# Table of Contents

1 **Introduction to Databases** ................................................................. 3  
   1.1 Introduction ................................................................. 3  
   1.2 Information Model ......................................................... 3  
      1.2.1 Entities ............................................................... 4  
      1.2.2 Entity Relationships .............................................. 5  
   1.3 Database Management ...................................................... 6  
   1.4 Database Languages ...................................................... 8  
   1.5 Data Protection ............................................................... 10  

2 **Basic Relational Data Model** ......................................................... 12  
   2.1 Introduction ................................................................. 12  
   2.2 Relation ................................................................. 13  
   2.3 Properties of a Relation .................................................. 16  
   2.4 Keys of a Relation ......................................................... 18  
   2.5 Relational Schema .......................................................... 19  

3 **Data Updating Facilities** ................................................................. 21  
   3.1 Introduction ................................................................. 21  
   3.2 Data Manipulation Facilities ........................................... 22  
   3.3 User Interface ............................................................... 24  
      3.3.1 Users ................................................................. 24  
      3.3.2 Interface ............................................................. 25  
      3.3.3 Self-Contained Language ...................................... 25  
      3.3.4 Embedded Host Language .................................... 28  
   3.4 Integrity ................................................................. 30  
      3.4.1 Implicit Integrity Constraints ................................. 31  
      3.4.2 Explicit Integrity Constraints ............................... 32  
   3.5 Database Transactions ..................................................... 34  

4 **Normalisation** .................................................................................. 37  
   4.1 Introduction ................................................................. 37  
   4.2 First Normal Form (1NF) ............................................... 40  
   4.3 Functional Dependencies ............................................. 41  
      4.3.1 The converse notation ......................................... 42  
   4.4 Second Normal Form (2NF) ........................................... 44  
   4.5 Third Normal Form (3NF) .............................................. 48  

5 **Relational Algebra (Part I)** .............................................................. 54  
   5.1 Relational Algebra and Relational Calculus .................... 54  
   5.2 Overview of Relational Algebra ..................................... 56  
   5.3 Selection ........................................................................... 57
1 Introduction to Databases

1.1 Introduction

We live in an information age. By this we mean that, first, we accept the universal fact that information is required in practically all aspects of human enterprise. The term ‘enterprise’ is used broadly here to mean any organisation of activities to achieve a stated purpose, including socio-economic activities. Second, we recognise further the importance of efficiently providing timely relevant information to an enterprise and of the importance of the proper use of technology to achieve that. Finally, we recognise that the unparalleled development in the technology to handle information has and will continue to change the way we work and live, ie. not only does the technology support existing enterprises but it changes them and makes possible new enterprises that would not have otherwise been viable.

The impact is perhaps most visible in business enterprises where there are strong elements of competition. This is especially so as businesses become more globalised. The ability to coordinate activities, for example, across national borders and time zones clearly depends on the timeliness and quality of information made available. More important perhaps, strategic decisions made by top management in response to perceived opportunities or threats will decide the future viability of an enterprise, one way or the other. In other words, in order to manage a business (or any) enterprise, future development must be properly estimated. Information is a vital ingredient in this regard.

Information must therefore be collected and analysed in order to make decisions. It is here that the proper use of technology will prove to be crucial to an enterprise. Up-to-date management techniques should include computers, since they are very powerful tools for processing large quantities of information. Collecting and analysing information using computers is facilitated by current Database Technology, a relatively mature technology which is the subject of this book.

1.2 Information Model

Information stored in computer memory is called data. In current computer systems, such data can (persistently) reside on a number of memory devices, most common of which are floppy disks, CD-ROMs, and hard disks.

Data that we store and manipulate using computers are meaningful only to the extent that they are associated with some real world object in a given context. Take, for example, the number ‘23’. This is a piece of data, but by itself a meaningless quantity. If it was associated with, say, a person and interpreted to denote that person’s age (in years), then it begins to be more meaningful. Or, if it was associated with, say, an organisation that sells electronic goods and interpreted to mean the number of television sets sold in a given month, then again it becomes more meaningful. Notice that in both preceding examples, other pieces of data had to be brought into context - a person, a person’s age, a shop, television sets, a given month, etc.

If the data is a collection of related facts about some enterprise (eg. a business, an organisation, an activity, etc), then it is called a database. The data stored need not
include every conceivable piece of fact about that enterprise. Usually, only facts relevant to some area of an enterprise are captured and organised, typically to provide information to support decision making at various levels (operational, management, etc). Such a constrained area of focus is also often referred to as the problem domain or domain of interest, and is typical of databases. In this sense, a database is an information model of some (real-world) problem domain.

1.2.1 Entities

Information models operate on so-called entities and entity relationships. In this section we will clarify what an entity is. Entity relationships are described in 1.2.2.

An entity is a particular object in the problem domain. For example, we can extend the electronics organisation above to identify three distinct entities: products, customers and sales representatives (see Figure 1-1). They are distinct from one another in the sense that each has characteristic properties or attributes that distinguish it from the others. Thus a product has properties such as type, function, dimensions, weight, brand name, cost and price; a customer has properties such as name, city of residence, age, credit rating, etc.; and a sales representative has properties such as name, address, sales region, basic salary, etc. Each entity is thus modelled in the database as a collection of data items corresponding to its relevant attributes. (Note that we distinguish between entities even if in the real world they are from the same class of objects. For example, a customer and a sales representative are both people, but a customer is a person who purchases goods while a sales representative is one who sells goods. The different ‘roles’ played distinguishes each from the other)

Note also the point made earlier that an information model captures only what is relevant in the given problem domain. Certainly, there are other entities in the organisation - regional offices, warehouses, store keepers, distributors, etc - but these may be irrelevant in the context of, say, analysing sales transactions and are thus omitted from the information model. Even at the level of entity attributes, not all conceivable properties need be captured as data items. A customer’s height, weight, hair colour, hobby, formal qualification, favourite foods, etc, are probably irrelevant and can thus omitted from the model.

Strictly speaking, the objects we referred to above as entities are perhaps more accurately called entity classes because they each denote a set of objects (individual entities), each of which exhibits the properties/attributes described for the class. Thus the entity class ‘customer’ is made up of individual entities, each of which has attributes ‘name’, ‘city of residence’, ‘age’, etc. Every individual entity will then have these attributes but one individual will differ from another in the values (data items) associated with attributes. For example, one customer entity might have the value ‘Smith’ as its ‘name’ attribute, while another might have the value ‘Jones’.
Notice now that in our information model an attribute is really a pair: an attribute description or name (such as ‘age’) and an attribute value (such as ‘56’), or simply, an ‘attribute–value’ pair. An individual entity is completely modelled only when all its attribute descriptions have been associated with appropriate attribute values. The collection of attribute–value pairs that model a particular individual entity is termed a data object. Figure 1-2 illustrates three data objects in the database, each being a complete model of an individual from its corresponding entity class.

An entity by itself is often not as interesting or as informative as when it relates in some way to some other entity or entities. A particular product, say a CD-ROM drive, itself only tells us about its intrinsic properties as recorded in its associated data object. A database, however, models more than individual entities in the problem domain. It also models relationships between entities.
In the real world, entities do not stand alone. A CD-ROM drive is supplied by a supplier, is stored in a warehouse, is bought by a customer, is sold by a sales representative, is serviced by a technician, etc. Each of these is an example of a relationship, a logical association between two or more entities.

Figure 1-3. Relationships between entities

Figure 1-3 illustrates such relationships by using links labelled with the type of association between entities. In the figure, a representative sells a product and a customer buys a product. Taken together, these links and the entities they link model a sales transaction: a particular sales transaction will have a product data object related (through the ‘sells’ relation) to a representative data object and (through the ‘buys’ relation) to a customer data object.

Like entity attributes, there are many more relationships than are typically captured in an information model. The choices are of course based on judgements of relevance given a problem domain. Once choices are made and the database populated with particular data objects and relationships, it can then be used to analyse the data to support decision making. In the simple example developed so far, there are already many types of analysis that can be carried out, eg. the distribution of sales of a particular product type by sales region, the performance of sales representatives (measured perhaps by total value of sales in some time interval), product preferences by customer age groups, etc.

1.3 Database Management

The real world is dynamic. As an organisation goes about its business, entities are created, modified or destroyed. Similarly with entity relationships. This is easy to see even for the simple problem domain above, eg. when a sales is made, the product sold is then logically linked to the customer that bought it and to the representative that sold it. Many sales transactions could take place each day and thus many new logical links created between many individual entities. New entities can also be introduced, eg. a new customer arrives on the scene, a new product is offered, or a new salesperson is hired. Likewise, some entities may no longer be of concern to the organisation, eg. a product is discontinued, a salesperson quits or is fired, etc (these entities may still exist in the real world but have become irrelevant for the problem domain). Clearly, an information model must also change to reflect the changes in the problem domain that it models.
If the problem domain is small, involving only a few entities and relationship, and the dynamic changes are relatively few or infrequent, manual methods may be sufficient to maintain an accurate record of the state of the business. But if hundreds or thousands of entities are involved and the business is very dynamic, then maintaining accurate records of its state becomes more of a problem. This is when computers with their immense power to handle large amounts of information become crucial to an organisation. Frequently, it is not just a question of efficiency, but of survival, especially in intensely competitive business sectors.

The need to use computers to efficiently and effectively store databases and to keep them current has developed over the years special software packages called *Database Management Systems* (DBMS). A DBMS enables users to create, modify, access and protect their databases (Figure 1-4).

![Figure 1-4 A DBMS is a tool to create and use databases](image)

In other words, a DBMS is a *tool* to be applied by users to build an accurate and useful information model of their enterprise.

Conceptually, database management is based on the idea of separating a database *structure* from its *contents*. Quite simply, a database structure is a collection of static descriptions of entity classes and relationships between them. At this point, it is perhaps simplest to think of an entity class description as a collection of attribute labels. Entity contents can then be thought of as the values that get associated with attribute labels, creating data objects. In other words, the distinction between structure and content is little more than the distinction made earlier between attribute label and attribute value.
Relationship descriptions likewise are simply labelled links between entity descriptions. They specify possible links between data objects, i.e. two data objects can be linked only if the database structure describes a link between their respective entity classes. Figure 1-5 illustrates this separation of structure from content.

The database structure is also called a schema (or meta-structure—because it describes the structure of data objects). It predefines all possible states of a database, in the sense that no state of the database can contain a data object that is not the result of instantiating an entity schema, and likewise no state can contain a link between two data objects unless such a link was defined in the schema.

Moreover, data manipulation procedures can be separated from the data as well! Thus the architecture of database systems may be portrayed as in Figure 1-6.

### 1.4 Database Languages

We see from the foregoing that to build a database, a user must

1. Define the Database Schema
2. Apply a collection of operators supported by the DBMS to create, store, retrieve and modify data of interest

A typical DBMS would provide tools to facilitate the above tasks. At the heart of these tools, a DBMS typically maintains two closely related languages:

1. A Data Description Language (DDL), which is used to define database schemas, and
2. A Data Manipulation Language (DML), which allows the user to manipulate data objects and relationships between them in the context of given database schemas

These languages may vary from one DBMS to another, in their underlying data model, complexity, functionality, and ease of use (user interface).

So far, we have talked about ‘users’ as if they were all equal in interacting with a DBMS. In actual fact, though, there may be several types of users distinguished by their role (a division of labour, often necessary because of highly technical aspects of DBMSs). For example, an organisation that uses a DBMS will normally have a Database Administrator (DBA) whose job is to create and maintain a consistent set of database schemas to satisfy the needs of different parts of the organisation. The DBA is the principal user of the DDL. Then there are application developers who develop specific functions around the database (eg. product inventory, customer information, point-of-sale transaction handling, etc). They are the principal users of the DML. And finally, there are the end-users who use the applications developed to support their work in the organisation. They normally don’t see (and don’t care to know about!) the DDL or the DML.

The DDL is a collection of statements for the description of data types. The DBA must define the target database structure in terms of these data types.

For instance, the data object, attribute and link mentioned above are data types, and hence may be perceived as a simple DDL. Thus the data structures in Figure 1- are notionally DDL descriptions of a database schema, as illustrated in Figure 1-7 (‘notional’ because the actual language will have specific syntactical constructs that may differ from one DBMS to another).
A DML is a collection of operators that can be applied to valid instances (ie. data objects) of the data types in the schema. As illustrated in Figure 1-8, the DML is used to manipulate instances, including the creation, modification and retrieval of instances. (Like the DDL above, the illustration here is notional; more concrete forms of these languages will be covered in later sections of this book).

1.5 Data Protection

Databases can be a major investment. There are costs, obviously, associated with the hardware and specialised software such as a DBMS. Less obvious, perhaps, are costs of creating databases. Large and complex databases can require many man-years of analysis and design effort involving specialist skills that may not be present in the organisation. Thus expensive consultants and other technical specialists may have to be brought in. Furthermore, in the long-term, an organisation must also develop internal capability to maintain the databases and deal with changing requirements. This usually means hiring and retaining a group of technical specialists, such as DBAs, who need to be trained (and re-trained to keep up with the technology). End-users too will need to be trained to use the system properly. In other words, there are considerable running costs as well. In all, databases can be very expensive.

Aside from the expenses above, databases often are crucial to a business. Imagine what would happen, say, if databases of customer accounts maintained by a bank were (accidently or maliciously) destroyed! Because of these actual and potential costs, databases must be deliberately protected against any conceivable harm.

Generally, there are three types of security that must be put in place:

1. Physical Protection: these are protective measures to guard against natural disasters (eg. fire, floods, earthquakes, etc), theft, accidental damage to equipment and other threats that can cause the physical loss of data. This is generally the area of physical installation management and is outside the scope of this book.
2. *Operational Protection*: these are measures to minimise or even eliminate the impact of human errors on the databases’ integrity. Errors can occur, for example, in assigning values to attributes—a value may be unreasonable (eg. an age of 213!) or of the wrong type (eg. the value ‘Smith’ assigned to the age attribute). These measures are typically embodied in a set of *integrity constraints* (a set of assertions) that must be enforced (ie. the truth of the assertions must be preserved across all database transactions). An example of an assertion might be ‘the price of a product must be a positive number’. Any operation then is invalid if it violates a stated constraint, eg. “Store … Price=−9.99”. These constraints are typically specified by a DBA in the database schema.

3. *Authorisational Protection*: these are measures to ensure that access to the databases are by authorised users only, and then only for specific modes of access (eg. some users may only be allowed to read while others can modify database contents). They are necessary to ensure that confidentiality and correctness of information is preserved. Access control can be applied at various levels in the system. At the installation level, access through computer terminals may be controlled using special access cards or passwords. At successively lower levels, control may be applied to an entire database, to its physical devices (or parts thereof), or to its logical parts (parts of the schema). In extremely sensitive problem domains, access control may even be applied to individual instances or data objects in the database.
2 Basic Relational Data Model

2.1 Introduction

Basic concepts of information models, their realisation in databases comprising data objects and object relationships, and their management by DBMS’s that separate structure (schema) from content, were introduced in the last chapter. The need for a DDL to define the structure of entities and their relationships, and for a DML to specify manipulation of database contents were also established. These concepts, however, were presented in quite abstract terms, with no commitment to any particular data structure for entities or links nor to any particular function to manipulate data objects.

There is no single method for organising data in a database, and many methods have in fact been proposed and used as the basis of commercial DBMS’s. A method must fully specify:

1. the rules according to which data are structured, and
2. the associated operations permitted

The first is typically expressed and encapsulated in a DDL, while the second, in an associated DML. Any such method is termed a Logical Data Model (often simply referred to as a Data Model). In short,

Data Model = DDL + DML

and may be seen as a technique for the formal description of data structure, usage constraints and allowable operations. The facilities available typically vary from one Data Model to another.

![Figure 2-1 Logical Data Model](image)

Each DBMS may therefore be said to maintain a particular Data Model (see Figure 2-1). More formally, a Data Model is a combination of at least three components:

1. A collection of data structure types
2. A collection of operators or rules of inference, which can be applied to any valid instance of data types in (1)
A collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or change of state or both.

It is important to note at this point that a Data Model is a logical representation of data which is then realised on specific hardware and software platforms (its implementation, or physical representation as illustrated in Figure 2-1). In fact, there can be many different implementations of a given model, running on different hardware and operating systems and differing perhaps in their efficiency, performance, reliability, user interface, additional utilities and tools, physical limitations (e.g. maximum size of databases), costs, etc. (see Figure 2-2). All of them, however, will support a mandatory minimal set of facilities defined for that data model. This is analogous to programming languages and their implementations, e.g. there are many C compilers and many of them implement an agreed set of standard features regardless of the hardware and software platforms they run on. But as with programming languages, we need not concern ourselves with the variety of implementations when developing database applications—knowledge of the basic logical data model is sufficient for us to do that.

**Figure 2-2** Multiple realisations of a single Data Model

It is also important not to confuse the terms information model and data model. The former is an abstraction of a real world problem domain and talks of entities, relationships and instances (data objects) specific to that domain. The latter provides a domain independent formal framework for expressing and manipulating the abstractions of any information model. In other words, an information model is a description, by means of a data model, of the real world.

### 2.2 Relation

Perhaps the simplest approach to data modelling is offered by the Relational Data Model, proposed by Dr. Edgar F. Codd of IBM in 1970. The model was subsequently expanded and refined by its creator and very quickly became the main focus of practically all research activities in databases. The basic relational model specifies a data structure, the so-called Relation, and several forms of high-level languages to manipulate relations.
The term *relation* in this model refers to a two-dimensional table of data. In other words, according to the model, information is arranged in columns and rows. The term *relation*, rather than matrix, is used here because data values in the table are not necessarily homogenous (i.e., not all of the same type as, for example, in matrices of integers or real numbers). More specifically, the values in any row are not homogenous. Values in any given column, however, are all of the same type (see Figure 2-3).

![Figure 2-3 A Relation](image)

A relation has a unique name and represents a particular entity. Each row of a relation, referred to as a *tuple*, is a collection of facts (values) about a particular individual of that entity. In other words, a tuple represents an *instance* of the entity represented by the relation.

![Figure 2-4 Relation and Entity](image)

Figure 2-4 illustrates a relation called ‘Customer’, intended to represent the set of persons who are customers of some enterprise. Each tuple in the relation therefore represents a single customer.

The columns of a relation hold values of attributes that we wish to associate with each entity instance, and each is labelled with a distinct attribute name at the top of the column. This name, of course, provides a unique reference to the entire column or to a particular value of a tuple in the relation. But more than that, it denotes a *domain* of values that is defined *over all relations* in the database.
The term domain is used to refer to a set of values of the same kind or type. It should be clearly understood, however, that while a domain comprises values of a given type, it is not necessarily the same as that type. For example, the column ‘Cname’ and ‘Ccity’ in Figure 2-4 both have values of type string (ie. valid values are any string). But they denote different domains, ie. ‘Cname’ denotes the domain of customer names while ‘Ccity’ denotes the domain of city names. They are different domains even if they share common values. For example, the string ‘Paris’ can conceivably occur in the Column ‘Cname’ (a person named Paris). Its meaning, however, is quite different to the occurrence of the string ‘Paris’ in the column ‘Ccicy’ (a city named Paris)! Thus it is quite meaningless to compare values from different domains even if they are of the same type.

Moreover, in the relational model, the term domain refers to the current set of values found under an attribute name. Thus, if the relation in Figure 2-4 is the only relation in the database, the domain of ‘Cname’ is the set \{Codd, Martin, Deen\}, while that of ‘Ccity’ is \{London, Paris\}. But if there were other relations and an attribute name occurs in more than one of them, then its domain is the union of values in all columns with that name. This is illustrated in Figure 2-5 where two relations each have a column labelled ‘C#’. It also clarifies the statement above that a domain is defined over all relations, ie. an attribute name always denotes the same domain in whatever relation it occurs.

This property of domains allows us to represent relationships between entities. That is, when two relations share a domain, identical domain values act as a link between tuples that contain them (because such values mean the same thing). As an example, consider a database comprising three relations as shown in Figure 2-6. It highlights a Transaction tuple and a Customer tuple linked through the C# domain value ‘2’, and the same Transaction tuple and a Product tuple linked through the P# domain value ‘1’. The Transaction tuple is a record of a purchase by customer number ‘2’ of product number ‘1’. Through such links, we are able to retrieve the name of the customer and the product, ie. we are able to state that the customer ‘Martin’ bought a ‘Camera’. They help to avoid redundancy in recording data. Without them, the Transaction relation in Figure 2-6 will have to include information about the appropriate Customer and Product in its table. This duplication of data can lead to integrity problems later, especially when data needs to be modified.
2.3 Properties of a Relation

A relation with $N$ columns and $M$ rows (tuples) is said to be of *degree* $N$ and *cardinality* $M$. This is illustrated in Figure 2-7 which shows the Customer relation of degree four and cardinality three. The product of a relation’s degree and cardinality is the number of attribute values it contains.

The characteristic properties of a relation are as follows:

1. All entries in a given column are of the same kind or type
2. The ordering of columns is immaterial. This is illustrated in Figure 2-8 where the two tables shown are identical in every respect except for the ordering of their columns. In the relational model, column values (or the value of an attribute of a given tuple) are not referenced by their position in the table but by name. Thus the display of a relation in tabular form is free to arrange columns in any order. Of course, once an order is chosen, it is good practice to use it every time the relation (or a tuple from it) is displayed to avoid confusion.
3. No two tuples are exactly the same. A relation is a set of tuples. Thus a table that contains duplicate tuples is not a relation and cannot be stored in a relational database.

4. There is only one value for each attribute of a tuple. Thus a table such as in Figure 2-9 is not allowed in the relational model, despite the clear intended representation, ie. that of customers with two abodes (eg. Codd has one in London and one in Madras). In situations like this, the multiple values must be split into multiple tuples to be a valid relation.

5. The ordering of tuples is immaterial. This follows directly from defining a relation as a set of tuples, rather than a sequence or list. One is free therefore to display a relation in any convenient way, eg. sorted on some attribute.

The extension of a relation refers to the current set of tuples in it (see Figure 2-10). This will of course vary with time as the database changes, ie. as we insert new tuples, or modify or delete existing ones. Such changes are effected through a DML, or put another way, a DML operates on the extensions of relations.

The more permanent parts of a relation, viz. the relation name and attribute names, are collectively referred to as its intension or schema. A relation’s schema effectively describes (and constrains) the structure of tuples it is permitted to contain. DML operations on tuples are allowed only if they observe the expressed intensions of the affected relations (this partially addresses database integrity concerns raised in the last chapter). Any given database will have a database schema which records the intensions of every relation in it. Schemas are defined using a DDL.
2.4 Keys of a Relation

A key is a part of a tuple (one or more attributes) that uniquely distinguishes it from other tuples in a given relation. Of course, in the extreme, the entire tuple is the key since each tuple in the relation is guaranteed to be unique. However, we are interested in smaller keys if they exist, for a number of practical reasons. First, keys will typically be used as links, i.e., key values will appear in other relations to represent their associated tuples (as in Figure 2-6 above). Thus keys should be as small as possible and comprise only nonredundant attributes to avoid unnecessary duplication of data across relations. Second, keys form the basis for constructing indexes to speed up retrieval of tuples from a relation. Small keys will decrease the size of indexes and the time to look up an index.

Consider Figure 2-11 below. The customer number (C#) attribute is clearly designed to uniquely identify a customer. Thus we would not find two or more tuples in the relation having the same customer number and it can therefore serve as a unique key to tuples in the relation. However, there may be more than one such key in any relation, and these keys may arise from natural attributes of the entity represented (rather than a contrived one, like customer number). Examining again Figure 2-11, no two or more tuples have the same value combination of Ccity and Cphone. If we can safely assume that no customer will share a residence and phone number with any other customer, then this combination is one such key. Note that Cphone alone is not - there are two tuples with the same Cphone value (telephone numbers in different cities that happen to be the same). And neither is Ccity alone as we may expect many customers to live in a given city.
Introduction to Databases & Relational DM

While a relation may have two or more candidate keys, one must be selected and designated as the primary key in the database schema. For the example above, C# is the obvious choice as a primary key for the reasons stated earlier. When the primary key values of one relation appear in other relations, they are termed foreign keys. Note that foreign keys may have duplicate occurrences in a relation, while primary keys may not. For example, in Figure 2-6, the C# in Transaction is a foreign key and the key value ‘1’ occurs in two different tuples. This is allowed because a foreign key is only a reference to a tuple in another relation, unlike a primary key value, which must uniquely identify a tuple in the relation.

2.5 Relational Schema

A Relational Database Schema comprises
1. the definition of all domains
2. the definition of all relations, specifying for each
   a) its intension (all attribute names), and
   b) a primary key

Figure 2-12 shows an example of such a schema which has all the components mentioned above. The primary keys are designated by shading the component attribute names. Of course, this is only an informal view of a schema. Its formal definition must rely on the use of a specific DDL whose syntax may vary from one DBMS to another.

| Domains: C#, Cname, Ccity, Cphone P#, Pname, Price, Date, Qnt; |
|-------------|---------------------|---------------------|---------------------|
| Relations:  | Customer            | Product             |
|             | C#                  | P#                  |
|             | Cname               | Pname               |
|             | Ccity               | Price               |
|             | Cphone              |                     |
|             | Transaction         |                     |
|             | C#                  | P#                  |
|             | Date                |                      |
|             | Qnt                 |                      |

Figure 2-12 An Example Relational Schema

There is, however, a useful notation for relational schemas commonly adopted to document and communicate database designs free of any specific DDL. It takes the simple form:

<relation name>: <list of attribute names>

Additionally, attributes that are part of the primary key are underlined.

Thus, for the example in Figure 2-12, the schema would be written as follows:

Customer: (C#, Cname, Ccity, Cphone)
Transaction: (C#, P#, Date, Qnt)
Product: (P#, Pname, Price)
This notation is useful in clarifying the overall organisation of the database but omits some details, particularly the properties of domains. As an example of a more complete definition using a more concrete DDL, we rewrite some the schema above using Codd’s original notation. The principal components of his notation are annotated alongside.

**Keyword to introduce a domain**

**Domain CName Character(20)**

**Domain CCity Character(20)**

**Domain CPhone Numeric(12)**

**Domain C# Numeric(5)**

... etc

**Keyword to introduce a relation**

**Relation Customer: (C#, Cname, Ccity, Cphone) KEY C#**

**Relation Transaction: (C#, P#, Date, Qnt) KEY C#, P#, Date**

... etc
3 Data Updating Facilities

3.1 Introduction

We have seen that a Data Model comprises the Data Description facilities (through the DDL) and the Data Manipulation facilities (through the DML). As explained in Chapter 2, a DDL is used to specify the schema for the database - its entities can be created, and its attributes, domains, and keys can be defined through language statements of the DDL. The structure of the entities is defined, but not the data within them. DDL thus supports only the declaration of the database structure.

In this chapter, we shall see how the second component, the DML, can be used to support the manipulation or processing of the data within the database structures defined by the DDL. Manipulation begins with the placement of data values into the structures. When the data values have been stored, the end user should be able to get at the data. The user would have a specific purpose for the piece of data he/she wants to get - perhaps to display the data on the screen in order to know the value, to write the data to an output report file to be printed, to use the data as part of a computation, or to make changes to it.

Figure 3.1: Data Definition and Manipulation facilities of a Data Model
3.2 Data Manipulation Facilities

The manipulative part of the relational model comprises the DML which contains commands to put new data, delete and modify the existing data. These facilities of a Database Management System are known as Updating Facilities or Data Update Functions, because unlike the DDL which executes operations on the structure of the database’s entities, the DML performs operations on the data within those structures.

Given a relation declared in a database schema, the main update functions of a DML are:

1. To insert or add new tuples into a particular relation
2. To delete or erase existing tuples from a relation
3. To modify or change data in an existing relation

Examples:

1. To insert a new tuple

The company receives a new customer. To ensure that its database is up-to-date, a new tuple containing the data that is normally kept about customers must be created and inserted.

Step 1. A user (through an application program) chooses a relation, say the Customer relation. It has 4 attributes, and 3 existing tuples.

