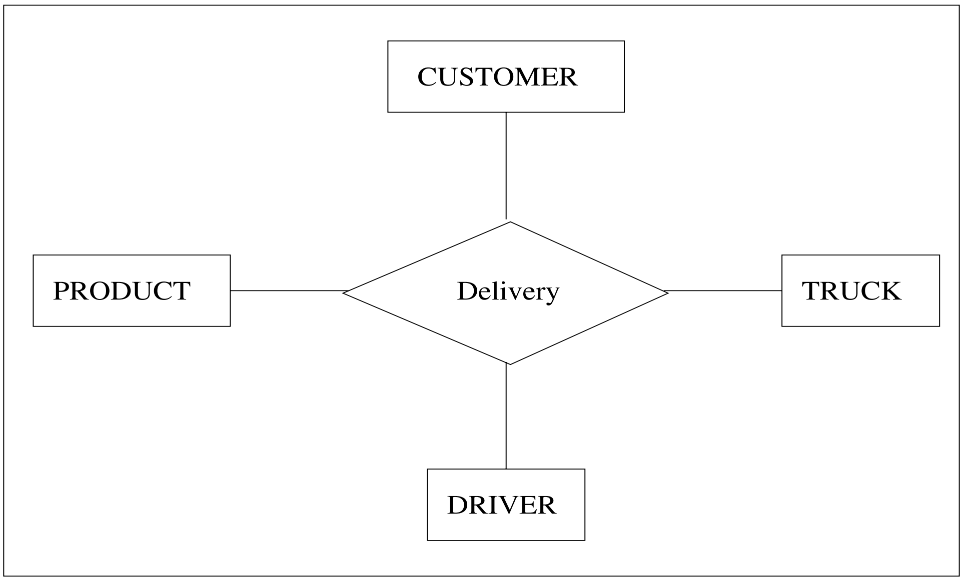


Figure 2.4: Delivery example (Multiway Relationship)

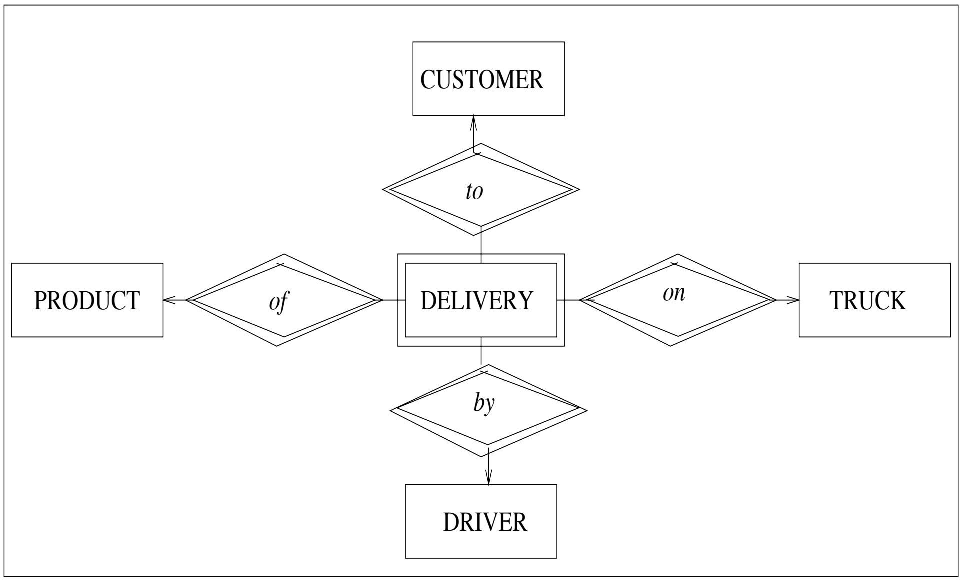
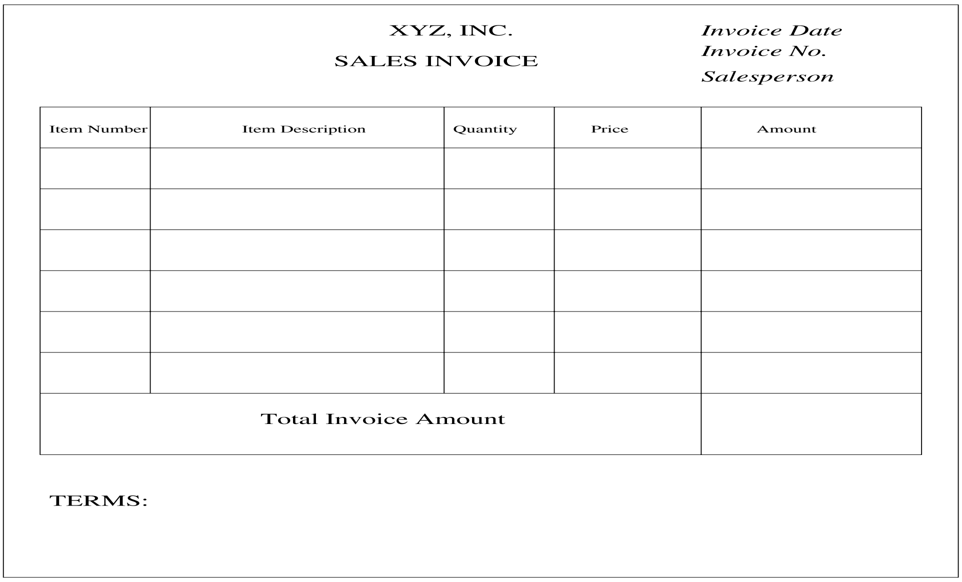


Figure 2.5: Delivery example (Binary Relationships)



XYZ, INC.

SALES INVOICE

Invoice Date
Invoice No.
Salesperson

Item Number	Item Description	Quantity	Price	Amount
Total Invoice Amount				

TERMS:

Figure 2.6: A Sales Invoice

Figure 2.7 gives the ERD for the sales invoice. It contains a many-to-many relationship `FOR` in that an invoice can contain many items, and an item can appear on many invoices. Converting such many-to-many relationship into an entity set, we can reduce the multiplicity of relationships to many-to-one as shown in Figure 2.8. The resultant entity set `INVOICE LINE` is a weak entity set. It does not have an existence apart from that given to it by the sales invoice and the item, ie., to understand the *meaning* of invoice line, we need to know which sales invoice and which item the invoice line pertains to. This dependence of invoice line is implemented in the ERD by it borrowing the key attributes of the two entity sets (`SALES INVOICE` and `ITEM`) with which it has many-to-one relationships. Accordingly, the relationships `CONTAINS` and `FOR` are enclosed in diamonds, and the weak entity set `INVOICE LINE` is also enclosed in a rectangle.

