Chapter 2

Data protection

Essential reading

"An Introduction to Database Systems", sixth edition, by C. J. Date, published by Addison-Wesley, 1995, [ISBN 0-201-82458-2], *Chapter13, Chapter 14 and Chapter 15* (pp. 347-437).

Alternatively

"Database Systems: A Practical Approach to Design, Implementation and Management", second edition, by T. Connolly and C. E. Begg, 1999, [ISBN 0-201-34287-1], *Chapter 17* (pp. 558-592) and *Chapter 16* (pp. 531-546).

Further reading

"Database Systems Concepts", second edition, by H. F. Korth and A. Silbershatz, published by McGraw-Hill, 1991, [ISBN 0-07-100804-7], *Chapter 10, Chapter 11 and Chapter 12* (pp. 313-422) — for a more concise and basic reading.

"Database Systems: A Practical Approach to Design, Implementation and Management", second edition, by T. Connolly and C. E. Begg, 1999, [ISBN 0-201-34287-1], *Chapter 17* (pp. 592-601) — for some more advanced topics.

Introduction

The database is situated at the core of the information system of an organisation; all the organisation's relevant information is stored in its database. This characteristic has two consequences.

- There are many users who, directly or through application programs, access and manipulate the data stored in the database. Moreover, their data manipulation requests can occur at any particular moment of time. This high demand for data access increases the probability of data getting corrupted.
- Many of the organisation's activities rely on this data. Consider, for example, a company that includes a few factories, a personnel department, a finance department, a sales department, etc., depend almost entirely upon the data stored in the database. Therefore, if this data gets corrupted, the consequences can be catastrophic for the organisation.

Even from this brief presentation, we can conclude that it is essential for the DBMS to guarantee that data is safely stored. The DBMS must guarantee that data cannot be lost or damaged by accident or by evil will. In short, the DBMS must provide mechanisms for *data protection.*

In order to guard oneself from a certain problem, one needs to know what the problem is. In order to specify the data protection mechanisms that a DBMS must provide, the problems that would lead to data corruption have to be identified. Let us consider the following three situations.

- The DBMS, or a part of it, might crash while executing certain operations on the database. Because the operations were not successfully completed, the database is left in an unpredictable state, most probably damaged.

- Two different application programs (or interactive users) might operate on the database at the same time – *concurrently*. In this case it is possible that during some update operations performed by an application A1, another application, A2, interferes and performs some update operations on the same data. Because A1 is not aware of this interference, the database might end in an undesired / unpredictable state, probably damaged.
- Non-authorised people (hackers) might get access to data that they should not be able to see. Motivated by evil purposes, they might perform certain operations that leave it in a damaged state.

The problems above illustrate three important issues related to data protection, namely:

- *recovery*;
- *concurrency control*;
- *security*.

They constitute the content of the present chapter.

Data recovery and concurrency control are both aspects of a more general topic, *transaction processing*. However, rather than presenting the general issues related to transaction processing, an application driven presentation was preferred (by the author) in this chapter.

The issues related to transaction processing are generic, in that they are not restricted to relational DBMSs. However, because (1) most of the research work on transactions was applied to the relational model and (2) you are familiar with the relational model, these issues are presented, in this chapter, in a relational context.

Data recovery

You must have experienced, at least once, the frustrating feeling of losing some important piece of work, because the system you were using crashed just before you managed to save it. If you were disciplined, you had some recent backups and you could restart your work from there. If you were lucky, the system you were using had automatically maintained a backup of your work, and so you did not loose everything. Otherwise, you simply lost everything. This example introduced the idea of data recovery.

Consider a more specific example, a situation you have almost certainly encountered. Suppose you were working on a complex document in Microsoft Word[1], a document that contained all kind of diagrams, graphs and pictures, requiring therefore a considerable amount of memory. Suppose you also had some other applications open at the same time, such as Excel, PowerPoint, a C compiler, an email system, an Internet browser, etc. You were cutting from these applications and pasting into your document. You were working for a few hours and you had forgotten to save the document. Then, possibly because you did not have a very powerful computer, it hanged and did not respond to any commands at all. The only solution left was to reboot.

However, you did not lose all your work; Microsoft Word automatically maintains temporary files as backups for the open documents. You remembered you were annoyed by the fact that every ten minutes the system seemed to stop for a few seconds, automatically saving the document in the backup (recovery) file and you even wanted to turn off that feature. After the crash, you were glad you did not do it. For, after you rebooted the system and restarted Word, your work was recovered from the backup file almost completely, due to the fact that Word automatically save your work just before the crash.

---

[1] The author is not trying to denigrate Microsoft (in fact, he mainly uses the Windows platform), but this was one of the biggest drawbacks of Windows 3.1 (and, subsequently, of the applications developed on this platform).

There are two ideas that can be inferred from the above example.

- Data recovery is based on data redundancy (duplication of data). Moreover, this redundant data must exist on the permanent support. It is not sufficient to have the data duplicated in the computer's memory (it would be lost in the case of a crash), it has to be written on the physical external (permanent) support.
- The better the data recovery features are, the more inefficient the system becomes. In the example above, in order to ensure that not too much data is lost in case of a crash, the frequency of the automatic saving feature has to be quite high (at least every 15 minutes). This might annoy the user, because every 15 minutes the system halts for a few seconds in order to save the work.

A system crash may occur during the use of a database system, leading, probably, to the loss of important data and / or to the deterioration (e.g. inconsistencies) of the data that is left in the database. Since this data is of a paramount importance for the proper functionality of the organisation (that owns and uses the database), data recovery mechanisms have to be provided by the DBMS.

Definition: Data recovery (in the context of databases) means the capability of restoring a database, after a system failure (crash), to a previously known correct state.

Given the complexity of a database system (e.g. size, possibility of being used concurrently by many users), the data recovery mechanisms are more elaborate than the one mentioned in the previous example. However, they are still based on the same principle, namely *redundancy*.

Apart from its actual content, the database must maintain information about a correct state, which could be used in the case of a system crash. This redundancy has to be provided at the level of permanent support (memory), otherwise it would be lost, too, after the crash. Moreover, the user, theoretically, should not be aware of it. If there is a crash, it is the DBMS alone that should be able to restore its database to a correct state. In other words, whether the DBMS supports data recovery or not, the logical structure of the data is the same. Therefore, this redundancy occurs at the physical and not at the logical level.

The mechanisms of data recovery are based on the concept of transaction. So, before we look at recovery, we are going to look at transactions.

Transactions and transaction recovery

Consider a database of a company, including information about its employees and their children (the company needs information about children because at special occasions, such as Christmas, it gives them presents). This information is stored in two relations as illustrated in Figure 1.