Step 2. The user prepares a new tuple of the relation (database) on the screen or in the computer’s memory

Step 3. Through a DML command specified by the user, the DBMS puts a new tuple into the relation of the database according to the definition of the DDL to place data in that row. The Customer relation now has 4 tuples.

This is thus a way to load data into the database.
2. To delete an existing tuple

An existing customer no longer does any business with the company, in which case, the corresponding tuple must be erased from the customer database.

\[\text{Step 1. The user chooses the relation, Customer.}\]
\[\text{Step 2. The user issues a DML command to retrieve the tuple to be deleted.}\]
\[\text{Step 3. The DBMS deletes the tuple from the relation.}\]

The updated relation now has one less tuple.

<table>
<thead>
<tr>
<th>Customer</th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Lamp1</td>
<td>Graz</td>
<td>832551</td>
<td></td>
</tr>
</tbody>
</table>

3. To modify an existing tuple

An existing customer has moved to a new location (or that the current value in the data field is incorrect). He has new values for the city and telephone number. These new values must replace the previous values.

\[\text{Step 1. The user chooses the relation.}\]
\[\text{Step 2. The user issues a DML retrieval command to get the tuple to be changed.}\]
\[\text{Step 3. The user modifies one or more data items.}\]
\[\text{Step 4. The DBMS inserts the modified tuple into the relation.}\]

Two types of modifications are normally done:

1. Assigned - an assigned modification entails the simple assignment of a new value into the data field (as in the example above)
2. Computed - in computed modification, the existing value is retrieved, then some computation is done on it before restoring the updated value into the field (e.g. all phone numbers beginning with the digit 5 are to be changed to begin with the digits 58).

Additionally, it is possible to insert new tuples into a relation with one or more unknown values. Such unknown values, called NULL-VALUEs, are denoted by ?

**To insert a tuple with unknown values**

<table>
<thead>
<tr>
<th>Customer</th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Deen</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

A new customer, Deen, is created. But Deen has yet to notify the company of his living place and telephone number. At a later point in time, when Deen has confirmed his living place and telephone, the tuple with his details can be modified by replacing the ?s with the appropriate values.

At this stage, we only mention these update functions via logical definitions such as above. In the implementation of DMLs, there exist many RDBMS systems with wide variations in the actual language notations. We shall not discuss these update functions of concrete data manipulation languages yet.

### 3.3 User Interface

#### 3.3.1 Users

Now that data values are stored in the database, we shall look at how users can communicate with the database system in order to access the data for further manipulation. First let us take a look at the characteristics of users and the processing and inquiries they tend to make.

There are essentially two types of users:

1. **End Users**: End users are those who directly use the information obtained from the database. They are likely to be computer novices or neophytes, using the computer system as an extended tool to assist them with their main job function (which may be to do financial analysis or to register students). End users may feel uncomfortable with computers or may simply be not interested in the technical details, but their lack of knowledge should not be a handicap to the main job which they have to do.

   End users should be able to retrieve the data in the database in any manner they wish, which are likely to be in the form of:

   - casual, unanticipated or *ad hoc* queries which often must be satisfied within a short space of time, if not immediately (e.g. “How many students over the age of 30 are taught by professors below the age of 30?”)
• standard or predictable queries that need to be executed on a routine basis (e.g. “Produce the monthly cash flow analysis report”)

2. Database specialists: Specialist users are knowledgeable about the technicalities of the database management system. They are likely to hold positions such as the database administrator, database programmer, database support person, systems analyst or the like. They are likely to be responsible for tasks like:
• defining the database schema
• handling complex queries, reports or tailored software applications
• defining data security and ensuring data integrity
• performing database backups and recoveries
• monitoring database performance

3.3.2 Interface

Interactions with the database would require some form of interface with the users. There are two basic ways in which the User-Database interface may be organized, i.e. the database may be accessed from a:

1. purpose-built, non-procedural Self-Contained Language, or a
2. Host Language (such as C, C++, COBOL, Pascal, etc.)

The Self-Contained Language tend to be the tool favored by end-users to access the database, whereas access through a host language is a tool of the technical experts and skilled programmers who use it to develop specialised software or database applications,

In either case, the access is still through the database schema and using the DML.

3.3.3 Self-Contained Language

Let us first take a look at the tool favored by the end users:

**Figure 3.2:** Two different interfaces to the database
Here we have a collection of DML statements (e.g. GET, SELECT) to access the database. These statements can be expanded with other statements that are capable of doing arithmetic operations, computing statistical functions, and so on. The DML statements, as we have seen, are dependent on the database schema. However, the additional statements for statistical functions etc. are not, and thus add a form of independence from the schema. This is illustrated in the Figure 3.3. Hence the name “Self-Contained” language for such a database language.

The self-contained language is most suitable for end users to gain rapid or online access to the data. It is often used to make *ad hoc* inquiries into the database, involving mainly the retrieval operation. It is often described as being user friendly, and can be learnt quite quickly. Instead of waiting for the database specialists or some technical whiz to program their requests, end users can now by themselves create queries and format the resulting data into reports.

The language is usually non-procedural and command-oriented where the user specifies English-like text. To get a flavor of such a language, let us look at some simple examples which uses a popular command-oriented data-sublanguage, SQL. (More will covered in Chapter 9).

Suppose we take the 3 relations introduced earlier:

- **Customer (C#, Cname, Ccity, Cphone)**
- **Product (P#, Pname, Price)**
- **Transaction (C#, P#, Date, Qnt)**

with the following sample values in the tuples:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Transaction</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
<td>Ccity</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 3.4: Sample database*
We shall illustrate the use of some simple commands:

- **SELECT * FROM CUSTOMER**
  
  This will retrieve all data values of the Customer relation, with the following resulting relation:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

- **SELECT CNAME FROM CUSTOMER WHERE CCITY=“London”**
  
  This will retrieve, from the Customers relation, the names of all customers whose city is London:

<table>
<thead>
<tr>
<th>Cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
</tr>
<tr>
<td>Deen</td>
</tr>
</tbody>
</table>

- **SELECT CNAME, P# FROM CUSTOMER, TRANSACTION WHERE CUSTOMER.C# = TRANSACTION.C#**
  
  This will access the two relations, Customer and Transaction, and in effect, retrieve from them the Names of customers who have transactions and the Part numbers supplied to them (note, customers with no transactions will not be retrieved). The resultant relation is:

<table>
<thead>
<tr>
<th>Cname</th>
<th>P#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
<td>1</td>
</tr>
<tr>
<td>Codd</td>
<td>2</td>
</tr>
<tr>
<td>Martin</td>
<td>1</td>
</tr>
<tr>
<td>Martin</td>
<td>2</td>
</tr>
</tbody>
</table>

- **SELECT COUNT (*), AVG (PRICE) FROM PRODUCT**
  
  Here, the DML/SELECT statement is expanded with additional arithmetic/statistical functions. This will access the Product relation and perform functions to

  1. count the total number of products and
  2. get the average of the Price values:

<table>
<thead>
<tr>
<th>Count</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1100</td>
</tr>
</tbody>
</table>
Once end users know how to define queries in terms of a particular language, it would seem that they can quite easily do their own queries like the above. It is a matter of a few lines of commands which may be quickly formulated to get the desired information. However if the query is too involved or complex, like the following example, then the end-users will have to be quite expert database users or will have to rely on help from the technical specialists.

```sql
SELECT DISTINCT P# FROM PRODUCT, TRANSACTION
WHERE NOT EXISTS
  (SELECT * FROM PRODUCT, TRANSACTION
   WHERE PRODUCT.P# = TRANSACTION.P#
   AND NOT EXISTS
     (SELECT * FROM PRODUCT, CUSTOMER
      WHERE PRODUCT.P# = TRANSACTION.P#
      AND CUSTOMER.C# = "3" ) )
```

Can you figure what this query statement does?

### 3.3.4 Embedded Host Language

Apart from simple queries, end users need specialised reports that require technical specialists to write computer programs to process them.

The interface for the skilled programmers is usually in the form a database command language and a programming language with utilities to support other data operations. Here in the second case, the DML statements are embedded into the text of an application program written in a general purpose host programming language. Thus SQL statements to access the relational database, for example, are embedded in C, C++ or Cobol programs.

![Figure 3.5: Embedding DML in a Host Language](image)

Embedded host language programs provide the application with full access to the databases to:

- manipulate data structures (such as to create, destroy or update the database tables),
- manipulate the data (such as to retrieve, delete, append or update the data items in the database tables),
Introduction to Databases & Relational DM

- manage groups of statements as a single transaction (such as to abort a group of statements), and
- perform a host of other database management functions as well (such as to create access permits on database tables).

The DML/SQL statements embedded in the program code is usually placed between delimiters such as

```
EXEC SQL
SELECT …
END-EXEC
```

The program is then pre-compiled to convert the DML into the host source code that can subsequently be compiled into object code for running.

Compared to the command lines of queries written in self-contained languages, an application program such as the above takes more effort and time. Good programming abilities are required. Applications are written in an embedded host language for various reasons, including for:

- large or complex databases which contain a hundred million characters or more
- a well known set of applications transactions, perhaps running hundreds of times a day (e.g. to make airline seat reservations) or standard/predictable queries that need to be executed on a routine basis (e.g. “generate the weekly payroll”)
- unskilled end-users or if the query becomes too involved or complicated for the end-user.

However, for end-users, again special interfaces may be designed to facilitate the access to these more complex programs.

![Easy-to-use End User Interface](image)

**Figure 3.6:** Easy-to-use End User Interface

Such interfaces are usually in the form of simple, user-friendly screens comprising easy-to-learn languages, a series of menus, fill-in-the-blank data-entry panels or report screens - sufficient for the user to check, edit and format data, make queries, or see the results, and without much technical staff intervention.
In this section we have seen that user interfaces are important to provide contact with the underlying database. One of the advantages of relational databases is the use of languages that are standardized, such as SQL and the availability of interface products that are easy-to-use. Often it takes just a few minutes to define a query in terms of a Self-Contained language (but the realization of such a query may take much more time). End users can thus create queries and generate their own reports without having to rely heavily on the programming staff to respond to their requests. More importantly, they can also be made more responsible for their own data, information retrieval and report generation. The technical staff must thus ensure that good communication exists between them and the end-users, that sufficient training is always given, and that good coordination all around is vital to ensure that these users know what they are doing.

These are the things that give relational databases their true flexibility and power (compared to the other models such as the hierarchical or network databases).

3.4 Integrity

Having seen what and how data can be manipulated in a database, we shall next see the importance of assuring the reliability of data and how this can be achieved through the imposition of constraints during data manipulation operations.

Apart from providing data access to the user who wishes to retrieve or update the data, a database management system must also provide its users with other utilities to assure them of the proper maintenance, protection and reliability of the system. For example, in single-user systems, where the entire database is used, owned and maintained by a single user, the problems associated with data sharing do not arise. But when an enterprise-wide database is used by many users, often at the same time, the problems of who can access what, when and how - confidentiality and update - become a very big concern. Thus data security and data integrity are crucial.

People should not see what is not intended for them (e.g. individuals must have privacy rights on their medical or criminal records, businesses must safeguard their commercially sensitive information, and the military must secure their operation plans). Additionally, people who are not authorized to update data, must not be allowed to change them (e.g. an electronic bank transfer must have the proper authorization).

While the issue of data security concerns itself with the protection of the database against unauthorized users who may disclose, alter or destroy data they are not supposed to have access to, data integrity concerns itself with protecting the database against authorized users. In this section, we shall focus on the latter (the former will be covered in greater detail in Chapter 11).

Thus integrity protection is an important part of the data model. By integrity we mean the correctness and reliability of the data maintained in the database. One must be confident that the data accessed is accurate, correct, relevant and valid. The protection can be viewed as a set of constraints that prevents any undesirable updates on the database. Two types of constraints may be applied:

- Implicit Integrity Constraints
- Explicit Integrity Constraints
3.4.1 Implicit Integrity Constraints

The term “Implicit Integrity Constraints” means that a user need not explicitly define these Integrity Constraints in a database schema. They are a property of the relational data model as such. The Relational Data Model provides two types of Implicit Integrity Constraints:

1. **Entity Integrity**

Recall that an attribute is usually chosen from a relation to be its primary key. The primary key is used to identify the tuple and is useful when one wishes to sort or access the tuples efficiently. It is a unique identifier for a given tuple. As such, no two tuples can have the same key values, and nor can the values be null. Otherwise, uniqueness cannot be guaranteed.

A primary key that is null would be a contradiction in terms, for it would effectively state that there is an entity that has no known identity. Hence, the term entity integrity.

Note that whilst the primary key cannot be null, (which in this case, C# cannot have a null value), the other attributes, may be so (for example, Cname, Ccity or Cphone may have null values).

Thus the rule for the “Entity Integrity” constraint asserts that no attribute participating in the primary key of a relation is permitted to accept null values.

2. **Referential Integrity**

We have seen earlier how a primary or secondary key in one relation may be used by another relation to handle many-to-many relationships. For example, the Transaction relation has the attribute C# which is also an attribute in the Customer relation. But C# is a primary key of Customer, thus making C# a foreign key in Transaction.

The foreign key in the Transaction relation cross-references data in the Customer relation, e.g. using the value of C# in Transaction to get details on Cname which is not found directly in it but in Customer. When relations make references to another relation via foreign keys, the database management system must ensure that data between the relations are valid. For example, Transaction cannot have a tuple with a C# value that is not found in the Customer relation for the tuple would then be referring to a customer that does not exist.

Thus, for referential integrity a foreign key can have only two possible values - either the relevant primary key or a null value. No other values are allowed.
Figure 3.8 above shows that by adding a tuple with C# 4 means that we have a foreign key that does not reference a valid key in the parent Customer relation. Thus a violation by an insert operation. Likewise, if we were to delete C# 2 from the Customer relation, we would again have a foreign key that no longer references a matching key in the base or parent relation.

Also note that the foreign key here, C# is not allowed to have a null value either since it is a part of Transaction’s primary key (which is the combined attributes of C#, P#, Date). But if the foreign key is a simple attribute, and not a combined/compound one, then it may have null values. In other words, a foreign key cannot be partially null, it must be wholly null if it does not refer to any particular key in the base relation. Unlike primary keys which are not permitted to accept null values, there may be instances when foreign keys have to be null. For example, a database about employees and departments would have a foreign key, say Dept# in the Employee relation which indicates the department to which the employee is assigned. But when a new employee joins the company, it is possible that the employee is not assigned to any department yet. Her Employee tuple may then have a null Dept#.

Thus the rule for the “Referential Integrity” constraint asserts that if a relation R2 includes a foreign key that matches the primary key of another relation R1, then every attribute value of the foreign key in R2 must either (i) be equal to the value of the primary key in some tuple of R1 or (ii) be wholly null.

3.4.2 Explicit Integrity Constraints

In addition to the general, implicit constraints of the relational model, any specific database will often have its own set of local rules that apply to it alone. This is again to ensure that the data values maintained are reliable. Specific validity checks are done on them, for otherwise unexpected or erroneous data may be created. Occasionally, one hears for example, of the case of the telephone subscriber getting an unreasonably large bill.

Various kinds of checks can be imposed. Amongst the usual constraints practised in data processing are tests on:

- class or data type, e.g. alphabetic or numeric type
- sign e.g. positive or negative number
- presence or absence of data, e.g. if spaces or null
- value, e.g. if value > 100
- range or limit, e.g. -10 ≤ x ≤ +10
- reasonableness, e.g. if y positive and < 10000
the consistency, e.g. if \( x < y \) and \( y < 100 \)

*In a Relational Data Model, explicit integrity constraints may be declared to handle the above cases as follows:*

1. **Domain Constraints**
   Such constraints characterize the constraints that can be defined on domains, such as value set restriction, upper and lower limits, etc. For example:

   
   ![Figure 3.9: Domain Constraint Violation](image)

   Whilst the Pname of the third tuple in Figure 3.9 complied with the allowable values, its Price should have been less than 2000. An error message is flagged and the tuple cannot be inserted into the relation.

2. **Tuple Constraints**
   The second type of explicit constraint characterizes the rules that can be defined on tuples, such as inter-attribute restrictions. For example:

   
   ![Figure 3.10: Tuple Constraint Violation](image)

   With such a declaration, then the above tuple with P# 3 cannot have a Price value that is greater or equal to 1500.

3. **Relation Constraints**
   Relation constraints characterize the constraints that can be defined on a relation structure, such as alternate keys or inter-relation restrictions. For example,

   
   ![Figure 3.11: Relational Constraint Violation](image)

   Pname, being an alternate key to the primary key, P#, should have unique values. However, Pname may be null (unlike P# which if null, would violate the entity integrity).
Other attributes too may have null or duplicate values without violating any integrity constraint.

3.5 Database Transactions

All these data manipulation operations are part of the fundamental purpose of any database system, which is to carry out database "transactions" (not to be confused with the relation named “Transaction” that has been used in many of the earlier examples). A database transaction is a logical unit of work. It consists of the execution of an application-specified sequence of data manipulation operations.

Recall that the terminology “Database system” is used to refer to the special system that includes at least the following components:

- the database
- the Database Management System
- the particular database schema, and
- a set of special-purpose application programs

Normally one database transaction invokes a number of DML operators. Each DML operator, in turn, invokes a number of data updating operations on a physical level, as illustrated in Figure 3.14 below.
If the “self contained language” is used, the data manipulation operators and database transactions have a “one-to-one” correspondence. For example, a transaction to put data values into the database entails a single DML INSERT operator command.

However, in other instances, especially where embedded language programs are used, a single transaction may in fact comprise several operations.

Suppose we have an additional attribute, TotalQnt, in the Product relation, i.e. our database will contain the following relations:

- **Customer (C#, Cname, Ccity, Cphone)**
- **Product (P#, Pname, Price, TotalQnt)**
- **Transaction (C#, P#, Date, Qnt)**

TotalQnt will hold the running total of Qnt of the same P# found in the Transaction relation. TotalQnt is in fact a value that is computed as follows:

\[
\text{Product.TotalQnt} \leftarrow \sum_{P\#} \text{Transaction.Qnt}
\]

Consider for example if we wish to add a new Transaction tuple. With this single task, the system will effectively perform the following 2 sequential operations:

- insert a new tuple into the Transaction relation
- update the Product relation such that the new Transaction.Qnt is added on to the value of Product.TotalQnt for that same P#

A transaction must be executed as an intact unit, for otherwise if a failure happens after the insert but before the update operation, then the database will left in an inconsistent state with the new value inserted but the total quantity not updated. But with transaction management and processing support, if an unplanned error or system crash occurs before normal termination, then those earlier operations will be undone - a transaction is executed in its entirety or totally aborted. It is this support to group operations into a
transaction that helps guarantee that a database would be in a “consistent” state in the event of any system failure during its data manipulation operations.

And finally, it must be noted that as far as the end-user is concerned, he/she “can see” database transactions as undivided portions of information sent to the system, or received from the system. It is also not important how the data is actually physically stored, only how it is logically available. This flexibility of data access is readily achieved with relational database management systems.
4 Normalisation

4.1 Introduction

Suppose we are now given the task of designing and creating a database. How do we produce a good design? What relations should we have in the database? What attributes should these relations have? Good database design needless to say, is important. Careless design can lead to uncontrolled data redundancies that will lead to problems with data anomalies.

In this chapter we will examine a process known as Normalisation—a rigorous design tool that is based on the mathematical theory of relations which will result in very practical operational implementations. A properly normalised set of relations actually simplifies the retrieval and maintenance processes and the effort spent in ensuring good structures is certainly a worthwhile investment. Furthermore, if database relations were simply seen as file structures of some vague file system, then the power and flexibility of RDBMS cannot be exploited to the full.

For us to appreciate good design, let us begin by examining some bad ones.

4.1.1 A Bad Design

E. Codd has identified certain structural features in a relation which create retrieval and update problems. Suppose we start off with a relation with a structure and details like:

```
Customer details
C# Cname Ccity
1 Codd London
2 Martin Paris
3 Deen London

Transaction details
P1# Date1 Qnt1 P2# Date2
1 21.01 20 2 23.01
2 26.10 25 2
3 29.01 20
```

Figure 4-1: Simple Structure

This is a simple and straightforward design. It consists of one relation where we have a single tuple for every customer and under that customer we keep all his transaction records about parts, up to a possible maximum of 9 transactions. For every new transaction, we need not repeat the customer details (of name, city and telephone), we simply add on a transaction detail.

However, we note the following disadvantages:

- The relation is wide and clumsy
- We have set a limit of 9 (or whatever reasonable value) transactions per customer. What if a customer has more than 9 transactions?
- For customers with less than 9 transactions, it appears that we have to store null values in the remaining spaces. What a waste of space!
• The transactions appear to be kept in ascending order of P#. What if we have to delete, for customer Codd, the part numbered 1—should we move the part numbered 2 up (or rather, left)? If we did, what if we decide later to re-insert part 2? The additions and deletions can cause awkward data shuffling.

• Let us try to construct a query to “Find which customer(s) bought P# 2”? The query would have to access every customer tuple and for each tuple, examine every of its transaction looking for

\[(P1# = 2) \text{ OR } (P2# = 2) \text{ OR } (P3# = 2) \ldots \text{ OR } (P9# = 2)\]

A comparatively simple query seems to require a clumsy retrieval formulation!

4.1.2 Another Bad Design

Alternatively, why don’t we re-structure our relation such that we do not restrict the number of transactions per customer. We can do this with the following structure:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
<th>P#</th>
<th>Date</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

This way, a customer can have just any number of Part transactions without worrying about any upper limit or wasted space through null values (as it was with the previous structure). Constructing a query to “Find which customer(s) bought P# 2” is not as cumbersome as before as one can now simply state: P# = 2.

But again, this structure is not without its faults:

• It seems a waste of storage to keep repeated values of Cname, Ccity and Cphone.

• If C# 1 were to change his telephone number, we would have to ensure that we update ALL occurrences of C# 1’s Cphone values. This means updating tuple 1, tuple 2 and all other tuples where there is an occurrence of C# 1. Otherwise, our database would be left in an inconsistent state.

• Suppose we now have a new customer with C# 4. However, there is no part transaction yet with the customer as he has not ordered anything yet. We may find that we cannot insert this new information because we do not have a P# which serves as part of the ‘primary key’ of a tuple. (A primary key cannot have null values).

• Suppose the third transaction has been canceled, i.e. we no longer need information about 20 of P# 1 being ordered on 26 Jan. We thus delete the third tuple. We are then left with the following relation:
But then, suppose we need information about the customer “Martin”, say the city he is located in. Unfortunately as information about Martin was held in only that tuple and having the entire tuple deleted because of its P# transaction, meant also that we have lost all information about Martin from the relation.

As illustrated in the above instances, we note that badly designed, unnormalised relations waste storage space. Worse, they give rise to the following storage irregularities:

1. **Update anomaly**
Data inconsistency or loss of data integrity can arise from data redundancy/repetition and partial update.

2. **Insertion anomaly**
Data cannot be added because some other data is absent.

3. **Deletion anomaly**
Data maybe unintentionally lost through the deletion of other data.

### 4.1.3 The Need for Normalisation

Intuitively, it would seem that these undesirable features can be removed by breaking a relation into other relations with desirable structures. We shall attempt by splitting the above Transaction relation into the following two relations, Customer and Transaction, which can be viewed as entities with a one to many relationship.

**Figure 4-2: 1:M data relationships**

<table>
<thead>
<tr>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Let us see if this new design will alleviate the above storage anomalies:

1. **Update anomaly**
If C# 1 were to change his telephone number, as there is only one occurrence of the tuple in the Customer relation, we need to update only that one tuple as there are no redundant/duplicate tuples.
2. **Addition anomaly**

Adding a new customer with C# 4 can be easily done in the Customer relation of which C# serves as the primary key. With no P# yet, a tuple in Transaction need not be created.

3. **Deletion anomaly**

Canceling the third transaction about 20 of P# 1 being ordered on 26 Jan would now mean deleting only the third tuple of the new Transaction relation above. This leaves information about Martin still intact in the new Customer relation.

This process of reducing a relation into simpler structures is the process of **Normalisation**. Normalisation may be defined as a step by step reversible process of transforming an unnormalised relation into relations with progressively simpler structures. Since the process is reversible, no information is lost in the transformation.

Normalisation removes (or more accurately, minimises) the undesirable properties by working through a series of stages called **Normal Forms**. Originally, Codd defined three types of undesirable properties:

1. Data aggregates
2. Partial key dependency
3. Indirect key dependency

and the three stages of normalisation that remove the associated problems are known, respectively, as the:

- First Normal Form (1NF)
- Second Normal Form (2NF), and
- Third Normal Form (3NF)

We shall now show a more formal process on how we can decompose relations into multiple relations by using the Normal Form rules for structuring.

### 4.2 First Normal Form (1NF)

The purpose of the First Normal Form (1NF) is to simplify the structure of a relation by ensuring that it does *not* contain data aggregates or *repeating groups*. By this we mean that no attribute value can have a set of values. In the example below, any one customer has a group of several telephone entries:

<table>
<thead>
<tr>
<th>Customer</th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5555910</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>832551</td>
</tr>
</tbody>
</table>

**Figure 4-3.** Presence of Repeating Groups
This is thus not in 1NF. It must be “flattened”. This can be achieved by ensuring that every tuple defines a single entity by containing only atomic values. One can either re-organise into one relation as in:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>City</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>5555910</td>
</tr>
<tr>
<td>2</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
<tr>
<td>2</td>
<td>Deen</td>
<td>London</td>
<td>832551</td>
</tr>
</tbody>
</table>

Figure 4-4: Atomic values in tuples

or split into multiple relations as in:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>City</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td></td>
<td>2263035</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td></td>
<td>5555910</td>
</tr>
<tr>
<td>2</td>
<td>Deen</td>
<td></td>
<td>2234391</td>
</tr>
<tr>
<td>2</td>
<td>Deen</td>
<td></td>
<td>832551</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C#</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2263035</td>
</tr>
<tr>
<td>1</td>
<td>5555910</td>
</tr>
<tr>
<td>2</td>
<td>2234391</td>
</tr>
<tr>
<td>2</td>
<td>832551</td>
</tr>
</tbody>
</table>

Figure 4-4: Reduction to 1NF

Note that earlier we defined 1NF as one of the characteristics of a relation (Lesson 2). Thus we consider that every relation is at least in the first normal form (thus the Figure 4-3 is not even a relation). The Transaction relation of Figure 4-2 is however a 1NF relation.

We may thus generalise by saying that “A relation is in the 1NF if the values in the relation are atomic for every single attribute of the relation”.

Before we can look into the next two normal forms, 2NF and 3NF, we need to first explain the notion of ‘functional dependency’ as these two forms are constrained by functional dependencies.

4.3 Functional Dependencies

4.3.1 Determinant

The value of an attribute can uniquely determine the value in another attribute. For example, in every tuple of the Transaction relation in Figure 4-2:

- C# uniquely determines Cname
- C# also uniquely determines Ccity as well as Phone
Given C# 1, we will know that its Cname is ‘Codd’ and no other. On the other hand, we cannot say that given Ccity ‘London’, we will know that its Cname is ‘Codd’ because Ccity ‘London’ will also give Cname of ‘Deen’. Thus Ccity cannot uniquely determine Cname (in the same way that C# can).

Additionally, we see that:
- (C#, P#, Date) uniquely determines Qnt

We can now introduce the definition of a “determinant” as being an attribute (or a set of non-redundant) attributes which can act as a unique identifier of another attribute (or another set of attributes) of a given relation.

We may thus say that:
- C# is a unique key for Cname, Ccity and Cphone.
- (C#, P#, Date) is a unique key for Qnt.

These keys are non-redundant keys as no member of the composite attribute can be left out of the set. Hence, C# is a determinant of Cname, Ccity, and Cphone. (C#, P#, Date) is a determinant of Qnt.

A determinant is written as:

\[ A \rightarrow B \]

and can be read as “A determines B” (or A is a determinant of B). If any two tuples in the relation R have the same value for the A attribute, then they must also have the same value for their B attribute.