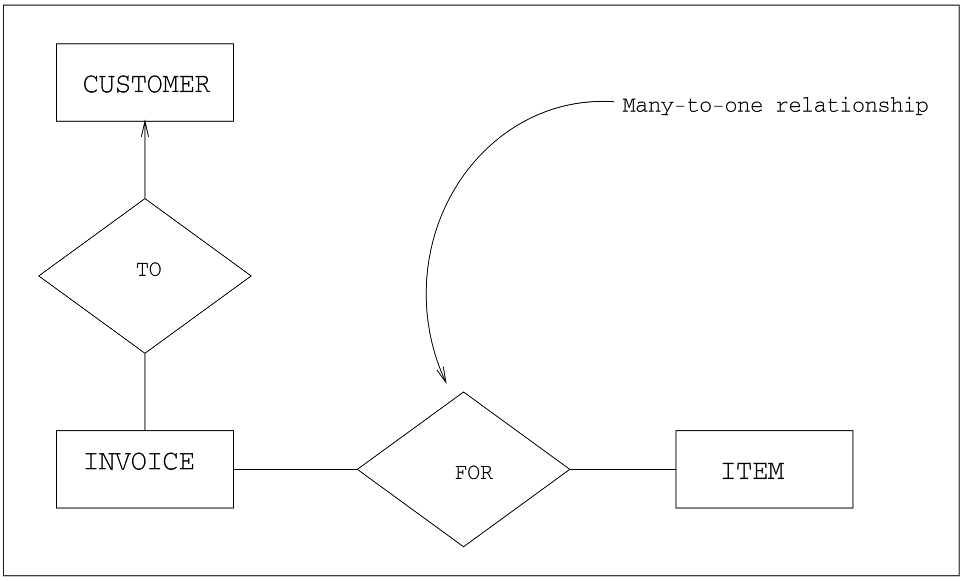


Figure 2.7: ERD for the Sales Invoice Example (Many-to-many Relationship)

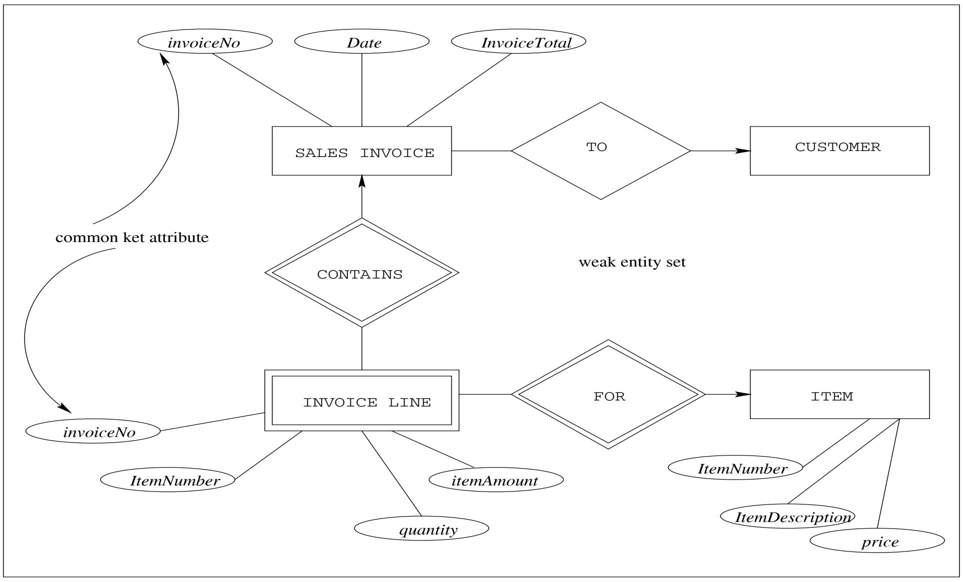


Figure 2.8: ERD for the Sales Invoice Example (Many-to-one Relationships with Weak Entity Set)

Chapter 3

The Relational Model & Database Design

> There is nothing that can be said by mathematical symbols and relations which cannot also be said by words. The converse, however, is false. Much that can be and is said by words cannot successfully be put into equations, because it is nonsense.
>
> C. Truesdell From *Six Lectures on Modern Natural Philosophy*

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental powers of the race. . . . It is a profoundly erroneous truism, repeated by all copy-books and by eminent people when they are making speeches, that we should cultivate the habit of thinking what we are doing. The precise opposite is the case. Civilisation advances by extending the number of important operations we can perform without thinking about them. Operations of thought are like cavalry charges in a battle *they are strictly limited in number, they require fresh horses, and must only be made at decisive moments.*
>
> A. North Whitehead

3.1 Introduction

In the previous two chapters, we have studied how to specify databases using ODL and the entity-relational diagrams. In this chapter, I shall discuss in order the basics of the relational model that underlies the relational database management systems, the translation of ODL and ERD specifications to relational schema, and finally the fundamentals of relational database theory (including the concept of functional dependency, discovery of relation keys, and the normalisation of relational databases).

Most currently used implementations of database systems are based on the relational model (even though one does find the older database systems based on the hierarchical and network models). While the object-oriented approach to data modeling provides a rich set of data types that enable us to model business data more realistically, ultimately such descriptions in ODL (or the ERD) will need to be translated into the relational model.

3.2 The Relational Model

The relational model has just one way to represent data in the database, as a table. The database consists of a set of tables. Each table has a name (*relation name*), columns (referred to as *attributes* of the relation), and rows (each row is referred to as a *tuple*). Each row represents an object (belonging to the class that is represented by the relation), or an entity belonging to the entity set representing the relation. For any entity, the value of any given attribute in the table is drawn from the domain (represented by the data type assigned) corresponding to the attribute.

3.2.1 An Example: Invoice Line

`invoiceNumber`	`itemNumber`	`itemQuantity`	`itemAmount`
235	43	25	28.35
235	24	10	43.25

Note that the table Invoice Line represents the entity set `invoiceLine`. An invoice line can be described by the attributes the `INVOICE` to which the line belongs (`invoiceNumber`), the identification of the inventory item appearing on the invoice line (`itemNumber`), quantity of the item ordered (*itemQuantity*), and the extension of `itemPrice` and `itemQuantity` (`itemAmount`). Remember that `itemPrice` is an attribute of the entity set `ITEM`.