Employees

E_id	EName	Department	Job	Salary	Children
Dev01	M. Black	Development	Programmer	28	2
Dev02	P. Hunt	Development	Analyst	31	0
Sal01	M. Briggs	Sales	Investigator	25	0
Supp01	P. Wells	User Support	Win-app-sup	22	1

Children

E_id	CName	Sex	DoB
Dev01	Joanne	F	3/04/95
Dev01	Mike	M	12/10/97
Supp01	Ann	F	30/05/96

Figure 1: The Employees and Children relations in the database of a company

Suppose that the development department employs another programmer, J. Smith, who has three children, Dan, Monica and William. The database must be accordingly updated. The following insert operations must be performed (expressed in SQL92):

```
INSERT ("Dev03", "J. Smith", "Development", "Programmer", 32, 3 )
      INTO Employees ;
INSERT ("Dev03", "Dan", "M", "4/03/94")        INTO Children ;
INSERT ("Dev03", "Monica", "F", "28/04/96")    INTO Children ;
INSERT ("Dev03", "William", "M", "15/08/98")   INTO Children ;
```

Suppose that an error occurs after the first two insert operations were successfully performed. The last two update operations will be lost (but the first two have been performed). As a result, the database will end in an inconsistent state: the Employees relation will specify that "Dev03" has three children, whereas the Children relation will contain information only about one child, "Dan".

Activity: You might think that the above drawback is due to the sequence in which the update operations were performed. Can you identify an order such that, if all the operations were initiated, but an error occurred in between, the database would not end into an inconsistent state?

The conclusion is that there are situations when a set of database operations only make sense if they are performed together; the performance of only a part of them would bring the database to an incorrect state (inconsistent state, more precisely). They represent a *logical unit of work.*

Such a situation occurs because a logical unit of work cannot be expressed in just one statement (of the database language) that is guaranteed to be either executed as a whole or not executed at all. For instance, the update above – a logical unit of work - expressed in natural language as

"J. Smith, who has three children, Dan, Monica and William, born on 4/03/95, 28/04/96 and 15/08/98, respectively, is employed for the Development Department, as a Programmer, for 32k a year"

can only be stated by means of *four* insert operations. Therefore, in order for these four operations to make sense (i.e. to express the update stated above in natural language), they have to be performed together, as a whole.

Definition: A transaction is a sequence of database operations that represent a logical unit of work. They transform the database from a consistent state to another consistent state.

Note that during the performance of a transaction (i.e. between the beginning and the end of the transaction) the database can be in an inconsistent state. It is only *between transactions* that the database is *guaranteed to be in a consistent state* (of course, we assume that the user of the database makes no mistakes; e.g. the user of the database will not attempt an update of the kind "J. Smith, who has *three* children, Dan and William, born on 4/03/95 and 15/08/98, respectively, is employed for the Development Department, as a Programmer, for 32k a year".

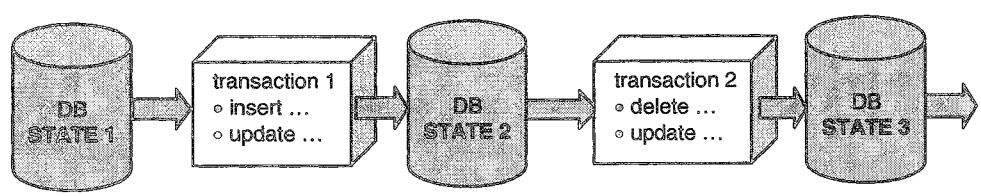

Figure 2: The evolution of a database

If a system supports transactions, then it should guarantee that all the constituent operations of the transaction will be executed once the transaction is initiated. This requirement is not achievable. However, a weaker requirement may be satisfied, namely:

Principle of transaction processing support: If some (not all) operations of a transaction are performed and a failure occurs (before the planned termination of the transaction), then the DBMS must guarantee that those operations will be undone.

This all-or-nothing feature ensures the *atomicity* of a transaction - a transaction cannot be split, *execution-wise*, into smaller components. Therefore, the evolution, in time, of

the database can be see as a sequence of correct states, the transition from one state to another being done only through successfully executed transactions (Figure 2).

The module of the DBMS that implements transaction support is called the *transaction manager*. There are two mechanisms that the transaction manager supports:

- COMMIT TRANSACTION – specifies a successful end of a transaction; i.e. it guarantees that all the corresponding operations of the transaction were performed (on the permanent support);
- ROLLBACK TRANSACTION – specifies an error occurred during a transaction; accordingly, all the successfully performed updates (prior to the error) have to be undone (rolled back), in order to preserve the consistency of the database.

Let us look at what happens during the execution of a transaction. We are going to illustrate this using an ad-hoc pseudo-code. A transaction can be specified as below.

```
BEGIN TRANSACTION ;
    INSERT ("Dev03", "J. Smith", "Development", "Programmer", 32, 3 )
           INTO Employees ;
    IF any-error-occurred THEN GO TO undo;
    INSERT ("Dev03", "Dan", "M", "4/03/94")          INTO Children ;
    IF any-error-occurred THEN GO TO undo;
    INSERT ("Dev03", "Monica", "F", "28/04/96")      INTO Children ;
    IF any-error-occurred THEN GO TO undo;
    INSERT ("Dev03", "William", "M", "15/08/98")     INTO Children ;
    IF any-error-occurred THEN GO TO undo;
COMMIT TRANSACTION ;
    GO TO continue ;
undo: ROLLBACK TRANSACTION ;
continue:
```

The beginning of the transaction is specified by a "BEGIN TRANSACTION" statement. Then, the first update (insert) operation is performed. There can be two possible outcomes.

1. The operation was successfully performed (usually, in the internal memory) and the transaction manager successfully recorded the result of this operation on the permanent support (memory).
2. An error occurred (i.e., either the operations was not successfully performed in the internal memory or its result could not be recorded on the permanent support).

If the result is the former, (1), then the DBMS will continue with the execution of the next operation of the transaction. If the result is the latter, (2), then it makes no sense to continue, for one operation of the transaction was unsuccessful. In this situation, the execution of the transaction is terminated by a jump (GO TO) to a ROLLBACK statement. When issued with a ROLLBACK, the transaction manager undoes all the operations of the current transaction that were performed so far. This is based on the information that was stored on permanent support (we referred to it as "the result of this operation" in 1 above).

During a transaction, the DBMS records, on permanent support, the description of each performed operation. This record is called the *log* or *journal* (Figure 3).

A clarification is required. The DBMS copies parts of the database in the internal memory (buffers) and performs the operations on these copies. The main reason for this is efficiency – the internal memory is much faster than the external memory. From time to time these buffers are forcibly written to the external support. In the situation above, performing an operation of a transaction might mean performing it only on the copies of base relations in the buffers. If the system crashes, these updates will be lost. Therefore, their description must be recorded in the log util they can be performed permanently.

The information recorded in the log is sufficient for:

- undoing any performed operation, in case an error occurs before the planned completion of a transaction;

- performing all the operations of a transaction on permanent support, even if everything (all the updates) was lost from the internal memory (as the situation would be in case of a system crash).

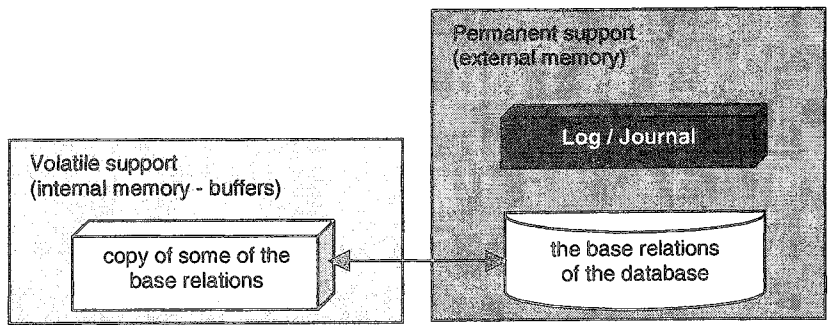

Figure 3: A log/journal is needed for transactions support

Let us now return to the main thread of discussion, the description of transaction execution. If all the operations of the transaction were successfully performed and the relevant information was successfully stored on the permanent support, a COMMIT TRANSACTION statement can be issued. A COMMIT statement guarantees that all the operations of the transaction may be performed on the permanent support – the log contains all the necessary information for this.