Applying this to the Transaction relation above, we may then say:

\[ C# \rightarrow Cname \]
\[ C# \rightarrow Ccity \]
\[ C# \rightarrow Cphone \]
\[ (C#, P#, Date) \rightarrow Qnt \]

The value of the attribute on the left-hand side of the arrow is the determinant because its value uniquely determines the value of the attribute on the right.

Note also that:

\[ (Ccity, Cphone) \rightarrow Cname \]
\[ (Ccity, Cphone) \rightarrow C# \]

### 4.3.1 The converse notation

\[ A \xrightarrow{X} B \]

can be read as A “does not determine” B.

Taking again the Transaction relation, we may say therefore that Ccity cannot uniquely determine Cname.
Introduction to Databases & Relational DM

Ccity \rightarrow Cname  
because there exists a number of customers living in the same city.
Likewise:

\[ \text{Cname} \rightarrow (\text{Ccity}, \text{Cphone}) \]
\[ \text{Cname} \rightarrow \text{C#} \]
as there may exist customers with the same name.

4.3.2 Functional Dependence

The role of determinants is also expressed as “functional dependencies” whereby we can say:

“If an attribute \( A \) is a determinant of an attribute \( B \), then \( B \) is said to be functionally dependent on \( A \)”

and likewise

“Given a relation \( R \), attribute \( B \) of \( R \) is functionally dependent on attribute \( A \) if and only if each \( A \)-value in \( R \) has associated with it one \( B \)-value in \( R \) at any one time”.

“C# is a determinant of Cname, Ccity and Cphone” is thus also “Cname, Ccity and Cphone are functionally dependent on C#. Given a particular value of Cname value, there exists precisely one corresponding value for each of Cname, Ccity and Cphone. This is more clearly seen via the following functional dependency diagram:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4-5:** Functional dependencies in the Transaction relation

Similarly, “(C#, P#, Date) is a determinant of Qnt” is thus also “Qnt is functionally dependent on the set of attributes (C#, P#, Date)”. The set of attributes is also known as a composite attribute.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4-6:** Functional dependency on a composite attribute
4.3.3 Full Functional Dependence

“If an attribute (or a set of attributes) \( A \) is a determinant of an attribute (or a set of attributes) \( B \), then \( B \) is said to be fully functionally dependent on \( A \)” and likewise

“Given a relation \( R \), attribute \( B \) of \( R \) is fully functionally dependent on attribute \( A \) of \( R \) if it is functionally dependent on \( A \) and not functionally dependent on any subset of \( A \) (\( A \) must be composite)”.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4-7:** Functional dependencies in the Transaction relation

For the Transaction relation, we may now say that:

- Cname is fully functionally dependent on C#
- Ccity is fully functionally dependent on C#
- Cphone is fully functionally dependent on C#
- Qnt is fully functionally dependent on (C#, P#, Date)
- Cname is not fully functionally dependent on (C#, P#, Date), it is only partially dependent on it (and similarly for Ccity and Cphone).

Having understood about determinants and functional dependencies, we are now in a position to explain the rules of the second and third normal forms.

4.4 Second Normal Form (2NF)

Consider again the Transaction relation which was in 1NF. Recall the operations we tried to do in Section 4.1.2 above and the problems encountered:

1. **Update**

What happens if Codd’s telephone number changes and we update only the first tuple (but not the second)?
2. Insertion

If we wish to introduce a new customer, we cannot do so unless an appropriate transaction is effected.

<table>
<thead>
<tr>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Entity Integrity Violation: P# is a part of primary key!

3. Deletion

If the data about a transaction is deleted, the information about the customer is also deleted. If this happens to the last transaction for that customer the information about the customer will be lost.

<table>
<thead>
<tr>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Clearly, the Transaction relation although it is normalised to 1NF still have storage anomalies. The reason for such violations to the database’s integrity and consistency rules is because of the partial dependency on the primary key.

Recall, the functional dependencies as shown in Figure 4-x. The determinant (C#, P#, Date) is the composite key of the Transaction relation - its value will uniquely determine the value of every other non-key attribute in a tuple of the relation. Note that whilst Qnt is fully functionally dependent on all of (C#, P#, Date), Cname, Ccity and Cphone are only partially functionally dependent on the composite key (as they each depend only on the C# part of the key only but not on P# or Date).

The problems are avoided by eliminating partial key dependence in favour of full functional dependence, and we can do so by separating the dependencies as follows:
The source relation into thus split into two (or more) relations whereby each resultant relation no longer has any partial key dependencies:

We now have two relations, both of which are in the second normal form. These are the same relations of Figure 4-3 above, and the discussion we had earlier clearly shows that the storage anomalies caused by the 1NF relation have now been eliminated:

1. **Update anomaly**
   
   There are no redundant/duplicate tuples in the relation, thus updates are done just at one place without any worry for database inconsistencies.

2. **Addition anomaly**
   
   Adding a new customer can be done in the Customer relation without concern whether there is a transaction for a part or not.
3. **Deletion anomaly**

Deleting a tuple in Transaction does not cause loss of information about Customer details.

More generally, we shall summarise by stating the following:

<table>
<thead>
<tr>
<th>R: (K1, K2, Π1, Π2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 a x</td>
</tr>
<tr>
<td>1 2 b y</td>
</tr>
<tr>
<td>2 1 c x</td>
</tr>
<tr>
<td>2 2 d y</td>
</tr>
</tbody>
</table>

1. Suppose, there is a relation \( R \) where the composite attribute \( (K1, K2) \) is the Primary Key. Suppose also that there exist the following functional dependencies:

\[
(K1, K2) \rightarrow I1
\]

i.e. a full functional dependency on the composite key \( (K1, K2) \).

\[
K2 \rightarrow I2
\]

i.e. a partial functional dependency on the composite key \( (K1, K2) \).

The partial dependencies on the primary key must be eliminated. The reduction of 1NF into 2NF consists of replacing the 1NF relation by appropriate “projections” such that every non-key attribute in the relations are fully functionally dependent on the primary key of the respective relation. The steps are:

1. Create a new relation \( R2 \) from \( R \). Because of the functional dependency \( K2 \rightarrow I2 \), \( R2 \) will contain \( K2 \) and \( I2 \) as attributes. The determinant, \( K2 \), becomes the key of \( R2 \).

2. Reduce the original relation \( R \) by removing from it the attribute on the right hand side of \( K2 \rightarrow I2 \). The reduced relation \( R1 \) thus contain all the original attributes but without \( I2 \).

3. Repeat steps 1. and 2. if more than one functional dependency prevents the relation from becoming a 2NF.

4. If a relation has the same determinant as another relation, place the dependent attributes of the relation to be non-key attributes in the other relation for which the determinant is a key.

**Figure 4-9:** Reduction of 1NF into 2NF
Introduction to Databases & Relational DM

Thus, “A relation R is in 2NF if it is in 1NF and every non-key attribute is fully functionally dependent on the primary key”.

4.5 Third Normal Form (3NF)

A relation in the Second Normal Form can still be unsatisfactory and show further data anomalies. Suppose we add another attribute, Salesperson, to the Customer relation who attends to the needs of the customer.

<table>
<thead>
<tr>
<th>Customer</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
<td>Ccity</td>
<td>Cphone</td>
<td>Salesperson</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>Ducruet</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234931</td>
<td>Smith</td>
</tr>
</tbody>
</table>

Its associated functional dependencies are:

\[ C# \rightarrow \text{Cname, Ccity, Cphone, Salesperson} \]

Consider again operations that we may want to do on the data

1. **Update**

Can we change the salesperson servicing customers in London? Here, we find that there are several occurrences of London customers (e.g. Codd and Deen). Thus we must ensure that we update all tuples such that ‘Smith’ is now replaced by the new salesperson, say ‘Jones’, otherwise we end up with a database inconsistency problem again.

2. **Insertion**

Can we enter data that ‘Fatimah’ is the salesperson for the city of ‘Sarawak’ although no customer exists there yet?

As C# is the primary key, a null value in C# cannot be allowed. Thus the tuple cannot be created.

3. **Deletion**

What happens if we delete the second tuple, i.e. we no longer need to keep information about the customer Martin, his city, telephone and salesperson? If this tuple is removed, we will also lose all information that the salesperson Ducruet services the city of Paris as no other tuple holds this information.

As another, more complex example, consider keeping information about parts that are being kept in bins, where the following relation called Stock
which contains information on:

- the bin number (B#)
- the part number (P#) of the part kept inside a given bin
- the quantity of pieces of the part in a given bin (QB)
- the lead time (LT) taken to deliver a part after an order has been placed for it
- the re-order level (RL) of the part number which indicates the an order must be placed to re-order new stock of that part whenever the existing stock quantity gets too low, i.e. when QB \leq RL

We further assume that:

- parts of a given part number may be stored in several bins
- the same bin holds only one type of part, i.e. it does not hold parts of more than one part number
- the lead time and re-order level are fixed for each part number

The only candidate key for this relation is B#, hence it must be selected as the primary key. Since the B# is a single attribute, the question of partial dependency does not arise (the relation is in Second Normal Form).

Its associated functional dependencies (which are full functional dependencies) are:

\[ B# \rightarrow P#, QB, LT, RL \]

But in this case, we also have data anomalies:

1. **Update**
   Suppose the re-order level for part number 1 is updated, i.e. RL for P# 1 must be changed from 1 to 4. We must ensure that we update all tuples containing P#1, i.e. tuples 1 and 2; any partial updates will lead to an inconsistent database inconsistency.

2. **Insertion**
   Suppose the stock for part number 3 needs to be updated with a new bin number 3. However, this is not allowed since B# is a primary key.
We cannot store LT and RL information for a new expected part number in the database unless we actually have a bin number allocated to it.

3. **Deletion**

If the data (tuple) about a particular bin is deleted, the information about the part is also deleted. If this happens to the last bin containing that part, the information about the concrete part (LT, RL) will also be lost.

From the two examples above, it is still evident that despite having relations in 2NF, problems can still arise and we should now try to eliminate them. It would seem we need a third normal form to eliminate these anomalies.

Scrutinising the functional dependencies of these examples, we notice the existence of “other” dependencies:

**Customer**

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
<th>Salesperson</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
<td>Ducruer</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
<td>Smith</td>
</tr>
</tbody>
</table>

**Figure 4-10**: All functional dependencies in the Customer relation

Notice for example that the dependency of the attribute Salesperson on the key C#, i.e. $C# \rightarrow Salesperson$ is only an indirect or transitive dependency, which is also indicated in the diagram as a dotted arrow $\rightarrow$.

This is considered indirect because $C# \rightarrow Ccity$ and $Ccity \rightarrow Salesperson$, and thus $C# \rightarrow Salesperson$.

Thus for the Stock relation:

$B# \rightarrow P#, QB$ and $P# \rightarrow LT, RL$

then $B# \rightarrow P#, QB, LT, RL$

**Figure 4-11**: All functional dependencies in the Stock relation

The Indirect Dependency obviously causes data duplication (e.g. note the two occurrences of $P#1$ and $LT 12$ in the first two tuples of Stock), which leads to the above anomalies. This can be eliminated by removing all indirect/transitive dependencies. We
do this by splitting the source relation into two or more other relations, as illustrated in the following example:

```
<table>
<thead>
<tr>
<th>B#</th>
<th>P#</th>
<th>QB</th>
<th>Stock:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P#</th>
<th>(LT, RL)</th>
<th>Part:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>2</td>
</tr>
</tbody>
</table>
```

where we can then get

```
<table>
<thead>
<tr>
<th>B#</th>
<th>P#</th>
<th>QB</th>
<th>LT</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>9</td>
<td>18</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P#</th>
<th>LT</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>2</td>
</tr>
</tbody>
</table>
```

We can say that the storage anomalies caused by the 2NF relation can now be eliminated:

1. **Update anomaly**
   To update the re-order level for part number 1, we need only change one (the first) tuple in the new Part relation without concern for other duplicates that used to exist before.

2. **Addition anomaly**
   We can now store LT and RL information for a new part number in the database by creating a tuple in the new Part relation, without concern whether there is a bin number allocated to it or not.

3. **Deletion anomaly**
   Deleting the tuple about a particular bin will remove a tuple form the new Stock relation. Should the part that was deleted from that bin be the only bin where it could be found, however does not mean the loss of data about that part. Information on the LT and RL of the part is still in the new Part relation.

More generally, we shall summarise by stating the following:

Suppose there is a relation R with attributes A, B and C. A is the determinant.

If $A \rightarrow B$ and $B \rightarrow C$ then
A → C is the ‘Indirect Dependency’

(Of course, if A → C and B does not exist, then A → C is a ‘Direct Dependency’)

The Indirect Dependencies on the primary key must be eliminated. The reduction of 2NF into 3NF consists of replacing the 2NF relation by appropriate “projections” such Indirect Key Dependencies are eliminated in favour of the Direct Key Dependencies.

The steps are:
1. Reduce the original relation R by removing from it the attribute on the right hand side of any indirect dependencies A → C. The reduced relation R1 thus contain all the original attributes but without C1.
2. Form a new relation R2 that contains all attributes that are in the dependency B → C.
3. Repeat steps 1. and 2. if more than one indirect dependency prevents the relation from becoming a 3NF.
4. Verify that every determinant in every relation is a key in that relation

\[
\begin{array}{c|c|c}
\text{R:} & \text{A} & \text{B} \quad \text{C} \\
1 & \ldots & x & 5 \\
2 & \ldots & y & 4 \\
3 & \ldots & x & 5 \\
\end{array}
\quad \begin{array}{c|c|c}
\text{R1:} & \text{A} & \text{B} \\
1 & \ldots & x \\
2 & \ldots & y \\
3 & \ldots & x \\
\end{array}
\begin{array}{c|c}
\text{R2:} & \text{B} \quad \text{C} \\
x & 5 \\
y & 4 \\
\end{array}
\]

Figure 4-12. Reduction of 2NF into 3NF

Thus, “A relation R is in 3NF if it is in 2NF and every non-key attribute is fully and directly dependent on the primary key”.

There is another definition of 3NF which states that “A relation is in third normal form if every data item outside the primary key is identifiable by the primary key, the whole primary key and by nothing but the primary key”.

In order to avoid certain update anomalies, each relation declared in the data base schema, should be at least in the Third Normal Form. Structurally, 2NF is better than 1NF, and 3NF is better than 2NF. There are of course other higher normal forms like the Boyce-Codd Normal Form (BCNF), the Fourth Normal Form (4NF) and the Fifth Normal Form (5NF).

However, the Third Normal Form is quite sufficient for most business database design purposes, although some very specialised applications may require the higher-level normalisation.

It must be noted that although normalisation is a very important database design component, we should not always design in the highest level of normalisation, thinking that it is the best. Often at the physical implementation level, the decomposition of
relations into higher normal form mean more pointer movements are required to access and the thus slower the response time of the database system. This may conflict with the end-user demand for fast performance. The designer may sometimes have to “denormalise” some portions of a database design in order to meet performance requirements at the expense of data redundancy and its associated storage anomalies.
5 Relational Algebra (Part I)

5.1 Relational Algebra and Relational Calculus

Many different DMLs can be designed to express database manipulations. Different DMLs will probably differ in syntax, but more importantly they can differ in the basic operations provided. For those familiar with programming, these differences are not unlike that found in different programming languages. There, the basic constructs provided can greatly differ in syntax and in their underlying approach to specifying computations. The latter can be very contrasting indeed—look at, for example, the differences between procedural (e.g., C) and declarative (e.g., Prolog) approaches to specifying computations.

Relational Algebra and Relational Calculus are two approaches to specifying manipulations on relational databases. The distinction between them is somewhat analogous to that between procedural and declarative programming. Thus, before looking at the details of these approaches, it is instructive to briefly digress and look at procedural and declarative programming a little more closely. We will then try to put into context how the algebra and calculus help in the design and implementation of DMLs.

Briefly, the procedural approach specifies a computation to be performed as explicit sequences of operations. The operations themselves are built from a basic set of primitive operations and structuring/control primitives that build higher level operations. An operation can determine the flow of control (i.e., determines the next operation to be executed) and/or cause data to be transformed. Thus programming in the procedural style involves working out the operations and their appropriate order of execution to effect the desired transformation(s) on input data.

The declarative approach, in contrast, would specify the same computation as a description of the logical relationship between input and output data. These relationships are typically expressed as a set of truth-valued sentences about data objects. Neither operations nor their sequences are made explicit by the programmer. Operations are instead implicit in the predetermined set of rules of inference used to derive new sentences from those given (or derived earlier).

In other words, if you used a procedural programming language, you must specify how input is to be transformed to desired outputs using the basic operations available in the language. However, if you used a declarative language, you must instead describe what relationships must be satisfied between inputs and outputs, but essentially say nothing about how a given input is to be transformed to the desired output (it is for the system to determine how, typically by some systematic application of inference rules).

Amidst such differences, users are right to raise some basic concerns. Are these differences superficial? Or do they mean that there can be computations specifiable in one but not in the other? If the latter were the case, programmers must avoid choosing languages that simply cannot express some desired computation (assuming we know exactly the limitations of each and every programming language)! Fortunately for programmers, this is not the case. It is a well-known result that every general-purpose programming language (whether procedural or declarative) is equivalent to every other.
That is, if something is computable at all, it can be specified in any of these languages. Such equivalence is established by reference to a fundamental model of computation\(^1\) that underlies the notion of computability.

Now what can we say about the different (existing and future) database languages? Can two different languages be said to be equivalent in some sense? To answer this question we must first ask whether, in relational databases, there is a fundamental model of database manipulation? The answer is yes—Relational Calculus defines that model!

Let us first state a fundamental definition:

A language that can define any relation definable in relational calculus is relationally complete.

This definition provides a benchmark by which any existing (and future) language may be judged as to its power of expression. Different languages may thus be equivalent in the sense of being relationally complete\(^2\).

Relational Calculus, as we shall see later, provides constructs (well-formed expressions) to specify relations. These constructs are very much in a declarative style. Relational Algebra, on the other hand, also provides constructs for relational database manipulations but in a more procedural style. Moreover, it has also been established that Relational Algebra is equivalent to Relational Calculus, in that every expression in one has an equivalent expression in the other. Thus relational completeness of a database language can also be established by showing that it can define any relation expressible in Relational Algebra.

Why then should we bother with different languages and styles of expression if they are all in some sense equivalent? The answer is that besides equivalence (relational or computational), there are other valid issues that different language designs try to address including the level of abstraction, precision, comprehensibility, economy of expression, ease of writing specifications, efficiency of execution, etc. Declarative constructs, by virtue of their descriptive nature (as opposed to the prescriptive nature of procedural constructs), are closer to natural language and thus easier to write and understand. Designers of declarative languages try to provide constructs that even end-users (with a little training) can use to formulate, for example, ad hoc queries. Declarative constructs, however, execute less efficiently than equivalent procedural ones. Attempts to make them more efficient typically involve first, automatic translation to equivalent procedural form, and second, optimising the resulting expressions according to some predetermined rules of optimisation. In fact, this was the original motivation for the Algebra, ie. providing a

\(^1\) The Universal Turing Machine (model of computation) is accepted as defining the class of all computable functions. Every programming language shown to be equivalent to it is therefore equivalent with every other.

\(^2\) Note however, that relational completeness is not the same as computational completeness, ie. Relational Calculus is not equivalent to general-purpose programming languages. It is a specialised calculus intended for the Relational Database Model. Thus while two languages may be relationally complete, each may have features over and above that required for relational completeness (but these need not concern us here).
set of operations to which expressions of the Calculus could be translated and subsequently optimised for execution. But the efficiency of even this approach cannot match that of carefully hand-coded procedural specifications. Thus for periodical and frequently executed manipulations, it is more efficient to use algebraic forms of database languages.

Because of the fundamental nature and role of the Relational Algebra and Calculus in relational databases, we will look at them in depth in the ensuing chapters. This will provide readers with the basic knowledge of database manipulations possible in the model. We begin in this chapter with Relational Algebra.

5.2 Overview of Relational Algebra

Relational Algebra comprises a set of basic operations. An operation is the application of an operator to one or more source (or input) relations to produce a new relation as a result. This is illustrated in Figure 5-1 below. More abstractly, we can think of such an operation as a function that maps arguments from specified domains to a result in a specified range. In this case, the domain and range happen to be the same, i.e., relations.

![Figure 5-1 A Relational Algebraic Operation](image)

This is no different in principle to, say, operations in arithmetic. For example, the ‘add’ operation in arithmetic takes two numbers and produces a number as a result. In arithmetic, we are used to writing expressions to denote operations and their results. Thus, the ‘add’ operation is usually written using the ‘+’ symbol (the operator) placed between its two arguments, e.g., 145 + 168. Moreover, because an expression denotes the result of the operation (which is of the same type as its input arguments), it itself can be written as an argument in another operation, allowing us to construct complex expressions to denote one result, e.g., 145 + 168 – 20 \times 3.

Complex expressions that combine different operations are evaluated by a sequence of reductions. The sequence, in the case of arithmetic expressions, is determined by the familiar precedence of operators. Thus, the expression 145 + 168 – 20 \times 3 would be reduced as follows:

\[
145 + 168 – 20 \times 3 \rightarrow 145 + 168 – 60 \rightarrow 313 – 60 \rightarrow 253
\]

This default precedence can be overridden with the explicit use of parenthesis. Thus,
Introduction to Databases & Relational DM

\[(145 + (168 - 20)) \times 3 \rightarrow (145 + 148) \times 3 \rightarrow 393 \times 3 \rightarrow 1179\]

All these would of course be elementary to the reader! The point though is that the basic operations of relational algebra are defined to allow manipulations of relations in much the same way that arithmetic operations manipulate numbers above. With appropriately defined symbols to denote operators and syntax to denote the application of operators to arguments (relations), relational expressions combining multiple operations can be constructed to denote a resultant relation. And as with arithmetic expressions, (algebraic) relational expressions are evaluated by reduction in some specified (default or explicit) order. Thus the earlier statement that relational algebra is basically procedural in nature, ie. operations and their sequencing are explicitly specified to achieve a particular transformation of input(s) to output.

The analogy with arithmetic above has been useful to highlight the basic nature of relational algebra. However, the algebra’s basic operations are much more complex than those of arithmetic. They are in fact much more related to operations on sets (viz. intersection, union, difference, etc). This is not surprising as relations are after all special kinds of sets.

As mentioned above, a relational operation maps relations to a relation. As a relation is completely defined by its intension and extension, the complete definition of a relational operation must specify both the schema of the resultant relation and the rules for generating its tuples from the input relation(s). In the following, we will do just that. Moreover, in the interest of clarity as well as precision of presentation, we will define each basic operation both informally and formally.

While notations used to denote the basic operators and operations may differ in the literature, there is no disagreement in their basic logical definitions. It will be necessary for us to use some concrete notation in what follows and, rather than introducing yet more notations, we have chosen in fact to use Codd’s original notation.

5.3 Selection

Selection is used to extract tuples from a relation. A tuple from the source relation is selected (or not) based on whether it satisfies a specified predicate (condition). A predicate is a truth-valued expression involving tuple component values and their relationships. All tuples satisfying the predicate are then collected into the resultant relation. The general effect is illustrated in Figure 5-2.
For example, consider the ‘Customer’ source relation below:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

The result of a selection operation applied to it with the condition that the attribute ‘Ccity’ must have the value “London” will be:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

because these are the only tuples that satisfy the stated condition. Procedurally, you may think of the operation as examining each tuple of the source relation in turn (say from top to bottom), checking to see if it met the specified condition before turning attention to the next tuple. If a tuple satisfies the condition, it is included in the resultant relation, otherwise it is ignored.

We shall use the following syntax to express a selection operation:

```
select <source-relation-name>
where <predicate>
giving <result-relation-name>
```

The <source-relation-name> must already have been defined in the database schema, or is the name of the result of one of previous operations.

In its simplest form, the <predicate> is a simple scalar comparison operation, ie. expressions of the form

```
<value_1> <comparator> <value_2>
```
where <comparator> is one of any comparison operator (ie. =, <, >, ≥, ≤, etc). <value_i> denotes a scalar value and is either a valid attribute name in the source relation, or a constant value. If an attribute name is specified, it denotes the corresponding value in the tuple under consideration. Thus, the example operation above could have been denoted by the following construct:

$$\text{select Customer where Ccity = "London"}$$

Of course, arguments to a comparator must reduce to values from the same domain.

The <predicate> may also take the more complex form of a boolean combination of simple comparison operations above, using the boolean operators ‘AND’, ‘OR’, and ‘NOT’.

The <result-relation-name> is a unique name that is associated with the result relation. It can be viewed as a convenient abbreviation that can be used as <source-relation-name>s in subsequent operations. Thus, the full selection specification corresponding to the example above is:

$$\text{select Customer where Ccity = "London" giving Result}$$

Note that the intension of the resultant relation is identical to that of the source relation. In other words, the result of selection has the same number of attributes (columns) and attribute names (column labels) as the source relation. Its overall effect, therefore, is to derive a ‘horizontal’ subset of the source relation.

As another example, consider the following relation. Each tuple represents a sales transaction recording the customer number of the customer purchasing the goods (C#), the product number of the goods sold (P#), the date of the transaction (Date) and the quantity sold (Qnt).

<table>
<thead>
<tr>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Suppose now that we are interested in looking at only those transactions which took place before February 26 with quantities of more than 25 or involving customer number 2. This need would be translated into the following selection:

$$\text{select Transaction where Date < 26.01 AND Qnt > 25 OR C# = 2 giving Result}$$

and would yield the relation:
This example illustrates the use of boolean combinations in the `<predicate>`. However, formulating complex predicates is not as simple and straightforward as it may seem. The basic problem is having to deal with ambiguities at two levels.

First, the informal statement (typically in natural language) of the desired condition may itself be ambiguous. The alert reader would have noted that the phrase (in the above example)

“…only those transactions which took place before February 26 with quantities of more than 25 or involving customer number 2…”

has two possible interpretations:

- a) a transaction is selected if it is before February 26 and its quantity is more than 25, or it is selected if it involves customer number 2
- b) all selected transactions must be those that are before February 26 but additionally, each must either involve a quantity of more than 25 or a customer number of 2 (or both)

Such ambiguities must be resolved first before construction of the equivalent boolean expression is attempted. In the above example, the first interpretation was assumed.

Second, the formal boolean expressions involving AND, OR and NOT may also be ambiguous. The source of ambiguity is not unlike that for natural language (ambiguity of strength or order of binding).

