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.

![Figure 3.1: Data Definition and Manipulation facilities of a Data Model](image)

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.
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.

**Step 1.** The user chooses the relation, Customer.

**Step 2.** The user issues a DML command to retrieve the tuple to be deleted.

**Step 3.** The DBMS deletes the tuple from the relation.
The updated relation now has one less tuple.

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.

**Step 1.** The user chooses the relation.

**Step 2.** The user issues a DML retrieval command to get the tuple to be changed.

**Step 3.** The user modifies one or more data items.

**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 Cphone 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></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>

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

### 3.32 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.

![Figure 3.2: Two different interfaces to the database](image)
3.31 Self-Contained Language

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

**Figure 3.3**: Expanding DML with additional functions

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>
</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.

```
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?

### 3.32 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.
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),
- 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.
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 prevent 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.

<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>1</td>
<td>Codd</td>
<td>London</td>
<td>2263035</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Martin</td>
<td>Paris</td>
<td>5555910</td>
</tr>
<tr>
<td>?</td>
<td>3</td>
<td>Deen</td>
<td>London</td>
<td>2234391</td>
</tr>
</tbody>
</table>

Figure 3.7: Entity Integrity Violation
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.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Cname</td>
</tr>
<tr>
<td>1</td>
<td>1 Codd</td>
</tr>
<tr>
<td>2</td>
<td>2 Martin</td>
</tr>
<tr>
<td>3</td>
<td>3 Deen</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 3.8**: Referential Integrity Violation

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)

   Figure 3.9: Domain Constraint Violation
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).

![Figure 3.12: Allowable Null Foreign Key](image)

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

Figure 3.13: Database System Components

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.
Figure 3.14: Correspondance between Transaction, DML and Physical Operations

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.