The point in time where a COMMIT TRANSACTION statement is issued is called a *commit point*, or *syncpoint*. Once a commit point is reached, it is certain that no update operation from the corresponding transaction will have to be undone – the respective update operations are permanent. Between a commit point and the beginning of the next transaction, the database is available for any retrieve operations. Between a begin transaction and the corresponding commit point, the database is probably in an inconsistent state and certainly inaccessible to operations other than those constituting the transaction (Figure 4).

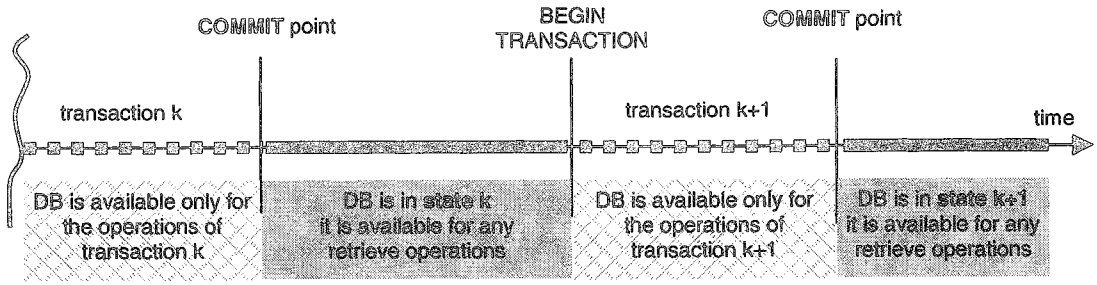

Figure 4: The states of a database in case of successful transactions

In case a transaction cannot be successfully completed, the database has to recover to the previous correct state; the ROLLBACK statement brings the database to the state it was left in by the last successful transaction. This mechanism is called *transaction recovery* and is depicted in Figure 5.

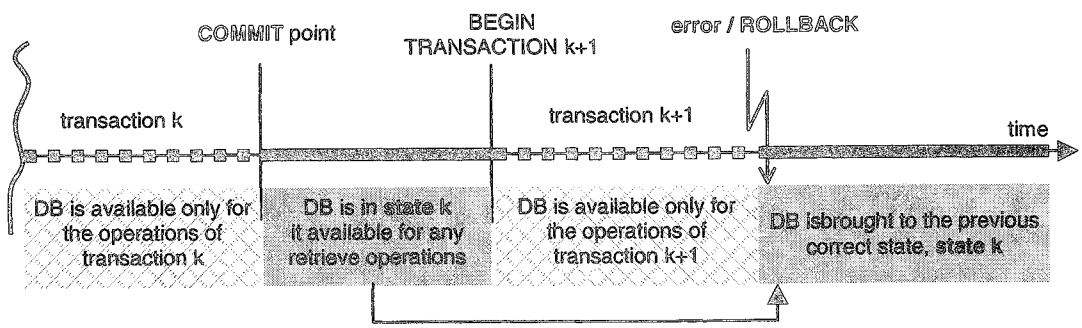

Figure 5: The states of a database in case of an unsuccessful transaction

It can be concluded, from the above description, that a transaction is also the unit of recovery, not just the unit of work. The log plays an important part in the mechanism of transaction processing. The log can successfully be used in transaction processing only if the relevant information about each operation of a transaction is completely entered before the operations is started. This rule is known as the *write-ahead* rule.

We have made a tacit assumption so far: the constituent operations of any transaction are atomic. An atomic operation cannot be performed partially; it is either completely performed or it is not performed at all. Consider, for instance, the result of the first insert operation in the above example. If this is successful, the result will be a new tuple in the relation Employees, containing *all* the information specified in the insert statement. If it is not successful, the relation Employees will not be modified at all. It is not possible to have, for instance, only {"Dev03", "J. Smith", "Development"} inserted into Employees.

This issue does not belong to transaction management. It is resolved by other mechanisms of the DBMS – the mechanisms that implement *set level* operations.

In the end of this section, four important properties possessed by transactions are enumerated (they are referred to as the ACID properties):

- *Atomicity* – either all the operations constituting a transaction are performed or non of them.
- *Consistency* - the transaction mechanism guarantees that the DB evolves (is transformed) from one consistent state to another.
- *Isolation* - transactions are isolated from one another, in that, when a transaction is performed on a database, no other transactions are (usually) allowed to be performed at the same time[2].
- *Durability* - once the transaction was committed, the updates it subsumed are guaranteed to be physically performed.

The transaction processing mechanism is at the basis of the database recovery mechanisms that is presented in the following section.

Database recovery

The previous section dealt with single transactions on the database. If an error occurred during a transaction, the DBMS *recovered* from it by undoing (rolling back) the operations that were already performed. However, a database is a shared resource. Therefore, it is highly probable that the DBMS has to deal with more than one transaction at a time.

It is not just a transaction that can encounter errors. The whole system (DBMS) itself can fail to function properly. The cause can be a power failure, an error that occurred in the operating system, etc. In the present chapter, we only refer to *system failure*, also known as *soft crash*; this means that the software system (i.e. the DBMS) ceases to function properly, but the database (i.e. by the permanent support or external memory) suffered no physical damage. In the case of a soft crash, the data stored on the permanent support remains unaltered. Even if parts of the internal memory or parts of the processing unit are physically damaged we are still dealing with a soft crash. This is the case as long as the permanent support remains undamaged. The other kind of failure, *media failure*, will not be discussed in this section.

Consider that, while the DBMS is processing several transactions, a failure occurs in the system. After the system is restarted, the database has to be brought to a consistent state; the system (DBMS) has to recover from the failure. This means that there must be a

[2] Two or more transactions can be performed concurrently on the same database providing they access different parts (disjoint) of the database. This is further explained in the section on concurrency.

way for the system to deal with the transactions that were in progress at the time of the failure. This is done based on the information kept in the log.

Much of the data processing takes place in buffers; the DBMS takes a copy of the data it needs and manipulates it in the internal memory. At certain moments of time the system automatically writes (*force writes*) on the permanent support, all the data existing in the buffers. Together with this, a list of all the transactions that were in progress at the time of writing is recorded in the log – i.e. the *checkpoint record*. Then the system carries on with the data processing until the next force writing point and so on.