Thus

\[ Cname = \text{“Codd” AND Ccity = “London” OR Ccity = “Paris”} \]

may be interpreted as

- a) a customer Codd who either lives in London or Paris (ie. the OR binds stronger and before AND)
- b) a customer Codd who lives in London, or any customer who lives in Paris (ie. the AND binds stronger and before OR)

Because of its more formal nature, however, such ambiguities are easier to overcome. It can in fact be eliminated through a convention of operator precedences and explicit (syntactical) devices to override default precedences. The conventions used are in fact as follows:
1. expressions enclosed within parentheses have greater precedence (ie. binds stronger). Thus,
   \[ \text{Cname} = \text{“Codd” AND (Ccity = “London” OR Ccity = “Paris”)} \]
   can only take interpretation (a) above

2. The order of precedence of the boolean connectives, unless overridden by parentheses, are (from highest to lowest) NOT, AND, OR. Thus,
   \[ \text{Cname} = \text{“Codd” AND Ccity = “London” OR Ccity = “Paris”} \]
   can only take interpretation (b) above

There is no limit to the level of ‘nesting’ of parentheses, ie. a parenthesised expression may have within it a parenthesised subexpression, which may in turn have within it a parenthesised sub-subexpression, etc. Given any boolean expression and the above conventions, we can in fact construct a precedence tree that visually depicts its unique meaning. Figure 5-3 illustrates this. A leaf node represents a basic comparison operation, whereas a non-leaf node represents a boolean combination of its children. A node deeper in the tree binds stronger than those above it. Alternatively, the tree can be viewed as a prescription for reducing (evaluating) a boolean expression to a truth-value (\text{true} or \text{false}). To reduce a node:

   a) if it is a leaf node, replace it with the result of its associated comparison operation
   b) if it is a non-leaf node, reduce each of its children; then replace it with the result of applying its associated boolean operation on the truth-values of its children

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{precedence_tree}
\caption{A boolean expression and its precedence tree}
\end{figure}

Using these simple conventions, we can check that expressions we construct indeed carry the intended meanings. (The reader can go back the last example and ascertain that the intended interpretation was indeed correctly captured in the predicate of the selection statement)
At this point, we should say a few words about the notation, particularly in the context of the analogy to arithmetic expressions in the last section. Strictly speaking, the full selection syntax above is not an expression that can be used as an argument to another operation. This does not contradict the analogy, however. The selection syntax, in fact, has an expression component comprising the select- and where-clauses only, i.e. without the giving-clause:

```
select <source-relation-name> where <predicate> giving <result-relation-name>
```

Thus,

```
select Customer where Ccity = "London"
```

is an expression that completely specifies a selection operation while denoting also its result, in much the same way that ‘2+3’ completely specifies an addition operation while also denoting the resultant number (i.e. 5). The expression, therefore, can syntactically occur where a relation is expected and it would then be valid to write:

```
select (select Customer where Ccity = "London") where C# < 3
```

Strictly speaking, this is all we need to define the selection operation. So, of what use is the giving-clause? The answer, in fact, was alluded to earlier when we described the clause as allowing us to introduce a convenient abbreviation. It is convenient, and useful, especially in simplifying and making more readable what may otherwise be unwieldy and confusing expressions. Even the simple double selection expression above may already look unwieldy to the reader (imagine what the expression would look like if it involved 10 algebraic operations, say!). It would be clearer to write:

```
select Customer where Ccity = "London" giving Result;
select Result where C# < 3 giving Result2
```

(of course, this operation could have been more simply and clearly written as a single selection operation involving a boolean combination of comparison operations; the reader should attempt to write it as an exercise)

Mathematical notation too have various devices to introduce abbreviations to simplify and make expressions more readable. What we are doing here with the giving-clause is analogous to, for example, writing:

```
let x = 2+3
let y = 7–2
let z = (x–y) × (x+y)
```

instead of the unabbreviated “((2+3)–(7–2)) × ((2+3)+(7–2))”. The giving-clause is thus mainly a stylistic device. It is important to note that that is all it is - introducing a temporary abbreviation to be used in another operation. In particular, it is not to be interpreted as permanently modifying the database schema with the addition of a new relation name.

In this book, we will favour this notational style because we think it leads to a simpler and more comprehensible notation. The reader should note, however, that while other
descriptions in the literature may favour and use only the expression forms, the differences are superficial.

**Formal Definition**

If $\sigma$ denotes a relation, then let

- $S(\sigma)$ denote the finite set of attribute names of $\sigma$ (ie. its intension)
- $T(\sigma)$ denote the finite set of tuples of $\sigma$ (ie. its extension)
- $\text{dom}(\alpha)$, where $\alpha \in S(\sigma)$, denote the set of allowable values for $\alpha$
- $\tau \bullet \alpha$, where $\tau \in T(\sigma)$ and $\alpha \in S(\sigma)$, denote the value of attribute $\alpha$ in tuple $\tau$

The **selection** operation takes the form

```
select $\sigma$ where $\pi$ giving $\rho$
```

where $\pi$ is a predicate expression. The syntax of a predicate expression is given by the following BNF grammar (this should be viewed as an abstract syntax not necessarily intended for an end-user language):

- $\text{pred\_exp} ::= \text{comp\_exp} | \text{bool\_exp} | ( \text{pred\_exp} )$
- $\text{bool\_exp} ::= \text{negated\_exp} | \text{binary\_exp}$
- $\text{negated\_exp} ::= \text{NOT} \text{pred\_exp}$
- $\text{binary\_exp} ::= \text{pred\_exp} \text{bool\_op} \text{pred\_expr}$
- $\text{bool\_op} ::= \text{AND} | \text{OR}$
- $\text{comp\_exp} ::= \text{argument} \text{comparator} \text{argument}$
- $\text{comparator} ::= > | < | \geq | \leq | =$
- $\text{argument}^3 ::= \text{attribute\_name} | \text{literal}$

$\pi$ is well-formed iff it is syntactically correct and

- for every attribute_name $\alpha$ in $\pi$, $\alpha \in S(\sigma)$
- for every comp_expr $\alpha_1 * \alpha_2$ (where ‘*’ denotes a comparator) such that $\alpha_1, \alpha_2 \in S(\sigma)$, either $\text{dom}(\alpha_1) \subseteq \text{dom}(\alpha_2)$ or $\text{dom}(\alpha_2) \subseteq \text{dom}(\alpha_1)$
- for every comp_expr $\alpha * \kappa$, or $\kappa * \alpha$ (where ‘*’ denotes a comparator) such that $\alpha \in S(\sigma)$ and $\kappa$ is a literal, $\kappa \in \text{dom}(\alpha)$

Further, let $\pi(\tau)$ denote the application of a well-formed predicate expression $\pi$ to a tuple $\tau \in T(\sigma)$. $\pi(\tau)$ reduces $\pi$ in the context of $\tau$, ie. the occurrence of any $\alpha \in S(\sigma)$ in $\pi$ is first replaced by $\tau \bullet \alpha$. The resulting expression is then reduced to a truth-value according to the accepted semantics of comparators and boolean operators.

Then $\rho$, the resultant relation of the selection operation, is characterised by the following:

---

3 The syntax of ‘attribute_name’ and ‘literal’ are unimportant in what follows and we leave it unspecified
5.4 Projection

Whereas a selection operation extracts rows of a relation meeting specified conditions, a projection operation extracts specified columns of a relation. The desired columns are simply specified by name. The general effect is illustrated in Figure 5-4.

![Figure 5-4: The projection operation](image)

We could think of selection as eliminating rows (tuples) not meeting the specified conditions. In like manner, we can think of a projection as eliminating columns not named in the operation. However, an additional step is required for projection because removing columns may result in duplicate rows, which are not allowed in relations. Quite simply, any duplicate occurrence of a row must be removed so that the result is a relation (a desired property of relational algebra operators).

For example, using again the customer relation:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

its projection over the attribute ‘Ccity’ would yield (after eliminating all columns other than ‘Ccity’):

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ccity</td>
</tr>
<tr>
<td>London</td>
</tr>
<tr>
<td>Paris</td>
</tr>
</tbody>
</table>

Duplicates
Note the duplication of row 1 in row 3. Projection can result in duplication because the resultant tuples have a smaller degree whereas the uniqueness of tuples in the source relation is only guaranteed for the original degree of the relation. For the final result to be a relation, duplicated occurrences must be removed, ie.

<table>
<thead>
<tr>
<th>Result</th>
<th>Ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td></td>
</tr>
</tbody>
</table>

The form of a projection operation is:

```
project <source-relation-name>
over <list-of-attribute-names>
giving <result-relation-name>
```

Thus the above operation would be written as:

```
project Customer
over Ccity
giving Result
```

As with selection, `<source-relation-name>` must be a valid relation—a relation name defined in the database schema or the name of the result of a previous operation. `<list-of-attribute-names>` is a comma-separated list of at least one identifier. Each identifier appearing in the list must be a valid attribute name of `<source-relation-name>`. And finally, `<result-relation-name>` must be a unique identifier used to name the resultant relation.

Why would we want to project a relation over some attributes and not others? Quite simply, we sometimes are interested in only a subset of an entity’s attributes given a particular situation. Thus, if we needed to telephone all customers to inform them of some new product line, data about a customer’s number and the city of residence are superfluous. The relevant data, and only the relevant data, can be presented using:

```
project Customer
over Cname, Cphone
giving Result
```

```
Result
Cname  | Cphone  |
-------|---------|
Codd   | 2263035 |
Martin | 5555910 |
Deen   | 2234391 |
```

Extending this example, suppose further that we have multiple offices sited in major cities and the task of calling customers is distributed amongst such offices, ie. the office
in London will call up customers resident in London, etc. Now the simple projection above will not do, because it presents customer names and phone numbers without regard to their place of residence. If it was used by each office, customers will receive multiple calls and you will probably have many annoyed customers on your hands, not to mention the huge phone bills you unnecessarily incurred!

The desired relation in this case must be restricted to only customers from a given city. How can we specify this? The simple answer is that we cannot - not with just the projection operation. However, the alert reader would have realised that the requirement to restrict resultant rows to only those from a given city is exactly the sort of requirement that the selection operation is designed for! In other words, here we have an example of a situation that needs a composition of operations to compute the desired relation. Thus, for the office in London, the list of customers and phone numbers relevant to it is computed by first selecting customers from London, then projecting the result over customer names and phone numbers. This is illustrated in Figure 5-5. For offices in other cities, only the predicate of the selection needs to be appropriately modified.

Note that the order of the operations is significant, i.e. a selection followed by a projection. It would not work the other way around (you can verify this by trying it out yourself).

![Figure 5-5 Combining operators to compute a desired relation](image)

**Formal Definition**

If \( \sigma \) denotes a relation, then let

- \( S(\sigma) \) denote the finite set of attribute names of \( \sigma \) (i.e. its intension)
- \( T(\sigma) \) denote the finite set of tuples of \( \sigma \) (i.e. its extension)
- \( \tau \cdot \alpha \), where \( \tau \in T(\sigma) \) and \( \alpha \in S(\sigma) \), denote the value of attribute \( \alpha \) in tuple \( \tau \)

The projection operation takes the form

**project** \( \sigma \) over \( \delta \) giving \( \rho \)

where \( \delta \) is a comma-separated list of attribute names. Formally, \( \delta \) (as a discrete structure) may be considered a tuple, but having a concrete enumeration syntax (comma-separated list).
Let $\text{S}_{\text{tuple}}(x)$ denote the set of elements in the tuple $x$. Then, $\delta$ must observe the following constraint:

$$\text{S}_{\text{tuple}}(\delta) \subseteq \text{S}(\sigma)$$

ie. every name occurring in $\delta$ must be a valid attribute name in the relation $\sigma$.

Furthermore, if $\tau \in \text{T}(\sigma)$ and $\tau'$ denotes a tuple, we define:

$$R(\tau, \delta, \tau') \equiv \forall \alpha \bullet \alpha \in \text{S}_{\text{tuple}}(\delta) \iff \tau \bullet \alpha \in \text{S}_{\text{tuple}}(\tau')$$

ie. a tuple element $\tau \bullet \alpha$ is in the tuple $\tau'$ if and only if the attribute name $\alpha$ occurs in $\delta$.

Then $\rho$, the resultant relation of the projection, is characterised by the following:

- $\text{S}(\rho) \equiv \text{S}_{\text{tuple}}(\delta)$
- $\text{T}(\rho) \equiv \{ \tau' | \tau \in \text{T}(\sigma) \land R(\tau, \delta, \tau') \}$

### 5.5 Natural Join

The next operation we will look at is the Natural Join (hereafter referred to simply as Join). This operation takes two source relations as inputs and produces a relation whose tuples are formed by concatenating tuples from each input source. It is basically a cartesian product of the extensions of each input source. However, not all possible combinations of tuples necessarily end up in the result. This is because it implicitly selects from among all possible tuple combinations only those that have identical values in attributes shared by both relations.

![Diagram of Join](image)

**Figure 5-6** The Join combines two relations over one or more common domains

Thus, in a typical application of a Join, the intensions of the input sources share at least one attribute name or domain (we assume here that attribute names are global to a schema, ie. the same name occurring in different relations denote the same attribute and value domain). The Join is said to occur over such domain(s). Figure 5-6 illustrates the general effect. The shaded left-most two columns of the inputs are notionally the shared attributes. The result comprise these and the concatenation of the other columns from each input. More precisely, if the degree of the input sources were $m$ and $n$ respectively, and the number of shared attributes was $s$, then the degree of the resultant relation is $(m+n-s)$. 

As an example, consider the two relations below:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
</tr>
</tbody>
</table>

These relations share the attribute ‘C#’, as indicated. To compute the join of these relations, consider in turn every possible pair of tuples formed by taking one tuple from each relation, and examine the values of their shared attribute. So if the pair under consideration was

\(<1, \text{Codd, London, 2263035}>\text{ and } <1, 1, 21.01, 20>\)

we would find that the values match exactly. In such a case, we concatenate them and add the concatenation to the resultant relation. It doesn’t matter if the second tuple is concatenated to the end of the first, or the first to the second, as long as we are consistent about it. By convention, we use the former. Additionally, we omit the second occurrence of the shared attribute in the result (repeated occurrence is superfluous). This gives us the tuple

\(<1, \text{Codd, London, 2263035, 1, 21.01, 20}>\)

If, on the other hand, the pair under consideration was

\(<3, \text{Deen, London, 2234391}>\text{ and } <1, 1, 21.01, 20>\)

we would ignore it because the values of their shared attributes do not match exactly.

Thus, the resultant relation after considering all pairs would be:

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

The foregoing description is in fact general enough to admit operations on relations that do not share any attributes at all \((s = 0)\). The join, in such a case, is simply the cartesian product of the input sources’ extensions (the condition that tuple combinations have identical values over shared attributes is vacuously true since there are no shared attributes!). However, such uses of the operation are atypical.

Syntactically, we will write the Join operation as follows:

\[
\text{join } \langle\text{source-relation-name}\rangle_1 \text{ AND } \langle\text{source-relation-name}\rangle_2 \\
\text{over } \langle\text{attribute-name-list}\rangle \\
\text{giving } \langle\text{result-relation-name}\rangle
\]
where again

- `<source-relation-name>` is a valid relation name (in the schema or the result of a previous operation)
- `<attribute-name-list>` is a comma-separated non-empty list of attribute names, each of which must occur in both input sources, and
- `<result-relation-name>` is a unique identifier denoting the resultant relation

With this syntax, particularly with the `over`-clause, we have in fact taken the liberty

(1) to insist that the join must be over at least one shared attribute, ie. we disallow expressions of pure cartesian products of two relations that do not share any attribute. This restriction is of no practical consequence, however, as in practice a Join is used to bring together information from different relations related through some common value.

(2) to allow a join over a subset of shared attributes, ie. we relax (generalise) the restriction that a Join is over all shared attributes.

If a Join is over a proper subset of shared attributes, then shared attributes not specified in the `over`-clause will each have its own column in the result relation. But in such cases, the respective column labels will be qualified names. We will adopt the convention of writing a qualified name as ‘ρ.α’, where α is the column label and ρ the relation name in which α appears. As an illustration, consider the relations below:

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>abc</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>def</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>ijk</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>pqr</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>xyz</td>
</tr>
</tbody>
</table>

The operation

**join** R1 AND R2 over A1 giving Result

will yield

<table>
<thead>
<tr>
<th>A1</th>
<th>R1.A2</th>
<th>X</th>
<th>R2.A2</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>ijk</td>
<td>3</td>
<td>pqr</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>ijk</td>
<td>2</td>
<td>xyz</td>
</tr>
</tbody>
</table>

To see why Join is a necessary operation in the algebra, consider the following situation (assume as context the Customer and Transaction relations above): the company decided that customers who purchased product number 1 (P# = 1) should be informed that a fault has been discovered in the product and that, as a sign of good faith and of how it values its customers, it will replace the product with a brand new fault-free one. To do this, we need to list, therefore, the names and phone numbers of all such customers.
First, we need to identify all customers who purchased product number 1. This information is in the Transaction relation and, using the following selection operation, it is easy to limit its extension to only such customers:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>P#</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Next, we note that the resultant relation only identifies such customers by their customer numbers. What we need, though, are their names and phone numbers. In other words, we would like to extend each tuple in A with the customer name and phone number corresponding to the customer number. As such items are found in the relation Customer which shares the attribute ‘C#’ with A, the join is a natural operation to perform:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>P#</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

With B, we have practically derived the information we need—in fact, more than we need, since we are interested only in the customer name (the ‘Cname’ column) and phone number (the ‘Cphone’ column). But as we’ve learned, the irrelevant columns may be easily removed using projection, as shown below.

<table>
<thead>
<tr>
<th>B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
</tr>
</tbody>
</table>

As a final example, let us also assume we have the Product relation, in addition to the Customer and Transaction relations:

<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

The task is to “get the names of products sold to customers in London”. Once again, this task will require a combination of operations which must involve a Join at some point because not all the information required are contained in one relation. The sequence of operations required is shown below.
select Customer where Ccity = “London” giving A

join A AND Transaction over C# giving B

join B AND Product over P# giving C

project C over Pname giving Result
Formal Definition

As before, if $\sigma$ denotes a relation, then let

- $S(\sigma)$ denote the finite set of attribute names of $\sigma$ (ie. its intension)
- $T(\sigma)$ denote the finite set of tuples of $\sigma$ (ie. its extension)
- $\tau\bullet\alpha$, where $\tau \in T(\sigma)$ and $\alpha \in S(\sigma)$, denote the value of attribute $\alpha$ in tuple $\tau$

Further, if $\tau_1$ and $\tau_2$ are tuples, let $\tau_1^\tau_2$ denote the tuple resulting from appending $\tau_2$ to the end of $\tau_1$.

We will also have need to use the terminology introduced in defining projection above, in particular, $S_{\text{tuple}}$ and the definition:

$$R(\tau, \delta, \tau') \equiv \forall \alpha \cdot \alpha \in S_{\text{tuple}}(\delta) \iff \tau\bullet\alpha \in S_{\text{tuple}}(\tau')$$

The (natural) join operation takes the form

**join $\sigma$ AND $\nu$ over $\delta$ giving $\rho$**

As with other operations, the input sources $\sigma$ and $\nu$ must denote valid relations that are either defined in the schema or are results of previous operations, and $\rho$ must be a unique identifier to denote the result of the join. $\delta$ is a tuple of attribute names such that:

$$S_{\text{tuple}}(\delta) \subseteq (S(\sigma) \cap S(\nu))$$

Let $\varepsilon = (S(\sigma) \cap S(\nu)) - S_{\text{tuple}}(\delta)$, ie. the set of shared attribute names not specified in the over-clause. We next define, for any relation $r$:

$$\text{Rename}(r, \varepsilon) \equiv \{ \alpha \mid \alpha \in S(r) - \varepsilon \vee (\alpha = 'r.p' \land p \in S(r) \cap \varepsilon) \}$$

In the case that $\varepsilon = \{\}$ or $S(r) \cap \varepsilon = \{\}$, $\text{Rename}(r, \varepsilon) = S(r)$.

The Join operation can then be characterised by the following:

- $S(\rho) \equiv \text{Rename}(\sigma, \varepsilon) \cup \text{Rename}(\nu, \varepsilon)$
- $T(\rho) \equiv \{ \tau_1^\tau_2 \mid \tau_1 \in T(\sigma) \land \tau \in T(\nu) \land R(\tau, \Delta, \tau_2) \land \forall \alpha \cdot \alpha \in S_{\text{tuple}}(\delta) \Rightarrow \tau_1\bullet\alpha = \tau\bullet\alpha \}$

where

$$S_{\text{tuple}}(\Delta) = S(\nu) - S_{\text{tuple}}(\delta)$$

---

**Table**

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pname</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>VDU</td>
</tr>
</tbody>
</table>
Relational Algebra (Part II)

6.1 Introduction

In the previous chapter, we introduced relational algebra as a fundamental model of relational database manipulation. In particular, we defined and discussed three important operations it provides: Select, Project and Natural Join. These constitute what is called the basic set of operators and all relational DBMS, without exception, support them.

We have presented examples of the power of these operations to construct solutions (derived relations) to various queries. However, there are classes of practical queries for which the basic set is insufficient. This is best illustrated with an example. Using again the same example domain of customers and products they purchase, let us consider the following requirement:

“Get the names of customers who had purchased both product number 1 and product number 2”

All the required pieces of data are in the relations shown above. It is quite easy to see what the answer is—from the Transaction relation, customers number 1 and number 2 are the ones we are interested in, and cross-referencing the Customer relation (to retrieve their names) the customers are Codd and Martin respectively. Now, how can we construct this solution using the basic operation set?

Working backwards, the final relation we wish to construct is a single-column relation with the attribute ‘Cname’. Thus, the last operation needed will be a projection of some relation over that attribute. Such a relation must first be the result of joining Customer and Transaction (over ‘C#’), since Customer alone does not have data on products purchased. Second, it must contain only tuples of customers who had purchased products 1 and 2, ie. some form of selection must be applied. This analysis suggests that the required sequence of operations is a Join, followed by a Select, and finally a Project.

The following then may be a possible solution:

```
join Customer AND Transaction over C# giving A
select A where P# = 1 AND P# = 2 giving B
project B over Cname giving Result
```

The join results in:
At this point, however, we discover a problem: the selection on A results in an empty relation!

The problem is the selection condition: no tuple can possibly satisfy a condition that requires a *single* attribute to have *two different values* (“P# = 1 AND P# = 2”). This is obvious once it is pointed out, although it might not have been so at first glance. Thus while the selection statement is syntactically correct, its logic is erroneous. What is needed, effectively, is to select tuples of a particular customer only if there exists one with P# = 1 and another with P# = 2, ie. the form of selection needed is dependent across tuples. But the basic Select operator cannot express this because it operates on each tuple *in turn and independently* of one another.4

Thus the proposed solution above is not a solution at all. In fact, no combination of the basic operations can handle the query or other queries of this sort, for example:

“Get the names of customers who bought the product CPU but not the product VDU”, or

“Get the names of customers who bought every product type that the company sells”, etc

These examples suggest that additional operations are needed. In the following, we shall present them and show how they are used.

We will round up this chapter and our discussion of relational algebra with a discussion of two other important topics: how operations handle “null” values, and how sequences of operations can be optimised for performance. A null value is inserted into a tuple field to denote an (as yet) unknown value. Clearly, this affects the evaluation of conditions involving attribute values. Exactly how will be explained in Section 6.4. Finally, we will see that there may be several different sequences of operations that derive the same result. In such cases, we may well ask which sequence is more efficient, ie. least costly or better in performance, in some sense. A more precise notion of ‘efficiency’ of operators and how a given operator sequence can be made more efficient will be discussed in section 6.5.

---

4 Some readers may have noted that if OR was used instead of AND in the selection operation, the desired result would be constructed. However, this is coincidental. The use of OR is logically erroneous—it means one or the other, but not necessarily both. To see this, change the example slightly by deleting the last tuple in Transaction and recompute the result (using OR). Your answer would still be Codd and Martin, but the correct answer should be Codd alone!
6.2 Division

As the name of this operation implies, it involves dividing one relation by another. Division is in principle a partitioning operation. Thus, $6 \div 2$ can be paraphrased as partitioning a single group of 6 into a number of groups of 2—in this case, 3 groups of 2. The basic terminology used in arithmetic will be used here as well. Thus in an expression like $x \div y$, $x$ is the dividend and $y$ the divisor. Division does not always yield whole groups of the divisor, eg. $7 \div 2$ gives 3 groups of 2 and a remainder group of 1. Relational division too can leave remainders but, much like integer division, we ignore remainders and focus only on constructing whole groups of the divisor.

The manner in which a relational dividend is partitioned is a little more complex. First though, we should ask what aspect of a relation is being partitioned? The answer simply is the set of tuples in the relation. Next, we ask how we decide to group some tuples together and not others? Not surprisingly, the basis for such decisions has to do with the attribute values in the tuples. Let’s take a look at an example first before we describe the process more precisely.

The illustration above shows how we may divide a relation $R$, which is a simple binary relation in this case with two attributes $A_1$ and $A_2$. For clarity, the values of attribute $A_1$ have been sorted so that a given value appears in contiguous rows (where there’s more than one). The question we’re interested in is which of these values have in common an arbitrary subset of values of attribute $A_2$.

For example,

```
“which values of A1 share the subset \{a,b\} of A2?”
```

By inspecting $R$, the reader can verify that the answer are the values 1 and 2, because only tuples with these $A_1$ values have corresponding $A_2$ entries of both ‘a’ and ‘b’. Put another way, the tuples of $R$ are grouped by the common denominator or divisor $\{a,b\}$. This is shown in the relation $R’$ where we emphasise the groups formed using double-line borders. Other tuples (the remainder of the division) are ignored. Note that $R’$ is not the final result of division—it is only an intermediate working result. The desired result are the values of attribute $A_1$ in it, or put another way, the projection of $R’$ over $A_1$.

From this example, we can see that a division of a relation $R$ is performed over some attribute of $R$. The divisor is a subset of values from that attribute domain and the result is a relation comprising the remaining attributes of $R$. In relational algebra expressions, the divisor is in fact specified by another relation $D$. For this to be meaningful at all, $D$ must
have at least one attribute in common with the R. The division is over the common attribute(s) and the set of values used as the actual divisor are the values found in D. The general operation is depicted in the figure below.

Figure 6-1. The Division Operation

Figure 6-2 shows a simple example of dividing a binary relation R1 by a unary relation R2. The division is over the shared attribute I2. The divisor is the set \{1,2,3\}, these being the values found in the shared attribute in R2. Inspecting the tuples of R1, the value ‘a’ occur in tuples such that their I2 values match the divisor. So ‘a’ is included in the result. ‘b’ is not, however, as there is no tuple <b,2>.

Figure 6-2 Division of a binary relation by a unary relation

We can now specify the form of the operation:

```
divide <dividend-relation-name> by <divisor-relation-name>
giving <result-relation-name>
```

<dividend-relation-name> and <divisor-relation-name> must be names of defined relations or results of previous operations. <result-relation-name> must be a unique name used to denote the result relation. As mentioned above, the divisor must share attributes with the dividend. In fact, we shall insist (on a stronger condition) that the intension of the divisor must be a subset of the dividend’s. This is not really a restriction as any relation that shares attributes with the dividend can be turned into the required form simply by projecting over them.
We can now show how division can be used for the type of queries mentioned in the introduction. Take the query:

“Get the names of customers who bought every product type that the company sells”

The Transaction relation records customers who have ever bought anything. For this query, however, we are not interested in the dates or purchase quantities but only in the product types a customer purchased. So we project Transaction over C# and P# to give us a working relation A. This is shown on the left side of the following illustration. Next, we need all the product types the company sells, and these may be obtained by projecting the relation Product over P# to give us a working relation B. This is shown on the right side of the illustration.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>P#</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Now as we are interested in only those customers that purchased all products (i.e. all the values in B), B is thus used to divide A to result in the working relation C. In this case, there is only one such customer. Finally, the details of the customer are obtained by joining C with the Customer relation over C#.
Formal Definition

To formally define the Divide operation, we will use the notation introduced and used in Chapter 5. However, for convenience, we repeat here principal definitions to be used.