The above can be cast in the *relational schema* as below:

```
invoiceLine(invoiceNumber, itemNumber, itemQuantity, itemAmount)
```

Since like entity set a relation is a set concept, the order of listing of the attributes does not influence the *meaning* of the relation. Also, shuffling the order of the tuples in the relation also does not alter the meaning of the relation, so long as one is consistent in the interpretation of the table after shuffling of attributes or tuples. Nevertheless, it is a good idea to specify the standard order of attributes so there is no confusion in the interpretation of the relation.

In a relation, no two tuples can be identical, since then the assumption of a relation as a set is violated. Should in a particular application it happen that two tuples are identical, it is necessary to include in the schema of the offending relation a new attribute whose values differ for the two such tuples.

Unlike in the object-oriented modeling, all the domains underlying the relation attributes must be atomic, ie., they can not be structures or contain operators that are type constructors. While this seems rather restrictive, in practice most relational database systems provide for native data types that are indeed not atomic (for example, most of them provide types for what are really structures, such as date). Since in a tuple each attribute can take on exactly one value from the domain of each attribute, we can express a tuple as a function from the attributes to values as done below for the first tuple in our example:

$$\boxed{\begin{array}{c}\texttt{invoiceNumber} \rightarrow \texttt{235}\\ \texttt{itemNumber} \rightarrow \texttt{43}\\ \texttt{itemQuantity} \rightarrow \texttt{25}\\ \texttt{itemAmount} \rightarrow \texttt{28.35}\end{array}}$$

While the relation *schema* does not change and is relatively immutable, the *instance* of the relation which gives us the set of tuples of that relation at any instant of time change because of updates/deletions/inserts of tuples to that relation in the normal course of database transaction processing. It is important to bear in mind the difference between the *schema* of a relation and its *instance*. The design of a relational database is expressed in terms if its *schema* and *not* its *instances*.

A relational database, the *relational database schema* or simply *database schema* consists of a set of schemas of the relations in the database.

3.3 ODL To Relational Designs

While it is entirely possible to design a relational database directly in Entity-Relationship Diagrams and convert them into relational schemas, design of the database first in ODL is a very useful exercise at least for two reasons. First, while the Entity-Relationship model is simple, it forces one to make implicit compromises in the process of shoe-horning complex business rules into simplistic diagrams. By drawing ODL specifications first, we first specify the complexities of such business rules in all their glory and then make compromises explicitly while realising that the elegance and simplicity of the relational model force them on us. Secondly, the Entity-Relationship model is in some sense an incomplete description of the database (for example, inverse relationships are not shown). Considering ODL designs forces one to fully appreciate the semantics of the data being modeled.

I will discuss the conversion of ODL to relational schema in two steps. First I will discuss the conversion in case of attributes, and then the conversion of relationships.

3.3.1 ODL To Relation Schema: Attributes

Attributes that are Atomic Types

If the attributes are all atomic types, the conversion is really simple. In the corresponding relational schema, the relation name is the interface name in ODL and each attribute in the interface will also be an attribute in the schema. For example, the ODL code for ITEM in our Invoice example is:

```
interface ITEM {
    attribute string itemNumber;
    attribute string itemDescription;
    attribute string itemPrice;

}
```

and the corresponding relation schema is:

```
ITEM(itemNumber, itemDescription, itemPrice)
```

Record Structures with Atomic Attributes

If an attribute in an ODL interface is a structure, the conversion is once again simple, since we can include in the relation schema each atomic type in the structure as a separate attribute. For example, for the interface CUSTOMER in our invoice example below,

```
interface CUSTOMER {
    attribute string customerName;
    attribute Struct address
        {string street, string city, string state, string zipCode}
            customerAddress;
    attribute string customerPhone;

}
```

the relation schema can be written as

```
CUSTOMER(customerName, street, city, state, zipCode)
```

If two attributes in an interface have fields that have the same name, then we may have to rename them so that the attribute names in the relation schema are unique.

If some attributes in the interface are enumeration types, then in the relation schema they can be defined to ne string or integer. It is important to note that though the relational model does not have representations for data types such as dates and enumeration types, most commercial database systems do provide them, and the Structured Query Language (SQL) does support them.

Attributes involving Collection Types (Multivalued attributes)

Since ODL permits us to represent attributes by complex types using type constructors such as sets, bags, lists, and arrays. Data redundancies/anomalies can arise when ODL code is converted into relation schema specifications. However, as we will see later, they can be handled by normalising the relations in the relational databases.

Condider the interface for `CUSTOMER` where associated with a customer is a set of addresses and a contact person. The ODL code is given below:

```
interface CUSTOMER {
    attribute string customerName;
    attribute Set< Struct address
        {string street, string city, string state, string zipCode}
            customerAddressSet;
    attribute string contactPerson;

}
```

The corresponding relation schema is:

```
CUSTOMER(customerName, street, city, state, zipCode,
contactPerson)
```

It should be apparent that some of the attributes will be repeated, resulting in redundancy in data storage. For example,

customerName	street	city	state	zipCode	contactPerson
Greg Appliances	Allen st.	Albany	NY	12206	Greg
Greg Appliances	Ontario st.	Albany	NY	12205	Greg
Greg Appliances	Partridge st.	Albany	NY	12209	Greg
Bettina's Boutique	State st.	Albany	NY	12208	Bettina
Bettina's Boutique	Western Ave.	Albany	NY	12204	Bettina

Sets, Bags & Arrays:

The above is an example of sets. In case of lists, the position in the list has information content, and therefore the list position would be one of the attributes in the relation schema. In case of array, the array length is fixed, and therefore, in the relation schema, we would have attributes corresponding to each array position. In case of bags, a member can be repeated, and in the corresponding relation schema the *count* of the number of repetitions of the member would be treated as an attribute.