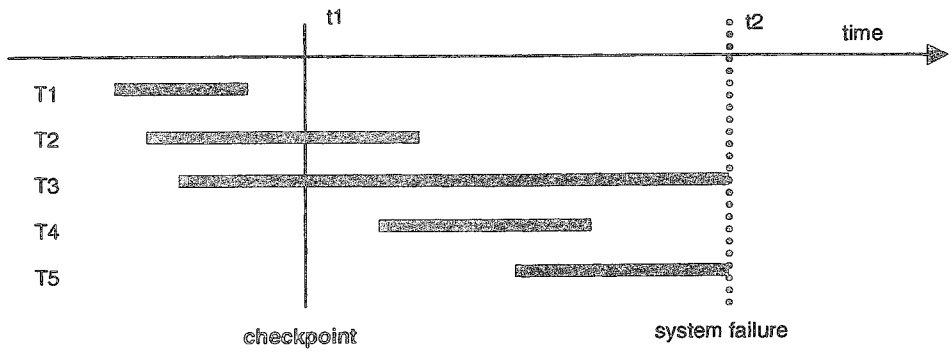

Figure 6: The five types of transaction possible with respect to a checkpoint

Suppose a system failure occurs at time t2 and that the most recent checkpoint was at time t1. There are five possible types of transactions in process that can be identified (Figure 6):

- T1 – was completely successfully before t1 (at time t1 it was written on permanent support and its successful termination was recorded in the log);
- T2 – was initiated before t1, was completely successfully after t1 but before t2 (it had not been written on permanent support, but its successful termination was recorded in the log);
- T3 – was initiated before t1 and was not completed at the time of failure (only its initiation was recorded in the log);
- T4 – was initiated after t1 and was successfully completed before t1 (its successful completion was recorded in the log);
- T5 – was initiated after t1 and was not completed at the time of the failure (only its initiation was recorded in the log);

The operations that the system undertakes to recover from the failure (i.e. just after the restart, before it starts any other data processing) are:

- the transactions of type T1 do not need to be consider during this process, because they have already been successfully completed (their results have been written on the permanent support);
- the transactions of type T2 and T4 have to be redone (based on the information recorded in the log), because they have been successfully completed but have not been recorded on permanent support (therefore they were lost during the crash);
- the transactions of type T2 and T5 have to be undone, because they were not successfully completed at the time of the crash and the log does not contain sufficient information in order for them to be re-initiated.

You can find the description of the restart procedure (based on the above points) in (Date 1995, p.381).

SQL support

SQL supports transaction-based recovery. This is based on the two mechanisms presented above, namely commit and rollback. SQL implements them in the two respective statements COMMIT and ROLLBACK.

There is a distinction from the process described above, in that SQL does not implement an explicit BEGIN TRANSACTION statement. In SQL, a transaction begins implicitly with the first *transaction initiating* statement issued just after the end of an explicit COMMIT or ROLLBACK statement. It suffices for you to know that any data definition or data manipulation statement discussed in the chapter about SQL is a transaction initiating statement. However, you should also know that there are other SQL statements that are transaction initiating.

For further details about SQL's support for transaction based recovery consult the manuals of the SQL dialect you are currently using.

Two-phase commit

A database can be split into several resources, each managed by its own *resource manager*. For instance, the database of a certain company can be split into several parts, relevant to each of the company's departments, i.e. Production, Sales, Finance and Personnel. They can all exist on different machines (or platforms), each having its own resource manager[3].

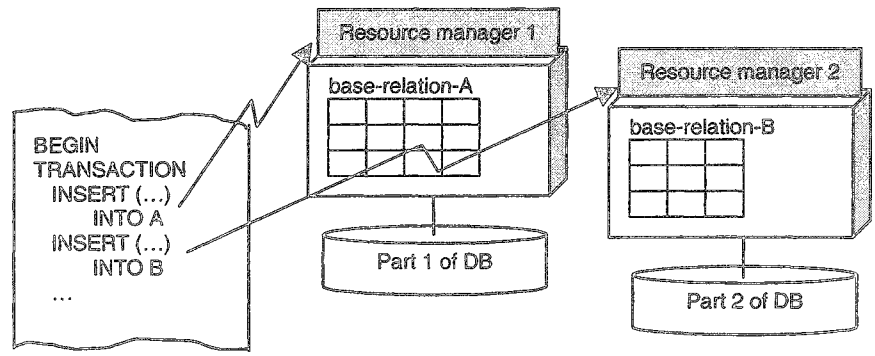

Figure 7: A transaction may involve more than one resource manager

A transaction may involve more than just one resource manager (Figure 7). For instance, the sale of certain products may require the updating of both some base relations belonging to the Sales department and of some base relations belonging to the Finance department. In such a case, the DBMS must make sure that the atomicity property of the transaction is preserved; the DBMS must ensure that either all resource managers perform their updates locally or that none of them performs any update. The situation in which, say, the records of the Sales department are updated but those of the Finance department fail to be updated would generate an inconsistent database.

The fact that there are different resource managers involved into a transaction adds a new dimension to the problem of recovery. The system (DBMS) has to implement a *global* COMMIT and a *global* ROLLBACK statement. Once that a global COMMIT was issued, the DBMS guarantees that all the operations of the transactions will be committed, even in the case of a soft-crash of any of the resource managers or of the DBMS itself. Conversely, in case one of the resource managers fails to execute successfully one of its operations, then the transaction's operations for all the resource managers have to be rolled back. This is achieved through a system component called the *co-ordinator*.

The co-ordinator works according to the following algorithm. Suppose that the transaction's constituent operations have completed successfully, thus the co-ordinator is issued with a (global) COMMIT statement.

---

[3] This will be discussed further in the chapter about distributed database.

- *Phase 1*: the co-ordinator prepares each resource manager that is involved in the transaction, by sending it a "get ready" message. On receiving this message, each resource manager must force write in their log the description of all the operations it performed. Once this information is recorded on permanent support, the respective operations will be able to be committed or rolled back, irrespective of any potential soft crash. If the force writing process is completed successfully, the resource manager sends an OK to the co-ordinator; otherwise it sends a NOT OK message.
- *Phase 2*: upon receiving all responses, the co-ordinator takes a decision regarding the transaction. The decision is to commit if all the answers were OK and to rollback otherwise. The co-ordinator records the decision in its own log (so that, in the eventuality of a crash, this decision can be recovered) and then sends it to all the participant resource managers. Even if a failure occurs, each resource manager is guaranteed to execute the co-ordinator's decision, because all the relevant information was recorded on permanent support.

This mechanism ensures the atomicity of transactions, but presents a drawback in that the resource managers depend on the co-ordinator. If the latter fails, some resource managers may be left in a wait status, awaiting the co-ordinator's decision.

Concurrency control

Concurrency problems

Because a database is a shared resource (this being one the advantages presented by the database approach), it is natural to expect the DBMS to allow *concurrent access* to data – access at the same moment in time - of different users or application programs. We gave, in the previous section, a classification of the transactions with respect to checkpoints. We tacitly assumed that it was possible for them to access the database concurrently.

Where each transaction (of a set of concurrent transactions) accesses disjoint parts of the database (i.e., that have no data in common) there should be no problems. However, problems occur when *the same data* is accessed by two or more transactions at the same moment of time.

There are three important problems that might occur as a result of concurrent access on the same data:

- the inconsistent analysis problem;
- the lost update problem;
- the uncommitted dependency problem.