If $\sigma$ denotes a relation, then let

- $S(\sigma)$ denote the finite set of attribute names of $\sigma$ (i.e., its intension)
- $T(\sigma)$ denote the finite set of tuples of $\sigma$ (i.e., its extension)
- $\tau\cdot\alpha$, where $\tau \in T(\sigma)$ and $\alpha \in S(\sigma)$, denote the value of attribute $\alpha$ in tuple $\tau$
- $S_{\text{tuple}}(x)$ denote the set of elements in tuple $x$

Furthermore, if $\tau \in T(\sigma)$, $\tau'$ denotes a tuple, and $S_{\text{tuple}}(\delta) \subseteq S(\sigma)$, we define:

$$R(\tau, \delta, \tau') \equiv \forall \alpha \cdot \alpha \in S_{\text{tuple}}(\delta) \Leftrightarrow \tau\cdot\alpha \in S_{\text{tuple}}(\tau')$$

The Divide operation takes the form

**divide $\sigma$ by $\nu$ giving $\rho$**

As with other operations, the input sources $\sigma$ and $\nu$ must denote valid relations that are either defined in the schema or are results of previous operations, and $\rho$ must be a unique identifier to denote the result of the division. The intensions of $\sigma$ and $\nu$ must be such that

$S(\nu) \subset S(\sigma)$

The Divide operation can then be characterised by the following:

- $S(\rho) \equiv S(\sigma) - S(\nu)$
- $T(\rho) \equiv \{ \tau \mid \tau_1 \in T(\sigma) \land R(\tau_1, \Delta, \tau) \land T(\nu) \subseteq \text{IM}(\tau) \}$

where

- $S_{\text{tuple}}(\Delta) = S(\rho)$,
- $S_{\text{tuple}}(\delta) = S(\nu)$, and
- $\text{IM}(\tau) = \{ t' \mid t \in T(\sigma) \land R(t, \delta, t') \land R(t, \Delta, \tau) \}$

6.3 Set Operations

Relations are basically sets. We should, therefore, be able to apply standard set operations on them. To do this, however, we must observe a basic rule: a set operation on two or more sets is meaningful if the sets comprise values of the same type. This is so that comparison of values from different sets is meaningful. It is quite pointless, for example, to attempt an intersection of a set of integers and a set of names. We can still perform the operation, of course, but we can already tell at the outset that the result will be a null set because any value from one will never be equal to any value from the other.
To ensure this rule is observed for relations, we need to state what it means for two relations to comprise values of the same type. As a relation is a set of tuples, the values we are interested in are the tuples themselves. So when is it meaningful to compare two tuples for equality? Clearly, the structure of the tuples must be identical, i.e., the tuples must be of equal length and their corresponding elements must be of the same type. Only then can two tuples be equal, i.e., when their corresponding element values are equal. The structure of a tuple, put another way, is in fact the intension or schema of the relation it occurs in. Thus, meaningful set operations on relations require that the source relations have identical intensions/schemas. Such relations are said to be union-compatible.

The set operations included in relational algebra are Union, Intersection, and Difference. Keeping in mind that they are applied to whole tuples, these operations behave in exactly the standard way. It goes without saying that their results are also relations with intensions identical to the source relations.

The Union operation takes the form

\[
<\text{source-relation-1}> \text{ union } <\text{source-relation-2}> \text{ giving } <\text{result-relation}>
\]

where \(<\text{source-relation-1}>\) are valid relations or results of previous operations and are union-compatible, and \(<\text{result-relation}>\) is a unique identifier denoting the resulting relation.

Figure 6-3 illustrates this operation.

![Figure 6-3 Relational Union Operation](image)

The Intersection operation takes the form

\[
<\text{source-relation-1}> \text{ intersect } <\text{source-relation-2}> \text{ giving } <\text{result-relation}>
\]

where \(<\text{source-relation-1}>\) are valid relations or results of previous operations and are union-compatible, and \(<\text{result-relation}>\) is a unique identifier denoting the resulting relation.

Figure 6-4 illustrate this operation.
The Difference operation takes the form

\[ \text{<source-relation-1> minus <source-relation-2> giving <result-relation>} \]

where <source-relation-1> are valid relations or results of previous operations and are union-compatible, and <result-relation> is a unique identifier denoting the resulting relation.

Figure 6-5 illustrate this operation.

As an example of the need for set operations, consider the query: “which customers purchased the product CPU but not the product VDU?”

The sequence of operations to answer this question is quite lengthy, but not difficult. Probably the best way to construct a solution is to work backwards and observe that if we had a set of customers who purchased CPU (say W1) and another set of customers who purchased VDU (say W2), then the solution is obvious: we only want customers that appear in W1 but not in W2, or in other words, the operation “W1 minus W2”.

The problem now has been reduced to constructing the sets W1 and W2. Their constructions are similar, the difference being that one focuses on the product CPU while the other the product VDU. We show the construction for W1 below.
Finally, details of such customers are obtained by joining Z1 and Customer, giving the desired relation W1.
The construction for $W_2$ is practically identical to that above except that the selection operation specifies the condition “Pname = VDU”. The reader may like to perform these steps as an exercise and verify that the following relation is obtained:

<table>
<thead>
<tr>
<th></th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

Now we need only perform the difference operation “$W_1$ minus $W_2$ giving Result” to construct a solution to the query:

<table>
<thead>
<tr>
<th></th>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
</tbody>
</table>

### Formal Definition

If $\sigma$ denotes a relation, then let

- $S(\sigma)$ denote the finite set of attribute names of $\sigma$ (ie. its intension)
- $T(\sigma)$ denote the finite set of tuples of $\sigma$ (ie. its extension)

The form of set operations is

$$\sigma <\text{set operator}> \lambda \text{ giving } \rho$$

where $<\text{set operator}>$ is one of ‘union’, ‘intersect’ or ‘minus’; $\sigma, \lambda$ are source relations and $\rho$ the result relation. The source relations must be union-compatible, ie. $S(\sigma) = S(\lambda)$.

The set operations are characterised by the following:

- $S(\rho) = S(\sigma) = S(\lambda)$ for all $<\text{set operator}>$s
- for ‘union’
  $$T(\rho) \equiv \{ t \mid t \in T(\sigma) \lor t \in T(\lambda) \}$$
- for ‘intersect’
  $$T(\rho) \equiv \{ t \mid t \in T(\sigma) \land t \in T(\lambda) \}$$
- for ‘minus’
  $$T(\rho) \equiv \{ t \mid t \in T(\sigma) \land t \notin T(\lambda) \}$$

### 6.4 Null values

In populating a database with data objects, it is not uncommon that some of these objects may not be completely known. For example, in capturing new customer information through forms that customers are requested to fill, some fields may have been left blank.
(some customers may take exception to revealing their age or phone numbers!). In these cases, rather than not have any information at all, we can still record those that we know about. But what value do we insert into the unknown fields of data objects? Leaving a field blank is not good enough as it can be interpreted as an empty string which may be a valid value for some domains. We need a value that denotes ‘unknown’ and that cannot be confused with valid domain values.

It is here that the *Null* value is used. We can think of it as a special value different from any other value from any attribute domain. At the same time, we may think of it as belonging to every attribute domain in the database, ie. it may appear as a value for any attribute and not violate any type constraints. Syntactically, different DBMSs may use different symbols to denote null values. For our purposes, we will use the symbol ‘?’.

How do null values affect relational operations? All relational operations involve comparing values in tuples, including Projection (which involves comparison of result tuples for duplicates). The key to answering this question is in how we evaluate boolean operations involving null values. Thus, for example, what does “? > 5” evaluate to? The unknown value could be greater than 5. But then again, it may not be. That is, the value of the boolean expression cannot be determined on the basis of available information. So perhaps we should consider the result of the comparison as unknown as well?

Unfortunately, if we did this, the relational operations we’ve discussed cease to be well-defined! They all rely on comparisons evaluating categorically to one of two values: TRUE or FALSE. For example, if the above comparison (“? > 5”) was generated in the process of selection, we would not know whether to include or exclude the associated tuple in the result if we were to admit a third value (UNKNOWN). If we wanted to do that, we must go back and redefine all these operations based on some form of three-valued logic.

To avoid this problem, most systems that allow null values simply interpret any comparison involving them as FALSE. The rationale is that even though they could be true, they are not demonstrably true on the basis of what is known. That is, the result of any relational operation conservatively includes only tuples that demonstrably satisfy conditions of the operation. Adopting this convention, all the operations defined previously still hold without any amendment. Some implications on the outcome of each operation are considered below.

For the Select operation, an unknown value cannot identify a tuple. This is illustrated in Figure 6-6 which shows two Select operations applied to the relation R. Note that between the two operations, the selection criteria ranges over the entire domain of the attribute I2. One would expect therefore, that any tuple in R1 would either be in the result of the first or the second. This is not the case, however, as the second tuple in R1 (<b,?>) is not selected in either operation—the unknown value in it falsifies the selection criteria of both operations!
For Projection, tuples containing null values that are otherwise identical are not considered to be duplicates. This is because the comparison “? = ?”, by the above convention, evaluates to FALSE. This leads to the situation as illustrated in Figure 6-7 below. The reader should note from this example that the symbol ‘?’ , while it denotes some value much like a mathematical variable, is quite unlike the latter in that it’s occurrences do not always denote the same value. Thus “? = ?” is not demonstrably true and therefore considered FALSE.

In a Join operation, tuples having null values under the common attributes are not concatenated. This is illustrated in Figure 6-8 (“?=1”, “1=?” and “?=?” are all FALSE).
In Division, the occurrence of even one null value in the divisor means that the result will be an empty relation, as any value in the dividend’s common attribute(s) will fail when matched with it. This is illustrated in Figure 6-9 below. Note, however, that this is not necessarily the case if only the dividend contains null values under the common attribute(s)—division may still be successful on tuples not containing null values.

![Figure 6-9 Division with null divisors](image)

In set operations, because tuples are treated as a single unit in comparisons, a single rule applies: tuples otherwise identical but containing null values are considered to be different (as was the case for Projection above). Figure 6-10 illustrates this for each set operation. Note that because of the occurrence of null values, the tuples in R2 are not considered duplicates of R1’s tuples. Thus their union simply collects tuples from both relations; subtracting R2 from R1 simply results in R1; and their intersection is empty.

![Figure 6-10 Set operations involving null values](image)

**6.5 Optimisation**

Each relational operation entails a certain amount of work: retrieving a tuple, examining a tuple’s attribute values, comparing attribute values, creating new tuples, repeating a process on each tuple in a relation, etc. For a given operation, the amount of work clearly varies with the cardinality of source relation(s). For example, a selection performed on a
relation twice the cardinality of another (of the same degree) would involve twice as much work.

We can also compare the relative amount of work needed between different operations based on the number of tuples processed. An operation with two source inputs, for example, need to repeat its logic on every possible tuple-pair formed by taking a tuple from each input relation. Thus if we had two relations of cardinalities $M$ and $N$ respectively, a total of $M \times N$ tuple-pairs must be processed, i.e. $M$ (or $N$) times more than, say, a selection operation on each individual relation. Of course, this is not an exact relative measure of work, as there are also differences in the amount of work expended by different operations at the tuple level. By and large, however, we are interested in the order of magnitude of work (rather than the exact amount of work) and this is fairly well approximated by the number of tuples processed.

We will call such a measure the efficiency of an operation. Thus, the efficiency of selection and projection is the cardinality of its single input relation, while the efficiency of join, divide and set operations is the product of the respective cardinalities of their two input relations.

Why should the efficiency of operations interest us? Consider the following sequence of operations:

\begin{verbatim}
join Customer AND Transaction over C# giving X;
select X where CCity = “London” giving Result
\end{verbatim}

Suppose the cardinality of Customer was 100 and that of Transaction was 1000. Then the efficiency of the join operation is $100 \times 1000 = 100000$. The cardinality of $X$ is 1000 (as it is certainly intended that the C# in every Transaction tuple matches a C# in one of the Customer tuples). Therefore, the efficiency of the selection is 1000. As these two operations are performed one after another, the efficiency of the entire sequence of operations is naturally the sum of their individual efficiencies, i.e. $100000 + 1000 = 101000$.

Now consider the following sequence:

\begin{verbatim}
select Customer where CCity = “London” giving X;
join X AND Transaction over C# giving Result
\end{verbatim}

The reader can verify that this sequence is relationally equivalent to the first, i.e. they produce identical results. But how does its efficiency compare with that of the first? Let us calculate using the same assumptions about the cardinalities. The efficiency of the selection is 100. To estimate the efficiency of the join, we need to make an assumption on the cardinality of $X$. Let’s say that 10 customers live in London. Then the efficiency of the join is $10 \times 1000 = 10000$, and the efficiency of the sequence as a whole is $100 + 10000 = 101000$—ten times more efficient than the first!

Of course, the reader may think that the assumption about $X$’s cardinality was contrived to give this dramatic performance improvement. The point, however, is that the second sequence can do no worse than the first, i.e. if all customers in the Customer relation live in London, then it performs as poorly as the first. More likely, however, we expect a performance improvement.
The above example illustrates a very important point about relational algebra: there can be more than one (sequence of) expression that describe a desired result. The main aim of optimisation, therefore, is to translate a given (sequence of) expression into its most efficient equivalent form. Such optimisation may be done manually by a human user or automatically by the database management system. Automatic optimisation may in fact do better because the automatic optimiser has access to information that is not readily available to a human optimiser, eg. current cardinalities of source relations, current data values, etc. But the overwhelming majority of relational DBMS's available today merely execute operations requested by users as is. Thus, it is important that users know how to perform optimisations manually.

For manual optimisation, it is perhaps less important to derive the most efficient form of a query than to follow certain guidelines, heuristics or rules-of-thumb that lead to more efficient expressions. Frequently the latter will lead to acceptable performance and expending more effort to find the optimal expression may not significantly improve that performance if good heuristics are used. There is, in fact, a simple and effective rule to remember when writing queries: delay as long as possible the use of expensive operations! In particular, we should wherever possible put selection ahead of other operations because it reduces the cardinality of relations. Figure 6-11 illustrate the application of this principle. The reader should be able to verify that the two sequences of operations are logically equivalent and that intuitively the selection operations before the joins can significantly improve the efficiency of the query.

**Query:**
Get names of customers who bought product CPU earlier than 24.01.

| Join Customer And Transaction Over C# Giving X; |
| Join X And Product Over P# Giving Y; |
| Select Y Over Pname=CPU And Date<24.01 Giving W; |
| Project W Over Cname Giving Result; |

✅ **Bad! The Joins operate on large relations.**

| Select Product Where Pname=CPU Giving PI; |
| Select Transaction Where Date<24.01 Giving TI; |
| Join Customer And TI Over C# Giving X; |
| Join X AND PI Over P# Giving W; |
| Project W Over Cname Giving Result; |

✅ **Good solution!**

---

**Figure 6-11** Delay expensive operations
7 Relational Calculus (Part I)

7.1 Introduction

We established earlier the fundamental role of relational algebra and calculus in relational databases (see 5.1). More specifically, relational calculus is the basis for the notion of relational completeness of a database language, ie. any language that can define any relation expressible in relational calculus is relationally complete.

Relational Algebra (see chapters 5 and 6) is one such language. Its approach is procedural, ie. it provides a number of basic operations on relations and successive applications of these operations must be properly sequenced to derive answers to database queries. The basic operators are in themselves quite simple and easy to understand. However, except for fairly simple queries, the construction of operation sequences can be quite complex. Furthermore, such constructions must also consider efficiency issues and strive to find optimal ones (see 6.5). A considerable amount of programming skill is therefore required to effectively use relational algebra.

Relational Calculus takes a different approach to the human–database interface. Rather than requiring users to specify how relations are to be manipulated, it only requires them to define what the desired result is. How the result is actually computed, ie. the operations used, their sequencing and optimisation, is left to the database management system to work out. As it doesn’t deal with procedures (ie. sequencing of operations), this approach is frequently termed non-procedural or declarative.

Relational Calculus is mainly based on the well-known Propositional Calculus, which is a method of calculating with sentences or declarations. Such sentences or declarations, also termed propositions, are ones for which a truth value (ie. “true” or “false”) may be assigned. These can be simple sentences, such as “the ball is red”, or they may be more complex involving one or more simple sentences, such as “the ball is red AND the playing field is green”. The truth value of complex sentences will of course depend on the truth values of their components. This is in fact what the calculus ‘calculates’, using rules for combining truth values of component sentences.

In Relational Calculus, the sentences we deal with are simpler and refer specifically to the relations and values in the database of interest. Simple sentences typically take the form of comparisons of values denoted by variables or constants, eg. $X \geq 3$, $X < Y$, etc. More complex sentences are built using logical connectives And (‘&’) and Or (‘|’), eg. $X > 7 \ & \ X < Y \ | \ X \leq 5$. Simple and complex sentences like these are examples of Well-Formed Formulae, which we will define fully later.

Regardless of their exact syntax, a formula is in principle a logical function with one or more free variables. For purposes of illustration, we will write such functions as in the following annotated example:
In the above example, there is one free variable, X. The value of the function can be computed for specific instances of X. Thus,

\[ F(15) \equiv (15 > 12 & 15 < 18) \equiv (\text{true} & \text{true}) \equiv \text{true} \]

\[ F(10) \equiv (10 > 12 & 10 < 18) \equiv (\text{false} & \text{true}) \equiv \text{false} \]

Additionally, free variables are deemed to range over a set of permitted values, i.e. only such values can instantiate them. We shall see the significance of this later, as applied to relations. But just to illustrate the concept for now, consider the following function over two free variables:

\[ F(X,Y) =: X > Y & Y < 12 \]

Suppose X ranges over \{8, 15\} and Y ranges over \{7, 14\}. Then F(8, 14) and F(15, 7) are allowable instantiations of the function, with truth values false and true respectively, whereas F(1000,200) is not a valid instantiation. Such restrictions of values over which free variables range become significant when we interpret a formula as the simple query: “get the set of values of free variables for which the formula evaluates to true”. Thus, for the above formula, we need only construct the following table involving only the permitted values:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>false</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>false</td>
</tr>
</tbody>
</table>

The desired set of values can then be read from the rows where F(X,Y) evaluated to true, i.e. the set \{(8,7), (15,7)\}.

Relational Calculus is an application of the above ideas to relations. We will develop these ideas in greater detail in the following sections.

### 7.2 Tuple Variables

Free variables in logical functions can in principle range over any type of value. A feature that distinguishes Relational Calculus from other similar calculi is that the free variables range over relations. More specifically, any free variable ranges over the extension of a designated relation, i.e. the current set of tuples in the relation. Thus, a free variable may be instantiated with a tuple selected from the designated relation.

Suppose, for example, we introduced a variable C to range over the relation Customer, as in Figure 7-1. Then C may be instantiated with any one of the three tuples at any one time. The example shows C instantiated with the second tuple. Equivalently, we may
sometimes say that C ‘holds’ a value instead of being instanntiated with that value. In any case, because variables like C range over tuples (or is only permitted to hold a tuple), they are termed tuple variables.

A tuple has component parts, and unless we have a means of referring to such parts, the logical functions we formulate over relations will have limited expressive power. Given, for example, two variables X and Y that range over two different relations with a common domain, we may want to specify a condition where their current instantiations are such that the values under the common domain are identical. Thus while X (and Y) denote a tuple as a whole, we really wish to compare tuple component values. The syntactic mechanism provided for this purpose takes the form:

\(<\text{tuple-variable-name}.\text{<attribute-name>}>\)

and is interpreted to mean the value associated with \(<\text{attribute-name}>\) in the current instantiation of \(<\text{tuple-variable-name}>\). Thus, assuming the instantiation of C as in Figure 7-1:

\[\begin{align*}
C.C# &= 2 \\
C.Cname &= 'Martin' \\
\text{etc}
\end{align*}\]

This denotation of a particular data item within a tuple variable is often referred to as a projection of the tuple variable over a domain (eg. “C.Cname” is a projection of tuple variable C over the domain Cname).

Relational Calculus is a collection of rules of inference of the form:

\(<\text{target list}> : \text{<logical expression}>\)

where \(<\text{target list}>\) is a list of free variables and/or their projections that are referenced in \(<\text{logical expression}>\). This list is thought of as the “target list” because the set of instantiations of the list items that makes \(<\text{logical expression}>\) true is the desired result. In other words, an inference rule may be thought of as a query, and may be informally understood as a request to find all variable instantiations that satisfy \(<\text{logical expression}>\) and, for each such instantiation, to extract the data items mentioned in \(<\text{target list}>\).

For example, consider the inference rule in Figure 7-2. It references one free variable, C, which ranges over Customer. The \(<\text{target list}>\) specifies items we are interested in - only

---

5 This terminology may perhaps be favoured by programmers who are used to programming language variables and to thinking about them as memory locations that can ‘hold’ one value at a time.
the phone number in this case - but only of those tuples that satisfy the <logical expression>. In other words, the rule may be paraphrased as the query to “get the set of phone numbers of customers who live in London”. Note that the use of the variable C both in <target list> and in <logical expression> denotes the same instantiation, thereby ensuring that “C.Cphone” is extracted from the same tuple that satisfies the comparison “C.Ccity = London”. The computed set in this case would be \{2263035, 2234391\}, corresponding to the phone numbers in the first and last tuples - these being the only tuples satisfying “C.Ccity = London”.

![Figure 7-2 An inference rule over the Customer relation](image)

The reader should note the simplicity and declarative character of the inference rule, which merely states the desired result (the <target list>) and the conditions that must be satisfied (the <logical expression>) for a value to be included in the result. Contrast this with relational algebra which would require the following construction:

\[
\begin{align*}
\text{select } & \text{Customer where } \text{Ccity} = \text{‘London’ giving } X; \\
\text{project } & X \text{ over Cphone giving Result}
\end{align*}
\]

The above example only used a single variable. However, a single variable can only range over a single relation, while often the data items of interest are spread over more than one relation. In such cases, we will need more than one tuple variable.

Figure 7-3 illustrates such a case involving two variables, P and T, ranging over relations Product and Transaction respectively. Note that the inference rule at the top of the figure

- lists items from both variables in the target list (ie. P.Pname, T.C#)
- compares in the logical expression projections of the two different variables over the same domain (T.P# = P.P#)

It further illustrates specific instantiations of each variable and evaluation of the logical expression in the context of these instantiations. In this case, the logical expression is true and therefore the items in the target list are extracted from the variables (shown in the “result” table). It is important to note that a given inference, as in this illustration, is entirely in the context of a specific instantiation of each tuple variable. It is meaningless, for example, to evaluate “T.P# = P.P#” using one instance of P and “P.Price > 1000” using another. The total number of inferences that can be attempted for any given rule is therefore the product of the cardinality of each variable’s range.

The inference rule in this example may be paraphrased as the query “find the customer numbers and product names, priced at more than 1000, that they purchased”. As an exercise, the reader should attempt to construct this query in relational algebra (hint: it will involve the basic operators Select, Project and Join).
7.3 Quantifiers

Logical expressions may also include variable quantifiers, specifically:

1. the existential quantifier, denoted by the symbol ‘∃’, and
2. the universal quantifier, denoted by the symbol ‘∀’

These quantifiers quantify variables. An existentially quantified variable, say x, is written “∃x” and is read as “there exists an x such that…”. A universally quantified variable is written as “∀x” and is read as “for all x …”.

Quantification is applied to a formula and is written preceding it. For example, 

\[ ∃x \ (x < y & y < 12) \]

would be read as “there exists an x such that x is less than y and y is less than 12”. The formula to which the quantification is applied is called the scope of quantification. Occurrences of quantified variables in the scope of quantification are said to be bound (existentially or universally). The scope is normally obvious from the written expressions, but if ambiguities might otherwise arise, we will use parenthesis to delimit scope.

Informally, the formula “∃x ( <expr> )” asserts that there exists at least one value of x (from among its range of values) such that <expr> is true. This assertion is false only when no value of x can be found to satisfy <expr>. On the other hand, if the assertion is true, there may be more than one such value of x, but we don’t care which. In other words, the truth of an existentially quantified expression is not a function of the quantified variable(s).⁶

As an example, consider the unquantified expression

\[ x < y & y < 12 \]

and suppose x ranges over \{4,15\} and y over \{7,14\}. The truth table for the expression is:

---

⁶ The truth of a quantified expression does depend, of course, on the range of permitted values of the quantified variables.
Now consider the same expression but with $x$ existentially quantified:

$$\exists x \ (x < y \land y < 12)$$

Since we don’t care which value of $x$ makes the expression true as long as there is at least one, its truth depends only on the unbound variable $y$:

<table>
<thead>
<tr>
<th>$y$</th>
<th>$\exists x \ (x &lt; y \land y &lt; 12)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>14</td>
<td>false</td>
</tr>
</tbody>
</table>

An existentially quantified expression therefore has a distinctly different meaning from the same expression unquantified. In particular, when $<$logical expression$>$ of an inference rule is existentially quantified, it becomes a query on the free variables only, since it is a function of only those variables.

Consider, for example, the Product and Transaction relations in Figure 7-4, with tuple variables $P$ ranging over the former and $T$ over the latter. The rule

$$P.Pname: \exists T \ (T.P\# = P.P\# \ And \ T.C\# = 1)$$

is interpreted as the query to find values of the free variable $P$ such that there exists at least one value of the bound variable $T$ satisfying the formula “$T.P\# = P.P\# \ And \ T.C\# = 1$”. As before, evaluation of the expression is in the context of some instantiations of the variables. All possible values of $P$ must be considered, but for each we need only find one value of $T$ to satisfy expression. Once we have done that, other possible values of $T$, if any, may be ignored. For example, with $P$ instantiated to the first tuple of Product, we consider in turn tuples of Transaction as values of $T$. We will find in fact that the first already satisfies the expression and we may therefore ignore the others. With $P$ set to the second tuple, however, we will find no value for $T$ to satisfy the expression. The result for this example therefore is only one value for $P$, with $P.Pname=CPU$. 

<table>
<thead>
<tr>
<th>Product</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P#$</td>
<td>$T#$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
As another example, consider the relations in Figure 7-5 with associated tuple variables as shown. Suppose, we are interested in finding the names of customers who bought the product CPU. That is, our target is the value \( X.Cname \), but only if \( X \) is a customer who has bought the product CPU. In other words, there must exist a \( Y \) such that “\( X.C# = Y.C# \)”. This would establish that \( X \) bought a product, denoted by \( Y.P# \) (the product number). Furthermore, this product must be a CPU, i.e. there must exist a \( Z \) such that “\( Y.P# = Z.P# \) and “\( Z.Pname = CPU \)”. Thus, the rule corresponding to our query is

\[
X.Cname: \exists Y \exists Z ( X.C# = Y.C# & Y.P# = Z.C# & Z.Pname = CPU )
\]

The reader can verify that the answer satisfying this query is \{Codd, Martin\}.

![Figure 7-5] Product, Customer and Transaction relations with associated tuple variables

Using again the relations in Figure 7-5, let’s look at a more complex query: get the names of customers who bought the product CPU and VDU. At first glance, this seems a simple extension of the above query:

\[
X.Cname: \exists T1 \exists T2 \exists P1 \exists P2 ( X.C# = T1.C# & X.C# = T2.C# &

T1.P# = P1.C# & P1.Pname = CPU &

T2.P# = P2.C# & P2.Pname = VDU )
\]

But the reader who remembers a similar example in section 6.1 would have noted a problem. Specifically, the subexpression “\( Z.Pname = CPU \) & \( Z.Pname = VDU \)” can never be true for a given value of \( Z \)—a field of a given tuple can only hold one value, so only one or the other can be true but not both! Of course what we mean to specify is that the customer purchased at least one product which is a CPU, and another which is a VDU. Since a tuple variable can hold only one value at a time, this clearly cannot be done using only one tuple variable. The solution, therefore, is to introduce additional distinct variables to range over the same relation when more than one tuple is to be considered at a time (note that relational calculus places no restriction on the number of distinct variables that can range over a relation). For this particular example, we need only introduce one additional variable each for the relations Transaction and Product respectively, as shown in Figure 7-6. This will allow us to consider two separate purchases at one time. The correct formulation, therefore, is:

\[
X.Cname: \exists T1 \exists T2 \exists P1 \exists P2 ( X.C# = T1.C# & X.C# = T2.C# &

T1.P# = P1.C# & P1.Pname = CPU &

T2.P# = P2.C# & P2.Pname = VDU )
\]
Figure 7-6 additionally shows particular values of these variables that satisfy our query.