3.3.2 ODL to Relation Schema: Relationships

Consider the Sales Invoice example in our previous chapter. We can give the ODL specifications for classes `SALES-INVOICE` and `CUSTOMER` as below.

```
interface SALES-INVOICE {
    attribute integer invoiceNumber;
    attribute date invoiceDate;
    attribute float invoiceTotal;
    relationship CUSTOMER to inverse CUSTOMER::hasSentTo;

}
```

```
interface CUSTOMER {
    attribute string customerName;
    attribute Set< Struct address
        {string street, string city, string state, string zipCode}>
            customerAddressSet;
    attribute string contactPerson;
    relationship Set<SALES-INVOICE> hasSentTo inverse SALES-INVOICE::to;

}
```

Consider conversion of `SALES-INVOICE` to relation schema. It appears that the relationship is just like any other attribute. However, the *value* of the relationship is an *object* belonging to the class `CUSTOMER`. In the object-oriented world, this is simple since the reference to an object belonging to the `CUSTOMER` class would be implemented as a *pointer* to such an object in the SALES-INVOICE object. However, since the attribute domains in the relational model must be simple types, it would appear that the schema would contain *every* property of objects belonging to the CUSTOMER class. This would complicate matters, since the specification of a CUSTOMER object has a set-oriented inverse relationship with SALES-INVOICE objects. Therefore, in the relational database schema such pointers are simulated by the values of the set of attributes of the corresponding CUSTOMER class that uniquely identifies an object belonging to that class, ie., the values of key attributes. Therefore, we can create the following relation schema for SALES-INVOICE, assuming that customerName is the key of CUSTOMER objects. Since the key of a *foreign* relation CUSTOMER is an attribute of a SALES-INVOICE relation, it is called a *foreign key*.

```
SALES-INVOICE(invoiceNumber, invoiceDate, invoiceTotal, customerName)
```

Now consider the CUSTOMER class. The relationship hasSentTo is set-oriented, and would be implemented in the relation schema by an attribute representing the key of the class SALES-INVOICE with which it has such a set-oriented relationship. This does lead to data redundancies, which can be minimised in the process of database normalisation. We have the relation schema:

```
CUSTOMER(customerName, street, city, state, zipCode,
              contactPerson, invoiceNumber)
```

3.4 From ERDs to Relational Designs

The conversion of ERDs into relation schemas is relatively simple when the following steps are followed.

- Convert all multiway relationships into binary relationships.
- Convert all many-to-many relationships into many-to-one relationships by introducing dependent entity sets. This step will create weak entity sets and double-diamond relationships.
- create relation schemas so that

– weak entity sets borrow the key attributes of the entity sets with which they have relationships.

– There are no relation schema corresponding to double-diamond relationships, since their attributes are subsets of the attributes of the weak entity sets.

– In case of one-to-many relationships, the entity set on the "many" side of the relationship borrows the key of the entity set on the "one" side.

– There are no relation schemas corresponding to many-to-one (and one-to-many) relationships, since the set of their attributes is a subset of the attributes of the entity sets with which the relationship exists.

3.5 Relational Database Design Theory

In this section, I will introduce the concept of functional dependencies, formally define keys of relations in terms of functional dependencies, discuss the algorithm for the computation of closure of attributes and its significance in the identification of keys, minimal basis for a set of functional dependencies, the inference rules for functional dependencies in the Armstrong axioms, and the normalisation of relational databases.

3.5.1 Functional Dependencies

Definition 1 (Functional Dependency) Let $R(A_1, A_2, \ldots, A_n)$ be a relation schema, and let μ and ν be any two tuples in R. For any two set of attributes X and Y, the relation R satisfies the *functional dependency* $X \rightarrow Y$ if for every two tuples μ and ν in R such that $\mu[X] = \nu[X]$ it is also true that $\mu[Y] = \nu[Y]$. □

So, for any relation, a set of attributes X functionally determine a set of attributes Y, if any tuples in the relation agree on the values of attributes in X, then they must also agree on the values of attributes in Y. For example, consider the relation `SALES-INVOICE`. Since a given invoice can not have been written on two separate dates, in the `SALES-INVOICE` relation if any two tuples have the same `invoiceNumber`, then they must have been written on the same day. We can therefore infer that

$$\boxed{invoiceNumber \rightarrow invoiceDate}$$

Some of the other functional dependencies that we have for the relation `invoiceNumber` include the following:

$invoiceNumber \rightarrow invoiceTotal$
$invoiceNumber \rightarrow CustomerName$

Earlier we had defined key of a relation informally. Armed with the concept of functional dependency, we can define it more rigorously as follows:

Definition 2 (Key of a relation) A set of attributes X is a *key* of a relation R if the attributes in X functionally determine all the remaining attributes in R, and no proper subset of X functionally determines the remaining attributes in R.

□

Key of a relation is composed of the *minimal* set of attributes that functionally determine the remaining attributes. Any *superset* of a key is called a *superkey*. Every key is a superkey, but not all superkeys are keys.

In our `SALES-INVOICE` example, `invoiceNumber` is the key, as is illustrated in Figure 3.1.

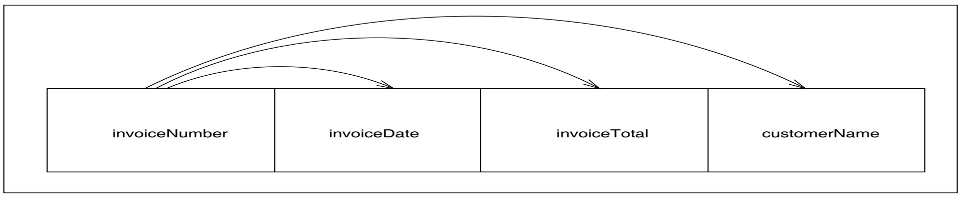


Figure 3.1: Relation key for `SALES-INVOICE`

Information regarding functional dependencies are usually obtained by asking client's operating personnel questions and studying the business processes. The keys of relations are often, as in our `SALES-INVOICE` example, apparent on examining the meaning of the attributes. However, it is important to have a procedure or algorithm for computing the key of a relation. It is to this issue that we now turn. But first a few definitions.

Definition 3 (Splitting/Combining Rule) The functional dependency

$$\boxed{A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m}$$

is equivalent to the set of functional dependencies below.

$$\boxed{\begin{array}{c} A_1A_2 \ldots A_n \rightarrow B_1 \\ A_1A_2 \ldots A_n \rightarrow B_2 \\ \ldots\ldots\ldots \\ \ldots\ldots\ldots \\ \ldots\ldots\ldots \\ A_1A_2 \ldots A_n \rightarrow B_m \end{array}}$$

□

The above rule allows us to split one functional dependency with many attributes on the right hand side into a set of functional dependencies, and also to combine many functional dependencies with the same attributes on the left hand side into a single functional dependency.

Definition 4 (Trivial Dependencies) Any functional dependency

$$\boxed{A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m}$$

is

- *Trivial* if the B's are a subset of A's.
- *Nontrivial* if at least one of the B's is not among the A's.
- *Completely nontrivial* if none of the B's is also one of the A's. □

We can use the definition of trivial dependencies to remove those attributes on the right hand side of functional dependencies in order to derive simpler functional dependencies.