To illustrate the inconsistency analysis problem, we look at a real life example. Consider the situation of an intermediate supplier, say "Alimento Inc.", who includes in their database information about the products in stock. One of the products they supply is sugar. Alimento Inc., in turn, purchases sugar from three factories, say A, B and C. There are three corresponding tuples in the database, say SugarA, SugarB and SugarC, that denote the quantity of sugar (in kilograms) that is in stock. Suppose their initial value, at time t0, was 500, 200 and 100 respectively.

At a time t1, a transaction T1 is initiated, to check the total quantity of sugar (Figure 8). At time t2, the value of SugarA is retrieved by transaction T1 and has the value of 500. At time t3, another transaction, T2, is initiated, to update the value of SugarA to zero, because, at that time, Alimento Inc. sent of it to one of their customers. T2 successfully (COMMIT) performs the update before T1 is completed. When T1 is finished, the result it provides is 800. However, according to the actual data in the database this should have been 300. So, the result provided by T1 is not consistent with the data in the database.

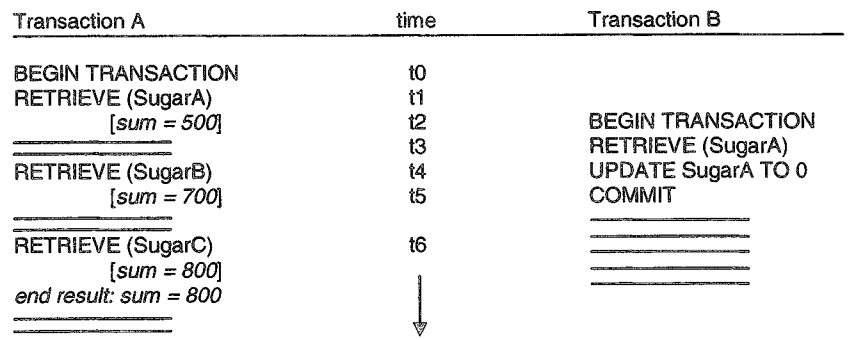

Figure 8: The inconsistent analysis problem

For the lost update problem, consider two transactions, A and B, attempting to update the same tuple r of a relation R, but with different values, say r1 and r2, respectively. Consider, further, that no concurrency control exists and that the two transactions are executed in time as depicted in Figure 9. The update that transaction A has performed at time t3 is overwritten by the update performed by transaction B at time t4. Therefore this update is simply lost, without the application that initiated the update being aware of this.

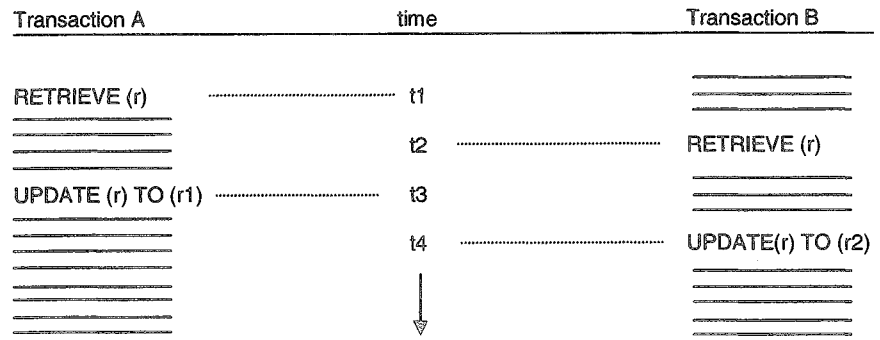

Figure 9: The lost update problem

Another situation that should be avoided is when a transaction performs an update, another transaction retrieves the data that has been updated, but at a later stage the initial transaction has to be rolled back. In this case the latter transaction uses data that does not exist in the database. This situation is illustrated in Figure 10.

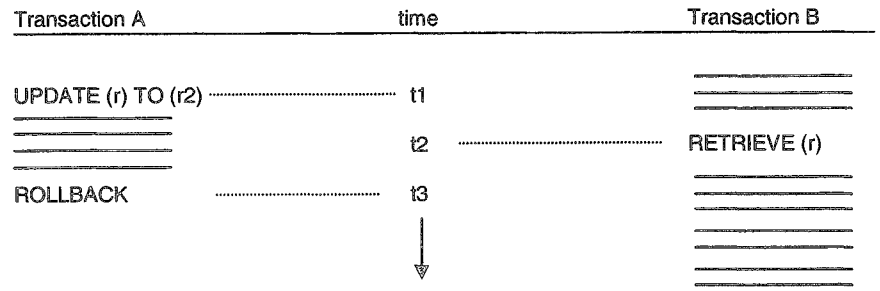

Figure 10: One version of the uncommitted dependency problem

Transaction A updates a tuple r of a relation R, from an initial value r1 to a new value r2. Subsequently, transaction B accesses tuple r and retrieves the value r2. At a later time, A fails and therefore has to be rolled back. The value of r is therefore r1 (and not r2). The problem is that the application that initiated B uses a value that did not exist in the database.

Another variant of the same problem occurs when instead of retrieving r, transaction B performs an update operation based on the value of r; at time t2 this value is r2 but this

value is lost at time t3. Thus, B performs an update based on a value that did not exist in the database.

All these problems are caused by the simultaneous (concurrent) access of two (or more) transactions to the same resource. In order to avoid such problems, a mechanism exists, that prevents this from happening; it is called *locking*.

Locking

By and large, the locking mechanism provides a means for preventing two or more transactions to concurrently update the same data.

Before a transaction uses some data, it needs to ask for permission to use it. Depending on whether this data is being used or not (and also on the kind of operation intended), the permission can be granted or refused. If the permission was granted, then the respective data is "marked" – *locked* – as being in use. In other words, a transaction can use an object only if it can acquire a *lock* on that object.

For the purpose of the present chapter we shall assume that the tuple is the only lockable database object. If you are interested[4], you can explore other kinds of lockable objects in (Date 1995, pp. 404-406).

There are two basic types of locks:

- *Exclusive* or *write locks*. Any transaction that needs to modify a certain tuple must acquire an exclusive lock on that tuple. Once a transaction has acquired an exclusive lock on a tuple t, no other transaction can access that tuple, i.e. acquire any kind of lock, until the former transaction releases the lock on t.
- *Shared* or *read locks*. Any transaction that needs to retrieve data from a certain tuple must acquire a shared lock on that tuple. There can be any number of shared locks on a tuple. If some transactions (each) hold a shared lock on a tuple t and another transaction requests a shared lock on the same tuple, the lock is granted. However, under the same assumption as above, if another transaction requests an exclusive lock on t, this lock is not granted.

This information is concisely expressed by the compatibility matrix (Figure 11). Explanations: "t" stands for any given tuple, "X" means an exclusive lock and "S" a shared lock. Referring to the first line of the matrix, an X lock on a tuple t can be granted only if there are no other locks on t.

request for lock on t \ existing lock on t	X	S	none
X	refused	refused	granted
S	refused	granted	granted
no request	-	-	-

Figure 11: The compatibility matrix

A *data access protocol* was defined based on the locking mechanism.

- A transaction that only needs to retrieve a tuple must first acquire an S lock on that tuple.
- A transaction that needs to update / modify a tuple must first acquire an X lock on that tuple.
- If a transaction T1 is refused a lock on a tuple t because another transaction T2 hold a lock on t, then T1 goes into a wait state until T2 releases the lock on t. When more than one transaction waits for a lock on the same tuple, the order in which they are served (considered) is decided by a sub-module of the concurrency control

[4] This is not a compulsory issue; explore further into it only if you are interested.

module, called a scheduler. The scheduler must make sure that no transaction is left to wait forever.