Let’s turn now to the universal quantifier. Informally, the formula “∀x ( <expr> )” asserts that for every value of x (from among its range of values) <expr> is true. Like the existential quantifier, the truth of an existentially quantified expression is not a function of the quantified variable(s). Consider, for example, the unquantified expression

\[ x < y \mid y < 12 \]

and suppose x ranges over \{4,15\} and y over \{7,14\}. The truth table for the expression is:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x &lt; y \mid y &lt; 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>true</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>false</td>
</tr>
</tbody>
</table>

Now consider the same expression but with x universally quantified:

∀x (x < y \mid y < 12)

In a sense, like the existentially quantified variable, we don’t care what the values of x are, as long as every one of them makes the expression true for any given y. Thus its truth table is:

<table>
<thead>
<tr>
<th>y</th>
<th>∀x (x &lt; y \mid y &lt; 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>14</td>
<td>false</td>
</tr>
</tbody>
</table>

The universal quantifier will be needed for queries like the following:
“get the names of customers who bought every type of product”

Assume the relations as in Figure 7-5. The phrase “every type of product” clearly means every tuple of the Product relation. However, the Product relation does not record purchases, which are found only in the Transaction relation, ie. a product is purchased (by someone) if there is a transaction recording its purchase. In other words, a customer (ie. X) satisfies this query if for every product (ie. ∀Z) there is a transaction (ie. ∃Y) recording its purchase by the customer. This can now be quite simply rewritten in the calculus:

\[ X.Cname: \forall Z \exists Y (X.C# = Y.C# \land Y.P# = Z.P#) \]

Note that the different types of quantifiers can be mixed. But note also that their order is significant, ie. \( \forall x \exists y (\text{<expr>}) \) is not the same as \( \exists y \forall x (\text{<expr>}) \). For example,

\[ \forall x \exists y (y \text{ is the mother of } x) \]

asserts that everyone has a mother. Whereas,

\[ \exists y \forall x (y \text{ is the mother of } x) \]

asserts that there is a single individual (y) who is the mother of everyone!

### 7.4 Well-Formed Formulae

Let us now be more precise about the valid forms of logical expressions involving tuple variables. Such valid forms are called well-formed formulae (wff) and are defined as follows:

1. \( A \theta M \) is a wff if
   - \( A \) is a projection of a tuple variable,
   - \( M \) is a constant or a projection of a tuple variable, and
   - \( \theta \) is one of the comparison operators: =, ≠, <, >, ≤, ≥

2. \( F1 \land F2 \) and \( F1 \lor F2 \) are wffs if \( F1 \) and \( F2 \) are wffs

3. \( (F) \) is a wff if \( F \) is a wff

4. \( \exists x (F(x)) \) and \( \forall x (F(x)) \) are wffs if \( F(x) \) is a wff with a free occurrence of the variable \( x \).

The operator precedence for the ‘&’ and ‘|’, operators follow the standard precedence rules, ie. ‘&’ binds stronger than ‘|’. Thus,

\[ F1 \land F2 \lor F3 \equiv (F1 \land F2) \lor F3 \]

Explicit use of parenthesis, as in rule (3) above, is required to override this default precedence. Thus if the intention is for the ‘|’ operator to bind stronger in the above expression, it has to be written as

\[ F1 \land (F2 \lor F3) \]

We can now be more specific about the form of a query in relational calculus:

\[ (<\text{target list}>):(<\text{wff}>) \]

As final examples for this chapter, consider the following queries:
Query 1: “Get the names, cities and phone numbers of customers who bought the product CPU before the 25th of January”

Assume the tuple variables C, T and P ranging over relations Customer, Transaction and Product respectively. The appropriate query is as follows:

(Q.Cname, Q.City, Q.Phone) :
\[ \exists T \exists P \ (Q.C# = T.C# \ & \ T.Date < 25.01 \ & \ T.P# = P.P# \ & \ P.Pname = CPU) \]

Query 2: “Get the names of products bought by customers living in London or by the customer named Smith”

Assume tuple variables as in Query 1 above. The appropriate query is as follows:

(P.Pname) :
\[ \exists T \exists C \ ( T.P# = P.P# \ & \ T.C# = Q.C# \ & \ ( Q.City = London \ | \ Q.Cname = Smith )) \]

Note that the use of parenthesis around the ‘or’ expression above is necessary.
8 Relational Calculus (Part II)

Relational Calculus, as defined in the previous chapter, provides the theoretical foundations for the design of practical data sub-languages (DSL). In this chapter, we will look at an example of one—in fact, the first practical DSL based on relational calculus—the Alpha.

Further to this, we will also look at an alternative calculus—still a relational calculus (i.e. relations are still the objects of the calculus) but based on Domain Variables rather than Tuple Variables. Because of this, the relational calculus covered earlier is more accurately termed Relational Calculus with Tuple Variables. The reader will recall that Tuple Variables range over tuples of relations and were central in the formulation of inference rules and in the definition of well-formed formulae. Domain Variables, on the other hand, range over domain values rather than tuples and consequently require a different construction of well-formed formulae. We will discuss this alternative in the second part of this chapter.

8.1 The Data Sub-Language Alpha

DSL Alpha is directly based on relational calculus with tuple variables. It provides, however, additional constructions that increase the query formulation power of the language. Such constructions are in fact found in most practical DSL in use today.

8.1.1 Alpha Command

DSL Alpha is a set of Alpha commands, each taking the form:

\[
\text{Get } \langle\text{workspace}\rangle\ (\langle\text{target list}\rangle) : \langle\text{WFF}\rangle
\]

\(<\text{workspace}>\) is an identifier or label that names a temporary working relation to hold the result of the command (similar to the named working relation in the ‘giving’ clause of relational algebra—see section 5.3). The attributes of this relation are specified by <target list> which is a list of tuple variable projections as in the previous chapter. <WFF> is of course a well-formed formulae of relational calculus that must be satisfied before the values in <target list> are extracted as a result tuple.

As an example, suppose the variable P ranges over the Product relation as shown in Figure 8-1. Then the following construction is a valid Alpha command:

\[
\text{Get } W(P.Pname) : P.Price \leq 1000
\]

The reader can see that except for the keyword ‘Get’ and the naming of the result relation (‘W’ in this example), the basic form is identical to the one used in the previous chapter, which would simply be written

\[
(P.Pname) : P.Price \leq 1000
\]

The semantics of the Alpha command is also exactly the same, except that the result is a named relation, as shown in the illustration.
8.1.2 Range Statement

In our exposition of relational calculus, tuple variables used in queries were introduced informally. We did this in the above example too (viz. “suppose the variable P …”). This will not do, of course, if we wish the language to be interpreted by a computer. Thus, tuple variables must be introduced and associated with the relations over which they range using formal constructions. In DSL Alpha, this is achieved by the range declaration statement, which takes the basic form:

```
Range <relation name> <variable name>
```

where <relation name> must name an existing relation and <variable name> introduces a unique variable identifier. The variable <variable name> is taken to range over <relation name> upon encountering such a declaration. The above example can now be written more completely and formally as:

```
Range Product P;
Get W(P.Pname) : P.Price ≤ 1000
```

DSL Alpha statements and commands, as the above construction shows, are separated by semi-colons (‘;’).

DSL Alpha also differs from relational calculus in the way it quantifies variables. First, for a practical language, mathematical symbols like ‘∀’ and ‘∃’ need to be replaced by symbols easier to key in. DSL Alpha uses the symbols ‘ALL’ and ‘SOME’ to stand for ‘∀’ and ‘∃’ respectively. Second, rather than using the quantifiers in the <WFF> expression, they are introduced in the range declarations. Thus, the full syntax of range declarations is:

```
Range <relation name> <variable name> [ SOME | ALL ]
```

Note that the use of quantifiers in the declaration is optional. If omitted, the variable is taken to be a free variable whenever it occurs in an Alpha command.

Let us look at a number of examples.

**Query 1:** “Get the names and phone numbers of customers who live in London”

Assume the Customer relation as in Figure 8-2. This query will only need a single free variable to range over customer. The Alpha construction required is:

```
Range Customer X;
Get WA(X.Cname, X.Cphone): X.Ccity = London
```

Figure 8-2 also highlights the tuples in Customer satisfying the WFF of the command and the associated result relation WA.
Query 2: “Get the names of products bought by Customer #2”

For this query, we will need to access the Transaction relation, with records of which customer bought which product, and the Product relation, which holds the names of products. Assume these relations are as given in Figure 8-3.

```
<table>
<thead>
<tr>
<th>Product</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
<td>Pname</td>
</tr>
<tr>
<td>1</td>
<td>CPU</td>
</tr>
<tr>
<td>3</td>
<td>VDU</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
</tr>
</tbody>
</table>
```

**Figure 8-3 Query 2**

The object of our query is the Pname attribute of Product, thus the tuple variable for Product must necessarily be a free variable:

```
Range Product A;
```

The condition of the query requires us to look in the Transaction relation for a record of purchase by Customer #2 - as long as we can find one such record, the associated product is one that we are interested in. This is a clear case of existential quantification, and the variable introduced to range over Transaction is therefore given by:

```
Range Transaction B SOME;
```

The Alpha command for the query can now be written:

```
Get W ( A.Pname ): A.P# = B.P# And B.C# = 2;
```

The associated tuples satisfying the WFF above are highlighted in the figure 8-3 (the result relation is not shown).

Query 3: “Get the names and phone numbers of customers in London who bought the product VDU”

This is a more complex example that will involve three relations, as shown in Figure 8-4. The target data items are in the Customer relation (names and phone numbers). So the tuple variable assigned to it must be free:
Range Customer X;

Part of the condition specified is that the customer must live in London (ie. X.Ccity = London), but the rest of the condition (‘… who bought the product VDU’) can only be ascertained from the Transaction relation (record of purchase by some customer) and Product relation (name of product). In both these cases, we are just interested in finding one tuple from each, ie. that there exists a tuple from each relation that satisfies the query condition. Thus, the variables introduced for them are given by:

Range Transaction Y SOME;
Range Product Z SOME;

The Alpha command can now be written as:

\[
\text{Get } W( \text{X.Cname, X.Cphone }): \\
\text{X.Ccity = London And X.C# = Y.C# And Y.P# = Z.P# And Z.Pname = VDU}
\]

Figure 8-4 highlights one instantiation of each variable that satisfies the above WFF.

Query 4: “Get the names of customers who bought all types of the company’s products”

As with the previous example, this one also requires access to three relations as shown in Figure 8-5. A customer will satisfy this query if for every product there is a transaction recording that he/she purchased it. This time, therefore, we have a case for universal quantification - “…all types of the company’s products” - which will require that the variable ranging over Product be universally quantified. The variable for Transaction, on the other hand, is existentially quantified (“…there is a transaction…”). The full Alpha construction therefore is:

Range Customer C;
Range Product P ALL;
Range Transaction T SOME;
Get W (C.Cname): P.P# = T.P# And T.C# = C.C#

Figure 8-5 highlights tuples from the various relations that satisfy this construction.

Note that the order of quantified variable declarations is important. The order above is equivalent to “\(\forall P \exists T\)”. If variable T was declared before P, it would be equivalent to “\(\exists T \forall P\)” which would mean something quite different! (see section 7.3)
Query 5: “Get the name of the most expensive product”

This query involves only one relation: the Product relation (assume the Product relation as in the above examples). Now, the “most expensive product” is that for which every product has a price less than or equal to it. Or, in relational calculus, X is such a product provided that “∀Y X.Price ≥ Y.Price”. Thus two variables are required, both ranging over Product but one of them is universally quantified:

Range Product X;
Range Product Y ALL;
Get W(X.Pname): X.Price ≥ Y.Price

It is perhaps interesting to note in passing that the choice by DSL Alpha designers to quantify variables at the point of declaration rather than at the point of use makes Alpha commands a little harder to read—it is not clear which variables are quantified just by looking at the Alpha command. One must search for the variable declaration to see how, if at all, it is quantified.

8.1.3 Additional Facilities

DSL Alpha provides additional facilities that operate on the results of its commands. While these are outside the realm of relational calculus, they are useful and practical functions that enhances the utility of the language. These facilities fall loosely under two headings: qualifiers, and library functions.

The qualifiers affect the order of presentation of tuples in the result relation, based on the ordering of values of a specified attribute in either an ascending or descending order, ie. they may be thought of as sort functions over a designated attribute. Note that in relational theory the order of tuples in a relation is irrelevant since a relation is a set of values. So the qualifiers affects only the presentation of a relation.

Syntactically, the qualifier is appended to the WFF and takes the following form:

\{ UP | DOWN \} <attribute name>

As an example, consider the requirement for the names of products bought by Customer #1 in descending order of their prices. The Alpha construction for this would be:
Introduction to Databases & Relational DM

Range Product X;
Range Transaction Y SOME;
Get UWA( X.Pname, X.Price ): (X.P# = Y.P# And Y.C# = 2) DOWN X.Price

Figure 8-6 shows the relations highlighting tuples satisfying the WFF. It also shows the result relation UWA which can be seen to be ordered in descending order of price.

---

The library functions, on the other hand, derive (computes) new values from the data items extracted from the database. Another way to put this is that the result relation of the basic Alpha command is further transformed by library functions to yield the final result. Why would we want to do this? Consider for example that we have a simple set of integers, say \{1,2,3\}. There are a variety of values we may wish to derive from it, such as

- the number of items, or cardinality, of the set (library function COUNT, ie. \(\text{COUNT}\{1,2,3\}=3\))
- the sum of the values in the set (library function TOTAL, ie. \(\text{TOTAL}\{1,2,3\}=6\))
- the minimum, or maximum, value in the set (library function MIN and MAX, ie. \(\text{MIN}\{1,2,3\}=1, \text{or MAX}\{1,2,3\}=3\))
- the average of values in the set (library function AVERAGE, ie. \(\text{AVERAGE}\{1,2,3\}=2\))

Extending this idea to relations, and in particular the Alpha command, library functions are applied to attributes in the target list, taking the form:

\(<\text{library function}>(<\text{attribute name}>)>\)

As an example, consider the need to find the number of customers who bought the product VDU. This is quite a practical requirement to help management track how well some products are doing on the market. Pure relational calculus, however, has no facility to do this. But using the library function COUNT in DSL Alpha, we can write the following:

Range Transaction T;
Range Product P SOME;
Get AAA( COUNT(T.C#) ): T.P# = P.P# And P.Pname = VDU

Figure 8-7 highlights the tuples satisfying the WFF and shows the result relation.
As another example, suppose we wanted to know how many products were bought by the customer Codd. The data items to answer this question are in the quantity field (Qnt) of the Transaction relation, but pure relational calculus can only retrieve the set of quantity values associated with purchases by Codd. What we need is the sum of these values. The library function TOTAL of DSL Alpha allows us to do this:

\[
\text{Range Transaction } T; \text{ Range Customer } C \text{ SOME;}
\]
\[
\text{Get BBB( TOTAL( } T.\text{Qnt } ) : T.C# = C.C# \text{ And } C.\text{Cname} = \text{Codd}}
\]

Figure 8-8 summarises the execution of this Alpha command.

As a final remark, we note that we have only sampled a few library functions. It is not our aim to cover DSL Alpha comprehensively, but only to illustrate real DSLs based on the relational calculus, and to look at added features or facilities needed to turn them into practical languages.

### 8.2 Relational Calculus with Domain Variables

#### 8.2.1 Domain Variables

As noted in the introduction, there is an alternative to using tuple variables as the basis for a relational calculus, and that is to use domain variables. Recall that a domain (see section 2.2) in the relational model refers to the current set of values of a given kind under an attribute name and is defined over all relations in the database, i.e. an attribute name denotes the same domain in whatever relation it occurs. A domain variable ranges over a designated domain, i.e. it can be instantiated to, or hold, any value from that domain.
For example, consider the domain Cname found in the Customer relation. This domain has three distinct values as shown in Figure 8-9. If we now introduced a variable, ‘Cn’, and designate it to range over Cname, then Cn can be instantiated to any of these values (the illustration shows it holding the value ‘Martin’).

As with tuple variables:
- a domain variable can hold only one value at any time
- domain variables can be introduced for any domain in the database
- more than one domain variable may be used to range over the same domain

Note also that the value of a domain variable is an atomic value, i.e., it does not comprise component values as was the case with tuple variables. Thus there is no need for any syntactic mechanism like the ‘dot notation’ to denote component atomic values of tuple variables. It also means that in constructing simple comparison expressions, domain variables appear directly without any embellishments, e.g., A > 1000, B = London, C ≤ 2000, D ≠ Paris, etc. (assuming of course that the variables A, B, C and D have been designated to range over appropriate domains).

In a relational calculus with domain variables we can write predicates of the form:

\[ \text{<relation name>}( x_1, \ldots, x_n ) \]

where
- \text{<relation name>} is the name of a relation currently defined in the database schema, and
- each \( x_i \) is a domain variable ranging over a domain from the intension of \text{<relation name>}

Thus, suppose we have the situation as in Figure 8-10. It is then syntactically valid to write:

Customer( A, B )

as ‘Customer’ is a valid relation name, and the variables ‘A’ and ‘B’ range over domains that are in the intension of the Customer relation.

The meaning of such a predication can be stated as follows:

a predicate ‘\text{<relation name>}( x_1, \ldots, x_n )’ is true for some given instantiation of each variable \( x_i \) if and only if there exists a tuple in \text{<relation name>} that contains corresponding values of the variables \( x_1, \ldots, x_n \).
Thus, for example, Customer(A,B) is true when A=Codd and B=London, since the first
tuple of Customer has the corresponding values. In contrast, Customer(A,B) is false when
A=Codd and B=Paris, as no tuple in Customer have these values. In fact, the values that
make Customer(A,B) true are:

<table>
<thead>
<tr>
<th>Cname</th>
<th>Ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
<td>London</td>
</tr>
<tr>
<td>Martin</td>
<td>Paris</td>
</tr>
<tr>
<td>Deen</td>
<td>London</td>
</tr>
</tbody>
</table>

that is, in relational algebra terms, a projection of <relation name> over the domains that
variables x₁, … , xₙ range over.

A query in relational calculus with domain variables take the form:

\[(<target list>) : (<logical expression>)\]

where

- `<target list>` is a comma-separated list of domain variable names, and
- `<logical expression>` is a truth-valued expression involving predicates and
  comparisons over domain variables and constants (the rules for constructing well-
  formed `<logical expressions>` will be detailed later)

The result of such a query is a set of instantiations of variables in `<target list>` that make
`<logical expression>` true.

For example, consider the database state in Figure 8-11 and the query

\[(x,y) : (Product(x,y) & y >1000)\]

which can be paraphrased as “get product names and their prices for those products
costing more than 1000”.

\[
\begin{array}{|c|c|}
\hline
P\# & Pname & Price \\
\hline
1 & CPU & 1000 \\
2 & VDU & 1200 \\
\hline
\end{array}
\]

Figure 8-11 Database state for the query “(x,y): (Product(x,y) & y > 1000)”

The only pair of (x,y) instantiation satisfying logical expression in this case is

\[(VDU,1200)\], ie. the result of the query is

\[
\begin{array}{|c|c|}
\hline
x & y \\
\hline
VDU & 1200 \\
\hline
\end{array}
\]

Domain variables, like tuple variables, may also be quantified with either the universal or
existential quantifier. Expressions involving quantified domain variables are interpreted
in the same way as for quantified tuple variables (see 7.3).

Consider the query: “get the names of products bought by customer #1”. The required
data items are in two relations: Product and Transaction, as follows.
We can paraphrase the query to introduce variables and make it easier to formulate the correct formal query:

\[(x) : \exists y \exists z \left( \text{Product}(x,y) \land \text{Transaction}(y,z) \land z = 1 \right)\]

where the subexpressions

- \(\text{Product}(x,y)\) captures the condition “there is a product number \(y\) for \(x\)”
- \(\text{Transaction}(y,z)\) captures the condition “there is a customer number \(z\) that purchases \(y\)”, and
- \(z = 1\) clearly requires that the customer number is 1

The reader should be able to work out the solution to the query as an exercise.

As a final example, consider the query: “get the names of customers who bought all types of the company’s products”. The reader can perform an analysis of this query as was done above to confirm that the relevant database state is as shown in Figure 8-12 and that the correct formal query is:

\[(x) : \forall y \exists z \left( \text{Customer}(x,z) \land \text{Transaction}(y,z) \right)\]

This example illustrates a universally quantified domain variable \(y\) ranging over \(P\#\). For this query, this means that the “\(\text{Transaction}(y,z)\)” part of the logical expression must evaluate to true for every possible instantiation of \(y\) given a particular instantiation of \(z\).
Thus, when \( x = \text{Codd} \) and \( z = 1 \), both \( \text{Transaction}(1,1) \) and \( \text{Transaction}(2,1) \) must evaluate to true. They do in this case and \( \text{Codd} \) will therefore be part of the result set. But, when \( x = \text{Martin} \) and \( z = 2 \), \( \text{Transaction}(1,2) \) is true but \( \text{Transaction}(2,2) \) is not! So \( \text{Martin} \) is not part of the result set. Continuing in this fashion for every possible instantiation of \( x \) will eventually yield the full result.

8.2.2 Well-Formed Formula

We have not formally defined above what constitutes valid <logical expression>s. We do so here, but for the sake of a uniform terminology, we will use the phrase well-formed formula (WFF) instead of <logical expression> just as we did for relational calculus with tuple variables. Thus a formal query in relational calculus with domain variables take the form:

\[
(<\text{target list}>): (\text{WFF})
\]

where \(<\text{target list}>\) is a comma-separated list of free variable names, and a WFF is defined by the following rules:

1. \( P(A,\ldots) \) is a WFF if \( P \) is a relation name and \( A,\ldots \) is a list of free variables
2. \( A \theta M \) is a WFF if \( A \) is a variable, \( M \) is a constant or a variable, and \( \theta \in \{\neq, <, >, \leq, \geq\} \)
3. \( F_1 \& F_2 \) and \( F_1 \mid F_2 \) are WFFs if \( F_1 \) and \( F_2 \) are WFFs
4. \( (F) \) is a WFF if \( F \) is a WFF
5. \( \exists x (F(x)) \) and \( \forall x (F(x)) \) if \( F(x) \) is a WFF with the variable \( x \) occurring free in it

As usual, the operator precedence for the ‘\&’ and ‘\mid’, operators follow the standard precedence rules, ie. ‘\&’ binds stronger than ‘\mid’. Thus,

\[
\text{‘} F_1 \& F_2 \text{' \mid F_3 \text{'\equiv ‘}( F_1 \& F_2 ) \mid F_3 \text{'}}
\]

Explicit use of parenthesis, as in rule (4) above, is required to override this default precedence. Thus if the intention is for the ‘\mid’ operator to bind stronger in the above expression, it has to be written as

\[
F_1 \& ( F_2 \mid F_3 )
\]
9 Data Sub-Language SQL

9.1 Introduction

In this chapter, we shall learn more about the essentials of the relational model’s standard language that will allow us to manipulate the data stored in the databases. This language is powerful yet flexible, thus making it popular. It is in fact one of the factors that has led to the dominance of the relational model in the database market today.

Following Codd’s papers on the relational model and relational algebra and calculus languages, research communities were prompted to work on the realisation of these concepts. Several implemented versions of the relational languages were developed, amongst the most noted were SQL (Structured Query Language), QBE (Query-By-Example) and QUEL (Query Language). Here, we shall look into SQL with greater detail as it the most widely used relational language today. One often hears of remarks that say, “It’s not relational if it doesn’t use SQL”. It is currently being standardised now as a standard language for the Relational Data Model.

SQL had its origins back in 1974 from IBM’s System R research project as Structured English Query Language (or SEQueL) for use on the IBM VS/2 mainframes. It was developed by Chamberlain et al. The name was subsequently changed to Structured Query Language or SQL. It is pronounced “sequel” by some and S-Q-L by others. IBM’s products such as SQL/DS and the popular DB2 emerged from this. SQL is based on the Relational Calculus with tuple variables. In 1986, the American National Standards Institute (ANSI) adopted SQL standards, contributing to its widespread adoption. Whilst many commercial SQL products exist with various “dialects”, the basic command set and structure remain fairly standard.

Although SQL is called a query language, it is capable of more than just getting data off relations in the databases. It can also handle data updates and even data definitions—add new data, change existing data, delete or create new structures. Thus SQL is capable of:

1. **Data Query**
   
   The contents of the database are accessed via a set of commands whereby useful information is returned to the end user

2. **Data Maintenance**
   
   The data within the relations can be created, corrected, deleted and modified

3. **Data Definition**
   
   The structure of the database and its relations can be defined and created

   The end user is given an interface, as we have seen in Chapter 3, to interact with the database via menus, query operations, report generators, etc. Behind this lies the SQL engine that performs the more difficult tasks of creating relation structures, maintaining the systems catalogues and data dictionary, etc.

SQL belongs to the category of the so-called Fourth-Generation Language (4GL) because of its power, conciseness and low-level of procedurality. As a non-procedural language it
allows the user to specify what must be done without detailing how it must be done. The user’s SQL request specification is then translated by the RDBMS into the technical details needed to get the required data. As a result, the relational database is said to require less programming than any other database or file system environment. This makes SQL relatively easy to learn.

9.2 Operations

9.2.1 Mapping: The SQL Select Statement

The basic operation in SQL is called mapping, which transforms values from a database to user requirements. This operation is syntactically represented by the following block:

```plaintext
Select (<Target List>)
From (<Relation>)
Where (<Predicate>)
```

Figure 9-1. SQL Select

This uncomplicated structure can be used to construct queries ranging from very simple inquiries to more complex ones by essentially defining the conditions of the predicate. It thus provides immense flexibility.

The SQL Select command combines the Relational Algebra operators Select, Project, Join, and the Cartesian Product. Because a single declarative-style command can be used to retrieve virtually any stored data, it is also regarded by many to be an implementation of the Relational Calculus. If we need to extract information from only one relation of the database, we may encounter similarities and a few differences between the Relational Calculus-based DSL Alpha and SQL. In this case we may substitute key words of DSL Alpha for matching key words of SQL as follows:

```plaintext
Select (<Target List>)
From (<Relation>)
Where (<Predicate>)
```

Figure 9-2. Similarities of DSL Alpha and SQL Select

Let us refer back to the earlier example with the Customer relation.

Suppose we wish to “Get the names and phone numbers of customers living in London”.
With DSL Alpha, we would specify this query as:

```plaintext
Range Customer X;
Get (X.Cname, X.Cphone): X.Ccity=London;
```
whereas in SQL its equivalent would be:

```sql
Select  Cname, Phone
From Customer
Where Ccity = 'London'
```

In either case, the result would be the retrieval of the following two tuples:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>City</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

This simple query highlights the three most used SQL clauses:

1. **The SELECT clause**

This effectively gets the *columns* that we are interested in getting from the relation. We may be interested in a single column, thus we may for example write “Select Cphone” if we only wish to list just the telephone numbers. We may also however be interested in listing the customer’s name, city and telephone number; in which case, we write “Select Cname, Ccity, Cphone”.

2. **The FROM clause**

We need to identify the *relations* that our query refers to and this is done via the From clause. The columns that we have chosen from the Select clause must be found in the relation names of the From clause as in “From Customer”.

3. **The WHERE clause**

This holds the *conditions* that allows us to restrict the tuples of the relation(s). In the example “Where Ccity=London” asserts that we wish to select only the tuples which contain the city name that is equal to the value ‘London’.

The system first processes the From clause (and all tuples of the chosen relation(s) are placed in the processing work area), followed by the Where clause (which chooses, one by one, the tuples that satisfy the clause conditions and eliminating those which do not), and finally the Select clause (which takes the resultant tuples and displays only the values under the Select clause column names).

### 9.2.2 Output Restriction

Most queries do not need every tuple in the relation but rather only a subset of the tuples.

As described previously in section 5.3, the following mathematical operators can be used in the predicate to restrict the output:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;, &gt;</td>
<td>Less, Greater</td>
</tr>
<tr>
<td>&lt;=, &gt;=</td>
<td>Less than or Equal, Greater than or Equal</td>
</tr>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not Equal</td>
</tr>
<tr>
<td>&amp;</td>
<td>And</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Not</td>
</tr>
</tbody>
</table>

Where `*Cname*` and `*Cphone*` are the columns in the `Customer` relation.
Additionally, the logical operators AND, OR and NOT may be used to place further restrictions. These logical operators, along with parentheses, may be combined to produce quite complex conditional expressions.