3.5.2 Finding Relation Keys

Earlier we defined a relation key as a set of attributes such that all the remaining attributes are functionally dependent on that set of attributes. When the relations have a small number of attributes, it is quite easy to find all the possible keys. When the number of attributes is large and the number of possible functional dependencies is also large we need a procedure, given a relation schema and the set of functional dependencies, that finds all the possible keys of the relation. It is to this issue that we now turn.

Consider a relation schema $R(A_1, A_2, \ldots, A_n)$ and a set of functional dependencies S. By our above definition of the key of a relation, for any subset K of the attributes of the relation R to be considered a superkey, we would need to show that with the functional dependencies in S and the attributes in K we can "reach" all attributes in R not in K in the sense that the functional dependencies in S imply functional dependencies from K to every attribute in R not in K.

Given a set of attributes of a relation K and a set of functional dependencies S, we can define a set of attributes, whose members include all attributes that are implied by the set of functional dependencies. If such a set includes all of the attributes in the relation, we can conclude that the attributes in K form a superkey of R. The algorithm for the computation of the closure of a set of attributes with respect to a set of functional dependencies provides us a means to determine if any subset of the attributes of a relation is a superkey with respect to a given set of functional dependencies.

Algorithm to Compute the closure of Attributes

INPUT: $R(A)$ a relation schema where $A = \{A_1, A_2, \ldots, A_n\}$, F a set of functional dependencies, and a set of attributes $K \subset A$.

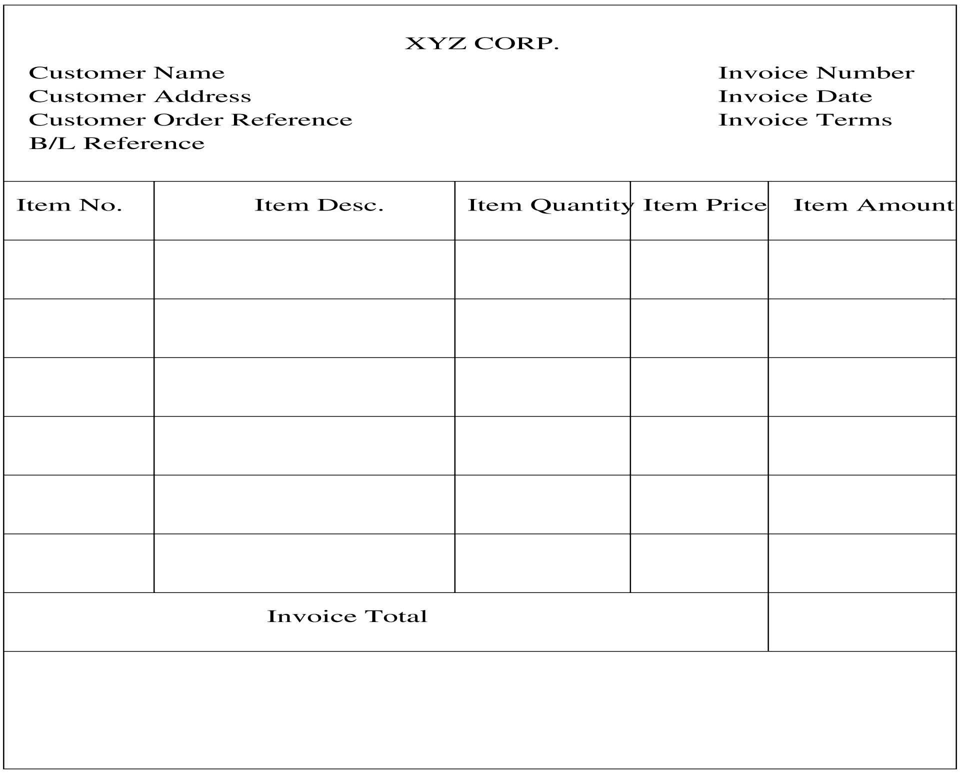
OUTPUT: Closure set K^+ of K with respect to F.

METHOD: Let $K^{(0)} = K$.

```
while $K^{(i+1)} \neq K^{(i)}$ do
    $K^{(i+1)} = K^{i)} \cup \{a \in A - K : \exists (Y \rightarrow Z) \in F, a \in Z, Y \in K^{(i)}$
end while
```

Since $K^{(i)} \subset K^{(i+1)}$ for all i and the number of attributes in R is finite, for some i, $K^{(i)} = K^{(i+1)}$. It can be shown that the above algorithm is both sound and complete: sound in the sense that if an attribute is in the closure set K^+, then it can be shown that there exists a functional dependency from K^+ to that attribute, complete in the sense that if there is a functional dependency where an attribute is on the right hand side, then the algorithm will include such an attribute in the closure set.

Now let us return to our SALES-INVOICE example. Figure 3.2 shows a sales invoice with a few additional attributes.



XYZ CORP.

Customer Name
Customer Address
Customer Order Reference
B/L Reference

Invoice Number
Invoice Date
Invoice Terms

Item No.	Item Desc.	Item Quantity	Item Price	Item Amount
Invoice Total				

Figure 3.2: SALES-INVOICE

Suppose we have just one relation schema for all the attributes that appear on the invoice. the relation schema would be

```
salesInvoice(invoiceNumber, invoiceDate, invoiceTerms, customerOrderRef,
    billOfLadingRef, invoiceTotal, itemNumber, itemDescription,
    itemPrice, itemQuantity, itemAmount, customerName, customerAddress)
```

Figure 3.3 gives the relation schema with the functional dependencies for `SALES-INVOICE`.