- All locks are normally held until the end of the transaction (i.e. it is the COMMIT or the ROLLBACK statement that releases the lock held by a transaction).

We revisit the concurrency problems, but this time the access to data is made according to the protocol previously described. Reconsider the uncommitted dependency problem presented in Figure 10. If the data access protocol is employed, the behaviour is that described in Figure 12.

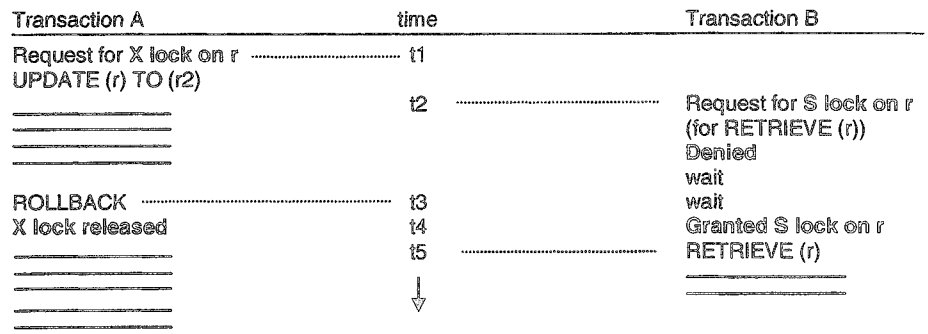

Figure 12: The uncommitted dependency problem is solved if the access is made by employing S and X locks

Transaction A, before updating r, requests an X lock on r. Supposing that no other transaction held a lock on r at t1, the request is granted (therefore the update is performed and the transaction continues). At t2, transaction B requests for an S lock on r (for the retrieve statement). Because this happens before A was completed, and so before the X lock was released, the request is denied and B enters a wait state.

At t3, because of an error, transaction A is rolled back. Th effects are: (1) the update is lost and (2) the X lock is released. When A releases its lock on r, B is in a position to receive the S lock. Supposing that the scheduler prioritises B amongst all the other transactions waiting for a lock on r, transaction B is granted the S lock. Therefore it can now perform the retrieve operation. This time, B retrieves the "real" value of r, i.e., the value that exists in the database. The uncommitted dependency problem is solved.

Nevertheless, the data access protocol is not the perfect solution to all the concurrency problems. There are situations when the concurrency problem is dealt away with, but another problem is introduced - deadlock. Such situations are described in the next section.

Deadlocks

Reconsider the lost update problem. The behaviour without locks was described in Figure 9. The new approach is now illustrated in Figure 13.

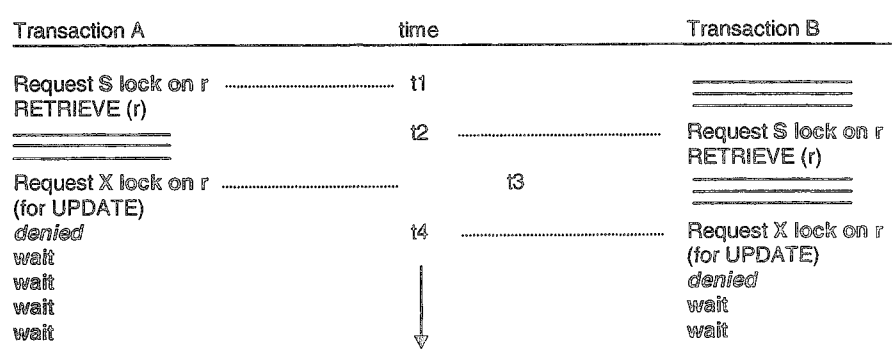

Figure 13: A deadlock generated by applying the data access protocol to the lost update problem

Transaction A, before retrieving r, has to acquire an S lock on r. Supposing r was unlocked, A's request is granted. At time t2, transaction B asks for an S lock on r (before the retrieve statement). In accordance with the compatibility matrix, this lock is granted as well. Now, both A and B have an S lock on r.

At time t3, A requires to update r, therefore it needs an X lock on it. Transaction A is denied this lock because of the S lock held by B, therefore it enters a wait state (until the S lock is released by B). At time t4, B requires an X lock on r. Transaction B, too, is put in the wait state, because A has an S lock on r.

After t4, both transactions, A and B, are waiting for each other to release the S lock. A waits for B, but B will not release the lock because it is in a wait state. Similarly, B waits for A, but A will not release the lock because it is in a wait state. This situation is called a *deadlock* and there is no "natural" way out of it. The only possibility to break such a deadlock is to forcibly terminate (rollback) one of the transactions.

In conclusion, the data access protocol based on S and X locks solved the lost update problem but generated another one, namely a deadlock.

A deadlock can also occur when two transactions work with the same two tuples (data objects, more general). Such a situation is depicted in Figure 14.

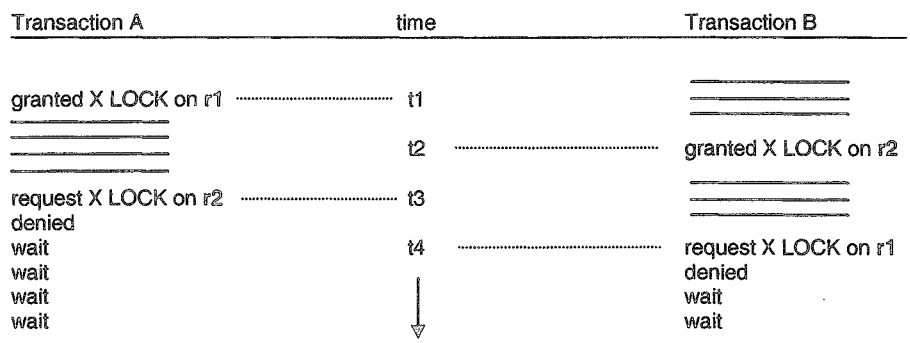

Figure 14: A deadlock due to accessing two data objects

Definition: A deadlock is a situation in which two (or more) transactions are in a wait state, each waiting for the other to release a lock.

It is possible to have more than two transactions in a deadlock, but this case is very rare in practice (Date 1995).

A deadlock will never break on its own, therefore the system (DBMS) must identify such situations and resolve them. Each system maintains a wait-for directed graph, stating which transaction is waiting for which.

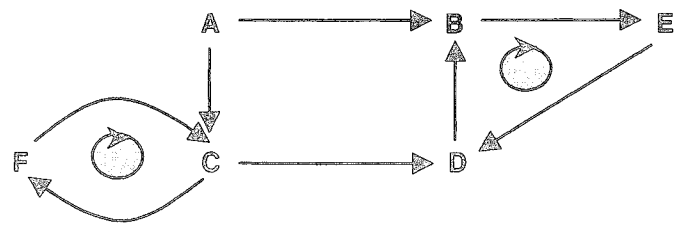

Figure 15: A wait-for graph

For instance, in Figure 15, A is waiting for B and C, C is waiting for D and F, B is waiting for E, etc. If the wait-for graph contains a cycle, then the respective transactions are in a deadlock situation. For instance, in the above example {B, E, D} are in a deadlock and {C, F} are in a deadlock. Note that the graph is directed, therefore ACDBA does not constitute a cycle. The systems checks for cycles in the wait-for graph at regular interval of times and thus identifies the deadlock states.