Suppose we need to retrieve the tuples from the Transaction relation such that the following conditions apply:

1. The transaction date is before 26 Jan and the quantity is at least 25
2. Or, the customer number is 2

The SQL statement that could get the desired result would be:

```
Select C#, Date, Qnt From Transaction
Where (Date < '21.01' And Qnt >= 25) Or C# = 2
```

### 9.2.3 Recursive Mapping: Sub-queries

The main idea of SQL is the recursive usage of the mapping operation instead of using the existential and universal quantifiers. So far in our examples, we always know the values that we want to put in our predicate. For example,

```
Where Ccity = 'London'
Where Date < '26.01' And Qnt > 25
```

Suppose we now wish to “Get the personal numbers of customers who bought the product CPU”. We could start off by writing the SQL statement:

```
Select C#
From Transaction
Where P# = ?
```

We cannot of course write “Where P#=CPU” because CPU is a part name not its number. However as we may recall, part number P# is stored in the Transaction relation, but the part name is in fact in another relation, the Product relation. Thus one needs to first of all get the part name from Product via another SQL statement:

```
Select P#
From Product
Where Pname = 'CPU'
```
Having obtained the equivalent P#, the value is then used to complete the earlier query. The way this is to be expressed is by writing the whole mapping operator in the right hand side of comparison expressions of another mapping operator. This effectively means the use of an inner block (sub-query) within the outer block (main query) as depicted in the figure below.

![Query nesting](image)

**Figure 9-3. Query nesting**

The query in the outer block thus executes by using the value set generated earlier by the sub-query of the inner block.

It is important to note that because the sub-query replaces the value in the predicate of the main query, the value retrieved from the sub-query must be of the same domain as the value in the main predicate.

### 9.2.4 Multiple Nesting

It is also possible that may be two or more inner blocks within an outer SQL block. For instance, we next wish to: “Get a date when customer Codd bought the product CPU”. The SQL statement we would start out with would probably look like this:

```sql
Select Date
From Transaction
Where P#=?
And C#=?
```

As in the earlier query, the part number P# can be obtained via the part name Pname in the relation Product. The customer name, Codd, however has to have its equivalent customer number which has to be obtained from C# of the relation Customer. Thus to complete the above query, one would have to work two sub-queries first as follows:
Figure 9-4. Interpretation of sub-queries

Note that the original SQL notation utilises brackets or parentheses to determine inner SQL blocks as:

```sql
Select Date
From Transaction
Where P# =
    ( Select P#
        From Product
        Where Pname = 'CPU'
    )
And C# =
    ( Select C#
        From Customer
        Where Cname = 'Codd'
    )
```

Similarly, an inner block may contain further inner SQL blocks. For instance, if we wish to “Get the names of customers who bought more than 20 pieces of the product CPU” we need to specify:

```sql
Select Cname
From Customer
Where C# =
    ( Select C#
        From Transaction
        Where P# =
            ( Select P#
                From Product
                Where Pname = 'CPU'
            )
    )
```
Thus we may visualise the nesting of sub-queries as:

```
Select …
From …. 
Where <attribute1> <operator>
   ( Select <attribute1>
   From …)
Where <attribute2> <operator>
   ( Select <attribute2>
   From …)
Where <attribute3> <operator>
   ( Select <attribute3>
   From …)
```

The number of inner blocks or levels of nesting may, however, be limited by the storage available in the workspace of the DBMS in use.

### 9.2.5 Multiple Data Items

Standard comparison operators ( =, >, <, >=, <=, <> ) operate on two data items, as in x = y or p >= 4. They cannot be applied to multiple data items. However, a particular SQL block normally returns a set of values (i.e. not a single value which can be used in a comparison).

For instance: “Get the product numbers of items which were bought by customers from London”.

```
Select P#
From Transaction
Where C# =
   ( Select C#
   From Customer
   Where Ccity = ‘London’ )
```

Given the sample database of the earlier examples, the result of the inner SQL block would yield two values for C#, which are 1 and 3, (or more precisely, the set {1, 3 } ). The outer SQL block, in testing C# = {1, 3 } would effectively test if {1,2 } = {1, 3 } or not. Thus the above SQL statement is not correct!

To overcome the error caused by the testing of multiple values returned by the sub-query, SQL allows the use of comparison expressions in the form:
This logical expression is true if the current value of an attribute is included (or not included, respectively) in the set of values.

For instance,

\[
\text{Smith In } \{ \text{Codd, Smith, Deen} \} \text{ is True,}
\]

and

\[
\text{Smith Not In } \{ \text{Codd, Smith, Deen} \} \text{ is False.}
\]

Thus in re-writing the earlier erroneous statement, we now replace the equal operator (=) with the set membership operator ‘In’ as follows:

\[
\text{Select P# From Transaction Where C# In} \\
( \text{Select C# From Customer Where Ccity = } \text{‘London’} )
\]

This time it would yield the outer SQL block would effectively test C# in \{1, 3\}. The outer SQL block would now only retrieve the P#s that are only in the set \{1, 3\} i.e. testing \{1, 2\} In \{1, 3\} This would result in returning P# 1 only, which is the expected right answer.

Illustrating with another example, consider the query to “Find the names of customers who bought the product CPU”. Its corresponding SQL statement would thus be:

\[
\text{Select Cname From Customer Where C# In} \\
( \text{Select C# From Transaction Where P# In} \\
( \text{Select P# From Product Where Pname = } \text{‘CPU’} ) )
\]

Executing this step-by-step:

(1) From the inner-most block,

\[
\text{Select P# From Product}
\]
Would first yield P# 1 from Product, i.e. \{1\}.

(2) The next block would thus be

\[
\text{Select C\# From Transaction}
\]
\[
\text{Where P\# In \{1\}}
\]

and this would yield C\#'s 1 and 2 (as they bought P\# 1), i.e. \{1, 2\}.

(3) And finally, the outer-most block would execute

\[
\text{Select Cname From Customer}
\]
\[
\text{Where C\# In \{1, 2\}}
\]

would result in the names of customers 1 and 2, which are Codd and Martin respectively.

We next go on to a slightly more complex example. Suppose we now wish to “Get a name of such customers who bought the product CPU but did not buy the product VDU”.

In SQL, the statement would be:

\[
\text{Select Cname From Customer}
\]
\[
\text{Where C\# In ( Select C\# From Transaction Where P\# In ( Select P\# From Product Where Pname = 'CPU' ) And C\# Not In ( Select C\# From Transaction Where P\# In ( Select P\# From Product Where Pname = 'VDU' ) ) )}
\]

Why don’t you try to figure out, step-by-step, the sequence of results from the inner-most blocks up to the final result of execution of the outer-most block?

Note that the comparison operators

\[
<\text{attribute name}> \begin{cases} \text{In} & \{<\text{set of values}>\} 
\end{cases}
\]

are used instead of existential qualifiers (\(\exists\)). It is an implementation of multiple logical OR conditions which is more efficiently handled.

Similarly, comparison expressions

\[
<\text{attribute name}> = \text{ALL } <\text{set of values}>
\]

are used instead of universal qualifiers (\(\forall\)).

This logical expression is valid (i.e. produces the logical value “True”) if the collection of attribute name values in the database includes the given set of values.
For instance, “Get personal numbers of those customers who bought all kinds of company’s products”, would have the following SQL statement for it:

```
Select C# From Transaction
Where P# =
    ALL ( Select P#
        From Product)
```

The inner block would yield the set \{1, 2\} of P# values. Executing the outer block would effectively test if the 3 customers in the Transaction relation, i.e. C# 1, 2 and 3 would have P# in \{1, 2\}

This test is as follows:

<table>
<thead>
<tr>
<th>C#</th>
<th>Transaction (C#, 1)</th>
<th>Transaction (C#, 1)</th>
<th>All P#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

The only customer that has P# equal to all P# as found in Product would be C# 1.

### 9.3 Further Retrieval Facilities

#### 9.3.1 Joining Relations

In the examples that have been used so far, our retrievals have been of values taken from one relation, as in “Select C# From Transaction”. However, often we have to retrieve information from two or more relations simultaneously. In other words, a number of relations names may be used in the From clause of SQL. For example, if we wish to access the relations Customer and Transaction, we may write the SQL statement as follows:

```
Select …
From Customer, Transaction
Where…
```

The target list in the Select clause may contain the attributes form various relations, as in

```
Select Cname, Date, Qnt
From Customer, Transaction
Where…
```

where, if you recall, Cname is an attribute of Customer and Date and Qnt are attributes of Transaction.
Similarly, comparison expressions in the Where clause may include attribute names from various relations,

Select Cname, Date, Qnt
From Customer, Transaction
Where (Customer.C# = Transaction.C#) And P# = 1

Note that a so-called qualification technique which is used to refer to attributes of the same name belonging to different relations. Customer.C# refers to the C# of the Customer relation whereas Transaction.C# refers to the C# of the Transaction relation.

![Figure 9-5. Qualifying attributes](image)

Thus the query “Get customer names, dates and number of pieces for transactions of the product number 1” will result in:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Martin</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Deen</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

It must be noted that the two (or more) relations that must be combined on at least one common linking attribute (as in the Relational Algebra’s JOIN operator). As in the above example, the link is established on C# as in the clause

Where Customer.C# = Transaction.C#

### 9.3.2 Alias

In order to avoid a possible ambiguity in a query definition SQL also allows to use an alias for the relation name in the From clause. The alias is an alternate name that is used to identify the source relation and the attribute names may include an alias as a prefix:

```
<COLUMN NAME>.<ATTRIBUTE NAME>
```

Suppose we use T and C as the aliases for the Transaction and Customer relations respectively. We may use these to label the attributes as in:

Select ... From Customer C, Transaction T
Where C.C# = T.C# And ...

An alias is especially useful when we wish to join a relation to itself because of grouping as in the query to “Find the names and phone numbers of customers living in the same city as the customer Codd”:

Select C2.Cname, C2.Cphone
From Customer C1, Customer C2
Where C2.Ccity = C1.Ccity
And C1.Cname = ‘Codd’

The resulting interpretation of the SQL statement is depicted in Figure 9-6 below:

<table>
<thead>
<tr>
<th>Customer = C1</th>
<th>Select C2.Cname, C2.Cphone</th>
<th>Customer = C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
<td>Ccity</td>
</tr>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
</tr>
<tr>
<td>3</td>
<td>Deen</td>
<td>London</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>C2.Cname</th>
<th>C2.Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
<td>2263035</td>
</tr>
<tr>
<td>Deen</td>
<td>2234391</td>
</tr>
</tbody>
</table>

Figure 9-6. Using an alias

9.4 Library Functions and Arithmetic Expressions

The SQL Select clause (target list) may contain also so-called SQL library functions that will perform various arithmetic summaries such as to find the smallest value or to sum up the values in a specified column. The attribute name for such library functions must be derived from the relations specified in the From clause as follows:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>To count the number of tuples containing a specified attribute value</td>
</tr>
<tr>
<td>SUM</td>
<td>To sum up the values of an attribute</td>
</tr>
<tr>
<td>AVG</td>
<td>To find the arithmetic mean (average value) of an attribute</td>
</tr>
<tr>
<td>MAX</td>
<td>To find the maximum value of an attribute</td>
</tr>
</tbody>
</table>
MIN To find the minimum value of an attribute

Examples

(1) Get the average quantity of VDUs per transaction

Select AVG (Qnt) From Transaction
Where P# =
     ( Select P# From Product
          Where Pname = ‘VDU’ )

Working first with the inner Select clause, we get a P# of 2 from the Product relation as the part number for the product named VDU. Thus the query is now reduced to

Select AVG(Qnt) From Transaction
Where P# = 2

Accessing the Transaction relation now would yield the following two tuples

<table>
<thead>
<tr>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

where the average quantity value is easily computed as (30+20)/2 which is 25.

(2) Get the total quantity of VDUs transacted would similarly be expressed as

Select SUM (Qnt) From Transaction
Where P# =
     ( Select P# From Product
          Where Pname = ‘VDU’ )

where the total value is easily computed as (30 + 20) giving 50.

An asterisk (*) in the Select clause is interpreted as “all attributes names of the relations specified in the From clause”.

Select * From Transaction
is equivalent to

Select C#, P#, Date, Qnt From Transaction

Thus a query to “Get all available information on customers who bought the product VDU” can be written as:
Select * From Customer
Where C# In
    ( Select C# From Transaction
        Where P# In
            ( Select P# From Product
                Where Pname = 'VDU' ) )

The interpretation of this query would be worked out as shown in the following sequence of accesses, starting from the access of the product relation to the Transaction and finally to the Customer relation:

![Figure 9-8. Working through 3 nested Selects](image)

The outcome would be the following relation:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
</tbody>
</table>

(3) Get a total number of such customers who bought the product VDU, would be written as:

Select COUNT (*) From Customer
Where C# In
    ( Select C# From Transaction
        Where P# In
            ( Select P# From Product
                Where Pname = 'VDU' ) )

and this would yield a value of 2 for Count (*).

Arithmetic expressions are also permitted in SQL, and the possible operations include:

- addition +
- subtraction -
- multiplication *
- division /

Expressions may be written in the Select clause as:

Select C#, P#, Qnt*Price From Transaction, Product
Introduction to Databases & Relational DM

Where Transaction.P# = Product.P#

which is used to “Get a total price for each transaction” resulting in:

<table>
<thead>
<tr>
<th>Product</th>
<th>Transaction</th>
<th>Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>P#</td>
<td>Pname</td>
<td>Price</td>
</tr>
<tr>
<td>1</td>
<td>CPU</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>VDU</td>
<td>1200</td>
</tr>
<tr>
<td>1</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29.01</td>
</tr>
</tbody>
</table>

Arithmetic expressions, likewise, can also be used as parameters of SQL library functions. For example, “Get a total price of all VDUs sold to customers” may be written as the following SQL statement:

```
Select SUM (Qnt*Price) From Transaction, Product
Where Transaction.P# = Product.P#
And Product.Pname = ‘VDU’
```

Work this out. You should get an answer of 60000.

The attribute names for both library functions and arithmetic expressions must be derived from the relations specified in the From clause.

Thus, it should be noted that the following query definition is NOT correct.

```
Select SUM (Qnt*Price) From Transaction
Where Transaction.P# = Product.P#
And Product.Pname = ‘VDU’
```

Additionally, SQL also permits the use of library functions not only in the Select clause but also in the Where clause as a part of comparison expressions.

The query to “Get all available information on such customers who bought the most expensive product” would be:

```
Select * From Customer
Where C# In
   ( Select C# From Transaction
      Where P# In
         ( Select P# From Product
            Where Price = MAX (Price) ) )
```
9.5 Additional Facilities

9.5.1 Ordering

The result of a mapping operation may be sorted in ascending or descending order of the selected attribute value.

The form of the Order clause is

Order By <attribute name> Up | Down

Examples

(1) Get a list of all transactions of the product CPU sorted in descending order of the attribute Qnt

Select * From Transaction

Where P# In

( Select P# From Product

Where Pname = ‘CPU’ )

Order By Qnt Down

The result would be

<table>
<thead>
<tr>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
</tbody>
</table>

If instead, the last clause had been “Order By Qnt Up”, the result would be listed in ascending order:

<table>
<thead>
<tr>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
</tbody>
</table>

The Order By clause is only a logical sorting process, the actual contents of the original relations are not affected.

Multi-level ordered sequence may also be performed as in:

Select * From Transaction

Order By C# Up,

Qnt Down

9.3.2 Handling Duplicates

The result of an SQL mapping operation is however not perceived as a relation, i.e. it may include duplicate tuples. Consider for example:
Select C# From Transaction
Where P# In
   ( Select P# From Product
    Where Price >= 1000 )

The result is actually

<table>
<thead>
<tr>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Imagine if we have thousands of transactions and yet a handful of customers. The result would yield hundreds (even thousands) of duplicates. Fortunately, duplicate tuples can be removed by using the Unique option in the Select clause of the operation as follows:

Select C# Unique From Transaction
Where P# In
   ( Select P# From Product
    Where Price >= 1000 )

and this will yield a much reduced result with only the distinct (unique) customer numbers:

<table>
<thead>
<tr>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

9.3.3 Grouping of Data

Usually, the result of a library function is calculated for the whole relation. For example, consider wanting to find the total number of transactions,

Select Count (*)
From Transaction

<table>
<thead>
<tr>
<th>Transaction</th>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>23.01</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
</tr>
</tbody>
</table>

Given this relation, the result of Count (*) is 4
However, sometimes we need to calculate a library function, not for the entire relation, but only for a subset of it. Such subsets of tuples are called groups. For instance, in the relation Transaction, a collection of tuples with the same value of attribute C# is a “group”. In this case, C# is called “Group By” attribute.

**Figure 9-9.** Grouping by customer numbers

The form of the Group By clause is

**Group By <attribute name>**

**Examples**

(1) “Get the list of all customer numbers and the quantity of products bought by each of them”. Note that the relation will have many transactions for any one customer. The transactions for each customer will have to be grouped and the quantities totaled. This is then to be done for each different customer. Thus the SQL statement would be:

```sql
Select C#, Sum(Qnt) From Transaction Group By C#
```

Thus all transactions with the same C#s are grouped together and the quantities summed to yield the summarised result:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>C#</th>
<th>P#</th>
<th>Date</th>
<th>Qnt</th>
<th>C# Sum(Qnt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>21.01</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>30</td>
<td>23.01</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>26.01</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29.01</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Why would the following statement be impossible to execute?

```sql
Select * From Transaction Group By P#
```

(2) Normally, the Where clause would contain conditions for the selection of tuples as in:

```sql
Select Cname, Sum (Qnt) From Customer, Transaction
Where Customer.C# = Transaction.C#
Group By C#
```
This statement will “Get a list of all customer names and the quantity of products bought by each of them” as follows:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>P#</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>Cname</th>
<th>Sum(Qnt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codd</td>
<td>50</td>
</tr>
<tr>
<td>Martin</td>
<td>45</td>
</tr>
</tbody>
</table>

**Figure 9-10.** Restriction followed by Grouping

### 9.3.4 Further Filtering: Having

We can further filter out unwanted groups generated by the Group By clause by using a “Having” clause which will include in the final result only those groups that satisfy the stated condition. Thus the additional “Having” clause provides a possibility to define conditions for selection of groups.

For example, if we wish to just “Get such customers who bought more than 45 units of products”, the SQL statement would be:

```
Select * From Customer
Where C# In
 ( Select C# From Transaction
     Group By C#
     Having SUM (Qnt) > 45 )
```

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>P#</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 9-11.** Grouping followed by Restriction

In this case, those grouped customers with 45 units or less will not be in the final result. The result will thus only be:

<table>
<thead>
<tr>
<th>C#</th>
<th>Cname</th>
<th>Ccity</th>
<th>Cphone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
</tbody>
</table>
It is important to note that in the further filtering of values, the Where clause is used to exclude values before the Group By clause is applied, whereas the having clause is used to exclude values after they have been grouped.
10 Query-By-Example (QBE)

10.1 Introduction

Data Query Languages were developed in the early seventies when the man-machine interface was, by today’s standards, limited and rudimentary. In particular, interaction with the computer was through the processing of batched jobs, where jobs (computation requests such as “run this program on that data”, “evaluate this database query”, etc) were prepared off-line on some computer readable media (eg. punch cards), gathered into a ‘batch’ and then submitted for processing. No interaction takes place between the user and computer while the jobs were processed. End results were instead typically printed for the user to inspect (again off-line) and to determine the next course of action. The batch cycle continued until the user had obtained the desired results.

This was pretty much the way database queries were handled (see Figure 10-1). As data coding devices were exclusively textual in nature and as processing is non-interactive, queries must be defined textually and each query must be self-contained (ie. has all the components required to complete the evaluation). The design of early languages were influenced by, and in fact had to comply with, these constraints to be usable. Thus, for example, the SQL query:

\[
\text{Select P# from Transaction} \\
\text{where C# IN ( Select C# from Customer} \\
\text{where Ccity = London )}
\]

could be easily encoded as a job for batched submission. Needless to say, the turnaround time in such circumstances were high, taking hours or even days before a user sees the results of submitted queries. Many hours are typically spent off-line for a job that would take seconds to evaluate, and it is even worse if you made an error in your submission!

Over the past 20 years, however, man-machine interfaces or human-computer interaction (HCI) has progressed in leaps and bounds. Today, graphical user interfaces (GUI) are taken for granted and the batched mode of processing is largely a past relic replaced by highly interactive computing. Nevertheless, many database query languages today still retain the old ‘batch’ characteristics and do not exploit features of interactive interfaces. This is perhaps not surprising as, first, a large body of techniques for processing textual
languages had grown over the years (eg. compiling and optimisation) and, second, they were well suited for embedding in more general purpose programming languages. The latter especially provides great flexibility and power in database manipulation. Also, as the paradigm shifted to interactive computing, its application to database queries was not immediately obvious. But end-user computing is, in any case, increasing and many tasks that previously required the skills of expert programmers are now being performed by end-users through visual, interactive interfaces.

Query-By-Example (QBE) is the first interactive database query language to exploit such modes of HCI. In QBE, a query is a construction on an interactive terminal involving two-dimensional ‘drawings’ of one or more relations, visualised in tabular form, which are filled in selected columns with ‘examples’ of data items to be retrieved (thus the phrase query-by-example). The system answers the query by fetching data items based on the given example and drawing the result on the same screen (see Figure 10-2).

![Figure 10-2 A QBE query and its results](image)

Typically, the ‘drawing’ of relations are aided by interactive commands made available through pull-down menus (see Figure 10-3). The menu selection is constrained to relations available in the schema and thus eliminates errors in specifying relation structures or attribute names as can occur in text-based languages like SQL. The interface provided is in effect a structured editor for a graphical language.
For the remainder of this chapter, we will focus exclusively on the principal features of QBE. In contrast to SQL, QBE is based on relational calculus with domain variables (see 8.2). To close this introduction, we should mention that QBE was developed by M.M. Zloof at the IBM Yorktown Heights Laboratory.

10.2 Variables and Constants

In filling out a selected table with an example, the simplest item that can be entered under a column is a free variable or a constant. A free variable in QBE must be an underlined name (identifier) while a constant can be a number, string or other constructions denoting a single data value. A query containing such combinations of free variables and constants is a request for a set of values instantiating the specified variables while matching the constants under the specified columns.

As an example, look at Figure 10-4. Two variables are introduced in the query: $a$ and $b$. By placing a variable under a column, we are in effect assigning that variable to range over the domain of that column. Thus, the variable $a$ ranges over the domain $P#$ while $b$ ranges over $Pname$.

The reader would have also noted that the variables are prefixed by “P.”. In QBE, this is required if the instantiation found for the specified variable is to be displayed, i.e. the prefix “P.” may be thought of as a command to print. We will say more about prefix commands like this later. Suffice it for now to say that if neither variable in Figure 10-4 was preceded by “P.” then the result table would display nothing!

The query in Figure 10-4 is in fact equivalent to the following construction of relational calculus with domain variables:

$$a \rightarrow P#; \quad b \rightarrow Pname;$$

$$(a, b): ( \text{Product}(a, b))$$

Assuming the usual Product relation extension as in previous chapters, the result of the query is shown in Figure 10-5.
Let us consider another simple example and walk through the basic interactions necessary to formulate the query and get the desired results. Suppose we wanted the names and cities of all customers. The basic interactions are summarised in Figure 10-6.

Figure 10-6 Basic sequence of interactions

1. The user first uses a pull-down menu as in
2. 
3. 

4. Figure 10-3 to select the appropriate relation(s) containing the desired items. For this query, the Customer relation would seem the most appropriate and selecting it would result in an empty template being displayed.

5. Inspecting the template, the user can ascertain that the desired data items are indeed in the selected template (viz. The Cname and Ccity columns). Next, the user invents variable identifiers (a and b) and types each under the appropriate column. This is all that is required for this query.

6. Finally, the example is evaluated by the system and the results displayed on the screen.

This is the basic interaction even for more complex queries - select relation templates, fill in example items, then let the system evaluate and display the results. Of course, with more complex queries, more than one relation may be used and constructing the example will usually involve more than just free variables, as we shall see in due course.

Free variables unconditionally match data values in their respective domains and thus, by themselves, cannot express conditional queries, such as “get the names and phone numbers of customers who live in London” (the italicised phrase is the condition). The simplest specification of a condition in QBE is a constant, which is a single data value entered under a column and interpreted as the condition:
Thus, the condition ‘live in London’ is quite simply captured by typing ‘London’ under the ‘City’ attribute in the Customer template, as shown in Figure 10-7.

More generally, the QBE syntax for conditions is:

\[
\text{[<comparator>] <constant>}
\]

where comparator is any one of ‘=’, ‘\neq’, ‘<’, ‘\leq’, ‘>’, and ‘\geq’, and is interpreted as the condition

\[
\text{<attribute name> <comparator> <constant>}
\]

If <comparator> is omitted, it defaults to ‘=’ (as in the above example). As an example of the use of other comparators, the query “get the names of products costing more than 1000” would be as shown in Figure 10-8.

A query can also spread over several rows. This is the QBE equivalent form for expressing complex conjunctions and disjunctions of conditions. To correctly interpret multiple row queries, bear in mind the following:

- the ordering of rows is immaterial
- a variable identifier denotes the same instantiation wherever it occurs

The second point above is particularly important when a variable occurs in more than one row. But let’s consider first the simpler case where distinct rows do not share any variable. In this case, the rows are unrelated and can be evaluated independently of one another and the final result is simply the union of the results of each row. The collective condition of such a query is thus a disjunction of the conditions specified in each row.
For example, consider the query: “Get the names of customers who either live in London or Paris and whose personal number is greater than 1”. The QBE query for this is shown in Figure 10-9. Looking at row 1, note that two conditions are specified. These must be satisfied by values from a single tuple, i.e. the condition may be restated as

\[ C\# > 1 \text{ AND } C\text{city}=\text{London} \]

Similarly, the condition specified in row 2 is

\[ C\# > 1 \text{ AND } C\text{city}=\text{Paris} \]

As the two rows do not share variables, the collective condition is a disjunction

\[ (C\# > 1 \text{ AND } C\text{city}=\text{London}) \text{ OR } (C\# > 1 \text{ AND } C\text{city}=\text{Paris}) \]

which may be simplified to

\[ C\# > 1 \text{ AND } (C\text{city}=\text{London} \text{ OR } C\text{city}=\text{Paris}) \]

**Figure 10-9** Multiple disjunctive rows

In contrast, if a variable occurs in more than one row, then the conditions specified for each row must be true for the same value of that variable. Consider, for example, the query in Figure 10-10 where the variable \( x \) occurs in both rows.

This means that a value of \( x \) must be found such that both row 1 and row 2 are simultaneously satisfied. In other words, the condition for this query is equivalent to

\[ \text{Ccity} = \text{London} \text{ AND } C\# > 1 \text{ AND } C\# < 4 \]

(Given the above Customer relation, only the value “Deen” satisfies both rows in this case.)

There is another possibly simpler way of describing the meaning and evaluation of multiple row queries. Specifically, we treat each row as a sub-query; evaluate each separately, then merge the results (a set of tuples for each sub-query) into a single table. The merging of two sets of tuples is simply a union, if their corresponding sub-queries do not share variables. Otherwise, their intersection over attributes that share variables is computed instead.

Thus, for the query in Figure 10-9, the first sub-query (row 1) results in the set \{Deen\}, while that of the second sub-query (row 2) is \{Martin\}. As the sub-queries do not share variables, the final result is simply the union of these results: \{Deen, Martin\}.

In contrast, for the query in Figure 10-10, the first sub-query (row 1) results in \{Deen\}, while the second (row 2) results in \{Codd, Deen\}. But as the sub-queries share the
variable \( x \) under attribute Cname, the merged result is the intersection of the two, i.e. \{Deen\}.

Before proceeding with the next section, we should just mention here some syntactical constraints and options of QBE. First, the prefix “P.” can be used on any example item, not just free variables. This underlines its earlier interpretation, i.e. it is a command to “print” or “display” the value of the item it prefixes (variable or comparison). Thus, if the query in Figure 10-10 had been:

```
Customer
C#  | Cname | City |
-----|-------|------|
P.>1 | P.x1  | P.London |
P.>1 | P.x2  | P.Paris  |
```

then the displayed result would be:

```
Customer
C#  | Cname | City | Cphone |
-----|-------|------|--------|
2    | Martin| Paris|        |
3    | Deen  | London|        |
```

Note that, in general, prefixing a comparison prints the value that satisfies it. Of course, in the case of a constant (implicitly a “=” comparison), the constant itself will be printed.