The functional dependencies are:

$$
\begin{array}{lcc}
customerName \rightarrow customerAddress & \text{or} & A \rightarrow B \\
itemNumber \rightarrow itemPrice & \text{or} & G \rightarrow C \\
itemNumber \rightarrow itemDescription & \text{or} & G \rightarrow D \\
invoiceNumber \rightarrow invoiceDate & \text{or} & H \rightarrow I \\
invoiceNumber \rightarrow invoiceTerms & \text{or} & H \rightarrow J \\
invoiceNumber \rightarrow customerOrderReference & \text{or} & H \rightarrow K \\
invoiceNumber \rightarrow B/LReference & \text{or} & H \rightarrow L \\
invoiceNumber \rightarrow invoiceTotal & \text{or} & H \rightarrow M \\
invoiceNumber \rightarrow customerName & \text{or} & H \rightarrow A \\
\{invoiceNumber, itemNumber\} \rightarrow itemQuantity & \text{or} & GH \rightarrow E \\
\{invoiceNumber, itemNumber\} \rightarrow itemAmount & \text{or} & GH \rightarrow F
\end{array}
$$

By applying the splitting/combining rule discussed earlier, we can summarise the abobe functional dependencies into:

$$
\boxed{\begin{array}{c}
A \rightarrow B \\
G \rightarrow CD \\
H \rightarrow AIJKLM \\
GH \rightarrow EF
\end{array}}
$$

In the above example, we have the set of attributes

$$A = \{A, B, C, D, E, F, G, H, I, J, K, L, M\} \tag{3.1}$$

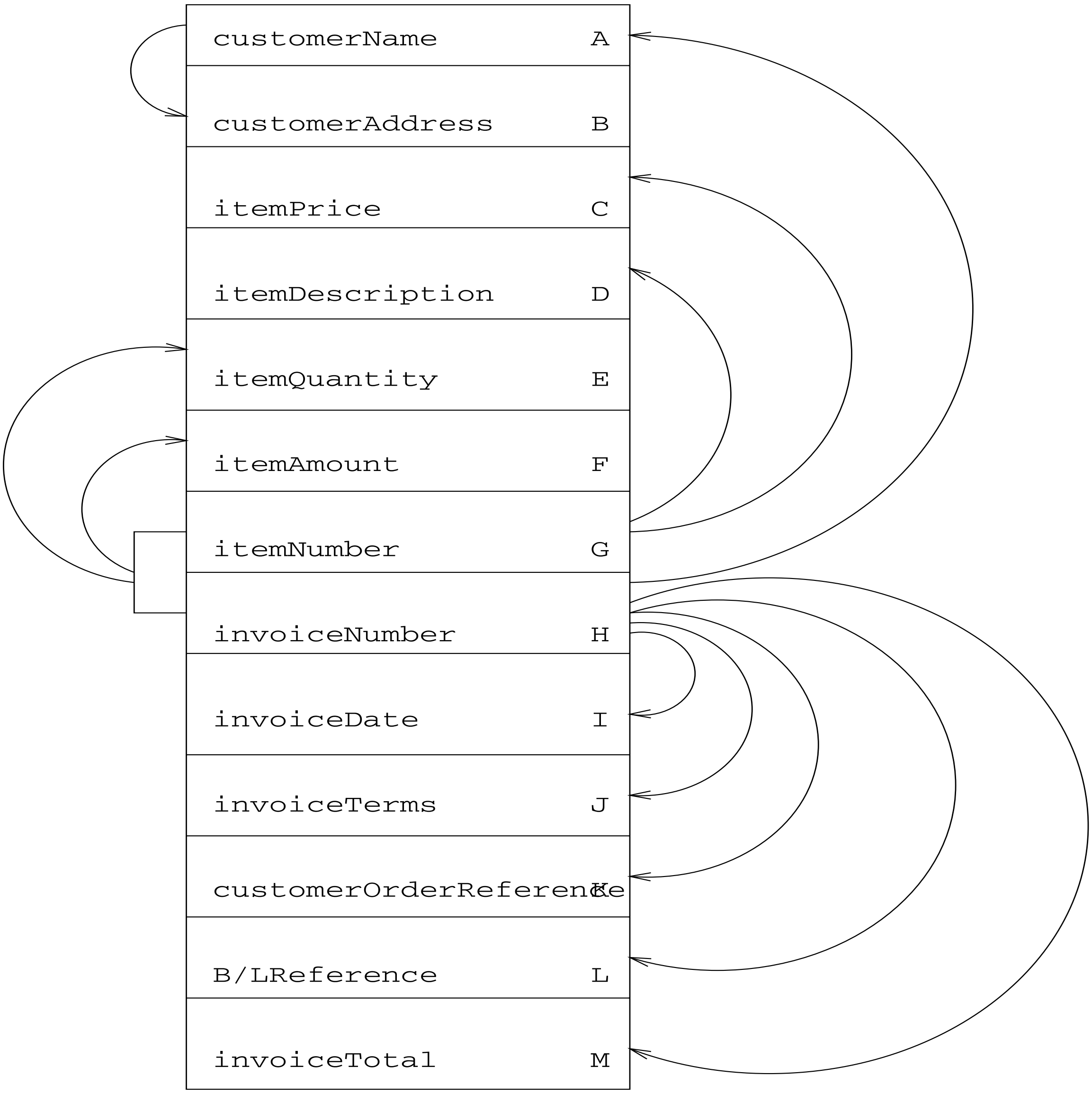


Figure 3.3: `SALES-INVOICE` Relation Schema with Functional Dependencies

in the schema of the relation SALES-INVOICE, with the set of functional dependencies:

$$F = \{A \rightarrow B, G \rightarrow CD, H \rightarrow AIJKLM, GH \rightarrow EF\} \tag{3.2}$$

Now let us examine if the set of attributes $K = \{AGH\}$ is a superkey of the relation SALES-INVOICE. For it to be so, we must have

$$\{AGH\}^{+} = \{A, B, C, D, E, F, G, H, I, J, K, L, M\} \tag{3.3}$$

Let us initialise this closure set to be computed by the closure of attributes algorithm to be $K^{(0)} = \{AGH\}$. First we search for the functional dependencies in the set S that have any of the subsets of $K^{(0)}$ on their left hand side. We find the following functional dependencies that satisfy this requirement:

$$\boxed{\begin{array}{c} A \rightarrow B \\ G \rightarrow CD \\ H \rightarrow AIJKLM \\ GH \rightarrow EF \end{array}}$$

Adding all the attributes on the right hand side of these functional dependencies, we can compute

$$K^{(0)} = \{A, B, C, D, E, F, G, H, I, J, K, L, M\} \tag{3.4}$$

Since $K^{(0)} = A$, we can conclude that $\{AGH\}$ is a superkey of the relation SALES-INVOICE.

Now consider the set $\{AG\}$. You can compute that

$$\{AG\}^{+} = \{A, B, C, D, G\} \tag{3.5}$$

Since this does not include all of the attributes of the SALES-INVOICE relation, we can conclude that $\{AG\}$ is *not* a superkey of that relation. I leave it as an exercise for you to findout that $\{GH\}$ is the relation key of SALES-INVOICE relation. (To demostrate that, you need to show that no subset of $\{GH\}$ is a superkey).

While the attribute closure set algorithm gives us a convenient way in which to identify if a particular subset of attributes of a relation form a superkey with respect to the functional dependencies for the relation, it is important to have an axiomatic way of reasoning about functional dependencies. It is to this topic that we now turn.