Once a deadlock is identified, the system terminates one of the transactions involved, by rolling it back. Such a transaction is usually called a *victim*. As a result of rolling back the victim, the locks it held are released, thus the deadlock is resolved. The methods of choosing the victim from the set of transactions in a deadlock are not within the scope of this chapter. Usually, the victim is automatically restarted at a later stage.

The locking mechanism, even if it might generate deadlocks, allows the development of a protocol that, if followed, guarantees that any concurrent execution of some transactions is correct. The next section presents this very important issue.

Serialisability

Let us reconsider some of the concurrency problems. In the case of the inconsistent analysis problem (Figure 8), if the end result (sum = 800) is written back, it would produce an incorrect (inconsistent) database; this is because the *actual* result, according to the data in the database should have been different (i.e. 300). In the case of the uncommitted dependency problem (Figure 10), if transaction B uses the retrieved value of r in some calculations and then writes the result back, it would generate an incorrect (inconsistent) database; this is because transaction B used the result of A's update operation as the value for r, but A was subsequently rolled back, so A's update operation had no actual effect on the database.

The reason for the database being left in an incorrect state is that the transaction were executed concurrently (hence interleaved) without the locking mechanism being employed. On the one hand, for practical reasons, transactions have to be allowed to be executed concurrently (inherently, interleaved). On the other hand, the DBMS should not allow transactions to be executed in such a way that their result is an incorrect database. Therefore, a criterion should be found, to decide whether a concurrent execution of a set of transactions would lead to an incorrect database or not.

Such a criterion exists, i.e. a criterion for correctness for concurrency control, and it is based on the concept of *serialisability*. The statement of this criterion is the following.

Statement: The interleaved execution of a set of transactions is correct if it is serialisable.

Definition: An interleaved execution of a set of transactions is serialisable if it produces the same result as some serial execution of the same set of transactions, one at a time.

For a justification of this statement refer to (Date 1995, pp. 400-401).

The way a set of transactions is executed is called a *schedule*. A schedule of a set of transactions specifies the order in which the transactions' operations are executed. If the operations in each transaction are executed in the given sequence, contiguously and not interleaved with other operations, then the schedule is said to be *serial*. Otherwise it is *interleaved*.

The above statement can be restated as: if an interleaved schedule of a set of transactions is equivalent to a serial schedule of the same transactions, then it is correct. An illustration of the concept of serialisability is given in Figure 16.

Interleaved schedule for transaction 1, 2 and 3

time	t1	t2	t3	t4	t5	t6	t7	t8	t9
Transaction 1	op11			op12			op13		op14
Transaction 2		op21			op22	op23			
Transaction 3			op31					op32	

Equivalent serial schedule (Transaction 1 then Transaction 3 then Transaction 2)

time	t1	t2	t3	t4	t5	t6	t7	t8	t9
Transaction 1	op11	op12	op13	op14					
Transaction 2							op21	op22	op23
Transaction 3					op31	op32			

Figure 16: A serialisable schedule

The interleaved transactions, of the examples mentioned at the beginning of this section, resulted into an incorrect database, because they were not serialisable. They were not serialisable because the effect they had on the database could not have been obtained by any serial execution (i.e. either A-then-B or B-then-A).

However, all the interleaved transaction executed according to the data access protocol (i.e. using the locking mechanism) were serialisable; all, including the ones that resulted in a deadlock. Therefore, their result, each time, was a correct database. As a matter of fact, it was the employment of the data access protocol that determined their execution to be serialisable.

A condition exists, that, if complied with, ensures that all possible interleaved executions of a set of transactions are correct.

Theorem: If all the transactions from a set S comply with the two phase locking protocol, then all the possible interleaved executions of the transactions of S are serialisable.

The *two phase locking protocol* (a generalisation of the data access protocol presented in the previous section), requires that:

- a transaction must acquire a lock on any object before accessing it;
- after releasing a lock a transaction must not attempt to acquire other locks.

The theorem above states that, given a set of transactions S, if they all comply with the two phase locking protocol, then for any possible interleaved schedule there exists a serial schedule that is equivalent to it. Note that the theorem does not state that any interleaved execution of a certain set of transactions will be equivalent to the *same* serial schedule (if the two phase protocol is complied with). Different interleaved schedules may be equivalent to different serial schedules, producing different effects on the database. For instance, the interleaved schedule of transactions 1, 2 and 3 in Figure 16 was equivalent to the serial schedule

Transaction 1 - Transaction 3 - Transaction 2

whereas the interleaved schedule, say

op31 - op11 - op12 - op13 - op21 - op14 - op 32 - op22 - op23

may be equivalent to the serial schedule

Transaction 3 - Transaction 1 - Transaction 2

However, each possible interleaved schedule has at least one serial schedule that it is equivalent to and therefore it produces a *correct* database.

SQL support

SQL92 does not provide support for explicit locking. However, it specifies that the updates made by a certain transaction do not become visible to other transactions, until the initiating transaction successfully terminated with a COMMIT statement. In case of a roll back, the updates are undone, and so are not be visible to other transactions.

Data security

The database is a shared resource where an institution stores all its relevant information. For instance, a university's database stores information about its departments, staff, courses, students, etc. The users are usually allowed to access (see or modify) only parts of it. For instance, students can see the information about the courses and about themselves, but they are not allows to modify any of it, nor are they allowed to see other parts of the database (such as information about departments or staff). Lecturers can see everything that students can, plus information about their own department and students. They are also allowed to modify some of the information about their students (e.g. certain exam results). And the list can continue. The conclusion is apparent: every DBMS must provide a mechanism that guarantees that users can do only the operations they are allowed to, in brief, a mechanism for *data security*.

The security restrictions can originate from different sources: legal, ethical, political, strategic, etc. For instance:

- *legal*: the personal information about the employees of a company (owned by the Personnel Department) has to be confidential;
- *ethical*: the exam results of the students can (legally) be made public, but, because some of them might feel embarrassed (bad results) in front of their peers, this information is better kept confidential;
- *strategic*: some universities may decide to make public the information about their financial situation because they are doing well and think this may be a way of advertising; other universities may decide to keep it private.

Activity: Select an application area of your choice and identify a set of security restrictions.

The problem of identifying security restrictions does not fall within the area of databases. For this course we assume that such security restrictions have already been identified and we focus on ways of expressing and enforcing them in a database system.

Each DBMS must provide a security mechanism that consists of

- a *definition language*, by means of which the security restrictions are made known to the system; not only must the rules be made known, but they also have to be remembered; therefore, they are *stored* in the catalogue;
- a *security subsystem*, by means of which the restrictions are enforced.

There are two main possible approaches to data security:

- *mandatory* – users access data according to certain classifications levels (for both);
- *discretionary* – users access data according to their privileges / authorities which are explicitly stated for each user and data object in part.