QBE also allows the user to simplify a query to only essential components. This is largely optional and the user may choose (perhaps for greater clarity) to include redundant constructs. Basically, there are two rules that can be applied:

1. If a particular variable is used only once, then it may be omitted. This saves the user the trouble of otherwise having to invent names. Application of this rule is illustrated in Figure 10-11, where it is applied to the first table (variables \( x_1 \) and \( x_2 \)) to result in the second. Note that unless this rule is kept in mind when reading simplified queries, the appearance of the prefix “P.” by itself may not only look odd but confusing too. The prefixes in the second table must be correctly read as prefixing implicit but distinct variables.

2. Duplicate prefixes and constants occurring over multiple rows may be “factorised” into just one row. This is illustrated also in Figure 10-11 where it is applied to the second table to result in the third. Again, unless this rule is kept in mind, queries such as that in the third table may seem meaningless.
While the above rules are optional, the following is a syntactic constraint that must be observed: if a free variable occurs in more than one row, then the prefix “P.” may be used on at most one of its occurrences.

The query below illustrates a valid construction - note that \( x \) occurs in two rows but only one of them has the P prefix.

10.3 Example Elements

Each row of a query table may be seen as an example of tuples from the associated relation—specifically, tuples that match the row. A tuple matches a row if each attribute value in the tuple matches the corresponding query item in the row. We have seen above exactly when a value matches a query item. In summary:

1. Any value matches a blank query item or a variable
2. A value matches a comparison item if it satisfies the specified comparison

Using these rules, it is relatively easy to ascertain tuples exemplified by a query row. This is illustrated in Figure 10-12. This is why variables in QBE are called example elements.

In extracting facts from several relations that share attribute domains, example elements are the key to finding related target tuples from the different relations. Consider the query:
“Get the names and phone numbers of customers that have purchased both product number 1 and product number 2”.

The Transaction relation has part of the information we are after. Specifically, we look for records of purchase of each item by the same customer, i.e. a tuple where the product number is 1, another where the product number is 2, but both with the same customer number. The entries in the Transaction template in Figure 10-13 capture this requirement.

However, this tells us only the customer number (the instantiation of $X$). Information about the customer’s name and phone number must be obtained from the Customer relation. We need to ensure, though, that these values are obtained from a customer tuple that represents the same customer found in the Transaction relation. In QBE, this is simply achieved by specifying the same example element $X$ in the customer number column of the Customer relation (as shown in the Customer template of Figure 10-13).

The query in Figure 10-13 may be evaluated, assuming the following extensions of Transaction and Customer, as follows.

1. The subquery in the first row of the Transaction template is matched by the first and third tuples of the Transaction relation, i.e. $X = \{1,2\}$
2. The subquery in the second row of the Transaction template is matched by the second and fourth tuples of the Transaction relation, i.e. $X = \{1,3\}$
3. The result of evaluating the Transaction template is therefore $\{1,2\} \cap \{1,3\} = \{1\}$.
4. The subquery in the Customer template matches all the tuples in the Customer relation, i.e. the entire relation is the result.
5. The final result is the intersection, over C#, of the results in (3) and (4), i.e. $\{<\text{Codd}, 2263035>\}$

Figure 10-14 shows another example of a multi-table query and illustrates also the relative ease in “reading” or paraphrasing QBE constructs. First, the Customer subquery makes it clear, from the use of “P.” prefix, that the desired result is a set of customer names and their phone numbers (the elements $a$ and $b$ respectively). The element $x$ links
Customer to Transaction, ie. a customer included in the result must have purchased something, denoted yet by another element \( y \). Furthermore, \( y \) must be such that it is the product CPU.

**Figure 10-14** Another example of a multi-table query with example elements

In other words, the query can be paraphrased as:

“Get the names and phone numbers of those customers who bought the product CPU”.

The preceding two examples should be enough for the reader to realise that (unadorned) example elements spread across tables are in fact existentially quantified. For example, there may be more than one Transaction tuple that can match the customer number found in Customer, but we don’t care which! The examples also show that, more generally, a QBE query can spread over a number of rows of a single relation and across other relations. A few further examples will serve to highlight QBE’s power and features.

In Figure 10-15, we see a complex-looking QBE query. A closer examination will reveal, however, that within each relation template the rows do not share elements, although the elements are shared across relations. In fact, there are two disjoint sets of rows - one taken from the first row of each relation and the other from the second row of each relation.

The first set is actually equivalent to the QBE query in Figure 10-14.

![Table 1](image1.png)

![Table 2](image2.png)

**Figure 10-15** Disjunctive multi-table query

The second differs only in the specified product (replace ‘CPU’ by ‘VDU’ in the above paraphrased query). By analogy with earlier constructions involving unrelated multiple rows, this type of construction therefore denotes a disjunctive query. In other words, combining the two sets of rows yield the query:

“Get the names and phone numbers of those customers who bought the product CPU or the product VDU”

Earlier, we’ve seen examples of elements used in multiple rows of the same relation. However, given now an understanding of multi-table queries, such constructions can equivalently be seen as a multi-table query involving the same table! This is shown in Figure 10-16 below.

![Table 3](image3.png)

![Table 4](image4.png)
Example elements may also be negated. Negated elements are written with the prefix ‘!’ eg. !X (read “not X”). The negated form can only be used if there is at least one occurrence of the unnegated element elsewhere in the query. It is then interpreted as matching any corresponding domain value that the unnegated form did not match.

Consider, for example, the illustration in Figure 10-17. There are two parts to the illustration, labelled (a) and (b), each with a query table and an extension of the corresponding relation. For purposes of this example, the two query tables constitute a multi-table query, i.e. the example element X is the same one in both. Note, however, that X is negated in (b).

Given the extension of Transaction as shown, the domain values matching the example element X in (a) is {1,2}. Turning now to the subquery in (b), the specification of ‘!X’ in it means that the only tuples that can match it are tuples such that the C# value is not in {1,2}. Given the extension of Customer as shown, this means that only the third tuple matches the example, i.e. the answer returned for elements A and B are ‘Deen’ and ‘2234391’ respectively.

### Figure 10-17 Negated Element

#### 10.4 The Prefix ALL

The prefix ALL can be applied to example elements. The occurrence of such an element in an arbitrary query row of an arbitrary relation denotes a set of values such that each, together with a particular instantiation of other items in the row, matches a tuple of the relation. As an example, consider the following relation and query:
In this case, there is only one other item in the query row: another element \( X \). The set of values denoted by ‘All.\( Y \)’ therefore needs to be determined for each value that \( X \) takes. Thus,

- when \( X = 1 \), there are two possible values for \( Y \), ie. 1 and 2. Thus, ‘All.\( Y \)’ is the set \( \{1,2\} \)
- when \( X = 2 \), there is only one value for \( Y \), ie. the set \( \{1\} \)
- when \( X = 3 \), there is also only one value for \( Y \), ie. the set \( \{2\} \)

If the query items had been prefixed with ‘P.’, the result displayed would be:

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>I2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>{1,2}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>{1}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>{2}</td>
<td></td>
</tr>
</tbody>
</table>

In the simplest case, a query row contains only one element prefixed with ALL. In this case, the element simply denotes the set of values in the corresponding domain. This is illustrated in Figure 10-19 below.

The use of ALL is more interesting when it involves multitable queries. For example, combining the query in Figure 10-18 and Figure 10-19 into a single query, we effectively restrict \( X \) to just the value 1. This is because ALL.\( Y \) occurs in both tables and must denote the same set, and the only set satisfying this is \( \{1,2\} \).

It should be clear now that ALL is used in QBE in the same way that a universal quantifier is used in relational calculus with domain variables. To highlight this, consider the query:

“Get the names of customers who bought all types of the company’s product”

Three relations are required to resolve this query: Customer, Transaction and Product. The QBE query is shown in Figure 10-20 which is also annotated with explanations.
Introduction to Databases & Relational DM

**Figure 10-20** The query “Get the names of customers who bought all types of the company’s product”

One final word about ALL: it does not remove duplicate values, in contrast to an unprefixed element which will return only unique matching values. This is illustrated in Figure 10-21 below. We shall see in the next section how this property is used (if fact, is necessary) in order to answer certain classes of practical queries.

**Figure 10-21** ALL does not remove duplicates!

### 10.5 Library Functions

As with SQL, QBE also provides arithmetic operations and a number of built-in functions which are necessary to manipulate the values in ways not otherwise within the scope of relational calculus, e.g. to count the number of occurrences of returned values or to sum them up. As you may expect by now, these operations are provided in the form of prefixes. For example, suppose we wish to know how many transactions were related to the purchase of a particular product, say product number 1. We can extract, for example, all customer numbers in transactions involving product number 1:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Transaction (Query)</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>P#</td>
<td>Date</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>23.01</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29.01</td>
</tr>
</tbody>
</table>

But what we are really interested in is counting the number of such values. QBE allows us to do this with the prefix CNT (equivalent to the function COUNT in SQL), which counts the number of values matching the element it prefixes.

Thus the same query above, different only in the addition of the CNT prefix, achieves the desired result:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Transaction (Query)</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>P#</td>
<td>Date</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>21.01</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>23.01</td>
</tr>
</tbody>
</table>
Note that the use of ALL is necessary. If the example element was simply “P.CNT.X”, the result would be 2! This is because without the ALL prefix, the values matching the element X are returned with duplicate values removed (as illustrated earlier in Figure 10-21).

Another frequently used function is SUM, which sums up the values matching the example element it prefixes. Suppose, we wish to know the total number of product number 1 that has been sold. Instead of counting the number of customers that purchased it, we sum instead the quantities recorded in the relevant transactions. Thus:

```
Transaction (Query)  | C#  | Date | Qty
---------------------|-----|------|-----
| 1                 | P.SUM.All.X | 65 |
```

QBE also allows us to group tuples in a relation based on a specified example element. That is, tuples with the same value of the example element are collected into one group (there will be as many groups as there are distinct values matching the example element). Grouping is specified using the G prefix (this is similar to the ‘Group By’ clause in SQL). Thus:

```
Transaction (Query)  | C#  | P#  | Date | Qty
---------------------|-----|-----|------|-----
| 1                 | P.G.X | 1 | 21.01 | 20 |
| 1                 | 2 | 23.01 | 30 |
| 2                 | 1 | 26.01 | 25 |
| 1                 | 1 | 29.01 | 20 |
```

Aritmetic functions may be applied to groups. Thus, if we wanted to know the total number of items purchased by each customer, we can modify the above query as follows:

```
Transaction (Query)  | C#  | P#  | Date | Qty
---------------------|-----|-----|------|-----
| 1                 | P.G.X | P.SUM.All.B | 70 |
| 1                 | 2 | 23.01 | 30 |
| 2                 | 1 | 26.01 | 25 |
| 1                 | 1 | 29.01 | 20 |
```

Groups may additionally be selected based on conditions that are specified in an additional column (this corresponds to the ‘Having clause’ of SQL). This additional conditions column may be created by means of a special menu item in the QBE interface.
Thus, if we are only interested in finding customers who have purchased more than 45 items, our query would be as follows:

In summary, grouping and arithmetic functions can be used in combination to obtain useful *derived* values from the database.
11 Architecture of Database Systems

11.1 Introduction

Software systems generally have an architecture, i.e., possessing of a structure (form) and organisation (function). The former describes identifiable components and how they relate to one another structurally; the latter describes how the functions of the various structural components interact to provide the overall functionality of the system as a whole. Since a database system is basically a software system (albeit complex), it too possesses an architecture. A typical architecture must define a particular configuration of and interaction between data, software modules, meta-data, interfaces and languages (see Figure 11-1).

The architecture of a database system determines its capability, reliability, effectiveness and efficiency in meeting user requirements. But besides the visible functions seen through some data manipulation language, a good database architecture should provide:

a) Independence of data and programs
b) Ease of system design
c) Ease of programming
d) Powerful query facilities
e) Protection of data

The features listed above become especially important in large organisations where corporate data are held centrally. In such situations, no single user department has responsibility over, nor can they be expected to know about, all of the organisation’s data. This becomes the job of a Database Administrator (DBA) who has a daunting range of responsibilities that include creating, expanding, protecting and maintaining the integrity of all data while addressing the interests of different present and future user communities. To create a database, a DBA has to analyse and assess the data requirements of all users and from these determine its logical structure (database

![Figure 11-1 General Database System Architecture](image-url)
Introduction to Databases & Relational DM

schema). This, on the one hand, will need to be efficiently mapped onto a physical structure that optimises retrieval performance and the use of storage. On the other, it would also have to be mapped to multiple user views suited to the respective user applications. For large databases, DBA functions will in fact require the full time services of a team of many people. A good database architecture should have features that can significantly facilitate these activities.

11.2 Data Abstraction

To meet the requirements above, a more sophisticated architecture is in fact used, providing a number of levels of data abstraction or data definition. The database schema, also known as Conceptual Schema, mentioned above represents an information model at the logical level of data definition. At this level, we abstract out details like computer storage structures, their restrictions, or their operational efficiencies. The view of a database as a collection of relations or tables, each with fixed attributes and primary keys ranging over given domains, is an example of a logical level of data definition.

The details of efficiently organising and storing objects of the conceptual schema in computers with particular hardware configurations are dealt with at the internal (storage) level of data definition. This level is also referred to as the Internal Schema. It maps the contents of the conceptual schema onto structures representing tuples, associated key organisations and indexes, etc, taking into account application characteristics and restrictions of a given computer system. That is, the DBA describes at this level how objects of the conceptual schema are actually organised in a computer. Figure 11-2 illustrates these two levels of data definition.

At a higher level of abstraction, objects from the conceptual schema are mapped onto views seen by end-users of the database. Such views are also referred to as External Schemas. An external schema presents only those aspects of the conceptual schema that are relevant to the particular application at hand, abstracting out all other details. Thus, depending on the requirements of the application, the view may be organised differently from that in the conceptual schema, e.g. some tables may be merged, attributes may be suppressed, etc. There may thus be many views created—one for each type of application. In contrast, there is only one conceptual and one internal schema. All views

Figure 11-2 The Logical and Internal Levels of Data Abstraction
are derived from the same conceptual schema. This is illustrated in Figure 11-3 which shows two different user views derived from the same conceptual schema.

Thus, modern database systems support three levels of data abstraction: External Schemas (User Views), Conceptual Schema, and Internal (Storage) Schema.

The DDL we discussed in earlier chapters is basically a tool only for conceptual schema definition. The DBA will therefore usually need special languages to handle the external and internal schema definitions. The internal schema definition, however, varies widely over different implementation platforms, ie. there are few common principles for such definition. We will therefore say little more about them in this book.

As to external schema definitions, note that in the relational model, the Data Sub-Languages can be used to both describe and manipulate data. This is because the expressions of a Data Sub-Language themselves denote relations. Thus, a collection of new (derived) relations can be defined as an external schema.

For example, suppose the following relations are defined:

- Customer( C#, Cname, Ccity, Cphone )
- Product( P#, Pname, Price )
- Transaction( C#, P#, Date, Qnt )

We can then define an external view with a construct like the following:

Define View My_Transaction_1 As
    Select Cname, Ccity, Date, Total_Sum=Price*Qnt
    From Customer, Transaction, Product
    Where Customer.C# = Transaction.C#
    & Transaction.P# = Product.P#

which defines the relation (view):

My_Transaction_1( Cname, Ccity, Date, Total_Sum )
This definition effectively maps the conceptual database structure into a form more convenient for a particular user or application. The extension of this derived table is itself derived from the extensions of the source relations. This is illustrated in Figure 11-4 below.

![Figure 11-4 External View Definition](image)

This is a very important property of the relational data model: *a unified approach to data definition and data manipulation.*

### 11.3 Data Administration

Functions of a DBA include:

- Creation of the database
  
  To create a database, a DBA has to analyse and assess the requirements of the users and from these determine its logical structure. In other words, the DBA has to design a conceptual schema and a first variant of an internal schema. When the internal schema is ready, the DBA must load the database with actual data.

- Acting as intermediary between users and the database

  A DBA is responsible for all user facilities determined by external schemas, ie. the DBA is responsible for defining all external schemas or user views.

- Ensuring data privacy, integrity and security

  In analysing user requirements, a DBA must determine who should have access to which data and subsequently arrange for appropriate privacy locks (passwords) for identified individuals and/or groups. The DBA must also determine integrity constraints and arrange for appropriate data validation to ensure that such constraints are never violated.

  Last, but not least, the DBA must make arrangements for data to be regularly backed up and stored in a safe place as a measure against unrecoverable data losses for one reason or another.
At first glance, it may seem that a database can be developed using the conventional “waterfall” technique. That is, the development process is a sequence of stages, with work progressing from one stage to the next only when the preceding stage has been completed. For relational database development, this sequence will include stages like eliciting user requirements, analysing data relationships, designing the conceptual schema, designing the internal schema, loading the database, defining user views and interfaces, etc, through to the deployment of user facilities and database operations.

In practice, however, when users start to work with the database, the initial requirements inevitably change for a number of reasons including experience gained, a growing amount of data to be processed, and, in this fast changing world, changes in the nature of the business it supports. Thus, a database need to evolve, learning from experience and allowing for changes in requirements. In particular, we may expect periodic changes to:

- improve database performance as data usage patterns changes or becomes clearer
- add new applications to meet new processing requirements
- modify the conceptual schema as understanding of the enterprise’s perception of data improves

Changing a database, once the conceptual and internal schemas have been defined and data actually loaded, can be a major undertaking even for seemingly small conceptual changes. This is because the data structures at the storage layer will need to be reorganised, perhaps involving complete regeneration of the database. A good DBMS should therefore provide facilities to modify a database with a minimum of inconvenience. The desired facilities can perhaps be broadly described to cover:

- performance monitoring
- database reorganisation
- database restructuring

By performance monitoring we mean the collection of usage statistics and their analysis. Statistics necessary for performance optimisation generally fall under two headings: static and dynamic statistics. The static statistics refer to the general state of the database and can be collected by special monitoring programs when the database is inactive. Examples of such data include the number of tuples per relation, the population of domains, the distribution of relations over available storage space, etc. The dynamic statistics refer to run-time characteristics and can be collected only when the database is running. Examples include frequency of access to and updating of each relation and domain, use of each type of data manipulation operator and associated response times, frequency of disk access for different usage types, etc.

It is the DBA’s responsibility to analyse such data, interpret them and where necessary take steps to reorganise the storage schema to optimise performance. Reorganising the storage schema also entails the subsequent physical reorganisation of the data themselves. This is what we mean by database reorganisation.

The restructuring of the conceptual schema implies changing its contents, such as:

- adding/removing data items (ie. columns of a relation)
- adding/removing entire relations
• splitting/recombining relations
• changing a relation’s primary keys
• …etc

For example, assuming the relations as on page 146, suppose we now wish to record also for each purchase transaction the sales representative responsible for the sale. We will need therefore to add a column into the Transaction relation, say with column name R#:

\[
\text{Transaction( C\#, P\#, R\#, Date, Qnt )}
\]

new attribute added

The intention, of course, is to record a unique value under this column to denote a particular sales representative. Details of such sales representatives will then be given in a new relation:

\[
\text{Representative( R\#, Rname, Rcity, Rphone)}
\]

A restructured conceptual schema will normally be followed by a database reorganisation in the sense explained above.

### 11.4 Data Independence

Data independence refers to the independence of one user view (external schema) with respect to others. A high degree of independence is desirable as it will allow a DBA to change one view, to meet new requirements and/or to optimise performance, without affecting other views. Relational databases with appropriate relational sub-languages have a high degree of data independence.

For example, suppose that the view

\[
\text{My\_Transaction\_1( Cname, Ccity, Date, Total\_Sum )}
\]

as defined on page 146 no longer meet the user’s needs. Let’s say that Ccity and Date are no longer important, and that it is more important to know the product name and quantity purchased. This change is easily accommodated by changing the select-clause in the definition thus:

Define View My\_Transaction\_1 As
\[
\text{Select Cname, Pname, Qnt, Total\_Sum=Price*Qnt}
\]
From Customer, Transaction, Product
Where Customer.C\# = Transaction.C\#
& Transaction.P\# = Product.P\#

This replaces the original specification of Ccity and Date items

If each view is defined separately over the conceptual schema, then as long as the conceptual schema does not change, a view may be redefined without affecting other views. Thus the above change will have no effect on other views, unless they were built upon My\_Transaction\_1.

Data independence is also used to refer to the independence of user views relative to the conceptual schema. For example, the reader can verify that the change in the conceptual schema in the last section (adding the attribute R# to Transaction and adding the new relation Representative), does not affect My\_Transaction\_1 - neither the original nor the changed view!. In general, if the relations and attributes referred to in a view definition
are not removed in a restructuring, the view will not be affected. Thus we can accommodate new (additive) requirements without affecting existing applications.

Lastly, data independence may also refer to the extent to which we may change the storage schema without affecting the conceptual or external schemas. We will not elaborate on this as we have pointed out earlier that the storage level is too diverse for meaningful treatment here.

### 11.5 Data Protection

There are generally three types of data protection that any serious DBMS must provide. These were briefly described in Chapter 1 and we summarise them here:

1. **Authorisational Security**
   
   This refers to protection against unauthorised access and includes measures such as user identification and password control, privacy keys, etc.

2. **Operational Security**
   
   This refers to maintaining the integrity of data, i.e. protecting the database from the introduction of data that would violate identified integrity constraints.

3. **Physical Security**
   
   This refers to procedures to protect the physical data against accidental loss or damage of storage equipment, theft, natural disaster, etc. It will typically involve making periodic backup copies of the database, transaction journalling, error recovery techniques, etc.

In the context of the relational data model, we can use relational calculus as a notation to define integrity constraints, i.e. we define them as formulae of relational calculus. In this case, however, all variables must be bound variables as we are specifying properties over their ranges rather than looking for particular instantiations satisfying some predicate. For example, suppose that for the Product relation, the Price attribute should only have a value greater than 100 and less than 99999. This can be expressed (in DSL Alpha style) as:

\[
\text{Range Product X ALL; (X.Price > 100 & X.Price < 99999 )}
\]

This is interpreted as an assertion that must always be true. Any data manipulation that would make it false would be disallowed (typically generating messages informing the user of the violation). Thus, not only does the relational data model unify data definition and manipulation, but its control as well.

In the area of physical security, database backups should of course be done periodically. For this purpose, it is perhaps best to view a database as a large set of physical pages, where each page is a block of fixed size serving as the basic unit of interaction between the DBMS and storage devices. A database backup is thus essentially a copy of the entire set of pages onto another storage medium that is kept in a secure and safe place. Aside from the obvious need for backups against damage of storage devices, theft, natural disasters and the like, backups are necessary to recover a consistent database in the event...
of a database ‘crash’. Such crashes can occur in the course of a sequence of database transactions, particularly transactions that modify the database content.

Suppose, for example, that the last backup was done at time \( t_0 \), and subsequent to that, a number of update transactions were applied one after another. Suppose further that the first \( n \) transactions were successfully completed, but during the \((n+1)\)th transaction a system failure occurred (e.g. disk malfunction, operating system crash, power failure, etc) leaving some pages in a corrupted state. In general, it is not possible to just reapply the failed transaction—the failure could have corrupted the updates performed by previous transactions as well, or worse, it could have damaged the integrity of the storage model as to make some pages of the database unreadable! We have no recourse at this point but to go back to the last known consistent state of the database at time \( t_0 \), i.e. the entire contents of the last backup is reinstated as the current database. Of course, in doing so, all the transactions applied after \( t_0 \) are lost.

At this point it may seem reasonable that, to guard against losing too much work, backups should perhaps be done after each transaction—then at most only the work of one transaction is lost in case of failure. However, many database applications today are transaction intensive typically involving many online users generating many transactions frequently (e.g. online airline reservation system). Many databases, on the other hand, are very large and an entire backup could take hours to complete. While backup is being performed the database must be inactive. Thus, it should be clear that this proposition is impractical.

As it is clearly desirable that transactions since the last backup are also somehow saved in the event of crashes, an additional mechanism is needed. Essentially, such mechanisms are based on journalling successful transactions applied to a database. This simply means that a copy of each transaction (or affected pages) is recorded in a sequential file as they are applied to the database.

The simplest type of journalling is the *Forward System Journal*. In this, whenever a page is modified, a copy of the modified page is also simultaneously recorded into the forward journal.

To illustrate this mechanism, let the set of pages in a database be \( P = \{p_1, p_2, \ldots, p_n\} \). If the application of an update transaction \( T \) on the database changes \( P_T \), where \( P_T \subseteq P \), then \( T(P_T) \) will be recorded in the forward journal. We use the notation \( T(P_T) \) to denote the set of pages \( P_T \) after the transaction \( T \) has changed each page in \( P_T \). Likewise, we write \( T(p_i) \) to denote a page \( p_i \) after it has been changed by transaction \( T \). Furthermore, if \( T \) was applied successfully (i.e. no crash during its processing), a separator mark, say ‘;’, would be written to the journal. Thus, after a number of successful transactions, the journal would look as follows

\[
< T(P_{T1}) ; T(P_{T2}) ; \ldots ; T(P_{Tk}) ; >
\]

As a more concrete example, suppose transaction \( T1 \) changed \( \{p_1, p_2, p_3\} \), \( T2 \) changed \( \{p_2, p_3, p_4\} \), and \( T3 \) changed \( \{p_3, p_4, p_5\} \), *in that order* and all successfully carried out. Then the journal would contain:

\[
< T1( \{p_1, p_2, p_3\} ) ; T2( \{T1(p_2), T1(p_3), p_4\} ) ; T3( \{T2(T1(p_3)), T2(p_4), p_5\} ) ; >
\]
Now suppose a crash occurred just after T3 has been applied. The recovery procedure consists of two steps:

a) replace the database with the latest backup

b) read the system journal in the forward direction (hence the term ‘forward’ journal) and, for each set of journal pages that precedes the separator ‘;’, use it to replace the corresponding pages in the database. Effectively, this duplicates the effect of applying transactions in the order they were applied prior to the crash.

The technique is applicable even if the crash occurred during the last transaction. In this case, the journal for the last transaction would be incomplete and, in particular, the separator ‘;’ would not be written out. Say that transaction T3 was interrupted after modifying pages p_3 and p_4 but before it could complete modifying p_5. Then the journal would look as follows:

\[
< T1( \{p_1, p_2, p_3\} ) ; T2( \{T1(p_2), T1(p_3), p_4\} ) ; T3( \{T2(T1(p_3)), T2(p_4), \ldots\} ) >
\]

Incomplete entry

In this case, recovery is exactly as described above except that the last incomplete block of changes will be ignored (no separator ‘;’). Of course, the work of the last transaction is lost, but this is unavoidable. It is possible, however, to augment the scheme further by saving the transaction itself until its effects are completely written to the journal. Then T3 above can be reapplied, as a third step in the recovery procedure.

While the forward journal can recover (almost) fully from a crash, its disadvantage is that it is still a relatively slow process—hundreds or even thousands of transactions may have been applied since the last full backup, and the corresponding journals of each of these transactions must be copied back in sequence to restore the state of the database. In some applications, very fast recovery is needed.

In these cases, the Backward System Journal will be the more appropriate journalling and recovery technique. With this technique, whenever a transaction changes a page, the page contents before the update is saved. As before, if the transaction successfully completes, a separator is written. Thus the backward journal for the same example as above would be:

\[
<\{ p_1, p_2, p_3 \} ; \{ T1(p_2), T1(p_3), p_4 \} ; \{ T2(T1(p_3)), T2(p_4), \ldots\} >
\]

Since each block of journal pages represents the state immediately before a transaction is applied, recovery consists of only one step: read the journal in the backward direction until the first separator and replace the pages in the database with the corresponding pages read from the journal. Thus, the backward journal is like an ‘undo’ file—the last block cancels the last transaction, the second last cancels the second last transaction, etc.

Features such as those discussed above can significantly facilitate the management of corporate data resources. Such features, together with the overall architecture and the Data Model examined in previous chapters, determine the quality of a DBMS and are thus often used as part of the principal criteria used in critical evaluation of competing DBMSs.