3.5.3 Reasoning about Functional Dependencies

The reasoning about functional dependencies is based on the following axioms that are collectively referred to as *Armstrong's axioms*.

- *Reflexivity/Trivial dependencies*: If $\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}$, then $A_1A_2A_3 \ldots A_n \rightarrow B_1B_2 \ldots B_m$.
- *Augmentation*: If $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m$, then $A_1A_2 \ldots A_nC_1C_2 \ldots C_k \rightarrow B_1B_2 \ldots B_mC_1C_2 \ldots C_k$.
- *Transitivity*: If $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m$ and $B_1B_2 \ldots B_m \rightarrow C_1C_2 \ldots C_k$, then $A_1A_2 \ldots A_n \rightarrow C_1C_2 \ldots C_k$.

It can be shown that for any attribute in the closure set K^+, by the application of the above three axioms we can prove that a functional dependency from attributes in K is impied by the functional dependencies in F.

Let us get back to our `SALES-INVOICE` example. We need to show that implied functional dependencies exist from $\{AGH\}$ to each attribute in $\{AGH\}^+$, ie., to the attributes. We will do this now.

- $\{AGH\} \rightarrow AGH$ (by Reflexivity axiom)
- To show $\{AGH\} \rightarrow B$:

  $A \rightarrow B$ (given)

  $\{AGH\} \rightarrow BGH$ (by augmentation)

  $\{AGH\} \rightarrow B$ (by splitting rule)
- To show $\{AGH\} \rightarrow C$:

  $G \rightarrow CD$ (given)

  $\{AGH\} \rightarrow ACDGH$ (by augmentation)

  $\{AGH\} \rightarrow C$ (by splitting rule)

- To show $\{AGH\} \rightarrow D$:

  $G \rightarrow CD$ (given)

  $\{AGH\} \rightarrow ACDGH$ (by augmentation)

  $\{AGH\} \rightarrow D$ (by splitting rule)

- To show $\{AGH\} \rightarrow E$:

  $GH \rightarrow EF$ (given)

  $\{GH\} \rightarrow E$ (by splitting rule)

  $\{AGH\} \rightarrow AEGH$ (by augmentation)

  $\{AGH\} \rightarrow E$ (by splitting rule)

- To show $\{AGH\} \rightarrow F$:

  $GH \rightarrow EF$ (given)

  $\{GH\} \rightarrow F$ (by splitting rule)

  $\{AGH\} \rightarrow AFGH$ (by augmentation)

  $\{AGH\} \rightarrow F$ (by splitting rule)

We have thus shown that functional dependencies from K to each of the attributes in the closure set K^+ are implied by the functional dependencies in F.

3.5.4 Relational Database Design Criteria

There are two ways in which databases are designed: *Decomposition*, and *Synthesis*. In *decomposition*, one starts with the assumption of a *universal relation* (whose relation schema includes *all* of the attributes in the enterprise) and a set of functional dependencies. The method decomposes this universal relation into smaller relations such that certain design criteria are satisfied. In the *systhesis* method, on the other hand, one starts with a set of functional dependencies which it uses to synthesise relation schemas such that the design criteria are satisfied. While synthesis method seems attractive, in most accounting (and business) situations, one usually has a set of relations given to us by the existing application, so decomposition method is appropriate..

In the process of designing relational databases, the main criterion used for guiding as well as evaluating design is:

Lossless-Join Decomposition:

In the process of decomposition, relational tables are split into smaller tables by *projecting* the original table over a subset of that relation attributes. If the tables so split, on *joining* together, yield the original table that was decomposed we say that the decomposition is *lossless*. Otherwise it is said to be *lossy*.

Preservation of Functional Dependencies:

The functional dependencies are integrity constraints on the database, and therefore it is important that they be preserved. When relations are decomposed, it is important that the decomposition do preserve all of the functional dependencies.

When a relation R is decomposed into relations R_1, R_2, $\ldots$, R_p, some of the functional dependencies in F can be lost because a decomposed relation may not contain all the attributes in a functional dependency. We say that a decomposition is functional dependency preserving if the union of all the dependencies preserved in the decomposition is F itself.

Formally, a decomposition of a relation $R(A, B, C)$ into two relations $R_1(A, B)$ and $R_2(A, C)$ is said to be lossless if for any attribute common to both R_1 and R_2, either $A \rightarrow B$ or $A \rightarrow C$.

Some Examples (Hawryszkiewycz, 1984)

Consider $R(A, B, C)$, $F = \{A \rightarrow B, C \rightarrow B\}$, and an instance of R given by the following:

A	B	C
a1	b1	c1
a3	b1	c2
a2	b2	c3
a4	b2	c4

If we decompose R into two relations $R_1(A, B)$ and $R_2(B, C)$ and populate these tables by taking the projection of R onto $\{A, B\}$ and $\{B, C\}$ respectively and removing any duplicates in the projected tuples, we have

$R_1(A,B)$

$A \to B$

A	B
a1	b1
a3	b1
a2	b2
a4	b2

$R_2(B,C)$

$C \to B$

B	C
b1	c1
b1	c2
b2	c3
b2	c4

If we join the two tables R_1 and R_2, we get the

A	B	C
a1	b1	c1
a1	b1	c2
a3	b1	c1
a3	b1	c2
a2	b2	c3
a2	b2	c4
a4	b2	c3
a4	b2	c4

To be a lossless-join decomposition, in our example we must have, for the attribute B that is common to both R_1 and R_2, either $B \to A$ or $B \to C$. Since neither of these functional dependencies hold, the decomposition is not lossless-join, but is *lossy*. In a lossy decomposition, the original relation table will be a subset of the relation table resulting from the joining of the decomposed relation tables, ie., $R \subset joinofR_1, R_2overX$ where X is the set of common attributes in R_1 and R_2.

It is important to note that the above is a lossy decomposition, but it preserves all the functiopnsl dependencies in F.

Now consider the relation $R(X,Y,Z)$ with $F = \{X \to Y, X \to Z, YZ \to X\}$ and its instance given by

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y2	z1
x4	y1	z2

If we decompose R into two relations $R_1(X, Y)$ and $R_2(X, Z)$ and populate these tables by taking the projection of R onto $\{X, Y\}$ and $\{X, Z\}$ respectively and removing any duplicates in the projected tuples, we have

$R_1(X, Y)$
$X \rightarrow Y$

X	Y
x1	y1
x2	y2
x3	y2
x4	y1

$R_2(X, Z)$
$X \rightarrow Z$

X	Z
x1	z1
x2	z2
x3	z1
x4	z2

You will notice that the functional dependency $\{YZ\} \rightarrow X$ is lost and therefore this decomposition is *not* functional dependency-preserving.