In the mandatory approach each data object is given a certain classification level and each user a certain clearance level. A user U can see an object O if the clearance level of U is greater than or equal to the classification level of O. A user U can modify an object O only if the clearance level of U is equal to the classification level of O. The rationale behind the latter rule is the prevention of users with a lower clearance level from *copying* important data of higher classification level. Note that the first rule does not prevent them from seeing it.

In the discretionary approach, security restrictions are specified by means of rules. For instance, suppose that the undergraduate director of a department is allowed to see all the information about the department's undergraduate students (kept in the Students-Info relation), but is only allowed to insert, delete and update tuples referring to undergraduates. This restriction can be expressed in the form of a rule, informally, as follows:

```
SECURTY RULE   modifications-made-by-undergrads-director
ALLOW          the undregraduate director identified by the user id sstaff001
TO             INSERT, DELETE, UPDATE
ON             Students-Info    WHERE    Type = "undergraduate"
ON VIOLATION   REJECT
```

This example illustrates that a security rule should have:

- a *name* – used for the rule's identification (e.g. for modifying it at a later stage or for referring to it in error messages);
- a *user specification* part – specifies the users which the rule should be applied to;
- an *operations enumeration* part – specifies the operations that the users, previously mentioned, are allowed to perform; these operations are usually referred to as *privileges*;

- a *scope definition* part –specifies the data objects (the part of the database) that the rules applies to; this is usually a relational expression (relational algebra or relational calculus);
- a *violation response* – specifies the action to be taken in case the rule was violated; this can usually be a procedure of any complexity.

SQL implements data security similarly to the way described above.

SQL support

SQL supports data security through its two statements GRANT and REVOKE. The former is used for defining, whereas the latter is used for removing security rules. Their general syntax is:

```
GRANT       @<privilege> | ALL PRIVILEGES
ON          <data object>
TO          @<user id> | PUBLIC
[WITH GRANT OPTION]

REVOKE      [GRANT OPTION FOR]  @<privilege>
ON          <data object>
FROM        @<user id> | PUBLIC
[RESTRICT | CASCADE]

<privilege> ::= USAGE | SELECT | INSERT | DELETE | UPDATE |
                REFERENCES
<data object> ::= DOMAIN <domain name> | [TABLE] <table name>
```

The Grant clause specifies a list of privileges. INSERT and DELETE can be column specific, but SELECT cannot. A USAGE privilege is necessary for being able to access a domain; a REFERENCES privilege is necessary to be able to refer to a specific table when defining an integrity constraint; the meaning of the other privileges is clear from their name.

The ON clause specifies the data object to which the security rule applies. It can be a domain or a named table (including views).

The TO clause specifies a list of users, by means of their ids, to which the rule applies. PUBLIC specifies all the users in the system.

WITH GRANT OPTION, if stated, specifies that the users who the rule applies to can grant any of the rule's privileges to other users.

In SQL security rules have no name. This is why the syntax for removing a rule (REVOKE) is a bit more complex. In the context of the above explanation, the REVOKE statement is self explanatory, apart from the last optional clause. If the option is RESTRICT, then the execution of the statement fails if some of the specified privileges were granted by some of the specified users to other users. For instance, suppose that the user U2 was awarded by U1 a privilege P on a relation R, with grant option, and then he granted it to another user U3. P cannot be revoked from U2, if the option was RESTRICT, as long as U3 still holds it. If the option was CASCADE, then P would have been revoked from U3 as well (and from other users that U3 granted it to, and so on).

In SQL, by default, the creator of any data object is automatically granted all privileges with grant option.

The grant statement is usually used in conjunction with views. The data object, to which a security rule applies, is defined by means of a view statement and the lists of users and privileges (on the data object, i.e. view) are specified by means of a grant statement.

Consider the two tables as illustrated in Figure 17, as part of the database of a bank branch.

Account_details

Acc_no	Owner	Type	Balance	OD_limit
0011234577	pjt01	current	+ 1789.45	500.00
1110023458	pjt01	savings	6400.00	0.00
0011134456	fjr01	current	- 345.67	1000.00
0022566433	jfk	current	10891.00	0.00

Customer_details

Owner	Name	Address	Telephone
pjt01	Paul J. Taylor	18A Cowley Road, Uxbridge	01985 134456
fjr01	Fiona J. Rider	133 Belmont Road, New Cross	0171 1233677
jfk	John F. Kent	15 Long Lane, Richmond	0181 9887624

Figure 17: A part of a bank database

Suppose that the branch has two secretaries (of user ids sec001 and sec002) who are allowed to get contact information for each of the accounts (i.e. the owner's name address and telephone), but are not allowed to see anything else, nor are they allowed to make any updates. Accordingly, they should only be allowed to see the Acc_no and Type attributes of the Account_details relation and all the attributes of Customer_details relation, except Owner. This rule is specified as follows:

```
DEFINE VIEW Contact_details AS
    SELECT      Acc_no, Type, Name, Address, Telephone
    FROM        Account_details, Customer_details
    WHERE       Account_details.Owner = Customer_details.Owner ;

GRANT     SELECT
ON        Contact_details
TO        sec001, sec002 ;
```

Paul J. Taylor (of user id cust1279) is allowed to see all the information about his accounts, but is not allowed to change any of it, nor is he allowed to see the information about the other customers' accounts (for correctness, in this case, we need to assume that customers are uniquely identified by their names). This rule is specified as

```
DEFINE VIEW PaulJTaylor-accounts AS
    SELECT      Acc_no, Type, Balance, OD_limit
    FROM        Account_details, Customer_details
    WHERE       Name = "Paul J. Taylor" AND
                Account_details.Owner = Customer_details.Owner ;

GRANT     SELECT
ON        PaulJTaylor-accounts
TO        cust1279 ;
```

Paul J. Taylor is also allowed to change his personal details (but not his owner id). This is expressed as:

```
GRANT     UPDATE (Name, Address, Telephone)
ON        Customer-details
TO        cust1279 ;
```

The security rules presented so far make no reference to the context in which the database system is used, such as user, terminal, date, time, etc. Therefore they are called context independent. Conversely, if system defined functions such as time(), date(), terminal(), user() are used in the definition of a security rule, that rule is said to be context dependent. For instance, user(), time() and date() can be used in conjunction to allow access to certain users only during weekdays and working hours (9:00 to 17:00); terminal() and user() can be used together in defining a rule that allows certain users access to the database only from certain terminals.

Activity: Explore how such rules (context dependent) can be defined in the SQL dialect you are using.

As a last idea in this context, a complex data security mechanism is useless if it can be bypassed. For instance it makes no sense for the DBMS to provide a very powerful language for security rules definition and a complex security subsystem if the operating system on which it runs provides little protection for its files.

Learning outcomes

On completion of this chapter you should be able to:

- explain the concept of transaction and the mechanisms that implement it (COMMIT and ROLLBACK);
- explain how transaction recovery is achieved in a DBMS;
- explain how database recovery can be achieved on the basis of transaction mechanism;
- describe the support for data recovery provided by the SQL92 dialect you are using;
- explain how the two-phase commit protocol can be used for providing recovery support for a distributed database system;
- discuss the possible problems due to concurrent access;
- define and explain the locking mechanism;
- define and explain the concept of deadlocks;
- discuss serialisability as a criterion for the correctness of concurrent executions;
- discuss the issue of data security;
- describe the support provided by SQL92 for security control.