When we join the decomposed relations R_1 and R_2, we get

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y2	z1
x4	y1	z2

which is the original relation R. Therefore this is a lossless-join decomposition.

The motivation for decomposition (or synthesis) include

- *Data Redundancy*: To mininise unnecessary duplication of data in the database.
- *Update Anomalies*: While updating the database, inconsistencies are possible if the updates are not done on all tuples containing the same attributes. For example, if all supplier information is subsumed in a vendor invoice relation, when the vendor's telephone number changes, in updating the vendor invoices relation it is necessary to update *all* tuples containing that vendor. Otherwise, the database can become inconsistent since different invoices for the same vendor will show different telephone numbers.
- *Deletion Anomalies*: There can be side effects to removing all tuples with certain values from a relation. For example, if a customer is deleted from a customer relation in a database, the referential integrity of the database may be compromised if invoices sent to that deleted customer still remain in the database.

3.5.5 Boyce-Codd Normal Form

One way to avoid the anomalies is to decompose a relation into relations such that they are all in the Boyce-Codd Normal Form.

A relation R is in *Boyce-Codd Normal Form* if for every nontrivial functional dependency $X \rightarrow Y$, X is a superkey of R.

The method for Boyce-Codd Normal Form Decomposition starts with a set of attributes, a set of functional dependencies. We find a nontrivial functional dependency $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m$ which violates BCNF. We add to the right side as many attributes as are functionally determined by $\{A_1, A_2, \ldots, A_n\}$.

Consider the `SALES-INVOICE` example with

$$R(A, B, C, D, E, F, G, H, I, J, K, L, M) \tag{3.6}$$

and the set of functional dependencies

$$F = \{A \rightarrow B, G \rightarrow CD, H \rightarrow AIJKLM, GH \rightarrow EF\} \tag{3.7}$$

Consider the decomposition below

$R_1(A, B)$	$F_1 = \{A \rightarrow B\}$
$R_2(G, C, D)$	$F_2 = \{G \rightarrow CD\}$
$R_3(GHEF)$	$F_4 = \{\{GH\} \rightarrow EF\}$
$R_4(HAIJKLM)$	$F_4 = \{H \rightarrow AIJKLM\}$

It is easily seen that this decomposition is in the Boyce-Codd Normal Form, the keys for the four relations in the decomposition being A, G, $\{GH\}$, and H respectively.

It can be proved that for any relation and a set of functional dependencies, there exists a Boyce-Codd Normal Form lossless-join decomposition. Unfortunately, however, it may not necessarily preserve all the functional dependencies. To illustrate this point, consider the relation in Figure 3.4.

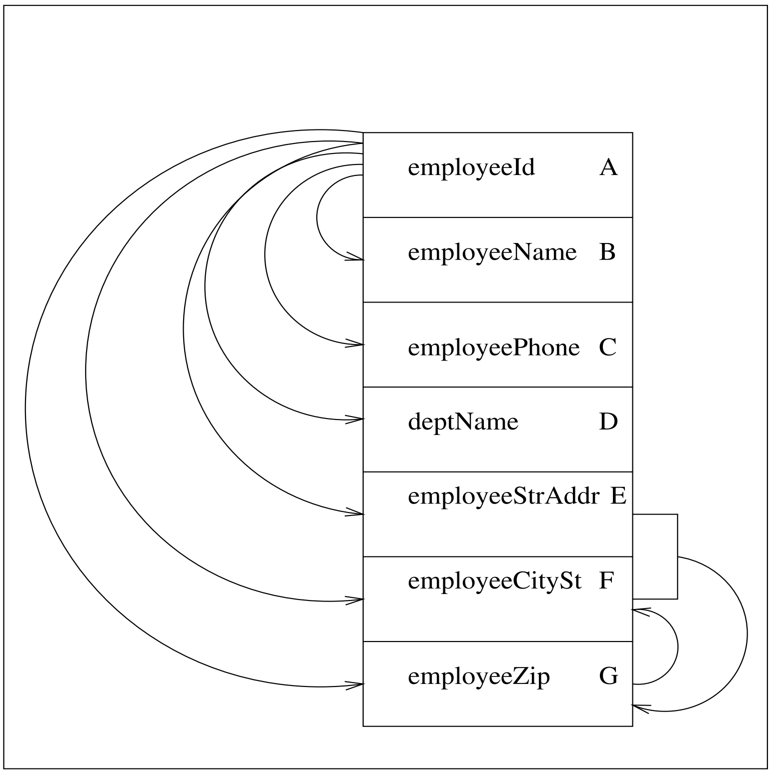


Figure 3.4: EMPLOYEE-DATA Relation Schema with Functional Dependencies

The BCNF decomposition of this relation is given by

$$\boxed{\begin{array}{ll} R_1(A) & F_1 = \{A \rightarrow BCDEF\} \\ R_2(G, F) & F_2 = \{G \rightarrow CF\} \\ R_3(GE) & F_4 = \{\phi\} \end{array}}$$

where ϕ is the empty set. The relation R_1 is in BCNF since A is also its superkey. The relation R_2 is also in BCNF since G is its superkey. The relation R_3 has no functional dependencies associated with it, and therefore its key consists of all its attributes $\{GE\}$, and is also in BCNF. The decomposition is of the lossless-join variety, but it is not functional dependency-preserving, since the dependency $EF \rightarrow G$ is not preserved by the decomposition.

A careful look at the example would reveal that it may be a good idea to combine into the following relation, since it probably is not necessary to split the street address and the city street into two separate relations.

$$\boxed{R_{2a}(G, E, F) \quad F_4 = \{EF \to G, G \to F\}}$$

It should, however, be obvious that this relation is not in BCNF since G is not its superkey.

Since functional dependencies reflect important business rules or relationships that have to do with database integrity, their preservation in the decomposition is often very important. Since that may not be possible to maintain, we look for a concept of a normal form that is lossless-join decomposition while at the same time preserving all the functional dependencies. The third normal form accomplishes this.

Third Normal Form

A relation R is in third normal form if, for any nontrivial functional dependency $A \to B$, either A is a superkey or B is a member of some key.

A more informal way to describe a third normal form relation is to say that a relation is in the third normal form if all non-key attributes functionally depend on the *whole* key and nothing but the key.